



Understanding BoundsChecker

Release 7.2



Technical support is available from our Technical Support Hotline or via our FrontLine Support Web site.

Technical Support Hotline:
1-800-538-7822

FrontLine Support Web Site:
<http://frontline.compuware.com>

This document and the product referenced in it are subject to the following legends:

Access is limited to authorized users. Use of this product is subject to the terms and conditions of the user's License Agreement with Compuware Corporation.

© 2004 Compuware Corporation. All rights reserved. Unpublished - rights reserved under the Copyright Laws of the United States.

U.S. GOVERNMENT RIGHTS

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in Compuware Corporation license agreement and as provided in DFARS 227.7202-1(a) and 227.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii)(OCT 1988), FAR 12.212(a) (1995), FAR 52.227-19, or FAR 52.227-14 (ALT III), as applicable. Compuware Corporation.

This product contains confidential information and trade secrets of Compuware Corporation. Use, disclosure, or reproduction is prohibited without the prior express written permission of Compuware Corporation.

ActiveCheck, BoundsChecker, DevPartner Studio, and FinalCheck are trademarks or registered trademarks of Compuware Corporation.

Acrobat® Reader copyright © 1987-2002 Adobe Systems Incorporated. All rights reserved. Adobe, Acrobat, and Acrobat Reader are trademarks of Adobe Systems Incorporated.

All other company or product names are trademarks of their respective owners.

US Patent Nos.: 5,987,249 and 6,332,213

Doc. 11472
March 23, 2004

A collage of various images including a globe, a person, a butterfly, a flower, and a person in a boat, arranged in a grid-like pattern. The images are semi-transparent and overlap each other, creating a layered effect. The colors are primarily blues, greens, and yellows, giving it a natural and serene feel.

Who Should Read This Manual	ix
Information for BoundsChecker64 Users	x
What This Manual Covers	x
Conventions Used In This Manual	xi
Customer Assistance	xi
For Non-technical Issues	xi
For Technical Issues	xii

Installing BoundsChecker

System Requirements	1
BoundsChecker	1
BoundsChecker64	2
Receiving Your BoundsChecker License File	2
Installing BoundsChecker	3
Your BoundsChecker64 Serial Number	4
Installing BoundsChecker64	4
Terminal Server Installation	5
Notes About the Installation Process	5
Removing an Installation	5

The BoundsChecker Solution

Check Early, Check Often—The BoundsChecker Philosophy	7
What's New in BoundsChecker	8
New for the BoundsChecker 7.2 Release	8

Review of Release 7.0	9
BoundsChecker Workflow	9
The Benefits of Using BoundsChecker	10
Comprehensive Error Detection	10
Flexible Debugging Environment	13
Integration with the Visual C++ Debugger	14
Detailed View of Memory and Resource Usage	14
Advanced Error Analysis	14
Open Error Detection Architecture	15

Chapter 3

The BoundsChecker User Interface

BoundsChecker Main Window	17
Results Pane	18
Details Pane	19
Source Pane	19
Settings Dialog Box	19
An Example: Call Validation	20
Using Other Settings Categories	21
Program Error Detected Dialog Box	21
Buttons on the Program Error Detected Dialog Box	22
Hints for Using the Program Error Detected Dialog Box	23
Memory and Resource Viewer Dialog Box	24
The Memory and Resource Viewer User Interface	24
Suppression and Filtering Dialog Boxes	26
Suppressing Errors	26
Filtering Errors	26
Suppression and Filtering User Interface	26
Creating and Saving Suppression and Filter Files	27
IDE Integration	28
Using BoundsChecker with Visual Studio 6	28
BoundsChecker Integration with Visual Studio .NET	29
Using BoundsChecker with Visual Studio .NET	31

Chapter 4

Workflow and Configuration Settings

BoundsChecker Workflow	33
Benefits of the BoundsChecker Workflow	34
Saving Error Detection Configurations	34
Customizing the BoundsChecker Settings	35
General	35

Data Collection	36
API Call Reporting	36
Call Validation	36
COM Call Reporting	37
COM Object Tracking	37
Deadlock Analysis	38
Memory Tracking	38
.NET Call Reporting	39
.NET Analysis	39
Resource Tracking	40
Modules and Files	40
Fonts and Colors	40
Configuration File Management	41

Chapter 5

Checking and Analyzing Programs

BoundsChecker Error Detection Tasks	43
Finding Leaks	43
Finding Pointer and Memory Errors	44
Finding Memory Corruption	44
Analyzing Legacy Code in .NET Applications	44
Validating Win32 API Calls	46
Searching for Application Deadlocks	46
Expanded Uses for BoundsChecker	46
Understanding Complex Applications	46
Reverse Engineering	48
Stress Testing	51

Chapter 6

BoundsChecker 7.2 for BoundsChecker 6 Users

Changes to the User Interface	55
Workflow Changes	56
Error Detection Settings	56
BoundsChecker 6 Normal and Maximum mode	57
BoundsChecker 6 Cross Reference	59
Error Detection Tab	59
General Settings	61
Event Reporting Tab	62
Program Information Tab	62
Error Suppression Tab	63
Modules and Files Dialog Box	64

Other BoundsChecker 7.2 Capabilities	64
--	----

Chapter 7

Analyzing Complex Applications

About Complex Applications	67
Image File Execution Options	68
An Example: Associating BoundsChecker with an Executable	69
Analyzing Limited Parts of Your Program	70
Using Modules and Files Settings	72
Deciding What to Monitor	73
How Does an Application Start Up?	74
Analyzing Services	75
Requirements and Guidelines	75
Analyzing a Service	76
Timing Problems and dwWait	76
Interactive Debugging	76
Alternate Method: Separating Control Logic from the Worker Thread	77
Custom Code to Turn the BoundsChecker Log On and Off	77
Common Service-related Issues	77
Analyzing ActiveX Controls Using the Test Container	78
Common Test Container Issues	79
Analyzing COM Objects Using MTS	80
Common MTS Issues	80
Analyzing Applications That Use COM+ 1.0 and 1.5	82
Common COM+ Issues	83
Analyzing ISAPI Filters Under IIS 5.0	84
Common ISAPI Filter Issues	86
Analyzing ISAPI Filters under IIS 6.0	87
IIS 5.0 Isolation Mode	87
IIS 6.0 Default Configuration	89
Common IIS 6.0 ISAPI Filter Issues	91
Frequently Asked Questions	92

Chapter 8

Working with User-Written Allocators

Introduction	95
Gathering Necessary Information	95
Finding the Names of User-Written Allocators	96
Examining Parameters of User-Written Allocator Functions	98
Special Assumptions Made By User-Written Allocators about Memory	98

Creating Entries in UserAllocators.dat 99

 Modules 100

 Allocator Records 101

 Deallocator Records 104

 QuerySize Records 107

 Reallocator Records 109

 Ignore Records 113

How to Diagnose Errors in UserAllocators.dat 115

 Token Parsing Errors 115

 Semantic Errors 115

 If Your Application becomes Unstable after Changing UserAllocators.dat . . 115

Chapter 9

Deadlock Analysis

Background: Single and Multi-threaded Applications 117

 Threads 118

 Critical Sections 118

Deadlock - A Basic Definition 119

Techniques for Avoiding Deadlocks 120

Potential Deadlocks 120

 The Dining Philosophers 121

 Monitoring Synchronization Objects 122

Other Synchronization Objects 123

Additional Information 124

 MSDN References 124

 Other References 124

Index 127

Preface



- ◆ Who Should Read This Manual
- ◆ What This Manual Covers
- ◆ Conventions Used In This Manual
- ◆ Customer Assistance

This manual describes concepts and procedures to help you understand the use of your Compuware® BoundsChecker™ software.

Who Should Read This Manual

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

New users should read Chapter 2, *The BoundsChecker Solution*, to get an overview of BoundsChecker concepts and subsequent chapters to learn how to use BoundsChecker most effectively.

Users of previous versions of BoundsChecker should read the Release Notes to see how this version of BoundsChecker differs from previous versions. Also read Chapter 6, *BoundsChecker 7.2 for BoundsChecker 6 Users* for information about performing BoundsChecker 6 tasks with BoundsChecker 7.

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

Information for BoundsChecker64 Users

The information in this guide applies to both BoundsChecker *and* BoundsChecker64. Throughout, the product name BoundsChecker refers to both versions of the product. Specific differences between the products will be noted in the text.

What This Manual Covers

This manual contains the following chapters and appendixes:

- ◆ **Chapter 1, *Installing BoundsChecker*** explains how to install the BoundsChecker software on your computer.
- ◆ **Chapter 2, *The BoundsChecker Solution***, presents the philosophy behind BoundsChecker and describes new features in this release.
- ◆ **Chapter 3, *The BoundsChecker User Interface***, describes the features of the most frequently used elements of the user interface.
- ◆ **Chapter 4, *Workflow and Configuration Settings***, explains how to configure BoundsChecker to solve various problems, ranging from simple API call validation to problems encountered in complex COM applications.
- ◆ **Chapter 5, *Checking and Analyzing Programs***, describes error detection tasks you can perform with BoundsChecker and other tasks, beyond error detection.
- ◆ **Chapter 6, *BoundsChecker 7.2 for BoundsChecker 6 Users***, explains how to use BoundsChecker 7 to handle tasks performed in BoundsChecker 6.
- ◆ **Chapter 7, *Analyzing Complex Applications***, provides information to help you use BoundsChecker more effectively when checking complex applications.
- ◆ **Chapter 8, *Working with User-Written Allocators***, explains how to customize the `UserAllocators.dat` file so you can analyze your own memory allocators.
- ◆ **Chapter 9, *Deadlock Analysis***, explains deadlocks, potential deadlocks, and synchronization objects. It also lists Web addresses and books that provide more information on these topics.

