DevPartner 10.6.1

Advanced Error Detection Techniques

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK
http://www.microfocus.com

Copyright © Micro Focus 2001-2012. All rights reserved.

MICRO FOCUS, the Micro Focus logo and Visual COBOL are trademarks or registered trademarks of Micro Focus IP Development Limited or its subsidiaries or affiliated companies in the United States, United Kingdom and other countries.

All other marks are the property of their respective owners.

Table of Contents

Preface . . . 7
Who Should Read This Manual . . . 7
What This Manual Covers . . . 7
Conventions Used In This Manual . . . 8
Getting Help . . . 8
Contact . . . 9

Chapter 1 · Workflow and Configuration Settings . . . 11
DevPartner Error Detection Workflow . . . 11
Benefits of the DevPartner Error Detection Workflow . . . 12
Saving Error Detection Configurations . . . 12
Using Error Detection from the Command Line . . . 12
Compiling, Instrumenting, and Building Unmanaged (Native) C++ Projects with NMDE-VENV . . . 14
Instrumenting Native C/C++ Code with nmvcbuild . . . 14
Customizing the DevPartner Error Detection Settings . . . 15
General . . . 16
Data Collection . . . 16
API Call Reporting . . . 17
Call Validation . . . 17
COM Call Reporting . . . 17
COM Object Tracking . . . 17
Deadlock Analysis . . . 18
Memory Tracking . . . 18
.NET Call Reporting . . . 19
.NET Analysis . . . 19
Resource Tracking . . . 20
Modules and Files . . . 20
Fonts and Colors . . . 20
Configuration File Management . . . 20

Chapter 2 · Checking and Analyzing Programs . . . 23
Error Detection Tasks . . . 23
Finding Leaks . . . 23
Finding Pointer and Memory Errors . . . 23
Finding Memory Corruption . . . 24
Analyzing Transitions to Legacy Code in .NET Applications . . . 24
Validating Win32 API Calls . . . 25
Searching for Application Deadlocks . . . 26
Expanded Uses for DevPartner Error Detection . . . 26
Understanding Complex Applications . . . 26

Reverse Engineering 28
Stress Testing 30

Chapter 3 · Analyzing Complex Applications 33
About Complex Applications 33
Wait for Process 34
Analyzing Limited Parts of Your Program 34
Using Modules and Files Settings 36
Deciding What to Monitor 38
How Does an Application Start Up? 39
Analyzing Services 39
Requirements and Guidelines 39
Analyzing a Service 40
Timing Problems and dwWait 40
Alternate Method: Separating Control Logic from the Worker Thread 40
Custom Code to Turn the DevPartner Error Detection Log On and Off 40
Common Service-related Issues 40
Analyzing ActiveX Controls Using the Test Container 41
Common Test Container Issues 42
Analyzing Applications That Use COM 43
Common COM Issues 44
Analyzing ISAPI Filters Under IIS 5.0 45
Common ISAPI Filter Issues 46
Analyzing ISAPI Filters under IIS 6.0 46
IIS 5.0 Isolation Mode 47
IIS 6.0 Default Configuration 48
Common IIS 6.0 ISAPI Filter Issues 48
Frequently Asked Questions 49

Chapter 4 · Working with User-Written Allocators 51
Introduction 51
Gathering Necessary Information 51
Finding the Names of User-Written Allocators 51
Special Assumptions Made By User-Written Allocators about Memory 53
Creating Entries in UserAllocators.dat 54
Modules 54
Allocator Records 55
Deallocator Records 59
QuerySize Records 61
Reallocator Records 64
Ignore Records 67
Coding UserAllocator Hook Requests 69
Code Requirements for UserAllocators 70
Allocator Function Hooks 70
Deallocator Function Hooks 71
Reallocator Function Hooks 71
Debugging UserAllocator Hooks 72
NoDisplay 72
Debug 72

How to Diagnose Errors in UserAllocators.dat 73
Token Parsing Errors 73
Semantic Errors 73
If Your Application becomes Unstable after Changing UserAllocators.dat 73

Chapter 5 · Deadlock Analysis 75
Background: Single and Multi-threaded Applications 75
Threads 75
Critical Sections 76
Deadlock - A Basic Definition 76
Techniques for Avoiding Deadlocks 77
Potential Deadlocks 77
The Dining Philosophers 78
Monitoring Synchronization Objects 79
Other Synchronization Objects 79
Additional Information 80
MSDN References 80
Other References 81

Appendix A · Troubleshooting Error Detection 83
Troubleshooting 83

Appendix B · Important Error Detection Files 93
Files and Their Purpose 93

Index 95

Preface

This manual describes concepts and procedures to help you understand in-depth use of Micro Focus® DevPartner Error Detection.

Who Should Read This Manual

This manual is intended for new DevPartner Error Detection users and for users of previous versions of DevPartner Error Detection who want an overview of new functions and interface changes.

New users should read the error detection chapter in Understanding DevPartner Studio to get an overview of DevPartner Error Detection concepts and then use this document to learn how to use DevPartner Error Detection most effectively.

Users of previous versions of DevPartner studio should read the Release Notes to see how DevPartner Error Detection differs from BoundsChecker, the error detection tool included with previous versions.

This manual assumes that you are familiar with the Windows interface and with software development concepts.

What This Manual Covers

This manual contains the following chapters and appendixes:

- **Chapter 1**, ***Workflow and Configuration Settings***, explains how to configure DevPartner Error Detection to solve various problems, ranging from simple API call validation to problems encountered in complex COM applications.
- **Chapter 2**, ***Checking and Analyzing Programs***, describes error detection tasks you can perform with DevPartner Error Detection and other tasks, beyond error detection.
- **Chapter 3**, ***Analyzing Complex Applications***, provides information to help you use DevPartner Error Detection more effectively when checking complex applications.
- **Chapter 4, *Working with User-Written Allocators,*** explains how to customize the `UserAllocators.dat` file so you can analyze your own memory allocators.
- **Chapter 5, *Deadlock Analysis,*** explains deadlocks, potential deadlocks, and synchronization objects. It also lists Web addresses and books that provide more information on these topics.
- **Appendix A, *Troubleshooting Error Detection,*** provides answers to some of the most common issues in a problem/solution format.

- **Appendix B, *Important Error Detection Files*,** provides a list of the important files associated with DevPartner Error Detection, and describes each file's purpose.

You will also find an index at the back of this manual.

Note: This manual contains information for all of the Visual Studio versions of DevPartner Studio. Notes throughout the text identify features that are available only for a specific release of Visual Studio.

Conventions Used In This Manual

This book uses the following conventions to present information.

- Screen commands and menu names appear in **bold typeface**. For example:

  Choose **Item Browser** from the **Tools** menu.

- Computer commands and file names appear in `monospace typeface`. For example:

  The *Understanding* ***DevPartner Error Detection*** manual (**`bc_vc.pdf`**) describes...

- Variables within computer commands and file names (for which you must supply values appropriate for your installation) appear in *`italic monospace type`*. For example:

  Enter **`http://servername/cgi-win/itemview.dll`** in the **Destination** field...

Getting Help

If ever you have any problems or you would like additional technical information or advice, there are several sources. In some countries, product support from Micro Focus may be available only to customers who have maintenance agreements.

If you obtained this product directly from Micro Focus, contact us as described below. If you obtained it from another source, such as an authorized distributor, contact them for help first. If they are unable to help, contact us as described below.

However you contact us, please try to include the information below, if you have it. The more information you can give, the better Product Support can help you. But if you don't know all the answers, or you think some are irrelevant to your problem, please give whatever information you have.

- The name, release (version), and build number of the product.
- Installation information, including installed options, whether the product uses local or network databases, whether it is installed in the default folders, whether it is a standalone or network installation, and whether it is a client or server installation.
- Environment information, such as the operating system and release on which the product is installed, memory, hardware/network specifications, and the names and releases of other applications that were running.
- The location of the problem in the product software, and the actions taken before the problem occurred.

- The exact product error message, if any.
- The exact application, licensing, or operating system error messages, if any.
- Your Micro Focus client, office, or site number, if available.

Contact

Our Web site gives up-to-date details of contact numbers and addresses. The product support pages contain considerable additional information, including the WebSync service, where you can download fixes and documentation updates. To connect, enter www.microfocus.com in your browser to go to the Micro Focus home page.

If you are a Micro Focus Product Support customer, please see your Product Support Handbook for contact information. You can download it from our Web site or order it in printed form from your sales representative. Support from Micro Focus may be available only to customers who have maintenance agreements.

Chapter 1

Workflow and Configuration Settings

DevPartner Error Detection can identify many different types of problems. The default DevPartner Error Detection settings have been chosen to find the most common errors with the minimum impact on performance.

By changing the settings, you can fine-tune DevPartner Error Detection to search for specific types of problems. Understanding the error detection settings will enable you to use DevPartner Error Detection to its fullest.

This chapter describes how to configure DevPartner Error Detection to solve various problems, ranging from simple API call validation to problems encountered in complex COM applications.

Note: Error Detection creates data files for each target application. You must ensure that you have write access to the directory containing the target executable before starting Error Detection.

DevPartner Error Detection Workflow

DevPartner Error Detection follows a program workflow that is more extensive than the workflow of earlier DevPartner Error Detection versions. This mechanism enables you to control the amount of data collected and reported.

Here are the four steps of the DevPartner Error Detection workflow:

1. Configure DevPartner Error Detection to collect the desired data
    a. Select the types of data you want to collect
    b. Define the portions of your application to be monitored
    c. Select the **Suppressions** and **Filters** you want to apply
2. Run your application
    a. As the program runs, review errors presented in the **Program Error Detected** dialog box
    b. Suppress errors that are not valid
    c. View the log and create filters if necessary
    d. Review memory and resource usage
3. View the data (after program termination)

   a Filter out events you do not want to see in the log

   b Create new suppressions to be applied to future runs of your application

4. If desired, save your settings, suppressions, and filters for future use

Benefits of the DevPartner Error Detection Workflow

The DevPartner Error Detection workflow enables you to:

- Select the type and amount of data to be collected
- Select the portions of the application to be monitored
- Suppress errors that report known issues, are handled by conditional code, or have been generated in third-party code
- Create filters to hide extraneous information in the log
- Save different configurations so that settings, suppressions and filters can be reused

DevPartner Error Detection provides defaults for each step in the workflow process. This means you can use DevPartner Error Detection with default settings, or you can change the settings to customize the way that DevPartner Error Detection analyzes your application.

Saving Error Detection Configurations

You can save an error detection configuration - a specific combination of settings (standalone versions) or options (Visual Studio) - to use again.

For example, you might create a configuration for memory and resource leaks, another for COM leaks and a third to do detailed **`lint`** type analysis. You can further refine settings and define configurations that look only at particular sections of a large application.

Using Error Detection from the Command Line

Use the following command syntax to check a program with **`BC.exe`** (the executable) from the command prompt. Brackets [ ] indicate that a command is optional.

```
BC.exe [/?]
```

```
BC.exe sessionlog.DPbcl
```

```
BC.exe [/B sessionlog.DPbcl] [/C configfile.DPbcc] [/M] [/NOLOGO]
[/X[S|D] xmlfile.xml] [/OUT errorfile.txt] [/S] [/W workingdir]
target.exe [target args]
```

Table 1-1. Command Line options

Option	Description
/?	Display usage information
sessionlog.DPbcl	Open an existing session log
/B sessionlog.DPbcl	Run in batch mode and save the session log to a log file `sessionlog.DPbcl`

Table 1-1. Command Line options

Option	Description
/C configfile.DPbcc	Use the **`configfile.DPbcc`** options
/M	Start **`BC.exe`** and minimize when running
/NOLOGO	Do not show the splash screen when loading **`BC.exe`**
/X xmlfile.xml	Generate XML output and save to the specified file. • When you specify an executable, Error Detection runs a session on the executable and then generates XML output from the results. • When you specify a session log file only (**`sessionlog.DPbcl`**), Error Detection converts the specified session log to XML and saves the output. **Note:** When you specify an executable, you must still specify an corresponding session log file using the `/B` switch.
/XS xmlfile.xml	The `/X` flag used with the `S` modifier instructs Error Detection to only save Summary data to the xml file. Information about the running of the Error Detection session (Session data) is always exported.
/XD xmlfile.xml	The `/X` flag used with the `D` modifier instructs Error Detection to only save Details data to the xml file. Information about the running of the Error Detection session (Session data) is always exported.
/OUT errorfile.txt	Output any error messages to a text file named **`errorfile.txt`** — this file will only contain error messages generated while trying to execute Error Detection, not the list of error and leaks discovered by Error Detection
/S	Run in silent mode — do not open the Program Error Detected dialog box on errors
/W workingdirectory	Set the target's working directory
target.exe [target args]	The executable to launch and its arguments

Note: You must specify the full directory path to your program executable if it is not located on the current path (the environment variable listing the directories that the system searches in order to find an executable).

You can specify multiple command options for one program. For example:

```
BC.exe /B test.dpbcl /S /M c:\testdir\test.exe
```

Compiling, Instrumenting, and Building Unmanaged (Native) C++ Projects with NMDEVENV

If you plan to build your unmanaged C++ project from the command line, and want to instrument it for performance or coverage analysis, you can use `NMDEVENV` from the command line. `NMDEVENV` lets you compile, instrument unmanaged code in a project or solution, and build the project or solution with a single command. It combines the code instrumentation and compile capability of the DevPartner `NMCL` command with build capability of the Visual Studio `DEVENV` command. `NMDEVENV` replaces the Visual Studio `DEVENV` command.

`NMDEVENV` uses Visual Studio instrumentation options set for the project or solution being instrumented. The build process uses the DevPartner `NMCL` compiler driver instead of the standard C++ compiler driver.

Use `NMDEVENV` with the `NMCL` option to enable instrumentation, and the `DEVENV /Build` or `/Rebuild` switch. Specify the project or solution, and whether to build with debugging. Note that if `Debug` is used and you have multiple debug configurations, `Debug` must be in quotes and entered with a specific qualifier (for example "`Debug|Win32`". No DEVENV IDE switches or other switches are needed.

Prerequisites

To run `NMDEVENV` you must have:

- DevPartner Studio 9.1.0 or later installed.
- A system environment set up to run the Visual Studio tools.
- `NMCL.EXE` and `DEVENV.EXE` in your path setting. By default `NMCL.EXE` is installed in the following location: **`\Program Files\Common Files\Micro Focus\NMShared`**

To build the debug configuration of a project with Error Detection instrumentation:

```
NMDEVENV /nmbcon sample.vcproj /build debug
```

Instrumenting Native C/C++ Code with nmvcbuild

If you plan to build your project from the command line, and want to instrument it for Error Detection, you need to use **`nmvcbuild.exe`** instead of the Microsoft **`vcbuild.exe`** compiler. The `vcbuild` compiler does not provide any way to replace the default compiler and linker, so there is no way to perform DevPartner native C/C++ instrumentation. **`nmvcbuild.exe`** is a command line utility, designed specifically for DevPartner, that allows it to perform native C/C++ instrumentation for `vcbuild`. It functions as a wrapper for `vcbuild` that watches for **`cl.exe`** and **`link.exe`** being started, and replaces them with **`nmcl.exe`** and **`nmlink.exe`**.

The `nmvcbuild` utility accepts the same command line parameters as `vcbuild` and `nmcl`. You can view the parameters for `vcbuild` and `nmcl` by entering `vcbuild ?` and `nmcl ?` at the command line. You can also embed any required parameters in the environment variable ***`nmcl`***, and then you only need to pass the `vcbuild` parameters when you call `nmvcbuild`. For example, the following entry would set `nmcl` parameters in the environmental variable:

```
set nmcl=/NMignore:StdAfx.cpp
```

For more information, refer to ***nmcl*** Options in the online help.

Prerequisites

To run **nmvcbuild.exe** you must have:

- DevPartner Studio installed on your system.
- A system environment set up to run the Visual Studio tools.
- **vcbuild.exe** and **nmvcbuild.exe** in your path setting. By default **nmvcbuild.exe** is installed in the following location:

```
\Program Files\Common Files\Micro Focus\NMShared
```

Note: For installs on 64-bit versions of Windows, the file is located at:

```
\Program Files (x86)\Common Files\Micro Focus\NMShared
```

Example

To build the debug configuration of the sample project with Error Detection instrumentation:

```
nmvcbuild /nmbcon sample.vcproj debug
```

Customizing the DevPartner Error Detection Settings

The DevPartner Error Detection settings provide the following types of customization:

- Restrict the types of information collected (e.g. memory and resource leaks)
- Further refine the types of information collected in each major category of analysis (for example, look only for resource leaks generated by graphics calls)
- Determine how much additional information such as call stacks, parameter data, return values, etc. is recorded along with the event or error
- Control the look and feel of the DevPartner Error Detection user interface. This includes changing fonts, colors, highlighting, or whether the Program Error Detected dialog box is displayed
- Save and restore DevPartner Error Detection settings created previously

By customizing the DevPartner Error Detection settings, you control how much data is collected and which portions of the application are monitored.

The DevPartner Error Detection settings are divided into these groups:

- General
- Data Collection
- API Call Reporting
- Call Validation
- COM Call Reporting
- COM Object Tracking
- Deadlock Analysis
- Memory Tracking
- .NET Analysis
- .NET Call Reporting
- Resource Tracking
- Modules and Files
- Fonts and Colors
- Configuration File Management

General

Use the check boxes under General settings to control following:

- Event logging

Note: Turning off event logging “silences” Error Detection. Error Detection will not report anything until event logging is turned on again.

- The Program Error Detected dialog box - to display on each error or not.
- Whether or not to display a prompt to save program results when closing Error Detection or starting another session.
- Whether or not to display the Memory Resource Viewer dialog box when the target application exits or not
- The directories to search for source and symbol files
- The working directory (available only when you use DevPartner Error Detection in standalone mode)
- Specify command line arguments (available only when you use DevPartner Error Detection in standalone mode)

Data Collection

Use the Data Collection settings to control the following features.

- The depth of various call stacks
- The amount of data to be stored for non-scalar parameters (for example, structures, classes, and pointers) and return values

If you are working with computers that have limited memory, or if you are analyzing large complex applications, you may want to restrict the size of the **Maximum call stack depth on allocation** to reduce memory requirements.

API Call Reporting

Use API Call Reporting settings to control the type of Windows API calls to be logged if **Enable API call reporting** has been selected. You can also control the logging of Windows messages.

To reduce log file sizes, selectively enable API calls for particular Windows module (for example, select GDI32 to log graphics calls).

Call Validation

Use Call Validation settings to control whether DevPartner Error Detection validates Windows API parameter and return values. By default, DevPartner Error Detection does not validate parameters.

If you are also tracking memory usage, you can select **Enable memory block checking**. When you select this option, DevPartner Error Detection performs more detailed parameter analysis using the knowledge gathered from the memory tracking system. Enabling this feature will detect more errors but will affect performance.

DevPartner Error Detection includes settings that enable you to restrict the types of validations performed on the Windows APIs. These settings enable you to de-select categories of errors that can generate spurious errors. Examples include flag checks, range checks, and enumeration checks. Explore these options if you want the detailed analysis of handles and pointers but are not interested in other types of validation.

DevPartner Error Detection enables you to select which Windows APIs to check. The default is to check all Windows APIs. If you are interested in a limited set of API calls, select only those modules. This can reduce the number of errors detected and improve performance.

COM Call Reporting

Use COM Call Reporting settings to control the COM interfaces that should be logged if **Enable COM method call reporting on objects that are implemented in the selected modules** has been selected.

By default, if **Enable COM method call reporting on objects that are implemented in the selected modules** is selected, DevPartner Error Detection will report on all known COM interfaces. For improved performance, select only the COM interfaces you need to check. Use the tree view that appears under COM Call Reporting. Decreasing the number of COM interfaces checked decreases the size of the log file and improves performance.

You can also select **Report COM method calls on objects implemented outside of the listed modules**.

COM Object Tracking

DevPartner Error Detection can monitor COM usage within an application and will report on any code that is leaking interfaces. If an interface leak is detected, DevPartner Error Detection will provide a COM use-count graph showing every `AddRef` and `Release` within the application. The graphs can be used to quickly spot missing `AddRef` or `Release` calls based on your knowledge of the application.

By default, DevPartner Error Detection does not enable COM object tracking. Select **Enable COM object tracking** to activate this feature. When COM object tracking is active, you can select **All COM classes** or you can select individual classes from the list provided.

Deadlock Analysis

Use Deadlock Analysis to monitor multi-threaded applications for deadlocks. This includes the following types of analysis:

- Monitoring and reporting of deadlocks as they occur in the application
- Monitoring the usage patterns of the synchronization objects within your application for potential deadlocks
- Monitor your application for synchronization object errors

Memory Tracking

Use Memory Tracking settings to control the type of memory leak detection performed on the application. Memory Tracking is enabled by default. If you do not want to perform memory leak detection, clear **Enable memory tracking**.

The Memory Tracking settings have been preset to generate acceptable results for most applications. The **Enable FinalCheck**, **Guard bytes**, **Fill on allocation** and **Poison on free** settings are of special note.

Enable FinalCheck

Selecting **Enable FinalCheck** has no effect unless your application is instrumented with FinalCheck. FinalCheck is on by default when you select **Instrumenting for Error Detection**. To enable instrumentation without having FinalCheck run, you can disable FinalCheck in the **Memory Tracking** pane of the **Error Detection Settings**.

The recommended usage is to leave **Enable FinalCheck** selected and to clear it only when you want to perform a less-detailed ActiveCheck analysis on an application that is already instrumented.

Guard Bytes

Guard bytes are used to detect memory overruns in ActiveCheck analysis. If you encounter heap corruption and DevPartner Error Detection does not detect the problem, consider increasing the **Count** setting to a larger value. Refer to the online documentation for tips on using these settings to track down heap errors that are hard to find.

Fill on allocation and *Poison on free*

Fill on allocation sets memory to a known state when it is allocated. **Poison on free** sets memory to a known state when it is deallocated.

The byte patterns used have been carefully selected to cause an application to generate errors if these byte patterns are accidentally used during program execution. Refer to the online documentation for additional information on these settings.

UserAllocators.dat

If you write your own memory allocation logic or override global `operator new`, see Chapter 4, “Working with User-Written Allocators” and review the documentation (in the form of comments) in the following file:

`C:\Program Files\Micro Focus\DevPartner Studio\BoundsChecker \Data\UserAllocators.dat`

Note: For installs on 64-bit versions of Windows, the file is located at: **`\Program Files (x86)\Micro Focus\DevPartner Studio\BoundsChecker\Data\UserAllocators.dat`**

.NET Call Reporting

Use .NET Call Reporting settings to control .NET assemblies that should be logged if **Enable .NET method call reporting** has been selected.

By combining .NET and COM call reporting, you can see both sides of COM Interop.

The .NET User Assemblies and .NET System Assemblies are displayed on separate branches of a tree view control.

Note: .NET Call Reporting can generate a large amount of data, and cause system slowdowns. Enable .NET Call Reporting only when necessary to debug and understand the framework, and even then select only the assemblies you need to check. Limiting the number of assemblies selected in the **All types** tree view decreases the size of the log file and improves performance.

.NET Analysis

DevPartner Error Detection supports mixed native and managed applications. If you are working in mixed environments, you can select **Enable .NET runtime analysis**. DevPartner Error Detection supports the following types of .NET analysis:

- Monitoring of unhandled exceptions being passed from native to managed code
- Analysis of .NET Finalizers
- Managed to native code interoperability
- Monitoring of garbage collection events

.NET Interoperability

The DevPartner Error Detection .NET Interoperability feature monitors the number of times an application transitions from managed to native code. Use this information to analyze usage patterns and target native code that could benefit from being rewritten in managed code. For best results, use this feature with the **Interop reporting threshold** parameter to specify your own lower limit for acceptable usage.

Resource Tracking

Use Resource Tracking settings to control the type of resource leak detection performed on the application. **Resource Tracking** is selected by default. If you do not want to perform resource leak detection, clear the **Enable resource tracking** check-box.

When resource tracking is selected, you can search for all resource leaks or limit the search to particular resources associated with specific libraries in the Windows API.

The resources have been grouped by library and within each library by the API call used to deallocate the resource. For example, if you have recently written a lot of code to manipulate the registry, you might want to de-select all libraries except **`ADVAPI32`**, then select only **`RegCloseKey`**.

Modules and Files

Use the Modules and Files settings to:

- Identify executables and libraries within your application that should be monitored or ignored
- Refine the list of executables and libraries to be monitored or ignored down to the source file level if symbols are available
- Identify a list of **System directories** that should be ignored by the DevPartner Error Detection analyzers

Use the Modules and Files settings to control the portions of your application that are monitored by DevPartner Error Detection. For example, you might consider using Modules and Files settings when writing large applications or applications such as ISAPI filters.

Note: Disabling all the modules in the Modules and Files settings will not prevent reporting of some error types. Error Detection always reports memory overruns within any module, and other types of events originating from the **`MFCxxxx.dll`** libraries.

For more information, see “Using Modules and Files Settings” on page 36.

Fonts and Colors

Use the Fonts and Colors settings to change the font, color and emphasis of each item in the DevPartner Error Detection user interface.

Configuration File Management

Use Configuration File Management to create multiple settings files for each project. Figure 1-1 on page 21 shows the Configuration File Management options available. You can then use these settings files throughout the software development cycle to perform various types of analysis. Consider these examples of settings files you might create:

- Use Call Validation and Modules and Files to select only your components; use these settings daily as you add new code to your application
- Use settings under Memory Tracking and Resource Tracking as you complete new components, or make non-trivial changes to existing components

- Create a settings file to be used in batch mode over the weekend to analyze the results of major milestones. You might also want to instrument the build with FinalCheck to obtain the most detailed information when you analyze the reports.
- Create a settings file with various sets of modules selected but all analysis features disabled. You can then load this settings file and select the options you want during an interactive session. This may be especially useful when you need to manage complex modules and files settings.

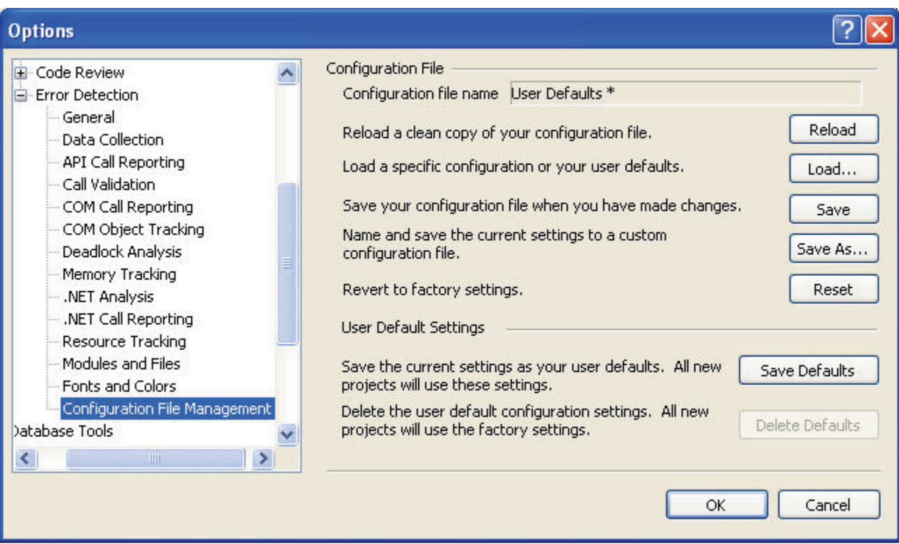


Figure 1-1. Configuration File Management settings

Configuration File Functions

The Configuration File Management page has the following functions available:

- **Configuration file name:** The full path and name of the configuration file.
- **Reload:** Loads the current configuration file again, discarding any changes. This returns you to the last saved version of the current configuration file.
- **Load:** Opens the **Load From** dialog box. Select **Internal User Defaults** to load your user default settings. If you select **Configuration File**, the **Load Configuration File** dialog opens. Use this to select a different configuration file to load.
- **Save:** Saves all active changes in the currently loaded configuration file.
- **Save As:** Opens the Save Configuration File dialog box. Use this to save the current configuration settings under a different file name.
- **Reset:** Resets all the program property settings to the default factory settings.
- **Save Defaults:** Save the current settings as your user defaults. All new projects will use these settings.
- **Delete Defaults:** Delete the user default configuration settings and revert to factory settings. All new projects will use the factory settings.

Chapter 2

Checking and Analyzing Programs

This chapter describes some of the error detection tasks you can perform with DevPartner Error Detection. It also describes other tasks that you can perform with DevPartner Error Detection.

Error Detection Tasks

DevPartner Error Detection typically includes tasks such as:

- Finding memory, resource and interface leaks
- Looking for pointer and memory errors
- Searching for memory corruption
- Analyzing the use of legacy code in .NET applications
- Validating Win32 API calls (32-bit and 64-bit versions)
- Searching for application deadlocks

Finding Leaks

DevPartner Error Detection excels at finding memory, resource and interface leaks. By default, DevPartner Error Detection searches for memory and resource leaks but not interface leaks. To search for interface leaks, select **Enable COM object tracking** in the COM Object Tracking settings.

DevPartner Error Detection provides two methods of detecting memory leaks, ActiveCheck and FinalCheck. ActiveCheck will search for memory leaks in any Windows 32-bit or 64-bit application. Leaks will be reported when your application shuts down. FinalCheck comprehensively reports memory leaks at run-time as they occur in 32-bit and 64-bit applications. Examples include when a local variable goes out of scope or when the last pointer to a block of memory is re-assigned, as well as dangling pointer usage and other hard to find errors.

Finding Pointer and Memory Errors

DevPartner Error Detection can search for pointer and memory errors using both ActiveCheck and FinalCheck technology. In ActiveCheck mode, DevPartner Error Detection will monitor pointers passed to Windows calls for errors. Alter the settings for Call Validation and Memory Tracking to configure the amount of checking done by DevPartner Error Detection.

If you re-compile your program using FinalCheck, DevPartner Error Detection will check every pointer reference in your program for correct usage. FinalCheck provides very detailed analysis of your program and will locate hard-to-find problems such as uninitialized variables, dangling pointers, unrelated pointer comparisons, array index errors, and so on.

Finding Memory Corruption

DevPartner Error Detection helps you find memory corruption problems caused by the following types of problems:

- Overrun allocated buffers
- Continued access to memory after it has been deallocated
- Deallocating a resource multiple times (e.g. double delete)

DevPartner Error Detection can detect many of these errors in ActiveCheck mode but provides the most detailed analysis with FinalCheck.

If you encounter memory overrun errors and you are restricted to using only ActiveCheck, see the online documentation which contains more information about **Check heap blocks at runtime** in the Memory Tracking settings.

Analyzing Transitions to Legacy Code in .NET Applications

DevPartner Error Detection provides the following types of analysis that can help you make the transition from native application to managed application programming:

- Complete analysis of the native portions of Windows applications
- Analysis of the transition layer between native and managed sections of applications that use mixed code
- Analysis of finalizers in managed applications

These types of analysis enable you to monitor:

- Unhandled exceptions being thrown from native applications and passed to managed code
- Garbage collector activity that might cause performance problems
- COM interoperability between managed and native code
- `P/Invoke` calls being made from managed code to native windows libraries
- The frequency of calls across the managed to native boundary

You can use this information to plan and monitor the process of migrating an application.

Migrating from Native to Mixed or Managed Code

The migration process involves the following steps:

1. Analyze COM usage for your native application to determine which objects are being used.
2. Rewrite a section of the application in managed code using `P/Invoke` and `COM` to call native portions of the application.
3. Under .NET Analysis, select **Enable .NET analysis** and **PInvoke interop monitoring** to analyze the transitions between the newly written code and the existing native code.
4. Make any necessary changes.
5. Under **.**NET Analysis, select **COM Interop monitoring** and **PInvoke interop monitoring** to monitor the number of calls made between managed and native code. Use the performance data to help make decisions on these additional changes:
   a. Determine which additional COM objects should be ported to managed code.
   b. Determine if new methods should be added to reduce the number of calls between managed and native code. For example, you might add a method to request data records 10-20 items at a time instead of one at a time.
   c. Determine if calls to native APIs (such as the Windows API) are being made efficiently.

You can also check for unhandled exceptions being thrown across the native-to-managed boundary. To do this, select **Exception monitoring** under **.NET Analysis**. Applications written in native code use exceptions to notify a caller that a call or method failed. As sections of your application are re-written in managed code, monitor the use of exceptions to catch exceptions before they transition to managed code.

Validating Win32 API Calls

DevPartner Error Detection recognizes thousands of Windows calls. This capability allows DevPartner Error Detection to validate pointers, flags, enumerations, handles, and return codes. Select **Enable call validation** to confirm that your applications are using Windows calls properly.

You can configure the following Call Validation features:

- Choose what types of Windows calls to monitor
- Selectively disable various types of validation such as flag, range, and enumeration checking

With these features, you can configure DevPartner Error Detection to validate important parameters such as handles and pointers and to report fewer errors that do not pertain to the task at hand.

Searching for Application Deadlocks

DevPartner Error Detection can identify code that will cause deadlocks in your application. Select **Enable deadlock analysis** to locate deadlocks. Additional controls enable you to fine-tune deadlock analysis.

Expanded Uses for DevPartner Error Detection

Beyond error detection tasks, DevPartner Error Detection can be used as:

- An aid to understand complex applications
- A reverse engineering tool
- A tool for stress testing an application

Understanding Complex Applications

DevPartner Error Detection contains several tools that help you better understand large, complex programs. Consider these three scenarios:

- A new developer joins an existing team and needs to understand how the various DLLs interact.
- A consultant has been brought onto a project to solve a problem (such as crashes, memory leaks, and so on) and needs to understand where to concentrate the most resources given a tight engineering schedule.
- A developer starts using a third-party library and wants to understand why the library is leaking Windows resources. In many cases, the problem is not with the library but in the way that the library is being used.

The following DevPartner Error Detection features can be used to address these scenarios.

COM Object Tracking

Many applications use COM objects that were provided by in-house developers, third-party vendors, or Microsoft. If these COM objects are not used correctly, interface leaks will occur. Interface leaks result in memory and resource leaks — objects allocated from the heap are not released properly, and in turn any memory allocated by those objects is not released properly.

The **COM Object Tracking** enables you to view leaked COM objects. This information can help you determine where the missing `Release` call should be made corresponding to an `AddRef` in your application.

Deadlock Analyzer

Many legacy applications, written before the common use of dual processors, may behave unpredictably when run on more current high-performance computer systems. For example, applications can become deadlocked when they use synchronization objects improperly.

Deadlock analysis under DevPartner Error Detection can identify code that may lead to deadlocks. Note that this analysis can also identify *potential* deadlocks. A potential deadlock is a deadlock waiting to happen when an undesirable set of conditions develop as an application runs. With DevPartner Error Detection, you can identify these potential deadlocks before they occur in a production environment.

Modules and Files

Complex applications are often developed across multiple organizations and include libraries purchased from outside vendors. By default, DevPartner Error Detection will report errors in any non-system DLL. Use the Modules and Files settings to restrict DevPartner Error Detection error reporting and call reporting to specific sections of your application. The result is a more meaningful error report that can be used to solve complex problems.

Note: Disabling all the modules in the Modules and Files settings will not prevent reporting of some error types. Error Detection always reports memory overruns within any module, and other types of events originating from the **`MFCxxxx.dll`** libraries.

The Modules Tab

The DevPartner Error Detection Modules tab (see Figure 2-1 on page 27) and associated details pane provide a view into your program. This view shows what DLLs are being loaded as the program runs. By carefully reviewing this report, you can answer the following questions and make better-informed decisions when you have to make trade-offs:

- Is this module instrumented, and how?
- Is a particular DLL really needed?
- Is it worth calling only one method in a DLL to incur the cost of *n* additional DLLs being loaded into the process?
- Why is my DLL being loaded at the non-preferred load address?
- Why are multiple versions of the same DLL being loaded into memory?