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

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 BoundsChecker* 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...

Customer Assistance

The following provides information on how to access customer assistance for non-technical, as well as technical issues.

For Non-technical Issues

Customer Service is available to answer any questions you might have regarding upgrades, serial numbers and other order fulfillment needs. Customer Service is available from 8:30 am to 5:30 pm EST, Monday through Friday.

North America

- ◆ 603 578 8400
- ◆ Toll free : 800 468 6342

International

- ◆ Europe: 00 800 6863 4248
- ◆ Finland: 990 800 6863 4248
- ◆ Israel: 014 800 6863 4248

All other international customers, check Compuware Worldwide Offices for the local phone number.

For Technical Issues

Technical Support can assist you with all your technical problems, from installation to troubleshooting.

Before contacting technical support please read the relevant sections of the product documentation and the ReadMe files.

You can contact Technical Support by:

E-Mail	Include your serial number and send as many details as possible to nashua.support@compuware.com
World Wide Web	Submit issues and access our support knowledge base at http://frontline.compuware.com/nashua/
Telephone	Telephone support is available as a paid* Priority Support Service from 8:30 am to 5:30 pm EST, Monday through Friday. Have product version and serial number ready. In the U.S. and Canada, call: 888 686 3427 International customers, call: +1 603 578 8100 * Technical Support handles installation and setup issues free of charge.
Fax	Include your serial number and send as many details as possible to 603 578 8401.

Before contacting Technical Support, please obtain and record the following information:

- ◆ Product name (or edition), or service pack
- ◆ Product version
- ◆ System configuration: operating system, network configuration, amount of RAM, environment variables, and paths
- ◆ Name and version of your compiler and linker and the options you used in compiling and linking
- ◆ Problem details: settings, error messages, stack dumps, and the contents of any diagnostic windows
- ◆ If the problem is repeatable, the details of how to create the problem

Chapter 1

Installing BoundsChecker



- ◆ System Requirements
- ◆ Receiving Your BoundsChecker License File
- ◆ Installing BoundsChecker
- ◆ Your BoundsChecker64 Serial Number
- ◆ Installing BoundsChecker64
- ◆ Terminal Server Installation
- ◆ Notes About the Installation Process
- ◆ Removing an Installation

This chapter explains how to install the BoundsChecker software on your computer.

System Requirements

BoundsChecker

These are the software requirements for BoundsChecker:

- ◆ Operating system:
 - ◇ Windows 2000 including Service Pack 1 or 2
 - ◇ Windows XP Professional Edition
 - ◇ Windows Server 2003
- ◆ Visual Studio version:
 - ◇ Visual Studio 6.0 SP5 or later
 - ◇ Visual Studio .NET Service Pack 1

These are the minimum hardware requirements:

- ◆ Pentium III 500 MHz

- ◆ 128 MB of memory
- ◆ 170 MB of disk space

If you intend to use BoundsChecker integrated with Visual Studio .NET, additional memory may be required. Consult the Visual Studio .NET hardware requirements for details.

BoundsChecker is not designed to operate on Itanium-based systems using the WOW64 emulator.

BoundsChecker64

These are the software requirements for BoundsChecker64:

- ◆ Operating system:
 - ◇ Windows XP Professional Edition
 - ◇ Windows Server 2003
- ◆ Compiler:
 - ◇ C++ Compiler provided with the Microsoft Platform SDK

These are the minimum hardware requirements:

- ◆ Itanium-based processor
- ◆ 256 MB of memory minimum
- ◆ 60 MB of disk space

Additional memory may be required depending on the size of the application and BoundsChecker64 options selected.

BoundsChecker64 is not designed to analyze 32-bit applications running in the WOW64 emulator.

Receiving Your BoundsChecker License File

Note: If you are installing BoundsChecker64, see “[Your BoundsChecker64 Serial Number](#)” on page 4.

The BoundsChecker software requires a license file supplied by Compuware to execute beyond a 14-day evaluation period. A license file, `License.dat`, is provided by Compuware for each license of BoundsChecker that you purchase.

Your license file(s) will arrive by e-mail around the time that your installation kit arrives, and is delivered to the person you designated when you placed your order.

Note: If you purchased BoundsChecker from a reseller, request a license file by calling Worldwide License Management at 1-800-538-7822 or register over the Web at <http://www.compuware.com/license>.

You may elect to install the BoundsChecker software as a 14-day evaluation. At any time during or after the evaluation period, you may install the license file that you receive from Compuware by running the License Administration Utility (Program Files\Common Files\Compuware\LAU.EXE). Note that you are limited to one 14-day evaluation period.

If your license file has not arrived, please contact Worldwide License Management: 1-800-538-7822. Outside the USA and Canada, please contact your local Compuware office or agent.

If you have other products that use FlexLM licensing and you choose to store your Compuware license file in the same folder as those other license files, we recommend that you rename the Compuware License file to prevent conflicts or accidental merging or replacing of license files.

If you already have a licensed Compuware product and would like to merge the license files, contact Technical Support (1-888-686-3427).

Installing BoundsChecker

Note: If you are installing BoundsChecker64, see “[Installing BoundsChecker64](#)” on page 4.

Note: If you are installing BoundsChecker on a system running Terminal Server, see “[Terminal Server Installation](#)” on page 5.

Use the following procedure to install BoundsChecker.

- 1 If you are running a previous version of BoundsChecker, uninstall your previous version and restart your machine.
- 2 Insert the BoundsChecker 7.2 CD to start the installation wizard.
- 3 Click **Next** on the Welcome screen and read and accept the License Agreement.
- 4 Enter your name and company information on the Customer Information screen and click **Next**.
- 5 When prompted to install a license, select either a 14-day evaluation or permanent installation. For a permanent installation, enter or browse to the license file location. If you are installing BoundsChecker and are using a license server, refer to the *Compuware License Installation Guide* included in your BoundsChecker kit.

Note: If you have multiple Compuware products and are maintaining your licenses in a master license file, complete the installation then open the License Administration Utility (default location Program Files/Common Files/Compuware/LAU.exe) click **Merge** and follow the prompts to identify your license file and your master license file, as described in the *Compuware License Installation Guide* included in your BoundsChecker kit.

- 6 On the **Custom Setup** screen, browse for a directory into which to install BoundsChecker, or accept the default, and click **Next**.
- 7 Verify that you have chosen the desired installation options and click **Install**.

A progress meter displays to help you track the progress of the installation. To halt the installation at any time, click **Cancel**.

- 8 Click **Finish** on the **Installation Complete** screen.

Your BoundsChecker64 Serial Number

The BoundsChecker64 software requires that you enter a valid serial number at installation. A serial number is provided by Compuware for each copy of BoundsChecker64 that you purchase.

Your serial number will be located on the software package, license agreement, registration card, and CD case. Keep at least one copy in a safe place.

Note: If you lose all copies of your serial number, you will need to call Compuware Customer Support and verify that you purchased a copy of BoundsChecker64, and they will send you a new package.

Installing BoundsChecker64

Note: If you are installing BoundsChecker64 on a system running Terminal Server, see [“Terminal Server Installation”](#) on page 5.

Follow this procedure to install BoundsChecker64.

- 1 Insert the BoundsChecker64 CD to start the installation wizard.
- 2 Click **Next** on the Welcome screen and read and accept the License Agreement.
- 3 Enter your name, company information and BoundsChecker64 serial number on the Customer Information screen and click **Next**.

The serial number is printed on the BoundsChecker64 packaging.

- 4 On the **Custom Setup** screen, browse for a directory into which to install BoundsChecker64, or accept the default, and click **Next**.
- 5 Verify that you have chosen the desired installation options and click **Install**.
A progress meter displays to help you track the progress of the installation. To halt the installation at any time, click **Cancel**.
- 6 Click **Finish** on the Installation Complete screen.

Terminal Server Installation

When installing BoundsChecker or BoundsChecker64 across a network using Terminal Server, an additional dialog box appears. This dialog box should remain open until the installation for all users is complete. Click **Finish** to indicate to the installer that you are done.

If you are running the installations from a command window, refer to the Microsoft documentation for `change user`.

Notes About the Installation Process

During the installation, certain portions of the process continue without advancing the progress bar on the installation wizard. If you believe the install is not responding, please check the Windows task manager before you stop the installation.

Removing an Installation

To remove BoundsChecker, use **Add/Remove Programs** in the Control Panel (**Start > Settings > Control Panel**). Select **Compuware BoundsChecker** and click **Remove**.