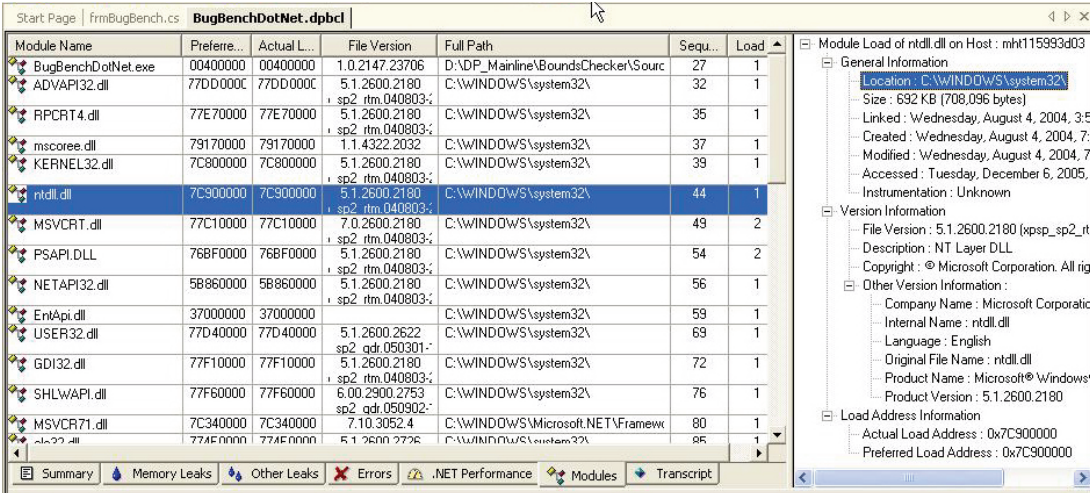


Figure 2-1. Modules tab and Details pane

Viewing and Sorting in the *Results* Pane

DevPartner Error Detection provides a wide variety of ways to view the data collected on your application. Initially, DevPartner Error Detection shows the **Summary** tab, a high-level report, in the **Results** pane. You can review the **Summary** tab and then double-click an entry to view more information.

This capability to navigate through multiple layers of information provides many different views of the data. For example:

- A technical lead might review the data looking for trends such as more or fewer memory leaks over time
- A developer might be interested in correcting memory overrun errors, dangling pointers, and so on.

This multi-level view enables you to identify the most relevant data and access a more detailed view in one of the tabs (**Memory Leaks**, **Other Leaks**, **Errors**, **.NET Performance**, or **Modules**) in the **Results** pane. When viewing data in one of the tabs, you can click column headers to further sort data by size, number of occurrences, location, and so on.

Reverse Engineering

DevPartner Error Detection can be used to analyze Windows applications. By creating a configuration with settings like those described in this section, you can use DevPartner Error Detection to monitor and report on the operations being performed by a Windows application.

Data Collection

Increase the **Call parameter encoding depth** parameter to generate more detailed API parameter information. Increasing the encoding depth will slow processing and increase the size of the log file.

API Call Reporting

Select **Enable API call reporting** to log API call and return values. The amount of detail DevPartner Error Detection gathers on parameters and classes passed as parameters is determined by the **Call parameter encoding depth** value under the **Data Collection** settings.

Select **Collect window messages** to record all window messages sent to the application. Selecting this option provides a view of how the application responds to various window events such as mouse clicks, repaint events, etc.

Note: Selecting either of these options will increase the size of the log file and will slow DevPartner Error Detection performance.

To minimize the overhead of API call reporting, select only system DLLs most relevant to the current task.

COM Call Reporting

Select **Enable COM method call reporting on objects that are implemented in the selected modules** to enable collection of COM method calls.

To keep the COM Call Reporting information manageable, select only the most relevant interfaces and clear the **All components** check box.

.NET Call Reporting

Select **Enable .NET Method Call Reporting** to enable collection of .NET method calls. To keep .NET call reporting manageable, select only .NET user assemblies (default setting).

.NET Analysis

Use **.NET Analysis** features when writing mixed native and managed code applications to:

- Monitor unhandled exceptions being thrown from native code into managed code
- Monitor calls (`P/Invoke` or `COM` method calls) being made from managed code to native code
- Select **Exception monitoring** to monitor exceptions.

To monitor calls from managed code to native code, select either **COM Interop monitoring** or **PInvoke monitoring**, then select an appropriate **Interop reporting threshold** value. When monitoring calls from managed to native code, select a sufficiently high reporting threshold value and use the **Modules and Files** settings to reduce unwanted information.

Function Groups to Turn Off for Reverse Engineering

Tip: Remember to select these features after you have finished your reverse engineering session.

DevPartner Error Detection provides tools to monitor many types of leaks and errors in Windows 32-bit and 64-bit applications. However, during reverse engineering sessions it may be desirable to turn off the DevPartner Error Detection error and leak detection logic. Follow these steps to disable these features in the **Program Settings** dialog box (in DevPartner Error Detection standalone) or the **Options** dialog box (in the Visual Studio IDE):

1. Under **Call Validation**, clear **Enable call validation**.
2. Under **COM Object Tracking**, clear **Enable COM object tracking**.
3. Under **Memory Tracking**, clear **Enable memory tracking**.
4. Under **Resource Tracking**, clear **Enable resource tracking**.
5. Under **Deadlock Analysis**, clear **Enable deadlock analysis**.

These features are intended to identify bugs in the code you are examining. By turning off these features, you can concentrate on information that may help you understand how the code in a component or API works.

Modules and Files

By default, DevPartner Error Detection will report on all portions of your application except those parts listed in the **System Directories** exclusion list.

When doing reverse engineering, you may want to monitor a few DLLs that would normally be excluded. By monitoring a DLL, you can trace into that DLL to see how it operates.

For example, to understand how a particular common control uses WIN32 API calls, you might explicitly include **COMCTL32.DLL** then enable **API Call Reporting**.

To monitor system DLLs explicitly, click **Add module** and add the desired DLLs.

Configuration File Management

You can use **Configuration File Management** to create and save settings designed for special tasks in your development cycle.

For example:

- Memory, Resource and COM leak detection
- Memory and Validation only
- Reverse engineering
- Any of the above, but with restricted sets of DLLs using custom **Modules and Files** settings.

To prevent DevPartner Error Detection from monitoring business-critical portions of your application (such as password checking), you can selectively disable DevPartner Error Detection logging by making calls to the DevPartner Error Detection callable interface at runtime. Please refer to the comments on event reporting in the following file for details:

C:\Program Files\Micro Focus\DevPartner Studio\BoundsChecker \ErptApi\NmApiLib.h

Note: For installs on 64-bit versions of Windows, the file is located at:

\Program Files (x86)\Micro Focus\DevPartner Studio\ BoundsChecker\ErptApi\NmApiLib.h

Stress Testing

A side effect of running DevPartner Error Detection is that it forces an application to deal with many unexpected situations that might only occur under heavy load situations.

Handling Non-zero Uninitialized Data

Many applications are written with the incorrect assumption that local variables and memory returned from dynamic memory allocation routines is initialized to some value. DevPartner Error Detection writes a known fill pattern over various types of memory when it is allocated to search for uninitialized data access. Examples include local variables, and memory allocated by `new`, `malloc`, `HeapAlloc` or `LocalAlloc`.

If your application has been written assuming that uninitialized memory will be zero, your program may crash or behave unpredictably when run under DevPartner Error Detection. If this occurs, instrument your application with FinalCheck and check it again with DevPartner Error Detection to locate the errors.

Note: If you have written your own memory allocation routine that does not follow these rules, add an entry for your routine in the UserAllocators.dat file. See Chapter 4, “Working with User-Written Allocators” for more information.

Pool Poisoning on Free

DevPartner Error Detection writes a known pattern on dynamically allocated memory after it has been deallocated. By doing so, applications that reference deallocated structures generate errors. In many cases, dangling pointer errors can be very difficult to diagnose and repair. Instrument your application with FinalCheck and check it again with DevPartner Error Detection to locate the errors.

Note: If you have written your own memory allocation routine that does not follow these rules, add an entry for your routine in the UserAllocators.dat file. See Chapter 4, “Working with User-Written Allocators” for more information.

Working in a Heavy CPU-Bound Environment

Many developers write applications on extremely fast and lightly-loaded systems. When the application is moved to a production environment, the program fails randomly. Tracking down timing and performance-related issues can be difficult and time-consuming.

DevPartner Error Detection monitors all aspects of program flow and places your application under a heavy CPU and memory workload. At the same time, DevPartner Error Detection can monitor calls to Windows functions for signs of failure; errors are reported in the Program Error Detected dialog box.

Detecting Problems with Multi-threaded Code

Many applications are written to make use of multiprocessor application servers. Unless a multi-threaded application is carefully designed, deadlock and resource deprivation issues can occur when the program is put under stressful conditions.

Running a multi-threaded application under DevPartner Error Detection causes the performance of various threads to deteriorate and may cause the program to display timing-related problems. Many such problems would normally occur in production situations when the program is under stress. By using DevPartner Error Detection, you may be able to find problems in the development process and correct them before going into production.

Run your application under DevPartner Error Detection with Deadlock Analysis enabled to check for deadlock, potential deadlock, and other synchronization bugs.

Detecting Memory and Pointer Reuse Errors

As applications have become more complex, the amount of memory and the number of pointers used in applications has increased dramatically. To deal with this problem, software developers use tools such as DevPartner Error Detection to search for memory and resource leaks. However, finding and plugging leaks is only one part of the task. Once memory has been deallocated, all outstanding pointers to the block should be declared as “dangling.” Attempts to reference dangling pointers should generate an error. The FinalCheck feature in DevPartner Error Detection has been designed to find and report on dangling pointers.

Undetected dangling pointers cause programs to reference blocks that have been deallocated or deallocated and reused by some other part of the system. A program run in a simple debugging environment may not show signs of failure. However, this same program could randomly crash, corrupt data or produce unexpected results when moved into a production environment.

Chapter 3
Analyzing Complex Applications

This chapter provides information to help you use DevPartner Error Detection more effectively when checking complex applications.

About Complex Applications

When you debug typical Windows applications, the default DevPartner Error Detection settings gather enough data to help you solve most common programming problems.

When you debug a complex application, you can benefit by customizing the Error Detection settings.

Complex applications can be divided into two groups:

- Large applications that contain many complex components
- Non-traditional applications such as Windows services, ActiveX components, MTS or COM components, ISAPI filters, and so on

Large Applications

Large Windows application are exceptional only because their size makes them difficult to monitor. Using DevPartner Error Detection, you can analyze a large application in logical, manageable sections, rather than trying to analyze the entire application at once. For example, if you are writing one DLL for a large application, you might:

- Exclude sections of the application from analysis
- Monitor only specific sections of the application
- Monitor only specific transactions within the application

Non-traditional Applications

Non-traditional applications may require different error detection strategies because of complex startup or configuration issues. You can configure DevPartner Error Detection to perform the special debugging or analysis operations required to monitor these types of applications.

DevPartner Error Detection Capabilities and Complex Applications

These Error Detection capabilities can help you analyze complex applications:

- Ability to Wait for Process

- Ability to restrict the modules and files monitored by your application

- Ability to enable or disable the Error Detection log at run-time

Wait for Process

Instead of running your program under Error Detection, you can have Error Detection initialize itself for your application and wait for it to start. You can then start your application manually, or using another means (such as the Service Control Manager). You can use this option to debug services such as IIS.

Note: When using Wait for Process, the full pathname of the application that starts must exactly match the full pathname of the application that Error Detection is looking for.

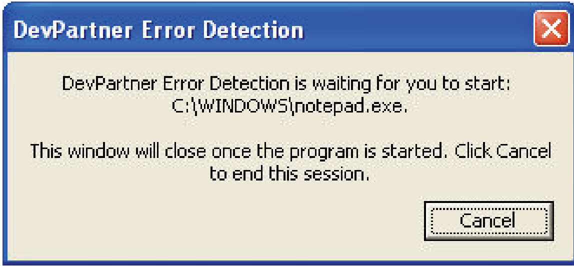
This option replaces the use of Image File Execution Options in previous releases of BoundsChecker and DevPartner Error Detection.

This option is only available when you are using the DevPartner Error Detection standalone application (**`bc.exe`**), and is not available when using Error Detection integrated into Visual Studio.

To debug an application or service using Error Detection in an "Initialize and Wait" manner:

1. Open the image you want to test inside the Error Detection application (**`bc.exe`**).
2. Configure Error Detection to watch for the errors that interest you.
3. Select **Wait for Process** from the **Program** menu.

   Error Detection initializes itself and displays a dialog box allowing you to cancel the session if desired.



4. Start your application as you normally would.

   If you normally start your application via the Service Control Manager, then start it that way. Error Detection closes the dialog when your application starts.

5. Exercise your application, and then cause it to exit.

Analyzing Limited Parts of Your Program

You can point DevPartner Error Detection at a limited problem area within a large or complex application and ignore the rest of the application. DevPartner Error Detection provides four mechanisms to help you analyze limited parts of your program:

- Use **Modules and Files** to exclude sections of your program from analysis.

- Use **Suppressions** and **Filters** to prevent undesirable information from either being logged or displayed.
- Use the **Program > Log Events** menu item or the Log Events toolbar button to toggle Error Detection logging.
- Add conditional code into your application to call `StartEvtReporting` and `StopEvtReporting`.

Note: `StartEvtReporting` and `StopEvtReporting` are DevPartner Error Detection functions that you can call from inside your application to control the writing of data into the DevPartner Error Detection log. If DevPartner Error Detection is not active, these calls return immediately.

Modules and Files

If you are working with large applications, you can use the Modules and Files settings to prevent sections of the application from being analyzed. This can reduce analysis time and decrease the number of unwanted error messages. These are some of the sections you can exclude:

- Unwanted DLLs, including third-party DLLs
- Individual source files from a DLL or EXE
- Entire DLL trees
- Exclude errors if source code is unavailable

Note: Disabling all the modules in the Modules and Files settings does not prevent reporting of some error types. Error Detection always reports memory overruns within any module, and other types of events originating from the **`MFCxxxx.dll`** libraries.

See “Using Modules and Files Settings” on page 36.

Suppression and Filtering

There are two ways to hide the errors and events that DevPartner Error Detection reports.

- **Suppression** prevents a specified type of error or event from being entered into the Error Detection log. To show a suppressed error, you need to remove the suppression instruction and re-run your application under DevPartner Error Detection.
- **Filtering** hides an error or event that has already been entered into the log. You can hide or display filtered errors.

Selective Event Logging

To monitor a small section of a large application, use the **Log Events** menu or tool bar button to turn the Error Detection log on and off. This technique can be especially useful when you select the following settings:

- API or COM call logging
- Call validation.

If you use selective event logging with any of the leak detection features (for example, memory tracking, resource tracking or COM interface tracking) be aware that many leaks are only detected at the end of the program. If logging is off when your program terminates, many of the leaks you are trying to find are reported.

When trying to detect leaks, use **Modules and Files** or **Suppression** to exclude unwanted information.

Conditional Code

You can modify your program to make calls into the DevPartner Error Detection data collection engine to enable or disable Error Detection logging. The following sample code shows how to disable Error Detection logging around unwanted areas:

```
// Requires library [installation folder]
\ERptApi\NMApiLib.lib

// Include file is located in [installation folder]\ErptApi

#include "nmapilib.h"

… [Code that can be monitored]

StopEvtReporting()

… [Code that should not be monitored]

StartEvtReporting()

… [Code that can be monitored]
```

You can also use the `StartEvtReporting` or `StopEvtReporting` API calls to prevent DevPartner Error Detection from logging business-critical sections of an application. Examples might include password validation or encryption routines. If DevPartner Error Detection is not active, the API calls return immediately.

Using Modules and Files Settings

To determine what to exclude from your application, follow these steps:

1. Open your executable in DevPartner Error Detection.
2. Disable all data collection.
   - ◇ In DevPartner Error Detection standalone:

     Select **Program** > **Settings** > **Error Detection**.
   - ◇ In Visual Studio:

     Select **DevPartner** > **Options**.

   In the Options or Settings dialog box, clear the settings for API Call Reporting, Call Validation, COM Call Reporting, COM Object Tracking, Deadlock Analysis, Memory Tracking and Resource Tracking.
3. Run your program in DevPartner Error Detection.

Error detection records all DLLs used by your application. Exercise the program in a way that causes all its DLLs to be loaded, then exit your application.

4 Open the DevPartner Error Detection Settings or Options dialog box and select data collection settings.

5 Select **Modules and Files** in the Settings or Options dialog box. DevPartner Error Detection automatically lists all executables and DLLs used by your application except for files located in the system folders.

6 Review the list of modules and files. Clear any listed DLLs that do not pertain to the task at hand. From this reduced list of DLLs, expand each DLL and select the source files to monitor.

7 To exclude all DLLs in a specific folder, click **System directories** and add the folder to the list of excluded folders. If there is a particular file you want to include from a system folder, click **Add module** to add it to the list of monitored DLLs. Clicking on the folder icon toggles it from a single folder to multiple folders. Table 3-1 explains the icon meanings.

Table 3-1. Meaning of folder icons in the Modules dialog box

Icon	Description
	The selected folder is excluded from testing (unless the specific dll is also listed in the Modules dialog box).
	The selected folder and all sub-folders are excluded from testing.

8 To exclude leaks and errors in portions of your program without source code, select **Show leaks and errors only if source code is available**.

Tip: If you plan to create multiple settings files you can name one of the settings files **Base Configuration**. You can then use the **Base Configuration** settings as a starting point to create other settings files.

9 After you create a logical subset of your application, use **Configuration File Management** to save your settings.

Table 3-2 provides a list of ways to use the Modules and Files: settings.

Table 3-2. Using the Modules and Files settings

When debugging...	Configure Error Detection to exclude...
An ActiveX control	All modules other than the DLL containing your ActiveX control including the ActiveX test container executable
A Windows service	Any modules that are not directly associated with the section of the service you are debugging
An ISAPI filter	All executables and DLLs in IIS or W3WP except your ISAPI filter

Table 3-2. Using the Modules and Files settings

When debugging...	Configure Error Detection to exclude...
A complex application	The sections of the application that do not apply to the problem you are trying to solve
An out of process COM object	Any modules that are not directly associated with your DLL, such as **`dllhost.exe`** or **`mtx.exe`**.

Note: If you exclude everything but your code, you might not see memory or resource leaks that are indirectly caused by your section of the application.

Deciding What to Monitor

When dealing with a complex application it is important to know which sections of an application to monitor. Deciding what to monitor and what to ignore affects your success when tracking down leaks and errors.

To decide what to monitor, consider these questions about your application:

- How does your application start up?
  - Do you start it directly?
  - Do you start it by running another program?
  - Do you launch it from the control panel?
  - Is your application launched indirectly?
- How many modules and files are in your application?
  - Do you own all the modules in your application (other than system modules)?
  - Do you have source for all of your modules?
- Are you interested in the entire application or only a part?
  - Do you care about errors in modules you don't control?
  - Is your application transactional? If so, do you want to watch the entire application or just a few transactions?
  - Does your application make use of resources passed to it from code you do not control?

Once you have answered these questions you can configure DevPartner Error Detection to monitor your application.

As you decide what to monitor, remember that other parts of the program may provide resources to your application. Be aware that if you narrow the focus too much, you may miss resources being passed between your selected analysis subset and the rest of your application.

For example, if you are writing an ActiveX control and running it under the test container, you want to know what happens in your DLL. However, if you call your object incorrectly, resource and interface leaks may occur. If you monitor only your control, errors are found but not errors caused by incorrect usage of your control.

How Does an Application Start Up?

If you are working with a console or Windows application, you can configure Error Detection to monitor your application by selecting **File > Open**. DevPartner Error Detection starts the application and analyzes any DLLs that are directly linked to it.

If you are working with a non-traditional application, it falls into one of two categories:

- It is started directly through a control program
- It is started indirectly based on a system action

The first type of application includes ActiveX controls or DLLs that are invoked by some test application. For example, if you have written an ActiveX control, you can analyze it using the Test Container application, which is available as an executable file with Visual Studio 2005 and 2008, and as a solution in Visual Studio 2010 in the **`Samples/1033`** folder.

If a system action directly invokes your application, use the Error Detection **Wait for Process** option to wait for the application to start (see “Wait for Process” on page 34). Examples of this include:

- Windows services
- Out-of-process COM servers

Many specialized applications, such as services and COM servers, are time critical. If your application is time critical, disable the time out logic when using DevPartner Error Detection for best results.

Analyzing Services

DevPartner Error Detection can monitor Windows services. When monitoring services consider the following:

- Is your service started at boot time or on demand?
- Does your service require a particular security context?
- Can your service be run interactively?
- Can you run your service without being a service?
- Does your service have timing issues?

DevPartner Error Detection can analyze services that can be started after the system is up and running. For best results, you should be able to manually start or stop your service throughout the debugging process.

Requirements and Guidelines

In order to monitor a service, DevPartner Error Detection requires that the account being used to run it have Administrative privileges. You might have additional problems if your application has tight timing requirements.

Analyzing a Service

Follow these steps to analyze your service with DevPartner Error Detection:

1. Stop your service.
2. Build the Debug configuration of your service with symbols and no optimization (optionally with FinalCheck).
3. Open your service's image with Error Detection and update the settings appropriately for this session.
4. Select **Wait for Process** from the **Program** menu.

   Error Detection initializes itself and displays a dialog box allowing you to cancel the session if desired.
5. Start your service as you normally would.

   If you normally start your service via the Service Control Manager, then start it that way. Error Detection closes the dialog when your application starts.

Timing Problems and dwWait

If your service fails to start up, or starts up and almost immediately terminates, you may need to alter the `dwWait` parameter in the **`ServiceStatus`** block passed to **`SetServiceStatus`**. If the value specified in your service is too small, the Windows Service Control Manager terminates the service. When using DevPartner Error Detection, set `dwWait` to a large value such as 4 million.

Note: After you finish using DevPartner Error Detection, restore the normal value for `dwWait`.

Alternate Method: Separating Control Logic from the Worker Thread

If you have written your service in a modular way, you may be able to separate the service control logic from the worker thread. One technique is to wrap a simple console application around the worker thread logic. This way, you can use DevPartner Error Detection to check your service worker thread as if it were a Windows console program.

Custom Code to Turn the DevPartner Error Detection Log On and Off

When dealing with a service that is not interactive, you can write custom code to turn DevPartner Error Detection logging on and off while the service is running. Write custom code to respond to control codes you pass from the `dwControl` parameter to **`ControlService`**.

You can make calls to the start and stop event reporting APIs in your service control logic. See "Conditional Code" on page 36.

Common Service-related Issues

The service starts and immediately hangs.

Make sure you are running your service with Administrative privileges. If you cannot get Administrative privileges, try the alternate method discussed above.

The service starts and almost immediately terminates.

The most likely cause is that the Windows Service Control Manager terminated your service. Increase the value of `dwWait` in your service's initialization logic and re-run your service.

Also, you should verify that DevPartner Error Detection has a valid working folder. Specify the working folder via the General settings, under **Settings** in the **Program** menu.

If the problem persists, consider modifying your service using the alternate method discussed above.

The service runs for a while then terminates unexpectedly.

Your service may be responding too slowly to a control message requesting your service state. Increase the time out value in `dwWait` when responding to service state requests.

Also, DevPartner Error Detection may have poisoned memory in your application, causing the crash. Disable the Memory Tracking feature in the Error Detection settings. If this eliminates the crash, instrument your service with FinalCheck, then re-run your application looking for uninitialized memory references, buffer overruns, and dangling pointers.

If the problem persists, consider modifying your service using the alternate method discussed above.

My service runs correctly, but terminates unexpectedly when it shuts down.

Your service is given a limited time to respond when it receives a shut down request from the Service Control Manager. When an application shuts down, DevPartner Error Detection performs many checks, looking for memory, resource and interface leaks, and re-checking allocated memory blocks for memory overruns. If the `dwWait` value specified for acknowledging the shut down request is too small, the Service Control Manager terminates the service. In this case, increase the `dwWait` value.

If the problem persists, consider modifying your service using the alternate method discussed above.

Analyzing ActiveX Controls Using the Test Container

You can use DevPartner Error Detection with the Test Container utility provided with Visual Studio to monitor ActiveX controls and any other COM object that can be used with the Test Container.

Follow these steps to use DevPartner Error Detection with the Test Container:

1. Run DevPartner Error Detection.
2. Select **File** > **Open** and choose the Test Container.

   If you installed Visual Studio in the standard folder, test container is in one of the following locations:

   ```
   C:\Program Files\Microsoft Visual Studio 8\Common7\Tools\
   TestCon32.exe
   ```

```
C:\Program Files\Microsoft Visual Studio 9\Common7\Tools\
TestCon32.exe
```

3. Select the Modules and Files settings.
4. Make sure **`TestCon32.exe`** is not selected.
5. Click **Add Module**.
6. Add the DLL that contains your ActiveX or COM control into the list of modules and files.
7. Add any additional DLLs required for your control.
8. Run your application.

When the Test Container application starts, follow these steps:

1. Click **New Control** on the toolbar.
2. Add your control from the list provided (for example, Calendar Control 8.0).
3. Use the Invoke Methods and Properties toolbar buttons to manipulate your control.
4. When you have finished exercising your control, exit from the Test Container.

During the run, DevPartner Error Detection reports errors as they are detected. When you exit Test Container, DevPartner Error Detection reports memory, resource and interface leaks that are not reported during the run.

Common Test Container Issues

DevPartner Error Detection reports errors in TestCon32.exe

By default, DevPartner Error Detection reports errors in the executable and all DLLs associated with a process unless the DLLs and EXEs were explicitly excluded using either Modules and Files or System folders. To prevent DevPartner Error Detection from reporting errors on **`TestCon32.exe`**, exclude this executable from the list of modules to check.

DevPartner Error Detection COM Call Reporting not logging calls to an object.

DevPartner Error Detection logs methods only for COM interfaces that it has been instructed to recognize. Tell DevPartner Error Detection about your ActiveX control by selecting **Enable COM method call reporting on objects that are implemented in the selected modules** in the COM Call Reporting settings to activate method logging.

DevPartner Error Detection not reporting COM interface leaks in an object.

To collect COM interface leak information, select **Enable COM object tracking** in the COM Object Tracking settings, then select the COM classes to monitor.

To track your own objects, review the list of COM classes in the COM Object Tracking settings and select only your classes. If you are unsure which classes to select, select **All COM classes**.

Analyzing Applications That Use COM

DevPartner Error Detection can analyze COM components. For DevPartner Error Detection to analyze COM components, you need to edit the COM Component Services settings to establish DevPartner Error Detection as the debugger for the COM component.

Follow these steps to set up DevPartner Error Detection as the debugger for your COM component.

1. Choose **Start** > **Settings** > **Control Panel** > **Administrative Tools** > **Component Services**.
2. Use the tree control in the **Component Services** window to open **COM Applications**.
3. Select your component from the tree control.
4. Right-click your component and choose **Properties**.
5. In the property sheet for your component, click the Advanced tab.
6. In the Advanced tab, select **Launch in debugger**.

Tip: To avoid deleting **`dllhost.exe`**, cut and paste or type the path instead of clicking **Browse**.

7. Change the **Debugger path** to point to **`bc.exe`**. Provide the full path. If you chose the default path when you installed DevPartner Error Detection, this path would be:

   `C:\Program Files\Micro Focus\DevPartner Studio\BoundsChecker\bc.exe`

   For installs on 64-bit versions of Windows, DevPartner Error Detection is located at: **`\Program Files (x86)\Micro Focus\DevPartner Studio\BoundsChecker\bc.exe`**.

Note: Do not delete **`dllhost.exe /ProcessID:{…}`** from the end of the debugger path.

8. Click **OK** to save the changes.

After you establish DevPartner Error Detection as the debugger for your component, follow these steps:

1. Start your component using one of the following methods:
   - ◇ Run an application that uses the component.
   - ◇ Start your application using **Component Services**.
     - – Select your component from the tree control.
     - – Right-click on the component and choose **Start**.
2. When DevPartner Error Detection starts up, select the settings you would like to use, then begin an error detection run.

Note: To see errors and events in your COM component only, remove **`dllhost.exe`** and any other DLLs from the list of modules in the DevPartner Error Detection Modules and Files settings.

3 After you finish exercising your component, shut down your component. Follow these steps:

   a In the **Component Services** window, select your component from the tree control.

   b Right-click the component and choose **Shut down**.

   DevPartner Error Detection performs the normal end-of-process error and leak detection.

4 After you finish debugging, clear the **Launch in debugger** check box:

   a Select the component in the tree view of the **Component Services** window.

   b Right-click the component and choose **Properties**.

   c Click the Advanced tab in the property sheet and clear **Launch in debugger**.

   d Click **OK**.

Common COM Issues

DevPartner Error Detection reports errors in dllhost.exe?

By default, DevPartner Error Detection reports errors in the executable and all DLLs associated with a process unless the DLLs and EXEs were explicitly excluded using either Modules and Files or System folders. To prevent DevPartner Error Detection from reporting errors on **`dllhost.exe`**, exclude this executable from the list of modules to check.

DevPartner Error Detection COM Call Reporting not logging calls to a component?

DevPartner Error Detection logs COM method calls only for interfaces that it can recognize. Select **Enable COM method call reporting (on objects that are implemented in the selected modules)** in the COM Call Reporting settings to activate method logging.

DevPartner Error Detection not reporting COM interface leaks in a component?

DevPartner Error Detection reports COM interface leak information only if you select **Enable COM object tracking** in the COM Object Tracking settings. You must also indicate which COM classes should be monitored.

To track only your interfaces, review the list of COM classes in the COM Object Tracking settings, select your classes and clear all others. If you are unsure what classes should be selected, select **All COM classes**.

DevPartner Error Detection appears to not respond for a long time after exercising a component.

DevPartner Error Detection is waiting for **`dllhost.exe`** to time out and terminate the process. When **`dllhost.exe`** terminates, DevPartner Error Detection performs the final memory, resource and interface leak detection.

To terminate **`dllhost.exe`** before it times out, locate your component in the **Component Services** window, then right-click your component and choose **Shut down**.

Debugging `dllhost.exe` using Wait for Process

Debugging **`dllhost.exe`** using **Wait for Process** is strongly discouraged. Given the number of components being created on Windows operating systems supported by DevPartner Error Detection, it is safer to use the supported mechanisms provided by COM using the component services debugging options.

Failure to use the supported debugging mechanisms could cause strange system failures when other COM components are requested. The components may not start up properly because you have associated all instances of **`dllhost.exe`** with DevPartner Error Detection.

Analyzing ISAPI Filters Under IIS 5.0

You can use DevPartner Error Detection to analyze ISAPI filters within an IIS process. Follow these steps to analyze your ISAPI filter with DevPartner Error Detection:

1. Build your ISAPI filter with the Debug configuration with symbols and no optimization (optionally with FinalCheck).
2. Stop the Internet Information Server (IIS) Service.
3. Configure Error Detection for **`inetinfo.exe`**:
   a. Open **`inetinfo.exe`** in the Error Detection application (**`bc.exe`**).
   You can find **`inetinfo.exe`** in:

   ```
   %WINDIR%System32\Inetsrv\inetinfo.exe
   ```

   b. Open **Modules and Files** under **Options/Settings** and clear all EXEs and DLLs.
   c. Click **Add Module** to add your ISAPI filter to the list of modules.
   d. Update the remaining settings appropriately for your ISAPI filter.
4. Configure the virtual directory that contains the ISAPI extension you want to test to use the High (Isolated) Application Protection mode.
   a. Open the Internet Information Services Manager.
   b. Browse to the virtual directory.
   c. Right-click and choose **Properties**.
   d. Configure **Application Protection** in the Virtual Directory tab to **High (Isolated) Application Protection Mode**.
5. Select **Wait for Process** from the **Program** menu.

   Error Detection initializes itself for IIS and waits for it to start.
6. Start the IIS Admin service from the **Services** control panel.
7. Generate a series of HTTP requests to the IIS server to exercise your ISAPI filter.

8 After you finish exercising your ISAPI filter, use the Service Control Panel to stop the IIS Service.

9 Error Detection then performs end-of-process error and leak detection.

Common ISAPI Filter Issues

Many of the common problems associated with debugging ISAPI filters have already been discussed in the Common issues for Services.

The following issues are specific to IIS and ISAPI filter debugging.

IIS start then stops responding

DevPartner Error Detection requires Administrative privileges to debug a service. If the account used does not have Administrator privileges, IIS either stops responding or terminates with an error.

DevPartner Error Detection log contains unneeded information

Use the Modules and files settings to exclude **`inetinfo.exe`** and all DLLs except your ISAPI filter.

When you run **`inetinfo.exe`** the first time, DevPartner Error Detection automatically adds any DLLs that were dynamically loaded into the process to the list of modules and files. Use the Modules and Files settings dialog to clear any unwanted DLLs. Do not remove them from the list as they are added back to the list and turned on during the next run.

Are there other sources of information about debugging services and ISAPI filters

- There are a number of excellent articles available on MSDN discussing debugging techniques for IIS and ISAPI filters.
- There are a number of knowledge base articles available on our web site.

Tips for debugging IIS interactively with DevPartner Error Detection

- You must be logged into an account with Administrative rights.
- If these suggestions do not solve the your problem, review the Microsoft Technical Note “Debugging ISAPI Applications”

  `http://msdn.microsoft.com/en-us/library/bewb5yw3`

Analyzing ISAPI Filters under IIS 6.0

You can use DevPartner Error Detection to analyze ISAPI filters if you have configured IIS 6.0 in one of the following ways:

- IIS 5.0 Isolation Mode
- IIS 6.0 default configuration

To analyze your ISAPI filter with DevPartner Error Detection, first build your ISAPI filter with Debug and no optimization (optionally with FinalCheck). Then, follow the instructions for the IIS configuration you are using.

IIS 5.0 Isolation Mode

When running IIS 6.0 in the IIS 5.0 Isolation Mode configuration, run DevPartner Error Detection against the **`inetinfo.exe`** executable.

Follow these steps to analyze your ISAPI filter:

1. Configure Error Detection for **`inetinfo.exe`**:
   a. Open **`inetinfo.exe`** in the Error Detection application (**`bc.exe`**).
      You can find **`inetinfo.exe`** in:

      `%WINDIR%System32\Inetsrv\inetinfo.exe`

   b. Open Modules and Files under **Options/Settings** and clear all EXEs and DLLs.
   c. Click **Add Module** to add your ISAPI filter to the list of modules.
   d. Update the remaining settings appropriately for your ISAPI filter.
2. Configure the virtual directory that contains the ISAPI extension you want to test to use the High (Isolated) Application Protection mode.
   a. Open the IIS Admin utility.
   b. Browse to the virtual directory.
   c. Right-click and choose **Properties**.
   d. Configure **Application Protection** on the **Virtual Directory** tab of this dialog box to be **High (Isolated) Application Protection Mode**.
3. Open the **Services** dialog box in the **Control Panel**.
4. Select **Wait for Process** from the **Program** menu.
   Error Detection initializes itself for IIS and waits for it to start.
5. Start the **IIS Admin** and **World Wide Web Publishing** services.

   DevPartner Error Detection starts and monitor the **`inetinfo.exe`** process and your ISAPI filter.
6. Generate a series of HTTP requests to the web server to exercise your ISAPI filter.
7. After you finish exercising your ISAPI filter, use the IIS Manager to stop IIS.

   When the **Shutting Down** dialog box appears, click **End Now**. This stops the **`inetinfo.exe`** process.

Note: If you click the **Stop** button in DevPartner Error Detection, both DevPartner Error Detection and the **`inetinfo.exe`** process terminates, and you lose any data you have collected.

Error Detection performs end-of-process error and leak detection.

IIS 6.0 Default Configuration

When running IIS 6.0 in the default configuration mode, start DevPartner Error Detection against the **`W3WP.exe`** executable.

Follow these steps to analyze your ISAPI filter:

1. Configure Error Detection for **`W3WP.exe`**:
   - **a** Open **`W3WP.exe`** in the Error Detection application (**`bc.exe`**).
     You can find **`W3WP.exe`** in:

     ```
     %WINDIR%System32\Inetsrv\W3WP.exe
     ```

   - **b** Open Modules and Files under **Options/Settings** and clear all EXEs and DLLs.
   - **c** Click **Add Module** to add your ISAPI filter to the list of modules.
   - **d** Update the remaining settings appropriately for your ISAPI filter.
2. Configure the virtual directory that contains the ISAPI extension you want to test to use the `MSSharePointAppPool`.
   - **a** Open the **Internet Information Server (IIS) Manager**.
   - **b** Browse to the virtual directory.
   - **c** Right-click and choose **Properties**.
   - **d** Configure **Application Pool** in the **Virtual Directory** tab to **MSSharePointAppPool**.
3. Select **Wait for Process** from the **Program** menu.