Chapter 2

The BoundsChecker Solution



- ◆ Check Early, Check Often—The BoundsChecker Philosophy
- ◆ What's New in BoundsChecker?
- ◆ The Benefits of Using BoundsChecker

Software complexity is growing rapidly; the opportunity for problems to develop is immense. Software defects can cripple a product, cause lengthy schedule delays, and ultimately cost the engineer, the development team, and the company dearly.

Developers spend most of their debugging time tracking down and repairing elusive bugs that were introduced early in development. The quality assurance staff does the bulk of the feature testing late in the development process when the schedule allows little time.

Unfortunately, testing at this late stage usually focuses only on the outward functionality of the product. Testers run a GUI regression test bed, achieve exit criteria, and declare the product ready to ship.

All too often, the product still has many hidden bugs that conventional testing techniques failed to identify. These bugs create customer dissatisfaction and poor product reputation. Updates, patches, and other expensive, retroactive fixes cost time and money that could be spent more profitably and creatively on product improvement and new product development.

Check Early, Check Often—The BoundsChecker Philosophy

To increase software quality, developers must thoroughly test their code early in the development process. Bugs must be caught and resolved as they are introduced to avoid surprises during integration, quality assurance, beta testing, and production. Briefly stated, “check early, check often.”

Most developers need to spend the majority of their time writing code if they are to release a product on schedule. Unfortunately, few developers have the time or resources to test their products thoroughly as they develop them.

BoundsChecker provides a solution to this problem. BoundsChecker automates the crucial process of error-detection and analysis, identifies elusive bugs that are beyond the reach of traditional debugging and testing techniques, and adds little or no time to the development process.

Industry figures show that 50 percent of the development effort on an average project is spent on debugging. Regularly using BoundsChecker can significantly reduce the amount of time needed to debug your applications.

What's New in BoundsChecker

New for the BoundsChecker 7.2 Release

BoundsChecker 7.2 adds the following features to those available in the 7.0 release:

- ◆ **Deadlock Analysis:** Detects actual and potential deadlocks between threads and synchronization objects.
- ◆ **Memory and Resource Viewer:** Enables you to inspect all current memory and resource allocation in their applications at a user-specified moment in time or interval of time.
- ◆ **COM Call Logging:** Enables you to log COM objects in specific user-selected module(s).
- ◆ **.NET Call Reporting:** Enables you to log .NET method calls.
- ◆ **CLR Garbage Collection Notifications:** Provides notification that a Garbage Collection event is occurring in the .NET Common-Language Runtime (CLR) along with appropriate statistics.

BoundsChecker 7.2 also adds support for these environments:

- ◆ Windows Server 2003
- ◆ Visual Studio .NET 2003

Review of Release 7.0

BoundsChecker 7.0, a major revision of the BoundsChecker product, added these features:

- ◆ Support for Visual Studio .NET 2002
 - ◇ Visual Studio .NET IDE Integration
 - ◇ FinalCheck support for Visual C++ .NET
 - ◇ Support for native and managed executables
 - ◇ .NET Performance monitoring
 - ◇ .NET unhandled exception monitoring
 - ◇ Finalizer analysis
- ◆ Cradle-to-grave monitoring of your application
 - ◇ BoundsChecker monitors your application from the moment of creation until the final moments before the process is unloaded from memory.
 - ◇ BoundsChecker monitors all DLL loads and unloads, static constructors and destructors as well as the normal flow of your application
 - ◇ Cradle-to-grave monitoring provides complete visibility into your application.
- ◆ COM use-count graphical analysis
- ◆ Completely re-designed suppression, filtering systems
- ◆ Re-designed user interface
 - ◇ The **Summary** tab provides a high-level view of your application
 - ◇ Detailed reports based on major categories such as Memory Leaks, Other Leaks, .NET Performance and Errors
- ◆ Re-designed Modules and Files support to include or exclude portions of your application from analysis
- ◆ Re-designed settings enable you to:
 - ◇ Tune BoundsChecker to collect only the information necessary to solve a particular problem
 - ◇ Make space vs. time trade-offs
 - ◇ Make performance vs. data collection trade-offs
 - ◇ Reduce reporting of errors caused by known problems

BoundsChecker Workflow

BoundsChecker 7.2 provides a more extensive program workflow than BoundsChecker 6. This workflow gives you greater control of the amount of data collected and reported. These are the steps of the BoundsChecker workflow:

- 1 Configure BoundsChecker to collect the desired data
- 2 Run your application
- 3 View the data
- 4 If desired, save settings, suppressions, and filters for reuse

Benefits of the BoundsChecker Workflow

The BoundsChecker 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 caused by known problems, that are handled by conditional code, or have been generated in third-party code
- ◆ Create filters to hide uninteresting information in the log
- ◆ Save multiple named configurations so that settings, suppressions and filters can be reused

The Benefits of Using BoundsChecker

BoundsChecker is the most comprehensive, automated debugging solution available for C and C++ development. If you develop Windows applications, you will benefit from these BoundsChecker features:

- ◆ Comprehensive error detection
- ◆ Flexible debugging environment
- ◆ Integration with Microsoft Visual C++ and Visual Studio .NET
- ◆ Detailed view of memory and resource usage
- ◆ Advanced error analysis
- ◆ Open error-detection architecture

Comprehensive Error Detection

BoundsChecker can analyze Windows applications with both ActiveCheck™ and FinalCheck™ technologies.

ActiveCheck

BoundsChecker uses ActiveCheck technology in all error detection sessions. ActiveCheck detects errors in your program without requiring you to recompile or relink.

ActiveCheck does the following:

- ◆ Reports API validation errors at run-time
- ◆ Reports memory and resource leaks when your program terminates

- ◆ Isolates errors to the line where the memory or resource was allocated or the error was generated

Note: BoundsChecker always enables ActiveCheck technology even when FinalCheck is also selected.

When you run your program under BoundsChecker, it automatically analyzes the internals of your program as it runs. BoundsChecker monitors your program's API calls, memory allocations and deallocations, windows messages, and other significant events, then uses this data to detect errors and to provide a complete trace of your program's execution. You can even check programs that do not have source code available.

Since ActiveCheck requires no compilation or relinking overhead, you can use it daily. Use ActiveCheck throughout the software development cycle to find API validation errors, resource leaks and COM interface leaks.

Table 2-1, below, and Table 2-2 on page 12 list errors detected by ActiveCheck.

Table 2-1. Deadlock-related, .NET, and Pointer and Leak errors detected by ActiveCheck

Deadlock-related Errors	.NET Errors	Pointer and Leak Errors
<ul style="list-style-type: none"> • Deadlock • Potential deadlock • Thread deadlocked • Critical section errors • Semaphore errors • Mutex errors • Event errors • Handle errors • Resource usage and naming errors • Suspicious or questionable resource usage • Windows event errors 	<ul style="list-style-type: none"> • Finalizer errors • GC.Suppress finalize not called • Dispose attributes errors • Unhandled native exception passed to managed code 	<ul style="list-style-type: none"> • Interface leak • Memory leak • Resource leak

Table 2-2. Memory Errors and API and COM Errors Detected by ActiveCheck

API and COM Errors	Memory Errors
<ul style="list-style-type: none"> • COM interface method failure • Invalid argument • Invalid COM interface method argument • Parameter range error • Questionable use of thread • Windows function failed • Windows function not implemented 	<ul style="list-style-type: none"> • Dynamic memory overrun • Freed handle is still locked • Handle is already unlocked • Memory allocation conflict • Pointer references unlocked memory block • Stack memory overrun • Static memory overrun

FinalCheck

FinalCheck is a patented technology that BoundsChecker can use to instrument Visual C or Visual C++ applications. Instrumentation inserts diagnostic logic into your code when you compile it. With FinalCheck, BoundsChecker can pinpoint errors to the exact statement where they occurred.

Use FinalCheck for key project milestones and for detecting errors that are difficult to find.

FinalCheck is a superset of ActiveCheck that finds all the errors ActiveCheck finds, plus those listed in [Table 2-3](#).

An Example Comparing ActiveCheck and FinalCheck

If you allocate a block of memory using `new` or `malloc` and store the pointer in a local variable, BoundsChecker will record that information. If you later re-assign another value into the local variable without first either deallocating the memory block or assigning the pointer to another variable, you have just created a leak in your application.

- ◆ **Using ActiveCheck:** BoundsChecker reports that the block allocated by `malloc` or `new` was leaked and points to the line where the memory was allocated. The error is reported when your application exits.
- ◆ **Using FinalCheck:** BoundsChecker reports the location where the block was allocated and highlights the line where you assigned the new value into the last remaining variable referencing the block. The error is reported when it occurs.

Table 2-3. Errors Detected by BoundsChecker FinalCheck

Memory Errors	Pointer and Leak Errors
<ul style="list-style-type: none">• Reading overflows buffer• Reading uninitialized memory• Writing overflows buffer	<ul style="list-style-type: none">• Array index out of range• Assigning pointer out of range• Expression uses dangling pointer• Expression uses unrelated pointers• Function pointer is not a function• Memory leaked due to free• Leak due to leak• Memory leaked due to reassignment• Memory leaked leaving scope• Returning pointer to local variable• Leak due to unwind• Leak due to module unload• Leak due to thread ending

Flexible Debugging Environment

BoundsChecker provides a flexible debugging environment which you can run:

- ◆ As an integrated part of Microsoft Visual Studio 6, Visual Studio .NET 2002 or 2003
- ◆ As an independent application
- ◆ From a DOS command line

When you use BoundsChecker as part of the Visual Studio .NET or Visual C++ environments, all of the BoundsChecker features can be accessed from within the IDE. You can configure BoundsChecker settings, check your program, and review detected errors.

When you use BoundsChecker as an independent application, it runs completely outside of the Microsoft IDE.

You can run BoundsChecker from a DOS command line, using `bc7.exe` or `bc7.com`.

- ◆ `bc7.exe` starts the UI for BoundsChecker standalone.
- ◆ `bc7.com` is a small console program that spawns `bc7.exe` and waits for it to complete.

The difference between `bc7.exe` and `bc7.com` is important for batch scripts. Invoking `bc7.exe` directly will start BoundsChecker and continue

on to the next command without waiting for `bc7.exe` to complete. If the next step in the script is to check for a result, it won't be available.

Note: If you type only `bc7`, the OS will choose `bc7.com` instead of `bc7.exe`.

Integration with the Visual C++ Debugger

BoundsChecker automatically integrates with the debuggers in Visual Studio .NET and Visual C++ 6.

The Program Error Detected dialog box includes a **Debug** button. When you click **Debug**, BoundsChecker drops into the IDE debugger, at the line of code that generated the error.

When you review logged errors in the **Details** pane of the BoundsChecker window, right-click an event and choose **Edit Source**. This opens the source file to the line of code that generated the error.

Detailed View of Memory and Resource Usage

The Memory and Resource Viewer dialog box, available with both ActiveCheck and FinalCheck analysis, enables you to better understand how your application uses memory and resources.

With the Memory and Resource Viewer dialog box, you can view allocated memory and resources at any time in your application. You can also use this feature to calculate memory and resource usage between two points in your application.

Advanced Error Analysis

Windows is an event-driven environment in which much of your program is executed in response to Windows messages and other events. BoundsChecker intercepts control when events occur and logs them, so you can use them to see a complete history of events that led to a problem. BoundsChecker logs these events:

- ◆ Windows messages and hooks. These events show how your program reacted to Windows messages.
- ◆ API calls and API returns along with argument information. These events define the order in which procedures are executed in your program.
- ◆ Output debug string messages from the program you are checking.
- ◆ Error messages, including all information BoundsChecker recorded in the event log.
- ◆ Notifications of deadlocks and potential deadlocks.
- ◆ Transitions from managed to unmanaged code within your mixed native and managed .NET applications.

- ◆ .NET method calls and returns, along with argument information. These events define the order in which .NET methods are executed in your program.
- ◆ .NET garbage collections. Monitoring these events can help track down performance problems in your application caused by frequent requests to collect unused memory from the heap.

Open Error Detection Architecture

When developing with C and C++, you can extend the COM functions BoundsChecker validates to include APIs you create. When you extend BoundsChecker to test your COM objects, BoundsChecker automatically validates their parameters, validates their return values, and logs their trace data, so you can analyze them.

You can also extend BoundsChecker by adding lines to `UserAllocators.dat` to describe user-written allocators in your application. [Chapter 8, “Working with User-Written Allocators”](#) provides more information about user-written allocators.

Chapter 3

The BoundsChecker User Interface



- ◆ BoundsChecker Main Window
- ◆ Settings Dialog Box
- ◆ Program Error Detected Dialog Box
- ◆ Suppression and Filtering Dialog Boxes
- ◆ IDE Integration

This chapter describes key elements of the BoundsChecker user interface.

BoundsChecker Main Window

The BoundsChecker main window is divided into three sections, called panes:

- ◆ Results pane
- ◆ Details pane
- ◆ Source pane

See [Figure 3-1](#) on page 18.

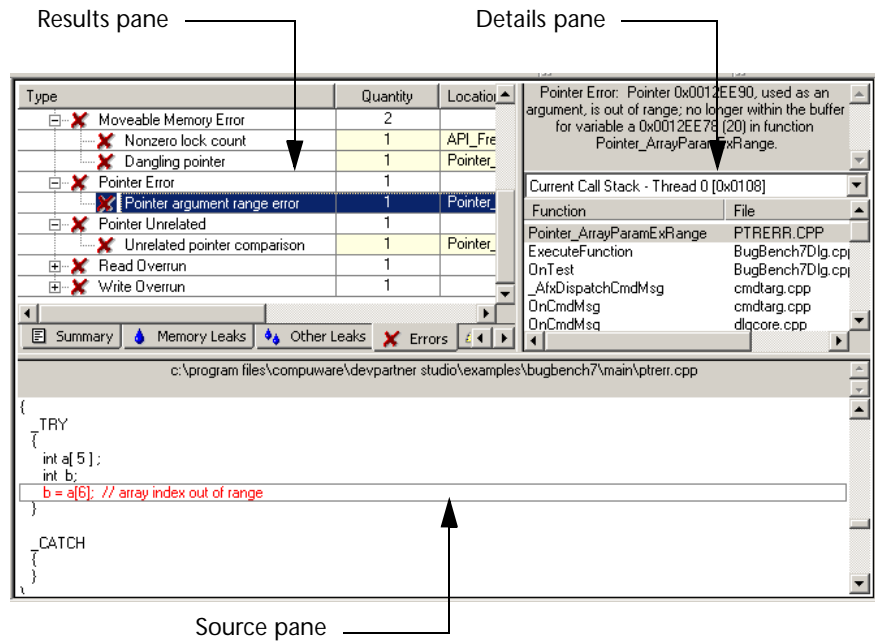


Figure 3-1. The BoundsChecker main window

Results Pane

The upper left section is the **Results** pane. The **Results** pane uses a series of tabs that enable you to navigate through the various types of information.

The **Summary** tab (first tab on the left of the pane) provides an overview of all errors and events detected in your BoundsChecker session. Double-click an event in the **Summary** tab to navigate to the event in the corresponding tab.

The rollout tabs include **Memory Leaks**, **Other Leaks**, **Errors**, **.NET Performance**, **Modules**, and **Transcript**.

- ◆ To sort the data on these tabs, click a column header (such as **Type**, **Quantity**, or **Deallocator** in the **Other Leaks** tab).
- ◆ For additional information about an event on a tab, right-click the event and choose **Explain**.
- ◆ To see an event in the context of other events in your application, right-click the event and choose **Locate in Transcript** from the shortcut menu. The **Transcript** tab provides a chronological list of all events that occurred within your application.

Details Pane

The upper right section is the **Details** pane. The **Details** pane consists of one or more sections, depending on the currently selected event. The top section describes the error or event in detail. The lower section(s) can display call stacks, P/Invoke use-count graphs, COM use-counts, and so on.

When you select an event, the **Details** pane displays the current call stack. If more than one call stack is accessible, the **Details** pane displays a drop-down list; use it to select a different call stack to view. You can select **Current Call Stack**, **Call Stack At Allocation** or **Call Stack At Deallocation**.

Source Pane

The bottom section is called the **Source** pane. It displays the source file associated with the call stack displayed in the **Details** pane. The source code changes when you select a different call stack in the **Details** pane.

Settings Dialog Box

The BoundsChecker settings enable you to:

- ◆ Select only the types of data collection needed for a particular problem
- ◆ Enable or disable portions of each major type of data collection
- ◆ Control what portions of your program are analyzed
- ◆ Use the default BoundsChecker settings to find the most common errors with the minimum impact on performance

The **Settings** dialog box has a tree control that shows the major settings categories. When you select a category, the dialog box displays the detailed settings for the category.

Note: In Visual Studio .NET, the term **Options** is used instead of **Settings**. See [“Using BoundsChecker with Visual Studio .NET”](#) on page 31.

The same tree control and settings dialog boxes are used in the BoundsChecker standalone application, in the Visual C++ IDE, and in the Visual Studio .NET IDE.

All groups of settings follow the same basic structure. You can enable or disable major types of data collection by selecting the top-level check box in the dialog box.

There are other settings under each top-level check box that further define how BoundsChecker will analyze your application. Change the settings to customize your error detection process.

For example, you can make trade-offs between detecting a broad or narrow range of errors.

- ◆ **Broad range:** Many data types; many related settings selected
 - ◇ Detects more errors
 - ◇ Has potential for more false positives
 - ◇ Reduces performance (due to larger number of errors detected)
 - ◇ Creates larger log files
- ◆ **Limited range:** Few data types, few related settings selected
 - ◇ Provides a narrow focus on a particular function
 - ◇ Detects fewer errors
 - ◇ Can miss relevant errors
 - ◇ Has a greater chance of seeing only those errors pertaining to the problem at hand
 - ◇ Provides faster performance
 - ◇ Creates smaller log files

An Example: Call Validation

For example, to activate **Call Validation**, select **Enable call validation**. (See [Figure 3-2](#) on page 21.) This makes all the Call Validation controls active. (By default, **Call Validation** is turned off.)

When you select **Call validation**, BoundsChecker validates over 5,000 Windows API calls. BoundsChecker will check for a large number of events, including (but not limited to) the following:

- ◆ Handle and pointer errors
- ◆ Flags
- ◆ Range checks
- ◆ API failures
- ◆ Invalid structure sizes
- ◆ Memory access failures

If you find flag checking or range checking generates unwanted errors that do not apply to the problem you are solving, clear **Flag, range and enumeration arguments**. **Call Validation** will continue checking return

values and, more importantly, handles and pointers passed to or from Windows calls.