   Error Detection initializes itself for IIS and waits for it to start.
4. Start the **IIS Admin** and **World Wide Web Publishing** services.

   Error Detection starts and monitors the **`W3WP.exe`** process and your ISAPI filter.
5. Generate a series of HTTP requests to the web server to exercise your ISAPI filter.
6. After you finish exercising your ISAPI filter, use the IIS Manager to stop IIS.

   When the **Shutting Down** dialog box appears, click **End Now**. This stops the **`W3WP.exe`** process.

Note: If you click the **Stop** button in DevPartner Error Detection, both Error Detection and the **`W3WP.exe`** process terminates, and you lose any data collected.

Error Detection performs end-of-process error and leak detection.

Common IIS 6.0 ISAPI Filter Issues

All of the items in the IIS 5.0 Common ISAPI Filter issues section apply. In a few cases you may need to substitute **`W3WP.exe`** for **`inetinfo.exe`**.

See "Common ISAPI Filter Issues" on page 46.

The following new issues apply to IIS 6.0:

- Microsoft has re-designed the IIS 6.0 default configuration to be more secure. One of these changes disables ISAPI Extensions by default. To debug an ISAPI extension, go to the Web Services Extensions tab in the IIS Admin tool and modify IIS to allow Unknown ISAPI extensions.
- Use the IIS Manager tool to Start, Stop or Restart IIS. To perform these operations, right-click the **<*MachineName*>** node in the tree and select **All Tasks->Restart IIS**. This opens a dialog box with controls that enable you to start and stop IIS.
- For best results, turn off logging functions, such as API Call Logging, before you monitor IIS. With logging functions on, DevPartner Error Detection creates extremely large log (.DPBcl) files and impacts the performance of the IIS server.

Note: Do not turn off **Log events** on the general dialog. Error Detection does not report anything as long as **Log events** is disabled. Use this feature only when you want to suppress all reporting until you specifically enable event logging via the menu bar button.

Frequently Asked Questions

What is the difference between DevPartner Error Detection ActiveCheck and FinalCheck or technologies?

DevPartner Error Detection has two modes of operation:

- **ActiveCheck -** In this mode DevPartner Error Detection operates on any 32-bit and 64-bit Windows program and intercepts all calls to the operating system and C run-time library looking for memory leaks, resource leaks, and usage of pointers that are passed to functions that aren't valid (or have been de-allocated).
- **FinalCheck -** In this mode you must re-compile your C or C++ program using DevPartner Error Detection FinalCheck instrumentation logic. To build using FinalCheck:, click DevPartner > **Native C/C++ Instrumentation Manager**

  With FinalCheck instrumentation, DevPartner Error Detection can watch every pointer fetch, store, or indirect occurring within the modules you instrument. DevPartner Error Detection can also watch variables go in and out of scope.

Note: Instrumenting code without a well defined order of evaluation can cause unexpected results, including erroneous data, hangs, and even crashes.

"Basically, in C and C++, if you read a variable twice in an expression where you also write it, the result is undefined." -Bjarne Stroustrup

The C/C++ standard explicitly does *not* define the order of evaluation when there are "side effects", such as storing a value in an object. For example, the following statement does not have a well defined order of evaluation: `i = ++i + 2;`

There are two points in this statement when values are stored in the variable *i*, and the language does not define what order they occur in. Instrumenting code like this may change the order of evaluation and change the result.

Note that when you run in FinalCheck mode, all ActiveCheck analysis is still performed along with the extended FinalCheck analysis.

FinalCheck specializes in finding dangling pointers, double deallocations, pointer overruns, uninitialized memory errors, read / write to unallocated memory.

When should I enable Call Validation?

Enabling Call Validation causes DevPartner Error Detection to find many more memory and pointer errors in your program. This feature is off by default because the volume of detected events can be large.

What is DevPartner Error Detection doing when I select the Enable memory block checking feature under Call Validation?

When you select **Enable memory block checking**, (which is turned off by default), DevPartner Error Detection performs a more detailed ActiveCheck analysis. Note that this feature can cause DevPartner Error Detection to run as much as 20% slower.

How do I use the DevPartner Error Detection Guard byte settings under Memory Tracking?

To alter the Guard bytes settings in the Memory Tracking configuration, first make sure that **Enable guard bytes** is checked.

Increase **Count** from 4 to a larger value of 8 or 16.

Increasing the number of guard bytes increases the spacing between heap blocks and provides an area between your blocks for DevPartner Error Detection to monitor for overruns.

Change the settings of **Check heap blocks at runtime** to either **Use adaptive analysis** or **On all memory API calls**.

This option tells DevPartner Error Detection to validate each heap block whenever you make a call into a memory function. This makes your program run slower but isolates heap corruption to a localized area of your program, making it easier to track down.

Chapter 4

Working with User-Written Allocators

This chapter provides information to help you implement user-written memory allocators.

Introduction

DevPartner Error Detection can perform memory analysis on user-written memory allocators. To do this, add descriptions of your memory allocators to a text file called **`UserAllocators.dat`**, which is installed in the data folder of the DevPartner Error Detection installation folder. After you add user-written allocators to this file, DevPartner Error Detection treats them like memory allocation routines provided with the operating system. If DevPartner Error Detection detects a leak caused by a user-written allocator described in **`UserAllocators.dat`**, the user-written allocator is shown in the Program Error Detected dialog box instead of the lower-level allocator within the user-written allocator.

Gathering Necessary Information

Before adding a user-written allocator to **`UserAllocators.dat`**, you need to gather the following information:

1. The exact names of the allocation, deallocation, reallocation and size functions of your user-written allocator.
2. Find out if your user-written memory allocator is statically linked to the application or is provided in a separate module (DLL).
3. The name of the module (DLL) that contains your user-written allocator.
4. Examine the parameters to your routines to determine how the size of the block and pointer associated with the memory block is either passed or returned to the caller.
5. Any special assumptions made in your allocator such as zeroing memory on allocation or a user-written allocator that stores data in deallocated blocks.

Finding the Names of User-Written Allocators

To add records to **`UserAllocators.dat`** you need to provide the exact name of your allocate, deallocate, reallocate and size functions.

Follow these steps to locate the name of the routine:

1. Determine the name of the following functions:
   - ◇ Allocation function (such as `malloc`, `calloc`, or `new`)
   - ◇ Deallocation function (such as `free` or `delete`)

- ◇ Reallocation function (such as `realloc` or `recalloc`)
- ◇ Memory block size function (such as `_msize`)

2 There are two basic cases to consider:

- ◇ You have symbols (pdb files) for the module that contains the function that you are defining.

  If a pdb file is available, internal symbols can be used. To find the mangled or expanded function name, use the linker/debugger options to create a map file when compiling the module. Then look under the **Publics by Value** section of the map file to determine the name of the function. This method always requires the `Static` keyword in the userAllocator function definition.

- ◇ You do not have symbols, or the symbols are invalid.

  If no pdb file is available, enter `dumpbin /exports yourlibrary.dll` at the Visual Studio command prompt. Use the function names as they appear in the output.

The name of your allocation functions may be either unmangled or mangled depending on the calling convention used and your choice of languages. If you are using C++, names are often mangled. Consider the following small C++ program:

```
#include <malloc.h>
#include <memory.h>

class SampleClass
{
public:
SampleClass(){}
void *operator new( size_t stAllocateBlock );
void operator delete( void * pBlock);
};
void *SampleClass::operator new( size_t stAllocateBlock )
{
void *pvTemp = malloc( stAllocateBlock );
if( pvTemp != 0 )
memset( pvTemp, 0, stAllocateBlock );
return pvTemp;
}
void SampleClass::operator delete(void * pBlock)
{
free(pBlock);
return;
}
int main(int argc, char * argv[])
{
SampleClass *pClass = new SampleClass;
return 0;
}
```

Before building the application, select Generate mapfile under the Visual Studio Project settings. After you build your application, open the map file and search for the operator new and operator delete methods. They look something like the following:

```
global operator new:??2SampleClass@@SAPAXI@Z
```

```
global operator delete:??3SampleClass@@SAXPAX@Z
```

These operators could be described to **`UserAllocators.dat`** as follows:

```
Allocator
    Module=myModule
    Function=??2SampleClass@@SAPAXI@Z
    MemoryType=MEM_CUSTOM1
    NumParams=1
    Size=1
    NoFill Static Debug
Deallocator
    Module=myModule
    Function=??3SampleClass@@SAXPAX@Z
    MemoryType=MEM_CUSTOM1
    NumParams=1
    Address=1
    Static Debug
```

Special Assumptions Made By User-Written Allocators about Memory

Normally, DevPartner Error Detection fills allocated memory with a fill pattern before returning the pointer to the user program and poisons the block of memory after it has been deallocated.

If your memory allocator initializes the block with special data, use the `NOFILL` flag so you do not overwrite the block and lose the data.

If your memory allocator assumes that it can read from the block after it has been deallocated, use the `NOPOISON` flag. There are various reasons why you might not want “poisoning” to occur:

- Your memory allocator stores data in the deallocated block to track the allocation status, link it to other deallocated blocks, etc.
- You application assumes that deallocated blocks can continue to be referenced until the block has been reallocated. Continuing to reference deallocated blocks is a dangerous practice, but many applications that do this are still in use.

Creating Entries in UserAllocators.dat

After you collect the information described in "Gathering Necessary Information" on page 51, you can describe your allocator in **`UserAllocators.dat`**.

To describe a function, create one record in the file per function. Each record follows this form:

```
Record_Type Parameter_1 Parameter_2 Etc...
```

Record_Type describes the type of function you are creating. The parameters that follow provide additional information about your allocator function.

When you create a record, separate each field with one or more space or tab characters. Records may span multiple lines.

Table 4-1 shows the Record Types that are currently defined.

Table 4-1. Record types

Record Type	Description
Allocator	Functions that allocate memory.
Deallocator	Functions that deallocate memory.
QuerySize	Functions that can query the size of a given memory block that was previously allocated by the Allocator function.
Reallocator	Functions that can adjust the size of a memory block that was previously allocated using the Allocator function. Note that the Reallocator function may or may not return the same block of memory.
Ignore	Allocator or deallocator functions that you want DevPartner Error Detection to ignore (not track memory).

Modules

Each of the UserAllocator record types requires you to specify the module that contains the function being described. There are three different types of modules that can be described.

Table 4-2. Module types

Module Type	Description
Named module	An explicitly named module (DLL) that contains the user allocation function or method (for example, **`foo.dll`**). **Note:** Module names do not support the use of wildcards.

Table 4-2. Module types (Continued)

Module Type	Description
Statically linked user allocator	An explicitly named module (DLL or executable) that contains the user allocation function or method. However, in this case, the function or method was originally part of a library (.lib file). Once linked into the module, the customer code can reference the functions or methods but they are not externally visible. You must provide debug symbols for the module and use the optional `STATIC` keyword to alert DevPartner Error Detection to look for the function or method in the debug symbols. **Note:** Failure to include the `STATIC` keyword in the optional parameters for the record prevents DevPartner Error Detection from properly monitoring the user-written allocation function or method.
*CRT	This is a special case that enables you to reference a function wherever it appears in your application. **Note:** The * in *CRT is not a wildcard character. *CRT covers the following 3 cases: • The Microsoft C Runtime Library • The statically linked C runtime library • Any user function that has the same mangled name as anything that you are patching (for example, `global operator new`.)

Allocator Records

Create an allocator record to describe a function that allocates memory. Follow this format:

```
Allocator Module=module_name Function=func_name MemoryType=mem_type
NumParams=param_num Size=size_value [Count=count_num]
[BufferLoc=buffer_loc] [Optional Parameters]
```

Note: By default, the address of the allocated block is the returned value of the specified function. Override this behavior by specifying the `BufferLoc` parameter to indicate that the address of the allocated block is returned in a parameter. (see `MAPIAllocateBuffer` in the **`UserAllocators.dat`** file).

DevPartner Error Detection assumes that the allocation failed if the location where it expects to find the address of the allocated block is `NULL` after the call (either the return value or the value in the parameter specified by `BufferLoc`).

Table 4-3. Allocator record parameters

Parameter	Description
Allocator	The first parameter in the record must be the word `Allocator` to indicate that you are describing an allocation routine.
module_name	The name of the module (Executable or DLL) that contains the user-written allocator. **Note:** Module names do not support the use of wildcards.
func_name	The name of the function that allocates blocks of memory in your user-written allocator. If you are using C++, this should be the 'mangled' name of the function. This parameter is case-sensitive.
mem_type	Use this parameter to describe what type of memory is being allocated. DevPartner Error Detection currently defines the following memory types: • `MEM_MALLOC` Blocks of memory returned from routines such as malloc, calloc, strdup, and so on. Memory of this type is freed using a routine similar to the C runtime library free routine. • `MEM_NEW` Blocks of memory that are returned from operator new and are freed by operator delete. • `MEM_CUSTOM1` through `MEM_CUSTOM9` Blocks of memory that must be paired with a particular deallocator. These types allow developers to declare their own custom memory allocators that do not interact with the standard memory allocators described above. DevPartner Error Detection verifies that a block of memory allocated with a given memory type is freed by a function of the same type. If the type does not match, DevPartner Error Detection displays a memory conflict error at runtime.
param_num	The number of parameters passed to your function. This value must be between 1 and 32. Provide the proper number of parameters when describing a user-written allocator function. Failure to provide the correct value can cause unpredictable results.
size_value	The number of the parameter that contains the size of the block to be allocated. The number of the first parameter is 1.

Table 4-3. Allocator record parameters

Parameter	Description
count_num	This optional parameter describes calloc-like functions that accept size and count parameters. If specified, the count indicates how many blocks of the specified size should be allocated. If you omit this parameter, DevPartner Error Detection assumes that the count is always 1.
buffer_loc	The number of the parameter containing the address that hold the address of the allocated block. The number of the first parameter is 1.
[Optional Parameters]	You can include the following optional parameters at the end of the record: • `DEBUG` If this parameter is specified, DevPartner Error Detection displays additional information about the hook. The output window (or dbgview) contains information on any errors that occurred when attempting to place the hook. The transcript pane displays statistics about each hook, including whether the function was successfully hooked and the number of times the hooked function was called. • `NODISPLAY` If this parameter is specified, DevPartner Error Detection stops displaying detailed information on each hook requested at the top of the transcript pane. • `NOFILL` If this parameter is specified, DevPartner Error Detection does not fill the buffer returned with the DevPartner Error Detection 'fill' character. **Note:** If your user-written allocator initializes the block with data, like calloc does, specify `NOFILL` to avoid corrupting your data. • `NOGUARD` If this parameter is specified, DevPartner Error Detection does not add any guard bytes at the end of the blocks created by this allocation function. • `STATIC` DevPartner Error Detection statically patches the user-written allocator. Specify the static option if your user-written allocator is linked into your application and is not provided in a separate DLL with an exported interface.

Sample Allocator Records

The following examples show hypothetical allocator record functions.

Example 1

In this example, the function `mallocXX` is located in a library called **`MyAlloc.dll`**. The function is assumed to be a malloc-type operator with one parameter with the size being passed in the first parameter. DevPartner Error Detection should not fill the memory block before returning it to the application program. Any `MEM_MALLOC` type function can free blocks allocated by this function.

```
Allocator Module=MyAlloc.dll Funtion=mallocXX
MemoryType=MEM_MALLOC NumParams=1 Size=1 NOFILL
```

Example 2

This example comes from the file used to track a custom global operator new in the Microsoft `iostream` code. Note that this function is located in the C runtime library. The record specifies *CRT as the module name so DevPartner Error Detection assumes that the function is located in one of the Microsoft C or C++ runtime libraries.

This function takes four parameters with the size being stored in the first parameter. DevPartner Error Detection is allowed to fill the block before returning to the program requesting the memory.

```
Allocator Module=*CRT Function=??2@YAPAXIHPBDH@Z
MemoryType=MEM_NEW NumParams=4 Size=1
```

Example 3

This example describes a function called `CustomAllocXX` that takes one parameter with the size being passed in the first parameter.

DevPartner Error Detection should not fill the buffer before returning it to the application program. Note that this record specifies `MEM_CUSTOM1` as the MemoryType. DevPartner Error Detection verifies that the memory allocated with this function is deallocated by a routine that is also of type `MEM_CUSTOM1`. Using other deallocation routines generates an allocation conflict message after freeing the memory.

```
Allocator Module=foo.dll Function=CustomAllocXX
MemoryType=MEM_CUSTOM1 NumParams=1 Size=1 NOFILL
```

Example 4

In this example, a function called `MyAlloc` has been built as a .LIB file and is statically linked to a data collection component called **`DataStore.dll`**. `MyAlloc` accepts four parameters. The first is the size of the data record; the second is the number of records to be allocated in a single block. The third parameter contains the address of the location holding the address of the allocated block. The memory retrieved from the application has been pre-initialized so DevPartner Error Detection should not fill the block.

```
Allocator Module=DataStore.dll Function=MyAlloc BufferLoc=3
MemoryType=MEM_MALLOC NumParams=4 Size=1 Count=2
NOFILL STATIC
```

Note: The `STATIC` keyword must be specified if the function name is not exported from the DLL.

Deallocator Records

Create a Deallocator record to describe a function that deallocates memory. Follow this format:

```
Deallocator Module=module_name Function=func_name MemoryType=mem_type
NumParams=param_num Address=address_value [Optional Parameters]
```

Table 4-4. Deallocator record parameters

Parameter	Description
Deallocator	The first parameter in the record must be the word Deallocator to indicate that you are describing a deallocation routine.
module_name	The name of the module (Executable or DLL) that contains the user-written allocator. **Note:** Module names do not support the use of wildcards.
func_name	The name of the function that deallocates blocks of memory in your user-written allocator. If you are using C++, this should be the 'mangled' name of the function. This parameter is case-sensitive.
mem_type	This parameter provides a mechanism to describe what type of memory is being deallocated. DevPartner Error Detection currently defines the following memory types: • `MEM_MALLOC` Describes blocks of memory returned from routines such as malloc, calloc, strdup, and so on. Memory of this type is freed using a routine similar to the C runtime library free routine. • `MEM_NEW` Describes blocks of memory that are returns from operator new and are freed by operator delete. • `MEM_CUSTOM1` through `MEM_CUSTOM9` Describes blocks of memory that must be paired with a particular deallocator. These types allow developers to declare their own custom memory allocators that do not interact with the standard memory allocators described above. DevPartner Error Detection verifies that a block of memory allocated with a given memory type is freed by a function of the same type. If the type doesn't match, DevPartner Error Detection displays a memory conflict error at runtime.