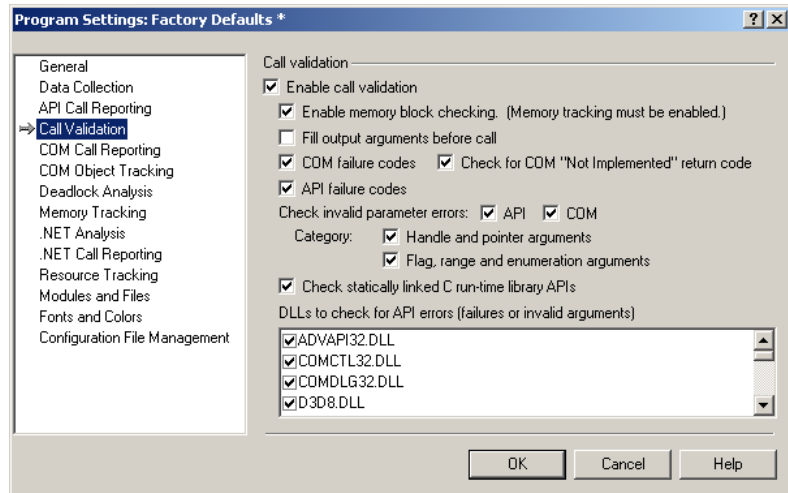


Figure 3-2. The Settings or Options dialog box showing the controls under Call validation.

Enable Memory Block Checking

By selecting **Enable memory block checking**, **Call Validation** will perform a more detailed analysis of all calls to the C run-time library and a number of other calls. By default, this setting is inactive. Selecting **Enable memory block checking** will decrease overall performance but may prove useful when diagnosing hard-to-find errors.

Using Other Settings Categories

The online help provides detailed information about each settings category and also describes some of the trade-offs associated with specific tools within a category.

Program Error Detected Dialog Box

The **Program Error Detected** dialog box (see [Figure 3-3](#) on page 22) displays the errors detected within your application.

Note: In BoundsChecker 6, the equivalent feature was called the BoundsChecker popup.

The top of the **Program Error Detected** dialog box describes the error detected. Below this will be one or more tabs. Each tab corresponds to a

call stack corresponding to a location within your application. Review the error along with the source information provided to help you locate the source of the problem and correct it.

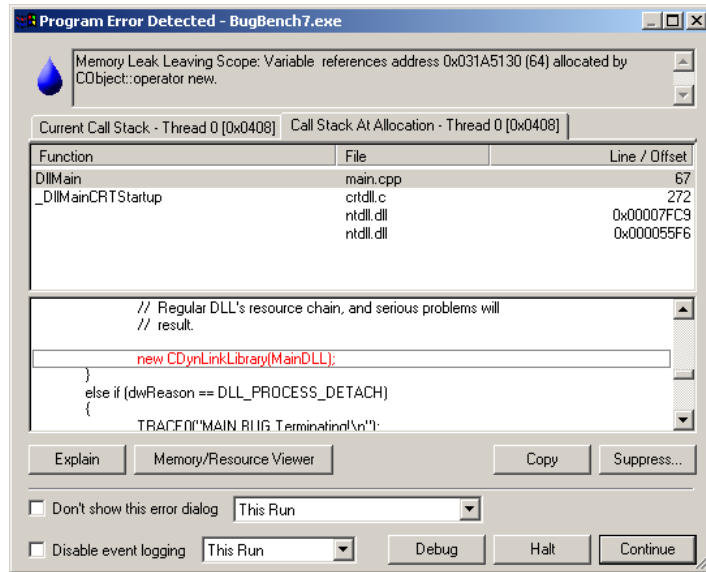


Figure 3-3. The Program Error Detected dialog box

Buttons on the Program Error Detected Dialog Box

Explain, **Memory and Resource Viewer**, **Submit**, **Debug**, **Copy** and **Suppress** buttons appear on the **Program Error Detected** dialog box.

Explain

Click **Explain** to obtain detailed explanations of each error along with sample code and a list of solutions to correct the problem.

Memory and Resource Viewer

Click **Memory/Resource Viewer** to view a detailed accounting of memory and resources that have not been freed. For more information, see [“The Memory and Resource Viewer dialog box.”](#) on page 25.

Submit

Submit is available if TrackRecord is part of your DevPartner installation. Click **Submit** to open TrackRecord to either a new defect or new task page.

Debug

Debug appears at the bottom of the dialog box when you are working from within Visual Studio .NET or Visual C++.

Click **Debug** to drop into the Visual Studio debugger. You can then examine variables or modify the source.

Copy

Click **Copy** to transfer the contents of all windows and tabs (except the **Source** pane) to the clipboard. You can then paste this information into other applications.

Suppress

Click **Suppress** to open a dialog box that enables you to suppress the current error. For more specific instructions, click **Help** in the suppression dialog box.

Hints for Using the Program Error Detected Dialog Box

Here are two hints that may help you check your applications more effectively.

Large Applications

If you are working with a large application and you are concerned only with errors in a particular portion of the program:

- 1 Start your error detection session.
- 2 Select **Don't show this error dialog** to prevent BoundsChecker from displaying the **Program Error Detected** dialog box.
- 3 Select **Disable event logging** to stop logging events into the BoundsChecker log.
- 4 When your application reaches the portion that you need to check, turn event logging on.
- 5 Exercise the portion of your application that you need to check, then turn event logging off and terminate your application.

Tip: Click the **Log Events** button on the **BoundsChecker** toolbar to turn event logging on or off.

See the online help for further details.

Suppressing Errors as You Check

Even after you fine-tune your settings, BoundsChecker may still flag an event that is not an error or is an error that is properly handled elsewhere

in your application. In this case, click **Suppress**. This opens the **Suppression** dialog which enables you to specify errors that you want BoundsChecker to skip over.

Memory and Resource Viewer Dialog Box

The **Memory and Resource Viewer** dialog box enables you to closely analyze memory and resource allocations that have not been freed.

For example, most analysis tools can determine that memory or resources have been leaked only at the end of an application. This information tells you little about usage during the middle of a program's execution. The BoundsChecker **Memory and Resource Viewer** can provide a snapshot taken at any point in a program's execution. You can also set a mark with the **Memory and Resource Viewer** so you can examine memory and resource allocations during a specific segment of your application's execution.

These capabilities can be especially useful in situations like these:

- ◆ 24/7 servers applications may never end during regular use
- ◆ An application may hang from resource exhaustion
- ◆ An application may consume large amounts of memory that is automatically cleaned up at program termination
- ◆ A large volume of uninteresting leaks may dilute the signal at end of process

The Memory and Resource Viewer User Interface

To access the Memory and Resource Viewer dialog box, click **Memory/Resource Viewer** in the **Program Error Detected** dialog box. This button is active after you select **Enable memory and resource viewer** under **General** settings or options.

Other ways to access the Memory and Resource Viewer:

- ◆ Automatically displayed at program termination (when enabled)
- ◆ Use the Memory and Resource Viewer programmable API
- ◆ From a Visual Studio break point:
 - ◇ Select Memory Resource Viewer from the BoundsChecker menu in Visual Studio 6, or;
 - ◇ Select DevPartner Memory / Resource Viewer from the Visual Studio .NET Debug menu.

The Memory and Resource Viewer dialog box is made up of four panes:

- ◆ **Memory contents pane:** Displays the contents of memory blocks in a variety of formats. This is similar to the Memory window in the Visual Studio debugger. Not available for resources.
- ◆ **Details pane:** Includes separate Memory, Resources and Summary tabs. Displays details about each memory and resource allocation.
- ◆ **Stack pane:** Displays callstack information for entries in the Memory tab; displays a description and callstack information for entries in the Resources tab.
- ◆ **Source pane:** When available, the source code corresponding to a callstack entry is displayed here.

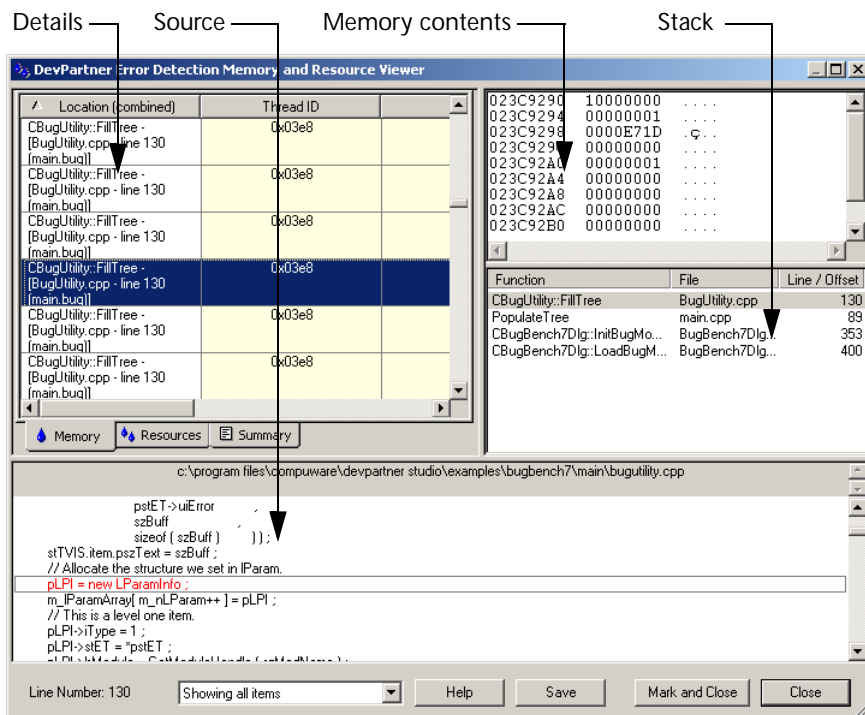


Figure 3-4. The Memory and Resource Viewer dialog box.