Table 4-4. Deallocator record parameters

Parameter	Description
param_num	The number of parameters passed to your function. This value must be between 1 and 32. Provide the proper number of parameters when describing a user-written allocator function. Failure to provide the correct value can cause unpredictable results.
address_value	The number of the parameter that contains the pointer to the block being deallocated. The number of the first parameter is 1.
[Optional Parameters]	The following optional parameters may be included at the end of the record: • `DEBUG` If this parameter is specified, DevPartner Error Detection displays additional information about the hook. The output window (or dbgview) contains information on any errors that occurred when attempting to place the hook. The transcript pane displays statistics about each hook, including whether the function was successfully hooked and the number of times the hooked function was called. • `NODISPLAY` If this parameter is specified, DevPartner Error Detection stops displaying detailed information on each hook requested at the top of the transcript pane. • `NOPOSION` When you specify `NOPOISON`, DevPartner Error Detection does not 'poison' the block of memory after it has been deallocated. If your user-written allocator stores data in the block after it has been deallocated or your application continues to use data in the block after it has been deallocated, specify `NOPOSION` to avoid corrupting your data. • `STATIC` DevPartner Error Detection statically patches the user-written allocator. Specify the static option if your user-written allocator is linked into your application and is not provided in a separate DLL with an exported interface.

DevPartner Error Detection does not check the return value from a deallocation function.

Sample Deallocator Records

The following examples show hypothetical deallocator records.

Example 1

In this example, a function called `freeXX` is located in a library called **`MyAlloc.dll`**. The function takes one parameter with the pointer to the block to be deallocated being passed in the first parameter. DevPartner Error Detection should not poison the memory before returning to the application program.

```
Deallocator Module=MyAlloc.dll Function=freeXX
MemoryType=MEM_MALLOC NumParams=1 Address=1 NOPOISON
```

Example 2

This example describes a function called `MyFree` in **`foo.dll`**. The function takes one parameter with the pointer to the block to be deallocated being passed in the first parameter. DevPartner Error Detection should poison the memory before returning to the application program. When the block is deallocated, DevPartner Error Detection verifies that the block was allocated by an allocator of type `MEM_CUSTOM1`. If the block was not from this group, an error would be generated.

```
Deallocator Module=foo.dll Funtion=MyFree
MemoryType=MEM_CUSTOM1 NumParams=1 Address=1
```

Example 3

In this example, a function called `MyFree` has been built as a .LIB file and is statically linked to a data collection component called **`DataStore.dll`**. MyFree accepts three parameters. The first and last parameters are of no interest to DevPartner Error Detection. The second parameter contains the address to be deallocated. Also, the private deallocation routine maintains private information in the deallocated block after the block has been freed.

```
Deallocator Module=DataStore.dll Function=MyFree
MemoryType=MEM_MALLOC NumParams=3 Address=2 NOPOISON
STATIC
```

Note: Specify `STATIC` if the function name is not exported from the DLL.

QuerySize Records

Create a QuerySize record to describe a function that returns the size of an allocated block of memory. Follow this format:

```
QuerySize Module=module_name Function=func_name MemoryType=mem_type
NumParams=param_num Address=address_value [Optional Parameters]
```

Note: if you omit a QuerySize record for a user-defined allocator, DevPartner Error Detection returns an incorrect block size for that function.

Table 4-5. QuerySize records

Parameter	Description
QuerySize	The first parameter in the record must be the word QuerySize to indicate that you are describing a size routine.

Table 4-5. QuerySize records

Parameter	Description
module_name	The name of the module (Executable or DLL) that contains the user-written allocator. **Note:** Module names do not support the use of wildcards.
func_name	The name of the function that returns the size of allocated blocks of memory in your user-written allocator. If you are using C++, this should be the 'mangled' name of the function. This parameter is case-sensitive.
mem_type	This parameter provides a mechanism to describe what type of memory is being queried. DevPartner Error Detection currently defines the following memory types: • `MEM_MALLOC` Describes blocks of memory returned from routines such as malloc, calloc, strdup, and so on. Memory of this type is freed using a routine similar to the C runtime library free routine. • `MEM_NEW` Describes blocks of memory that are returns from operator new and are freed by operator delete. • `MEM_CUSTOM1` through `MEM_CUSTOM9` Describes blocks of memory that must be paired with a particular deallocator. These types allow developers to declare their own custom memory allocators that do not interact with the standard memory allocators described above. DevPartner Error Detection verifies that a block of memory allocated with a given memory type is freed by a function of the same type. If the type doesn't match, DevPartner Error Detection displays a memory conflict error at runtime.
param_num	The number of parameters passed to your function. This value must be between 1 and 32. Provide the proper number of parameters when describing a user-written allocator function. Failure to provide the correct value can cause unpredictable results.
address_value	The number of the parameter that contains the pointer to the block being queried. The number of the first parameter is 1.

Table 4-5. QuerySize records

Parameter	Description
[*Optional Parameters*]	The following optional parameter may be included at the end of the record: • `DEBUG` If this parameter is specified, DevPartner Error Detection displays additional information about the hook. The output window (or dbgview) contains information on any errors that occurred when attempting to place the hook. The transcript pane displays statistics about each hook, including whether the function was successfully hooked and the number of times the hooked function was called. • `NODISPLAY` If this parameter is specified, DevPartner Error Detection stops displaying detailed information on each hook requested at the top of the transcript pane. • `STATIC` DevPartner Error Detection statically patches the user-written allocator. Specify the static option if your user-written allocator is linked into your application and is not provided in a separate DLL with an exported interface.

The return value from this function is assumed to be a size_t that provides the size of the block.

Sample QuerySize Records

The following examples show hypothetical QuerySize records.

Example 1

In this example, a function called `MySize` is located in a library called **`foo.dll`**. The function takes one parameter with the pointer to the block being queried in the first parameter

```
QuerySize Module=foo.dll Function=MySize
MemoryType=Mem_Custom1 NumParams=1
Address=1
```

Example 2

In this example, a function `MySize` has been statically linked into a data collection component called **`DataStore.dll`**. The `MySize` function accepts two parameters and the address being queried passes to the first parameter.

```
QuerySize Module=DataStore.dll Function=MySize
MemoryType=MEM_NEW NumParams=2
Address=1 STATIC
```

Note: Specify `STATIC` if the function name is not exported from the DLL.

Reallocator Records

Create a Reallocator record to describe a function that reallocates memory. Follow this format:

Reallocator Module=*module_name* Function=*func_name* MemoryType=*mem_type* NumParams=*param_num* Address=*address_value* Size=*size_value* [Count=*count_num*] [BufferLoc=*buffer_loc*] [*Optional Parameters*]

Note: By default, the address of the allocated block is the returned value of the specified function. Override this behavior by specifying the `BufferLoc` parameter to indicate that the address of the allocated block is returned in a parameter. (see `MAPIAllocateBuffer` in the **`UserAllocators.dat`** file).

Table 4-6. Reallocator record parameters

Parameter	Description
Reallocator	The first parameter in the record must be the word Reallocator to indicate that you are describing a Reallocation routine.
module_name	The name of the module (Executable or DLL) that contains the user-written allocator. **Note:** Module names do not support the use of wildcards.
func_name	The name of the function that reallocates blocks of memory in your user-written allocator. If you are using C++, this should be the 'mangled' name of the function. This parameter is case-sensitive.
mem_type	This parameter provides a mechanism to describe what type of memory is being reallocated. DevPartner Error Detection currently defines the following memory types: • `MEM_MALLOC` Describes blocks of memory returned from routines such as `malloc`, `calloc`, `strdup`, and so on. Memory of this type is freed using a routine similar to the C runtime library free routine. • `MEM_NEW` Describes blocks of memory that are returns from operator new and are freed by operator delete. • `MEM_CUSTOM1` through `MEM_CUSTOM9` Describes blocks of memory that must be paired with a particular deallocator. These types allow developers to declare their own custom memory allocators that do not interact with the standard memory allocators described above. DevPartner Error Detection verifies that a block of memory allocated with a given memory type is freed by a function of the same type. If the type doesn't match, DevPartner Error Detection displays a memory conflict error at runtime.

Table 4-6. Reallocator record parameters

Parameter	Description
param_num	The number of parameters passed to your function. This value must be between 1 and 32. Provide the proper number of parameters when describing a user-written allocator function. Failure to provide the correct value can cause unpredictable results.
address_value	The number of the parameter that contains the pointer to the block being reallocated. The number of the first parameter is 1.
size_value	The number of the parameter that contains the size of the block to be reallocated. The number of the first parameter is 1.
count_num	This optional parameter describes `calloc`-like functions that accept size and count parameters. If specified, the count indicates how many blocks of the specified size should be allocated. If you omit this parameter, DevPartner Error Detection assumes that the count is always 1.
buffer_loc	The number of the parameter containing the address that holds the address of the reallocated block. The number of the first parameter is 1.

Table 4-6. Reallocator record parameters

Parameter	Description
[*Optional Parameters*]	The following optional parameters may be included at the end of the record: • `DEBUG` If this parameter is specified, DevPartner Error Detection displays additional information about the hook. The output window (or dbgview) contains information on any errors that occurred when attempting to place the hook. The transcript pane displays statistics about each hook, including whether the function was successfully hooked and the number of times the hooked function was called. • `NODISPLAY` If this parameter is specified, DevPartner Error Detection stops displaying detailed information on each hook requested at the top of the transcript pane. • `NOFILL` If this parameter is specified DevPartner Error Detection does not fill any additional bytes added to the end of the previous allocation with the DevPartner Error Detection 'fill' character. **Note:** If your user-written allocator initializes the block with data, like calloc does, specify `NOFILL` to avoid corrupting your data. • `NOGUARD` If this parameter is specified, DevPartner Error Detection does not add any guard bytes at the end of the blocks created by this allocation function. • `STATIC` DevPartner Error Detection statically patches the user-written allocator. Specify the static option if your user-written allocator is linked into your application and is not provided in a separate DLL with an exported interface.

DevPartner Error Detection checks the return value from the reallocation function and assumes that a NULL value indicates error. Non-NULL addresses are assumed to be the address of the newly allocated block of memory.

Sample Reallocator Records

The following examples show hypothetical Reallocator records:

Example 1

A function called `reallocXX` that is declared in a module called **`foo.dll`**. This function accepts two parameters. The first parameter is the address of the existing memory block and the second parameter is the size of the requested block. Since no optional parameters were specified, DevPartner Error Detection fills any new memory (assuming the block is larger) with the fill pattern before returning control to the application program.

```
Reallocator Module=foo.dll Function=reallocXX
MemoryType=MEM_MALLOC NumParams=2 Address=1 Size=2
```

Example 2

A function called `reallocClear` is declared in a module called **`foo.dll`**. This function accepts three parameters. The first parameter is the address of the existing memory block and the third parameter is the size of the requested block. This reallocation routine performs its own fill operation on any additional memory allocated in the new block so DevPartner Error Detection should *not* fill additional memory in new blocks.

Note: DevPartner Error Detection ignores the contents of parameter 2 since it is not of interest.

```
Reallocator Module=foo.dll Function=reallocClear
MemoryType=MEM_MALLOC NumParams=3 Address=1 Size=3
NOFILL
```

Example 3

A function called `MyRealloc` has been built in a .LIB file and is statically linked into a data collection component called **`DataStore.dll`**. `MyRealloc` accepts four parameters. The first and fourth parameters are of no interest to DevPartner Error Detection. The second parameter contains the address of the existing block and the third parameter contains the new size of the block. The data collection routine pre-loads new data into the block on reallocation.

```
Reallocator Module=DataStore.dll Function=MyRealloc
MemoryType=MEM_ALLOC NumParams=4 Address=2 Size=3
NOFILL STATIC
```

Note: Specify `STATIC` if the function name is not exported from the DLL.

Ignore Records

Create an Ignore record to describe a function that should be ignored by the DevPartner Error Detection memory tracking system. Follow this format:

```
Ignore Module=module_name Function=func_name [Optional Parameters]
```

Use Ignore records to instruct DevPartner Error Detection to either ignore a user-written allocator or ignore a lower-level access routine used by a user-written allocator. Ignore records tell the DevPartner Error Detection memory tracking system not to track APIs that would normally be monitored.

Note: If you create Ignore records, the DevPartner Error Detection memory tracking system no longer monitors memory allocated or freed from those APIs. As a result, DevPartner Error Detection is no longer aware of this memory. This may cause the Call Validation and FinalCheck analysis modules to generate incorrect or incomplete error messages. If you have any questions about the use of this feature please contact Customer Care for assistance.

Table 4-7. Ignore record parameters

Parameter	Description
Ignore	The first parameter in the record must be the word Ignore to indicate that you are describing an API to be ignored.
module_name	The name of the module (Executable or DLL) that contains the function to be ignored. **Note:** Module names do not support the use of wildcards.
func_name	The name of the function to be ignored by the memory tracking system. If you are using C++, this should be the 'mangled' name of the function. This parameter is case-sensitive.
[Optional Parameters]	• `DEBUG` If this parameter is specified, DevPartner Error Detection displays additional information about the hook. The output window (or dbgview) contains information on any errors that occurred when attempting to place the hook. The transcript pane displays statistics about each hook, including whether the function was successfully hooked and the number of times the hooked function was called. • `NODISPLAY` If this parameter is specified, DevPartner Error Detection stops displaying detailed information on each hook requested at the top of the transcript pane. • `STATIC` DevPartner Error Detection statically patches the user-written allocator. Specify the static option if your user-written allocator is linked into your application and is not provided in a separate DLL with an exported interface.

Sample Ignore Records

The following examples show hypothetical ignore records.

Example 1

This example creates an ignore record that causees DevPartner Error Detection to monitor memory allocated by `GlobalAlloc` but does not see any requests to free the memory returned to the operating system using `GlobalFree`.

Note: This causes DevPartner Error Detection to report a large number of false memory leaks.

```
Ignore Module=Kernel32.dll Function=GlobalFree
```

Example 2

This example tells DevPartner Error Detection to ignore memory manipulated by the `GlobalAlloc` family of calls.

Note: This causes DevPartner Error Detection to report a large number of false Call Validation and FinalCheck errors.

```
Ignore Module=Kernel32.dll Function=GlobalAlloc

Ignore Module=Kernel32.dll Function=GlobalReAlloc

Ignore Module=Kernel32.dll Function=GlobalFree
```

Note: Adding these three lines to `UserAllocators.dat` is not recommended.

Example 3

This example tells DevPartner Error Detection to ignore memory manipulated by a statically linked function within a specified module. You might need to add lines like these if you write your own replacement memory allocation library and use the same names as the standard C runtime library and don't want DevPartner Error Detection to monitor your library usage.

```
Ignore Module=MyDLL.dll Function=Malloc STATIC

Ignore Module=MyDLL.dll Function=free STATIC

Ignore Module=MyDLL.dll Function=realloc STATIC
```

Note: Before adding lines like this to your `UserAllocators.dat` file, you should contact Micro Focus Customer Care for assistance.

Coding UserAllocator Hook Requests

When you create UserAllocator hook requests, keep the following important notes in mind:

- Use `MEM_CUSTOM1` through `MEM_CUSTOM9` to segregate your memory allocations from the system's allocations.
- Designate unique custom allocator types for each distinct and separate allocator type.
- Always deallocate memory using the type that corresponds to the allocator. For example, if the allocator was hooked with type `MEM_CUSTOM1`, you must free the memory with a deallocator that is also hooked with type `MEM_CUSTOM1`. If the deallocator is different, an allocation conflict error is reported.

Code Requirements for UserAllocators

To make use of the UserAllocator memory allocation hooks, the application code must include functions that control memory allocation and deallocation.

Allocator Function Hooks

- The allocator function being called must include a parameter specifying the number of bytes of memory requested.
- The function must either return the location of the allocated memory, or include a parameter containing an address pointing to the location.
- Other parameters may also be included in the function.

Examples

```
void *GetMemory(int BytesRequested);

Allocator

    Module=bcheap.dll

    Function=GetMemory

    MemoryType=MEM_CUSTOM1

    NumParams=1

    Size=1

    Static

    Noguard

    NoFill

    Debug
```

Or

```
void *pVoid;

HRESULT GetMemoryAgain(int BytesRequested, &pVoid);

// in the above example, the memory allocator places the location of
the allocated memory in the pVoid pointer.

Allocator

    Module=bcheap.dll

    Function=GetMemoryAgain

    MemoryType=MEM_CUSTOM1

    NumParams=2

    Size=1
```

```
BufferLoc=2

Static

Noguard

NoFill

Debug
```

Deallocator Function Hooks

- The deallocator function being called must include a parameter specifying the address of the memory being deallocated.
- Other parameters may also be included in the function.

Example

```
; static void *mark_free(void *p, int nLen, void *pExclude, int
nExcludeLen)

Deallocator

    Module=bcheap.dll

    Function=mark_free

    MemoryType=MEM_CUSTOM1

    NumParams=4

    Address=1

    Static

    NoPoison

    Debug
```

Reallocator Function Hooks

- The reallocator function being called must include a parameter specifying the number of bytes of memory requested.
- The reallocator function must include a parameter specifying the address of the ‘old’ memory being reallocated.
- The reallocator function must either return the location of the ‘new’ memory that was allocated, or include a parameter containing an address pointing to the location.
- Other parameters may also be included in the function.

Example

```
; void *DoMyRealloc(MyHeap *me, void *p, uint32 uSize)

ReAllocator
```

```
Module=bcheap.dll

Function=DoMyRealloc

MemoryType=MEM_CUSTOM1

NumParams=3

Address=2

Size=3

NoFill

NoGuard

Static

Debug
```

Debugging UserAllocator Hooks

Error Detection provides two keywords you can use to modify the presentation of information about your UserAllocator hooks, making debugging those hooks easier.

NoDisplay

By default, Error Detection displays the details of the hook as interpreted from the file request at the top of the Transcript pane. For example:

```
Allocator Module=bcheap.dll Function=MarkNodeAllocated
MemoryType=MEM_CUSTOM1 NumParams=3 Size=1 BufferLoc=3 NoFill NoGuard
Static Debug
```

If you do not want the detailed hook information displayed at the top of the Transcript pane, add the keyword `NoDisplay` to the hook request.