Saving Memory and Resource Viewer Contents

Click **Save** to record the current contents of the Memory and Resource Viewer dialog box as a tab-separated text file that you can review later using Microsoft Excel or a text editor.

Mark and Close

Click **Mark and Close** to set a reference point for recording memory and resource data. This enables you to compare memory and resource allocations before and after the event where you marked the reference point.

Suppression and Filtering Dialog Boxes

Suppression and filtering enable you to reduce the data collected or displayed. The intent of either method is to limit the data to a manageable subset for analysis.

There are two ways to hide an error:

- ◆ Suppression
- ◆ Filtering

Suppressing Errors

BoundsChecker will skip over any future occurrences of errors you suppress. Suppressed errors are not recorded in the log and they are not displayed in the **Program Error Detected** dialog box.

Filtering Errors

Filtering hides events already recorded in a BoundsChecker log.

Select errors that you want to filter. BoundsChecker finds these errors but either hides them from view in the **Results** pane or displays them with the appearance specified under **Fonts and Colors**.

If you later cancel a filtering instruction, the filtered errors will appear in the Results pane.

Suppression and Filtering User Interface

Click the filter icon on the toolbar to turn filtering on or off.

The BoundsChecker **Suppression** and **Filter** dialog boxes provide numerous controls over the suppression and filtering processes.

For example, you can suppress call validation errors from `FindResourceA` in `Kernel32` or for all calls in `Kernel32`. After you make this selection, you can apply it to a variety of different selection criteria within your application. BoundsChecker defaults to the least restrictive option. See [Figure 3-5](#) on page 27.

You can make the following additional choices for suppressions and filters:

- ◆ Enter a comment to describe why a given suppression or filter was created.
- ◆ Choose to apply the suppression or filter to the current run or future runs.
- ◆ Create suppression or filter files to store the suppression or filter instructions.

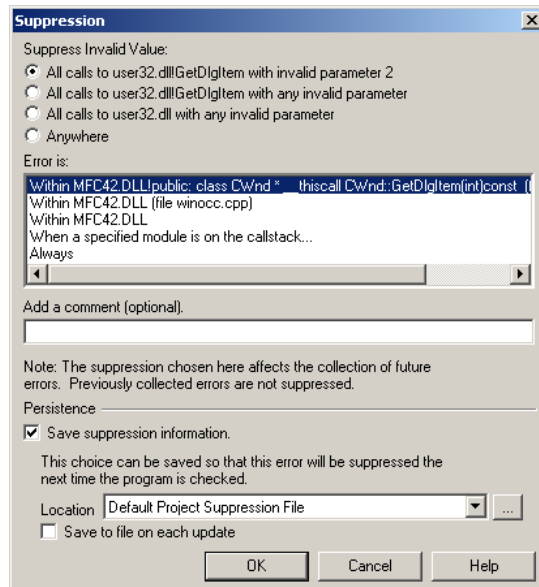


Figure 3-5. The **Suppression** dialog box. The **Filtering** dialog box uses the same design.

Creating and Saving Suppression and Filter Files

You can create multiple suppression files. You can use this feature to create additional suppression libraries for the various DLLs that make up a large application. By doing so, suppressions can easily be reused or shared among members of a development team.

See the online documentation for detailed information on each field in the **Suppression** and **Filter** dialog boxes.

IDE Integration

BoundsChecker provides integrated development environment support for Visual Studio 6 and Visual Studio .NET.

Using BoundsChecker with Visual Studio 6

BoundsChecker integrates into the Visual Studio menus, toolbars, debugger, and build system. See [Figure 3-6](#).

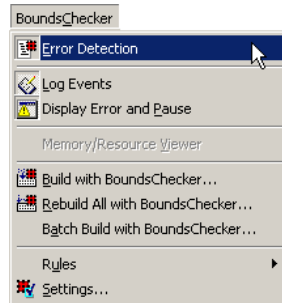


Figure 3-6. The BoundsChecker menu in Visual C++ 6.0.

Follow these steps to run BoundsChecker while debugging your application with Visual Studio:

- 1 Create or open an existing project.
- 2 Build your application.

To analyze your program with ActiveCheck, create an executable with debug symbols and, preferably, with optimizations disabled. To analyze your program with FinalCheck, instrument your program:

 - ◇ Select the debug configuration.
 - ◇ Select either **BoundsChecker > Build with BoundsChecker** or **BoundsChecker > Rebuild All with BoundsChecker**.

This action will re-compile your application. The Build tab of the Output window will display messages stating that BoundsChecker is instrumenting your application.
- 3 Choose the BoundsChecker settings to use for the analysis:
 - ◇ Select **BoundsChecker > Settings**, then make selections in the **Settings** dialog box.

Example: To validate Windows API calls made by your application:

 - Select **Call Validation** from the tree view in the settings dialog box.

- Select **Enable call validation**.

For most applications, the default settings produce acceptable results.

- 4 Press F5 to start debugging.
- 5 As your program executes, BoundsChecker will display errors in the **Program Error Detected** dialog box. Review the errors, then click **Debug** or **Continue**.
- 6 After your program terminates, review errors in the **Results** pane of the BoundsChecker window. To correct an error, right-click the event in the **Details** pane and choose **Edit Source**.
- 7 Correct your errors, recompile your application and continue testing.

BoundsChecker Integration with Visual Studio .NET

BoundsChecker is tightly integrated into Visual Studio .NET menus, toolbars, the **Solution Explorer**, the debugger, and the build system.

- ◆ Choose **Tools > Options** to open the **Options** dialog box, then select **BoundsChecker** to see the BoundsChecker options. See [Figure 3-7](#).

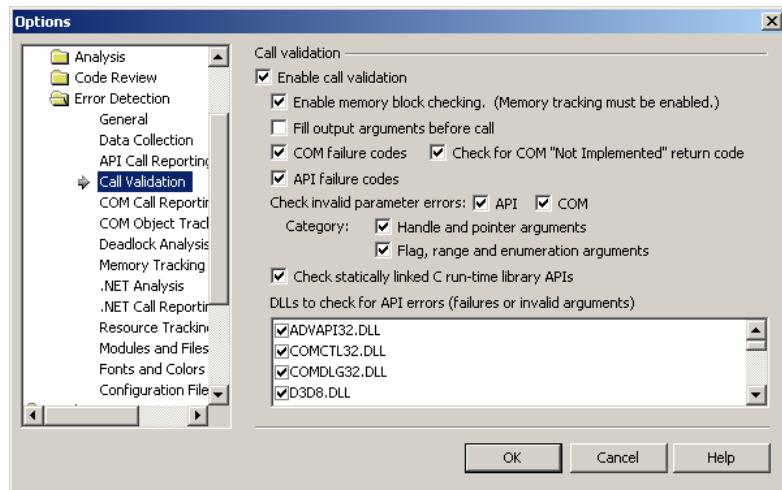


Figure 3-7. The Options dialog box.

- ◆ Choose **Tools > BoundsChecker** to access the BoundsChecker menu. Commands on this menu control FinalCheck instrumentation. There are also commands that control BoundsChecker analysis, BoundsChecker logging and the **Program Error Detected** dialog box. See [Figure 3-8](#).

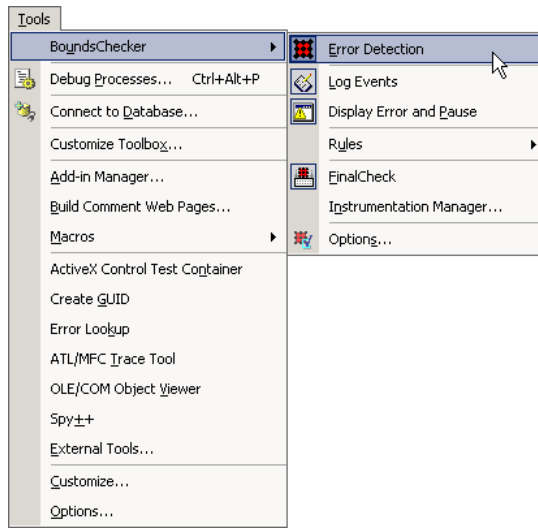


Figure 3-8. The BoundsChecker menu in Visual Studio .NET.

- ◆ You can use the BoundsChecker toolbar (see Figure 3-9) to toggle various commonly used BoundsChecker functions such as integrated debugging, automatic FinalCheck instrumentation, view filters, the BoundsChecker log and display of the **Program Error Detected** dialog box.