Debug

Error Detection also provides the `Debug` keyword, which provides extended details when you add it to the hook request.

The Transcript Pane

The presence of the `Debug` keyword in a hook request shows extended details at the bottom of the Transcript pane after Error Detection completes the run. The details displayed at the bottom of the Transcript pane include statistics about each hook, including whether the function was successfully hooked and the number of times the hooked function was called.

Error Details

If the bottom of the Transcript pane indicates that the function was not hooked, Error Detection provides more details to aid in debugging the failed hook:

If you are working inside Visual Studio, the output window provides the error details.

If you are running the standalone version of Error Detection, the **DbgView** tool provides the error details.

How to Diagnose Errors in UserAllocators.dat

If you add records to **`UserAllocators.dat`** you may receive one or more of the following types of errors:

- File access errors

  `UserAllocators.dat` is a text file stored in the `Data` folder of the DevPartner Error Detection installation folder. If the file is deleted or made non-readable, DevPartner Error Detection reports an error.

- File write errors

  When the DevPartner Error Detection session is started, it creates a file named **`UserAllocators.nlb`**. If the location specified in **Options >Data > NLB File Directory** is invalid or read-only, a failure occurs. To resolve this, either change the folder specification under **Options**, or change the properties of the folder to make it writeable.

- Parsing errors

  If DevPartner Error Detection encounters errors while parsing the **`UserAllocators.dat`** file, it logs the errors in the Errors tab. If Memory Tracking or Resource Tracking are enabled when these **`UserAllocators.dat`** errors are encountered, these features are disabled.

Token Parsing Errors

DevPartner Error Detection parses the file one line at a time using the following rules:

- Blank lines and lines that start with semi-colons (;) or double-slashes (//) are ignored.
- Each UserAllocator definition that is added must begin with a valid record type.
- All parameters for each definition must be separated by one or more spaces or tabs, and must follow the rules for each record type.

Semantic Errors

Each record type is parsed according to the rules for each parameter. Parameters may be case-sensitive and in some cases must fall inside a valid range (for example, the maximum number of parameters supported on a function is 32).

Duplicate entries in the same file may also generate errors if the records conflict with one another. **`UserAllocators.dat`** is assumed to be an advanced feature, so extensive cross checking is not performed.

If Your Application becomes Unstable after Changing UserAllocators.dat

When you add records to **`UserAllocators.dat`**, you are telling DevPartner Error Detection to monitor calls to and from your user-written allocator. If you did not properly describe the APIs to DevPartner Error Detection, your application may crash or function unpredictably.

Specifying the incorrect number of parameters for your functions is one of the most common causes of such problems.

You may also encounter problems if you depend on the contents of memory remaining constant and you did not add the `NOFILL` or `NOPOSION` options to your description.

If you encounter an error and are unsure how to proceed please contact Micro Focus Customer Care for assistance. When you contact Customer Care, please provide the following information:

- The version of DevPartner you are running
- A copy of your **`UserAllocators.dat`** file
- A description of the problem you are encountering

In some cases, you may be asked to provide a copy of the DLL containing your user allocator functions and a map file used to link the DLL.

If possible, avoid using Ignore records. Ignore records can cause DevPartner Error Detection to respond unpredictably when you analyze an application.

Chapter 5

Deadlock Analysis

Deadlock Analysis provides an automated method to search for deadlocks, potential deadlocks and other synchronization errors in your customer applications.

This chapter provides:

- An overview of the terms used in deadlock analysis
- Examples of deadlocks and potential deadlocks
- Sources of additional information on synchronization topics.

Background: Single and Multi-threaded Applications

Old style C/C++ programs had a simple main routine that called a number of functions, performed various operations, and then exited. These programs used a single thread of execution. In other words, the program executed one instruction at a time. If you were to step through the program using a debugger, you could watch every operation pass by like the frames in a movie.

Threads

A *thread* is a flow of control. A multi-threaded application has two or more flows of control. You can create additional threads by calling the Windows `CreateThread` function. `CreateThread` accepts a number of parameters including the address of a function that should be run on the newly created thread. If the `CreateThread` function is successful, the application has an additional thread of execution.

There are many ways to implicitly create a thread. A few examples include calling `_beginthread`, using third-party libraries, using COM or DCOM, or by using the Common Language Runtime.

If you have more than one thread executing in your program, it is possible for the two threads to try to access the same resources at the same time. This might include variables, files, handles, Windows resources, and so on. If multiple threads try to access the same resource at the same time, synchronization problems can occur. For example, if two threads, called `T1` and `T2`, both attempt to print out the numbers from 1 to 100, the output from each thread might appear as follows:

1 2 3 4 5 6 7 8 9 10 11 12 … 95 96 97 98 99 100

When both threads run at the same time, the output be jumbleds, as in the following example. The output from thread `T1` is in plain type; the output from thread `T2` is in bold italic type.

1 2 3 4 ***1 2*** 5 6 ***3 4 5 6*** 7 8 ***7*** … 95 96 97 ***94 95 96 97 98 99*** 98 99 100 ***100***

Critical Sections

To prevent such problems, you need to coordinate the interactions between threads. Most modern operating systems provide a series of synchronization functions that can be called to coordinate access to shared resources. The easiest to use and most common synchronization object is called a *critical section.* A critical section is a simple function that allows only one thread to have access to a resource at a time.

Consider the example of threads `T1` and `T2`, each written to print the numbers from 1 to 100, as described above. Defining a critical section C1 prevents jumbled output when both threads are running. This critical section controls access to the output stream. The functions executed by threads `T1` and `T2` would need to be modified as follows:

1. One of the threads would create the critical section C1.
2. Each thread would then perform the following steps:
   a. Request critical section C1
   b. Print out the list of numbers from 1 to 100
   c. Release critical section C1
3. The threads would then go off and do whatever additional processing was required that did not interact with the other thread.

Step 2- a translates into an `EnterCriticalSection` call asking the operating system to grant the thread exclusive access to critical section C1. If the critical section is not available, the operating system pauses the thread and wait until C1 becomes available.

Once a thread has access to the critical section, any other thread following the critical section rules for C1 does not attempt to print its output. After the thread prints the numbers from 1 to 100, step 2- c tells the operating system to `LeaveCriticalSection`. This releases the critical section for some other thread.

There is no rule that states that every thread in your program must use the critical section to print its output to the terminal. However, if you follow this rule, your output always appears correctly.

This same rule can be applied to accessing variables, structures, files, or any other shared resource.

Note: In most cases you do not need to wrap console output in critical sections unless you are writing code that causes the two output streams to collide with each other.

Deadlock - A Basic Definition

Based on the preceding example, a critical section seems to be a very simple mechanism to grant access to a shared resource. However, it can cause problems.

Consider a program that creates multiple critical sections named C1, C2 and C3. Each of these critical sections is used to guard access to a separate resource shared between the threads.

If a thread is granted access to one critical section (for example, C1) and then attempts to gain access to another critical section (for example, C2), it is possible that C2 has already been allocated by another thread. If the other thread quickly releases C2, there is no problem. The first thread waits until C2 becomes available then grants access to C2 so the operation can continue.

On the other hand, if the thread that holds C2 needs to wait for some other synchronization object to become available (such as C1), both threads stall waiting to gain access to the necessary resources. When two or more threads stall while waiting for resources that never become available, the result is called a *deadlock*.

Techniques for Avoiding Deadlocks

Deadlocks occur when multiple threads attempt to use shared resources but are unable to gain access to those resources. There are a number of methods to avoid deadlocks.

- Request access to synchronization objects only when you need them. Once you gain access to the objects, use them as quickly as possible and release the objects so other threads can use them.
- If you need to gain access to multiple synchronization objects at once to perform a given operation, request the first object and then try to gain access to the second object. If the second object is not available, release both objects and wait a short random interval. After the wait completes, attempt to gain access to the resources again. It is very important to release ownership of a resource if your thread becomes blocked waiting for another resource. Failure to release the object might cause a “deadly embrace” which makes the deadlock situation worse.
- Always ask for resources in the same order. For example, if you are required to gain access to C1, C2 and C3 to perform an operation, always access them in the same order (C1, C2, and C3) and release them in the opposite order (C3, C2, C1).
- Once you have acquired all the synchronization objects you need to perform an operation, do not perform an operation that might block waiting for another resource.

There are many more techniques for dealing with synchronization objects. “Additional Information” on page 80 lists MSDN resources and books that discuss synchronization objects.

Potential Deadlocks

DevPartner Error Detection reports a *potential deadlock* when it detects that you are not accessing resources in a safe manner. An example would be an application with three threads T1, T2, and T3, all of which make use of a series of resources controlled by critical sections C1, C2, and C3.

Table 5-1 illustrates the critical sections each thread requires to perform a given operation:

Table 5-1. Potential deadlock example: Threads and their required critical sections

Thread	Critical Section
T1	C1, C2

Table 5-1. Potential deadlock example: Threads and their required critical sections

Thread	Critical Section
T2	C2, C3
T3	C3, C1

Each thread can run independently and acquire the necessary critical sections to perform their designated tasks. However, a problem can occur when all three threads try to perform these operations at the same time.

The Dining Philosophers

The *Dining Philosophers* is a classic example often used to illustrate potential deadlocks in computer science classes. The DevPartner Error Detection software contains sample code for a version of the Dining Philosophers. You can find it under:

`...\DevPartner Studio\Examples\DeadlockPhilosophers`

The Dining Philosophers problem is based on a group of philosophers sitting at a round table with a large plate of food in the middle. Between each philosopher is a single chopstick.

The philosophers seated around the table can do three different things:

1. **Rest:** Resting philosophers sit and do nothing. They rest for a random period of time.
2. **Talk:** Talking philosophers speak to anyone else at the table interested in listening. They talk for a random period of time.
3. **Eat:** A hungry philosopher attempts to eat. To do this, they try to pick up a chopstick. In the simplest case, the philosopher always tries to pick up the left chopstick first, and, if successful, tries to pick up the right chopstick. A philosopher with both left and right chopsticks eats for a random amount of time, and then put down the chopsticks and either rest or start talking.

   A philosopher who can't pick up the first chopstick waits a few seconds and tries again. A philosopher who succeeds then attempts to pick up the right chopstick. If the right chopstick isn't available, the philosopher waits a few seconds and then tries again.

The problem occurs when all philosophers pick up the left chopstick at once. If this occurs, none of them ever put down the left chopstick and all starve to death (deadlock).

Depending on how you configure the Dining Philosopher algorithm, it might deadlock immediately or might run for several minutes before deadlocking. If you add philosophers and chopsticks to the table, the number of actual deadlocks tends to decrease. However, the potential for all philosophers to get hungry at once still exists. This is called a *potential deadlock*.

Potential deadlocks are often the hardest deadlocks to track down because they tend to occur on heavily loaded production systems. Attempts to duplicate these problems on development systems are usually time consuming and often don't find the real root of the problem.

DevPartner Error Detection notifies you if a potential deadlock is detected long before the actual deadlock occurs. DevPartner Error Detection also provides detailed information describing how the actual deadlock occurs. This can make it easier to modify your code to prevent the problem from occurring.

Monitoring Synchronization Objects

Deadlock analysis also monitors all the synchronization objects in your application for errors and questionable usage such as:

- Waits over a user specified duration
- Threads that re-enter an already owned critical section
- A wait on a mutex that is already owned by the thread
- Exiting a thread without releasing a synchronization object

In addition, you can configure DevPartner Error Detection to verify that synchronization objects that can be named follow your naming rules. For example, you might decide that your synchronization objects should be unnamed so that they cannot be accessed from outside the process. Any named synchronization objects are flagged as potential errors. You can then use this list to verify that the named synchronization objects contain the necessary security descriptors to prevent unwanted access by other processes on the system.

A complete list of synchronization errors appears in the online help, under *Deadlock Errors* in the *Descriptions of Detected Errors* section.

Other Synchronization Objects

The Windows operating system provides many different types of synchronization objects beyond the critical sections described on page 76. Below is a list of synchronization objects and excerpts of their definitions from MSDN. Excerpted text is printed in italics. With each term, the URL for the complete definition and a URL for a related code example are provided.

Critical Section

Critical section objects provide synchronization similar to that provided by mutex objects, except that critical section objects can be used only by the threads of a single process.

Complete definition:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/Critical_Section_Objects.asp

Code example:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/using_critical_section_objects.asp

Event

An event object is a synchronization object whose state can be explicitly set to be signaled by use of the `SetEvent` function. The event object is useful in sending a signal to a thread indicating that a particular event has occurred.

Complete definition:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/event_objects.asp

Code example:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/using_event_objects.asp

Mutex

A mutex object is a synchronization object whose state is set to signaled when it is not owned by any thread, and nonsignaled when it is owned. Only one thread at a time can own a mutex object.

Mutex objects are considerably slower than critical sections.

Complete definition:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/mutex_objects.asp

The following URL shows an example of using Mutex objects:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/using_mutex_objects.asp

Semaphore

A semaphore object is a synchronization object that maintains a count between zero and a specified maximum value. The count is decremented each time a thread completes a wait for the semaphore object and incremented each time a thread releases the semaphore.

Complete definition:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/semaphore_objects.asp

Code example:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/using_semaphore_objects.asp

Additional Information

MSDN References

For more information on synchronization objects please refer to the following links on MSDN:

Synchronization Overview:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/about_synchronization.asp

Synchronization Objects:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/synchronization_objects.asp

Wait Functions:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/wait_functions.asp

Using Synchronization Objects:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/using_synchronization.asp

Synchronization Reference:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/synchronization_reference.asp

Other References

The following books contain more information on synchronization objects:

Win32 Multithreaded Programming, by Aaron Cohen and Mike Woodring

Debugging Applications for Microsoft .NET and Microsoft Windows, by John Robbins

Debugging Windows Applications, 1st Edition, by John Robbins

Operating Systems, *4th Edition*, by William Stallings

Foundations of Multithread, Parallel and Distributed Programming, by Gregory R. Andrews.

Appendix A
Troubleshooting Error Detection

Troubleshooting

If you encounter the following problems, try the corresponding solutions. If you encounter further difficulties, contact Micro Focus Customer Care.

Problem	Solution
Error Detection is stepping on my memory.	• Disable **Poison on Free** under **Memory Tracking** in the Options/Settings dialog box. • Disable **Enable Fill on Allocation** under **Memory Tracking** in the Options/Settings dialog box.
Error Detection fails to stop in the debugger when the Program Error Detected dialog box is dismissed with the **Debug** button.	• When you close the Program Error Detected dialog box with the **Debug** button, Error Detection puts a temporary breakpoint in your application at the start of the first line after the error. The breakpoint is removed when the application stops in the debugger. If your application has an exception handler that catches the breakpoint exception and continues execution, the debugger does not get a chance to catch the breakpoint and stop the application.
I am getting a *Memory Allocation Conflict: Function Mismatch* error, and Error Detection is reporting that the block was deallocated by `Free`.	• If you are using `new` and `delete`, and Error Detection is complaining that the block was deallocated by `free`, then you may be using a release version of the C Run Time Library DLL (`/MD`). To fix this, build against the debug version of the C Run Time Library DLL (`/MDd`). This is controlled on the **C++ Code Generation** properties page by the **Runtime Library** entry.
Error Detection complains about an allocation conflict between operator new and free.	• Make sure that you have added all user written allocators to **`userallocators.dat`**. See the comments in the **`...\Data\UserAllocators.dat`** file.

Problem	Solution
Error Detection appears to be ignoring my memory and resource allocations. Why?	• If running Error Detection on a managed C++ application with the managed debugger, memory and resource allocations appears to be coming from **`mscorwks.dll`** or **`mscorsrv.dll`** instead of the application, and Error Detection ignores them. To use Error Detection on a mixed or managed code application, follow these steps: 1 Open the Visual Studio version you want to debug your process. 2 Start the DevPartner Error Detection standalone application. 3 Open your target process inside Error Detection, and select **Program > Start** to run your process. 4 Attach the debugger to your target process by clicking **Tools > Attach to Process** **Note:** Specify **Managed Code Debugging** (not **Mixed** or **Auto**) to allow the debugger to hit your managed code breakpoints correctly. Once you complete the steps, you can set and hit breakpoints in your managed code, and Error Detection detects your memory and resource allocations.

Problem	Solution

I am getting an *Invalid String* error when I run my application under Error Detection.	• When ASCII strings are cast to wide-char strings and used as API parameters, errors may be reported that do not appear to be related to the actual problem. A cast means the data type is trusted, so it is not detectable by Error Detection. For example:
	
	`BSTR m_DSNName;`
	`m_DSNName=SysAllocString(BSTR(""));`
	
	Is interpreted by the compiler as:
	
	`m_DSNName=SysAllocString((wchar_t *)"");`
	
	Error Detection reports:
	
	`Invalid String: In call to SysAllocString, address 0x0FE2751F is not null terminated in block 0x0FDF0000 (290816).`
	
	The `SysAllocString` statement casts an empty ASCII string to a wide-char string to use as the input parameter to `SysAllocString`. `SysAllocString` looks for the first double-byte null to find the end of the `wchar_t` string. It uses the length determined by the location of this double-byte null to determine the size of the new `BSTR`. `SysAllocString` validly creates a new `BSTR` for `m_DSNName` that holds a copy of all of the junk that exists between the beginning of the single-byte ASCII string & the first double-byte null that happened to occur later in memory.
	
	In the example above, Error Detection examines the input `wchar_t` string (which is actually a 1-byte empty ASCII string) and reports an invalid string (not null terminated). In the test code used for this case, DevPartner Error Detection knows about a valid block of memory that physically exists between the beginning of the single-byte ASCII string and the accidental double-byte null. It scans the input string only as far as the memory is not allocated for anything else. So in this case only a few bytes are scanned, reporting indicates a double-byte null in that area of memory is not found. To avoid this problem, change the code to use a wide-char string as input.