Figure 3-9. The BoundsChecker toolbar.

- ◆ BoundsChecker is tightly integrated into the Visual Studio .NET environment and also into the Solution Explorer.

- ◆ BoundsChecker will automatically register BoundsChecker sessions with the **Solution Explorer**. See [Figure 3-10](#).

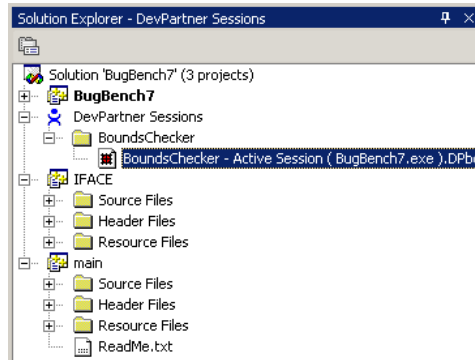


Figure 3-10. BoundsChecker sessions are automatically registered with the **Solution Explorer**.

Using BoundsChecker with Visual Studio .NET

Follow these steps to use BoundsChecker with Visual Studio .NET:

- 1 Create or open an existing solution.
- 2 Build your application.

To analyze your program with ActiveCheck, create an executable with these attributes:

- ◇ A Debug build, preferably with optimizations disabled
- ◇ With debug symbols
- ◇ With Visual C++ basic run-time checking disabled, do one of the following:
 - Remove /RTC from the CL command line.
 - Set Basic runtime checks in the C/C++ Code Generation settings to Default.

To analyze your native C/C++ program with FinalCheck, re-compile your application. You can instrument your program by doing one of the following:

- ◇ Enable FinalCheck instrumentation using one of the following procedures:

Tip: After the build has completed, you may want to disable **FinalCheck** instrumentation by clicking on **FinalCheck** in the **BoundsChecker** toolbar.

- Select **FinalCheck** from the **BoundsChecker** toolbar
- Choose **Tools > BoundsChecker > FinalCheck**.

◇ Build your application with Visual Studio .NET

If you have solutions with multiple projects, you may want to use the **BoundsChecker Instrumentation Manager** to select which projects should be instrumented. The **Instrumentation Manager** can be found on the **Tools > BoundsChecker** menu.

3 Choose the **BoundsChecker** options you want to use.

- ◇ Choose **Tools > Options**, then review the **BoundsChecker** options.

For example, to validate Windows API calls made by your application, select **Call Validation**. Under **Call Validation**, select **Enable call validation**. For most applications, the default settings should produce acceptable results.

4 Press F5 to start debugging.

5 As your program executes, **BoundsChecker** will display any errors encountered. Review the errors, then click either **Debug** or **Continue**.

6 After your program terminates, review errors in the **Details** pane of the **BoundsChecker** window. To correct an error, right-click an event and choose **Edit Source**.

7 Correct your errors, recompile your application and continue testing.

Chapter 4

Workflow and Configuration Settings



- ◆ BoundsChecker Workflow
- ◆ Customizing the BoundsChecker Settings

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

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

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

BoundsChecker Workflow

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

Here are the four steps of the BoundsChecker workflow:

- 1 Configure BoundsChecker 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 BoundsChecker Workflow

The BoundsChecker 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

BoundsChecker provides defaults for each step in the workflow process. This means you can use BoundsChecker as-is or you can change the settings to customize the way that BoundsChecker analyzes your application.

Saving Error Detection Configurations

You can save an error detection configuration - a specific combination of settings (Visual C++ and standalone versions) or options (Visual Studio .NET) - 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.

Customizing the BoundsChecker Settings

The BoundsChecker 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. are recorded along with the event or error
- ◆ Control the look and feel of the BoundsChecker user interface. This includes changing fonts, colors, highlighting, or whether the **Program Error Detected** dialog box is displayed
- ◆ Save and restore BoundsChecker settings created previously

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

The BoundsChecker 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 Call Reporting
- ◆ .NET Analysis
- ◆ Resource Tracking
- ◆ Modules and Files
- ◆ Fonts and Colors
- ◆ Configuration File Management

General

Use the checkboxes under **General** settings to control following:

- ◆ Event logging
- ◆ The **Program Error Detected** dialog box - to display on each error or not

- ◆ Whether or not to display a prompt save program results when the target application exits
- ◆ Whether or not to display the **Memory Resource Viewer** dialog box when the target application exits or not
- ◆ The location of source files
- ◆ The working directory and symbol files
- ◆ Specify command line arguments (available only when you use BoundsChecker 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
- ◆ Whether BoundsChecker should automatically learn about new COM objects (dynamic NLB generation)

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 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 libraries (for example, select GDI32 to log graphics calls).

Call Validation

Use **Call Validation** settings to control whether BoundsChecker validates Windows API parameter and return values. By default, BoundsChecker does not validate parameters. This is a change from BoundsChecker 6.

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

BoundsChecker 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.

BoundsChecker 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 libraries. 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, BoundsChecker 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**.

To monitor your own COM classes, enable **Generate NLB files dynamically** in the **General** settings so that BoundsChecker can automatically learn each of the methods associated with your COM object.

COM Object Tracking

BoundsChecker can monitor COM usage within an application and will report on any classes that are leaking interfaces. If an interface leak is detected, BoundsChecker will provide a COM use-count graph showing every `AddRef` and `Release` pair 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, BoundsChecker 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 Tracker** 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.

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 BoundsChecker 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 8, “Working with User-Written Allocators”](#) and review the documentation (in the form of comments) in the following file:

```
C:\Program Files\Compuware\BoundsChecker  
\Data\UserAllocators.dat
```

.NET Call Reporting

Use **.NET Call Reporting** settings to control the .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.

.NET Analysis

BoundsChecker supports mixed native and managed applications. If you are working in mixed environments, you can select **Enable .NET runtime analysis**. BoundsChecker 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 BoundsChecker .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 **Enable resource tracking**.

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 BoundsChecker analyzers

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

For more information, see [“Using Modules and Files Settings”](#) on page 72.

Fonts and Colors

Use the **Fonts and Colors** settings to change the font, color and emphasis of each item in the BoundsChecker user interface.

Configuration File Management

Use **Configuration File Management** to create multiple settings files for each project. You can then use these settings 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 interface leak detection 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.

Chapter 5

Checking and Analyzing Programs



- ◆ BoundsChecker Error Detection Tasks
- ◆ Expanded Uses for BoundsChecker

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

BoundsChecker Error Detection Tasks

BoundsChecker 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
- ◆ Searching for application deadlocks

Finding Leaks

BoundsChecker excels at finding memory, resource and interface leaks. By default, BoundsChecker 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.

BoundsChecker provides two methods of detecting memory leaks, ActiveCheck and FinalCheck. ActiveCheck will search for memory leaks in any Windows application. Leaks will be reported when your application shuts down. FinalCheck will comprehensively report

memory leaks at run-time as they occur in your application. Examples include when a local variable goes out of scope or when the last pointer to a block of memory is re-assigned.

Finding Pointer and Memory Errors

BoundsChecker can search for pointer and memory errors using both ActiveCheck and FinalCheck technology. In ActiveCheck mode, BoundsChecker 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 BoundsChecker.

If you re-compile your program using FinalCheck, BoundsChecker 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

BoundsChecker 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)

BoundsChecker 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 Legacy Code in .NET Applications

BoundsChecker 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 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 native 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 transitioning to managed code.

Validating Win32 API Calls

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

BoundsChecker introduces the following new **Call Validation** features:

- ◆ Ability to selectively choose what types of Windows calls to monitor.
- ◆ Ability to selectively disable various types of validation such as flag, range, and enumeration checking.

With these features, you can configure BoundsChecker 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

BoundsChecker 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 BoundsChecker

Beyond error detection tasks, BoundsChecker can be used as:

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

Understanding Complex Applications

BoundsChecker 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 BoundsChecker 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. A side effect of interface leaks can include memory and resource leaks caused by objects not properly freeing up system resources.

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 BoundsChecker 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 BoundsChecker, you can identify these potential deadlocks before occur running your application in a production environment.

Modules and Files

Complex applications are often developed across multiple organizations and include libraries purchased from outside vendors. By default, BoundsChecker will report errors in any non-system DLL. Use the **Modules and Files** settings to restrict BoundsChecker 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.

The Modules Tab

The BoundsChecker **Modules** tab provides a view into your program. This view accurately 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 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?

Viewing and Sorting in the *Results* Pane

BoundsChecker provides a wide variety of ways to view the data collected on your application. Initially, BoundsChecker 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

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

Data Collection

Select **Generate NLB files dynamically** so that BoundsChecker automatically learns about new COM interfaces and methods by extracting type libraries associated with COM components. BoundsChecker uses NLB files to provide information about your application.

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 BoundsChecker 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 BoundsChecker performance.

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

COM Call Reporting

Select **Collect COM method calls and returns** 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 interfaces** checkbox.

.NET Analysis

When writing mixed native and managed code applications, use the **.NET Analysis** features 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 **Detection threshold** value. When monitoring calls from managed to native code, select a sufficiently high detection threshold value and use the **Modules and Files** settings to reduce unwanted information.