Problem	Solution
Error Detection is not reporting any errors.	• Enable **Log Events** under **General** in the Options/Settings dialog box. • There are two situations in which DCOM or COM-based applications or components try to run under the restrictions of the `aspnet` account. By default, when a DCOM or COM application or component is launched from within an `ASP.NET` enabled web page, it runs in the context of the `aspnet` account. For security reasons, the `aspnet` account is a restricted account (it is a member of the Users group and has equivalent privileges). In this situation your COM component does not have the security privileges required for Error Detection to function properly. To work around this issue, you must configure your DCOM or COM application or component to execute within the context of the interactive user (via `dcomcnfg.exe`). To configure your DCOM or COM application or component to run under the context of the interactive user: 1 Open a command prompt and run `dcomcnfg.exe`. 2 Expand **Component Services > Computers > My Computer > DCom Config**. 3 Right-click on your COM component, and select **Properties**. 4 Select the Identity tab. 5 Ensure that you have selected **The interactive user**. 6 Click **OK**.
Error Detection is running exceedingly slow.	• Verify that you have not turned on all of the Error Detection options. Some options are very expensive when it comes to processor and memory usage, and you should only use those you really need. • Limit your maximum call stack depth on allocation (via **Data Collection** in the Options/Settings dialog box). Deep allocation call stacks consume large amounts of memory. • Limit your analysis by excluding modules and files (via **Modules and Files** in the Options/Settings dialog box). • Only use FinalCheck at major development milestones. • Disable .NET Call Reporting, or limit the number of assemblies selected in the **All types** tree view. • If you are only interested in detecting leaks in your application, consider enabling leak-only analysis. Checking the **Enable Leak Analysis Only** check box under **Memory Track** options disables everything in Memory Track, with the exception of monitoring for leaks. Memory Track does not look for overruns, use of uninitialized memory, or dangling pointers. Call Validation **Memory Block Checking** is also disabled because Memory Track is not evaluating any memory allocated by system modules.

Problem	Solution
The log is far too large.	• Disable **Log Events** under **General** in the Options/Settings dialog box. • Limit the Parameter Encoding depth (via **Data Collection** in the Options/Settings dialog box). • Limit your analysis by excluding modules and files (via **Modules and Files** in the Options/Settings dialog box). • Use Filters and Suppressions to limit the scope of what is reported. • Disable **.NET Call Reporting**, or limit the number of assemblies selected in the **All types** tree view. • Disable **Collect API Method Calls and Returns** under **API Call Reporting**. • Disable **Resource Tracking**, or limit the resources being tracked.
I get an error stating that Error Detection failed to create the `UserAllocators.log` file.	• If attempting to run an application with Error Detection in a folder to which you do not have write access, you may receive an error similar to the following: `UserAllocatorsError: An error was discovered when processing the UserAllocators.dat file. Failed to create UserAllocators.log file Error:0x00000005` You can control the folder that the **`UserAllocators.log`** file is written to by setting the **NLB File Directory** on the **Data Collection** settings page.
I receive errors when I try to open my target application using Error Detection under Windows Vista.	• Error Detection creates data files for each target application. You must ensure that you have write access to the folder containing the target executable before starting Error Detection.
Error Detection has incomplete or erroneous call stacks.	• Error Detection cannot locate the symbols, or the symbols are out of date. Enable Microsoft Symbol Server to retrieve symbols that match your current files, then rerun your application. • The application under test uses a mixture of managed and unmanaged code. Transitions between the two kinds of modules can throw off the call stack.
Symbol Server is taking far too long.	• Error Detection is trying to retrieve symbols from Microsoft Symbol Server every time you run. Disable Microsoft Symbol Server once you have retrieved the symbols that match your current files, and run your application using local symbols. • Download and permanently install a symbol server package for you specific operating system. See Microsoft's symbol download page for details.

Problem	Solution
COM Object Tracking (or COM Call Reporting) does not appear to be working correctly.	• Check your **Modules and Files** settings in the Options/Settings dialog boxto verify that you have not disabled reporting of events for certain modules. • Check your **Suppression and Filter** settings in the Options/Settings dialog box to verify that you have not previously set a filter or suppression for certain COM Tracking event(s). • Allow Error Detection to monitor more COM interfaces via the **All Interfaces** check-box under **COM Call Reporting** in the Options/Settings dialog box.
My application does not start, or it starts and immediately crashes.	• Clear the **Enable Poison On Free** check box under **Memory Tracking** in the Options/Settings dialog box. The application may be referencing deallocated memory. • Clear the **Fill Output Arguments Before Call** check box under **Call Validation** in the Options/Settings dialog box. • Investigate your user written allocators if you have any (see the **`...\Data\UserAllocators.dat`** file). • If your system has OS symbols present, you could have a bad symbol file. • If the crash is timing related, disable some of the Error Detection features and try running the application again.
My service starts and immediately hangs.	• Make sure you are running your service with Administrative privileges.
My service starts and almost immediately terminates.	• The most likely cause is that the Windows Service Control Manager terminated your service. Increase the value of `dwWait` in your service's initialization logic and re-run your service. • Verify that Error Detection has a valid working folder. Specify the working folder via the General settings, under **Settings** in the **Program** menu.
My service runs for a while then terminates unexpectedly.	• Your service may be responding too slowly to a control message requesting your service state. Increase the time out value in `dwWait` when responding to service state requests. • Error Detection may have poisoned memory in your application, causing the crash. Disable the **Memory Tracking** feature in the Options/Settings dialog box. If this eliminates the crash, instrument your service with FinalCheck, then rerun your application looking for uninitialized memory references, buffer overruns, and dangling pointers.
My service runs correctly, but terminates unexpectedly when it shuts down.	• Your service is given a limited time to respond when it receives a shut down request from the Service Control Manager. When an application shuts down, Error Detection performs many checks, looking for memory, resource and interface leaks, and re-checking allocated memory blocks for memory overruns. If the `dwWait` value specified for acknowledging the shut down request is too small, the Service Control Manager terminates the service. In this case, increase the `dwWait` value.

Problem	Solution
Error Detection fails to analyze my service.	Error Detection may fail to analyze a service. A common cause is that one (or more) of the directories used to monitor the process is not writable to the process being created. The following directories must be writable: • The `TMP` and `TEMP` environment variables must reference a folder that is writable to the process. If you have configured your service to run the `LOCAL_SYSTEM` context, assign these environment variables system wide. • The service must have write access to the NLB File folder. • The service may require write access to the working folder.
Visual Studio stops responding when I try to run an application with Error Detection.	Narrow down the cause by using at least one of the following methods. • Set breakpoints at various locations in the application test code. • Reconfigure the Error Detection settings by disabling various sub-systems or modules. See "Changing Program Settings" in the online help for more details.
I see memory growth in Task Manager, but Error Detection is not reporting any leaks. Why not?	• Make sure that you have fully instrumented your application with FinalCheck. • Make sure **Log Events** is enabled under **General** in the Options/Settings dialog box. • Make sure your application test modules are enabled via **Modules and Files** in the Options/Settings dialog box.
When running Error Detection on a managed C++ application, I am not seeing as many leaks and errors reported as I was expecting.	• This is a known limitation of Error Detection. You must be using the native debugger (or both the managed and native debugger) for Error Detection to see the memory allocations and report them properly. When using just the managed debugger, memory and resource allocations appear to be coming from **`mscorewks.dll`** instead of the application, and are ignored.
ActiveCheck does not find a memory leak, but FinalCheck does. Why?	• Instrumentation with FinalCheck allows for more robust memory tracking. FinalCheck is scope-aware, and allows Error Detection to track every pointer as well as all memory usage. For more information, refer to *ActiveCheck and FinalCheck* in the Error Detection section in the Quick Reference.

Problem	Solution
My application fails when run under Error Detection (usually with Memory Track enabled).	• Clear the **Enable Poison On Free** check box under **Memory Tracking** in the Options/Settings dialog box. • Try using alternate fill patterns for the **Enable Guard Bytes** and **Enable Fill On Allocation** settings under **Memory Tracking** in the Options/Settings dialog box. • You may have bad PDBs. Create an ASCII text file named **NMSYMDATA.DAT** in the NMShared\4.7 folder. This file should contain the names of the modules associated with bad PDB files followed by “,0x0”. For example: ADVAPI32.DLL,0x0 • If you are debugging an application generated using the Visual Studio MFC application wizard, and have enabled the Memory Track subsystem’s memory filling features, Error Detection may cause the application to crash. MFC sets an obscure pragma that causes the compiler to generate “minimal debugging information”. If the OS structures you are using have had additional fields added to them, Error Detection may get the wrong structure size from the debugging information when it attempts to determine how big the structure is, causing it to fill memory it should not be touching. Add _AFX_FULLTYPEINFO to the Preprocessor Definitions on the C++ Preprocessor settings page for your project, and then rebuild your solution. • If creating the **NMSYMDATA.DAT** file does not solve the problem, you may have to exclude the entire module. To exclude an application module, create an ASCII text file named **EXCLUDEDMODULES.DAT** in the Data folder where Error Detection is installed on your system. For example: **<InstallationRoot>\Data\EXCLUDEDMODULES.DAT** Add the names of each module you want to exclude on a separate line of the file. For example: **MYCUSTOMDRIVER.DLL**
I cannot debug my Web application using Error Detection integrated in Visual Studio.	• To debug applications and services such as Web applications, use the **Wait for Process** option (see “Analyzing Services” on page 39) available from the Error Detection application (**BC.EXE**). This option is not available when running Error Detection integrated in Visual Studio.

Problem	Solution
I have a module located in the System files folder that I want to debug. The System files are restricted, and I cannot debug my module. What can I do?	• Enable your modules via **Modules and Files** in the Options/Settings dialog box, and you can debug it for the current project. • To make the module valid for all projects and solutions, edit the file named **`Unrestricted_modules.txt`** in the `Data` folder where Error Detection is installed on your system. For example: **`<Installation-Root>\Data\Unrestricted_modules.dat`** Add the names of each module you want to include on a separate line of the file. For example: **`MYCUSTOM-DRIVER.DLL`**
Error Detection is reporting errors in **`dllhost.exe`** or **`TestCon32.exe`**.	• To prevent Error Detection from reporting errors on **`dll-host.exe`** or **`TestCon32.exe`**, exclude the executable from the list of modules to check.
COM Call Reporting is not logging calls to my object or component.	• Error Detection logs methods only for COM interfaces that it has been instructed to recognize. To tell Error Detection about your ActiveX control, select **Enable COM method call reporting** on objects that are implemented in the selected modules under **COM Call Reporting** in the Options/Settings dialog box.
Error Detection is not reporting COM interface leaks in my object or compo-nent.	• To collect COM interface leak information, select **Enable COM object tracking** under **COM Object Tracking** in the Options/Settings dialog box, then select the COM classes to monitor. • To track your own objects, review the list of COM classes in the **COM Object Tracking** settings and select only your classes. If you are unsure which classes to select, select **All COM classes**.
Error Detection appears to hang and not respond for a long time after I stop exercising my component.	• Error Detection is waiting for **`dllhost.exe`** to time out and terminate the process. When **`dllhost.exe`** terminates, Error Detection performs the final memory, resource and interface leak detection.
Why does IIS startup and then hang?	• Error Detection requires Administrative privileges to debug a service. If the account used does not have Administrator privileges, IIS eitherstops respondiog or terminates with an error.
I cannot instrument my code for Error Detection when I compile automatically via my command line batch process (using VCBUILD.EXE).	• To instrument your code when you call the compiler from the command line, you need to use the `NMVCBUILD` variant installed with DevPartner. Refer to "Instrumenting Native C/C++ Code with nmvcbuild" for more information about com-piling with `NMVCBUILD`.
I receive errors when I try to build my target application with instrumentation for Error Detection under Windows Vista.	• Error Detection creates data files for each target applica-tion. You must ensure that you have write access to the folder containing the target executable before starting Error Detection.

Problem	**Solution**
When I run my application under Error Detection, I receive bad data and/or unexpected results.	• Check to ensure that you do not have expressions in your code that rely on an undefined order of evaluation. Instrumenting code without a well defined order of evaluation can cause unexpected results, including erroneous data, hangs, and even crashes. The C/C++ standard explicitly does *not* define the order of evaluation when there are side effects, such as storing a value in an object. For example, the following statement does not have a well defined order of evaluation: `i = ++i + 2;` There are two points in this statement when values are stored in the variable *i*, and the language does not define what order they occur in. Instrumenting code like this may change the order of evaluation and change the result.
Error Detection is reporting false leaks and overruns in a mixed-mode (managed / unmanaged) environment.	• If you are running Error Detection on a managed C++ application with the managed debugger, memory and resource allocations appear to be coming from **`mscorwks.dll`** or **`mscorsrv.dll`** instead of the application, and Error Detection ignores them. To use Error Detection on a mixed or managed code application, follow these steps: 1 Open the Visual Studio version you want to debug your process. 2 Start the DevPartner Error Detection standalone application. 3 Open your target process inside Error Detection, and select **Program > Start** to run your process. 4 Attach the debugger to your target process by clicking **Tools > Attach to Process** **Note:** Specify **Managed Code Debugging** (not **Mixed** or **Auto**) to allow the debugger to hit your managed code breakpoints correctly. Once you complete the steps, you can set and hit breakpoints in your managed code, and Error Detection detects your memory and resource allocations.

Appendix B

Important Error Detection Files

Files and Their Purpose

The following table lists the files Error Detection uses to control and define behavior during a session. Included are the file location, name, purpose, and whether the file is user-modifiable.

Filename and Path	Purpose
<Program Root>\Data\ctisafe.dat	Specifies functions known to accept pointers without storing the pointer values. MemTrack and FinalCheck use this file to keep track of safe functions. This information prevents an error from being triggered when a pointer visits an unknown function. When a function is listed in this file, MemTrack and FinalCheck assume that the function has not saved a copy of the pointer. Functions may be added when necessary. Consult with Micro Focus Customer Care before removing any default functions from this list.
<Program Root>\Data\BCDefault.DPRul	Lists the default set of suppression and filter files that Error Detection will load. Add to this list using the suppression and filter editing dialogs within Error Detection; the information you add is valid for the current project directory only. To make information valid system-wide, edit the file manually and include the full path name to the suppression/filter file being added.
<Program Root>\Data*.DPFlt **<Program Root>\Data*.DPSup**	Defines the filters and suppressions Error Detection should use. Each one of these **.DPFlt** and **.DPSup** files contains specific filters and suppressions for system modules. Add, modify, or delete suppressions and filters via the Error Detection suppression and filter editing dialogs. Do not edit these files manually.

Filename and Path	Purpose
<Program Root>\Data\Unrestricted_modules.txt	Specifies modules that should be unrestricted DLLs despite being found in system directories. A module existing in a system directory needs to be marked as unrestricted to be examined for errors. By default, **Unrestricted_modules.txt** lists the various versions of MFC modules. Edit this file manually to add the names of specific modules existing in system directories so Error Detection examines them for errors.
<Program Root>\Data\UserAllocators.dat	Specifies custom allocators. For more information on this file and how it is used by Error Detection, see Chapter 4, "Working with User-Written Allocators".
<Program Root>\ERptApi\NMApiLib.*	Provides access to the Error Detection user-callable interface. **NMApiLib.h** defines and documents the user-callable interface to Error Detection, which is implemented by linking **NMApiLib.lib** into your project. For more information on the user-callable interface, see Chapter 3, "Analyzing Complex Applications".
<Program Root>\Data\ExcludedModules.dat	(User-created File) Contains a list of excluded modules. Each module is listed on a separate line of the file. For example: **MYCUSTOM-DRIVER.DLL**
<Program Root>\DPSErrorDetection.xsd	Contains the schema information used when exporting session file data to XML. Do not edit this file.
<NMShared Root>\NMSymData.dat	(User-created File) Contains the names of any modules associated with bad PDB files, followed by ",0x0". For example: ADVAPI32.DLL,0x0
<Program Root>\DPSErrorDetection.xsd	Specifies the schema to be used by the Data Export to XML feature.

Key

<Program Root> = C:\Program Files\Micro Focus\DevPartner Studio\BoundsChecker

<NMShared Root> = C:\Program Files\Common Files\Micro Focus\NMShared

Note: For installs on 64-bit versions of Windows, DevPartner Studio is located at: **\Program Files (x86)\Micro Focus\DevPartner Studio** and **\Program Files (x86)\Common Files\Micro Focus**

Index

A
ActiveCheck 49
ActiveX 39
  components 33
  debugging controls 37
administrative privileges 40, 46, 91
applications
  complex 33
  multi-threaded 31
  single-threaded 75
  transactional 38

B
BCDefault.DPRul 93

C
call parameter encoding depth 28
call validation 23, 25
CLR analysis 25
COM
  components 33
  servers 39
  usage 17
command line 12
complex applications 33
  analyzing 26
  debugging 38
conditional code 35
configuration file management 20
critical section, synchronization object 76
CTISafe.dat 93

D
dangling pointers 31
deadlock 77
  potential 77
default settings 23
dwWait 40

E
ExcludedModules.dat 94
executable files
  dllhost.exe 43

F
file extensions
  .DPFlt 93
  .DPRul 93
  .DPSup 93
filters 35
FinalCheck 49

G
guard bytes 18

I
IIS 34
  process 45
important files
  BCDefault.DPRul 93
  CTISafe.dat 93
  ExcludedModules.dat 94
  NMApiLib 94
  NMSymData.dat 94
  Unrestricted_modules.txt 94
  UserAllocators.dat 94
instrumenting native code 14
interface
  command line 12
  leaks 26

ISAPI filters 20, 33, 37, 45, 46

L

log file 28

M

managed code 19, 24, 25
memory
  leaks 26
  overrun 24
  poisoning 18
  tracker 20
modules and files 20, 34, 37
  and complex applications 27
  and reverse engineering 30
modules tab 27
multiprocessor application servers 31
multi-threaded applications 31

N

native code 19, 24, 25
NMApiLib 94
NMSymData.dat 94
NMVCBUILD 14

P

P/Invoke 24, 29
  interop monitoring 25
pointers, dangling 31
poisoning memory 18, 31
potential deadlock 77

R

resource leaks 20, 26
resource tracker 20

S

service control logic 40
services, debugging 34
settings 20, 21
  default 12
  refining 12
single-threaded applications 75
StartEvtReporting 36
StopEvtReporting 36
suppression 35

T

test container 38
third-party software 12, 26, 35
thread, definition of 75
transactional applications 38
troubleshooting 83

U

Unrestricted_modules.txt 94
UserAllocators.dat 94

V

VCBUILD 14

W

Windows
  service control manager 41, 88
  services 33, 39
  services, debugging 37
worker thread 40