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

Function Groups to Turn Off for Reverse Engineering

BoundsChecker provides tools to monitor many types of leaks and errors in Windows applications. However, during reverse engineering sessions it may be desirable to turn off the BoundsChecker error and leak detection logic. Follow these steps to disable these features in the **Program Settings** dialog box (in BoundsChecker standalone and in the Visual C++ 6 IDE) or the **Options** dialog box (in the Visual Studio .NET 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, BoundsChecker 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 BoundsChecker from monitoring business-critical portions of your application (such as serial number or password checking), you can selectively disable BoundsChecker logging by making calls to the BoundsChecker callable interface at runtime. Please refer to the comments in the following file for details:

C:\Program Files\Compuware\BoundsChecker\ErptApi\NmApiLib.h

Stress Testing

A side effect of running BoundsChecker 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 typically zero. BoundsChecker 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 BoundsChecker. If this occurs, instrument your application with `FinalCheck` and check it again with BoundsChecker 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 8, “Working with User-Written Allocators”](#) for more information.

Pool Poisoning on Free

BoundsChecker writes a known pattern on dynamically allocated memory after it has been deallocated. By doing so, applications that attempt to reference deallocated structures will 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 BoundsChecker 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 8, “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.

BoundsChecker monitors all aspects of program flow and places your application under a heavy CPU and memory workload. At the same time, BoundsChecker 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 BoundsChecker will cause 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 BoundsChecker, you may be able to find problems in the development process and correct them before going into production.

Run your application under BoundsChecker 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 BoundsChecker 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 BoundsChecker has been designed to find and report on dangling pointers.

Undetected dangling pointers allow 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 6

BoundsChecker 7.2 for BoundsChecker 6 Users



- ◆ Changes to the User Interface
- ◆ Workflow Changes
- ◆ Error Detection Settings
- ◆ BoundsChecker 6 Cross Reference
- ◆ Other BoundsChecker 7.2 Capabilities

BoundsChecker 6 provided a simple user interface and limited configuration options. BoundsChecker 7.2 provides an improved user interface and a greater number of configuration options.

This chapter describes how to map BoundsChecker 6 functionality into BoundsChecker 7.2.

Changes to the User Interface

If you used BoundsChecker 6, you will notice numerous changes to the user interface. Some of the key differences are listed here.

The BoundsChecker 7.2 main window includes a **Summary** pane, new tabs and additional rollup options.

The BoundsChecker 6 popup has been replaced by the **Program Error Detected** dialog box. Previous BoundsChecker 6 features have been retained and new features, such as the **Copy** button and stack tabs have been added. BoundsChecker 7.2 provides additional options to control the BoundsChecker log and the frequency of the **Program Error Detected** dialog box (previous versions had a simple on/off switch).

The BoundsChecker 7.2 settings provide more flexibility and enable you to save named configurations for common scenarios. Previous versions of BoundsChecker provided only **Normal**, **Maximum** and **Custom** settings.

The BoundsChecker command line has been changed to BC7 instead of BC. Also, there are two versions of BC7. The first is BC7.exe, which invokes the graphical user interface. The second is BC7.com, which invokes the command line interface and will not terminate the command until the BC7 executable has completed. The BC7.com executable is the preferable solution for batch script operation. Omitting the .EXE file extension in your scripts will automatically invoke the BC7.com executable.

The BoundsChecker dumper utility is included and has been renamed bcdump.exe. Output from the new dumper utility has changed significantly.

Workflow Changes

BoundsChecker 6 followed a four-step workflow:

- 1 Open the executable
- 2 Select **Normal**, **Maximum** or **Custom**
- 3 Run your application
- 4 View the data

BoundsChecker 7.2 has a similar workflow but provides greater flexibility in steps 2 through 4. If you do not require advanced configurations, you should be able to start using BoundsChecker with only a few changes. If you use advanced configurations, see [Chapter 4, “Workflow and Configuration Settings”](#) for advanced setting information.

Error Detection Settings

The BoundsChecker 6 settings provided a limited solution to error detection. With the expanded capabilities of the new workflow, you can fine-tune BoundsChecker 7.2 settings. For example, you can save settings configurations for: different phases of your development process; different types of applications; or for different views of the same application.

To get the most out of BoundsChecker 7.2, review the rest of this section to understand how to go beyond the BoundsChecker 6 model of **Normal**, **Maximum** and **Custom**.

BoundsChecker 6 Normal and Maximum mode

BoundsChecker 7.2 can be configured to collect similar information to BoundsChecker 6 by making the following changes to the BoundsChecker 7.2 settings:

Normal

Start with the BoundsChecker 7.2 defaults and make the following changes:

- ◆ General
 - ◇ Enable memory and resource viewer: on
 - ◇ Show memory and resource viewer when application ends: off
- ◆ Data Collection
 - ◇ Maximum Call Stack on allocation: 4
 - ◇ Maximum Call Stack on error: 19 (selectable from 1-19 in version 6)
 - ◇ Call parameter encoding depth: 0 (not recommended)
 - ◇ Generate NLB files dynamically: Off

Note: Setting the Encoding depth to 0 most closely matches BoundsChecker 6. However, it does not report deeper levels of detail referenced by pointers. If you use the BoundsChecker 7.2 default setting, 1, BoundsChecker will gather more information.

- ◆ Call Validation
 - ◇ Enable call validation: On
 - ◇ COM failure codes: Off
 - ◇ BoundsChecker 7.2 validates many more Windows API functions than BoundsChecker 6. Disable unwanted libraries at the DLL level.

Note: BoundsChecker 6 did not provide COM use-count graphing.

- ◆ COM Object Tracking
 - ◇ Select All COM classes.
- ◆ Memory Tracking
 - ◇ Guard byte pattern: 0xBF
- ◆ Resource Tracking
 - ◇ BoundsChecker 7.2 detects many new resource types. Disable new resource types by deallocator type.
- ◆ Deadlock Analysis
 - ◇ Enable deadlock analysis: off

Note: BoundsChecker 6 did not provide deadlock analysis.

- ◆ .NET Call Reporting
 - ◇ Collect .NET method events: off
- ◆ .NET Analysis
 - ◇ Enable .NET analysis: off

Note: BoundsChecker 6 did not provide support for .NET.

Maximum

Start with the BoundsChecker 7.2 defaults and make the following changes:

- ◆ General
 - ◇ Enable memory and resource viewer: on
 - ◇ Show memory and resource viewer when application ends: off
- ◆ Data Collection
 - ◇ Maximum Call Stack on allocation: 19 (selectable from 1-19 in version 6)
 - ◇ Maximum Call Stack on error: 19 (selectable from 1-19 in version 6)
 - ◇ Call parameter encoding depth: 1 (but not as detailed as version 6)
 - ◇ Generate NLB files dynamically: Off
- ◆ Call Validation
 - ◇ Enable call validation: On
 - ◇ COM failure codes: Off
 - ◇ BoundsChecker 7.2 validates many more Windows API functions than BoundsChecker 6. Disable unwanted libraries at the DLL level.
- ◆ COM Object Tracking
 - ◇ Select All COM classes.

Note: BoundsChecker 6 did not provide COM use-count graphing.

- ◆ Memory Tracking
 - ◇ Guard byte count: 8 bytes
 - ◇ Guard byte pattern: 0xBF
- ◆ Resource Tracking
 - ◇ BoundsChecker 7.2 detects many new resource types. You can disable new resource types by deallocator type.
- ◆ Deadlock Analysis
 - ◇ Enable deadlock analysis: off

Note: BoundsChecker 6 did not provide deadlock analysis.

- ◆ .NET Call Reporting
 - ◇ Collect .NET method events: off

- ◆ .NET Analysis
 - ◇ Enable .NET analysis: off

Note: BoundsChecker 6 did not provide support for .NET.

BoundsChecker 6 Cross Reference

The following tables map BoundsChecker 6 settings to BoundsChecker 7.2.

Error Detection Tab

Table 6-1. Error Detection Tab - Memory Error Checking

BoundsChecker 6	BoundsChecker 7.2
Check for memory errors	Memory Tracking <ul style="list-style-type: none">• Enable memory tracking
Check for uninitialized memory errors	Always enabled if Memory tracker is selected.
Use multiple methods	No corresponding feature. BoundsChecker 7.2 uses a different set of algorithms to detect uninitialized memory.
Fill character	Memory Tracking <ul style="list-style-type: none">• Fill on allocation pattern
Add this number of guard bytes...	Memory Tracking <ul style="list-style-type: none">• Guard byte count

Table 6-2. Error Detection Tab - Pointer and Leak Error Checking

BoundsChecker 6	BoundsChecker 7.2
Check for pointer errors	Always on if either Call validation or Memory Tracking is selected.
Maximum call depth for errors	Data Collection <ul style="list-style-type: none"> Maximum call stack on error
Check for memory, resource and OLE interface leaks	Enables the following features: <ul style="list-style-type: none"> Call Validation Memory Tracking COM Object Tracking Resource Tracking
Report leaks in real time	Memory Tracking <ul style="list-style-type: none"> Report leaks immediately
Report call stack information on allocation	No corresponding feature. BoundsChecker 7.2 always reports the call stack on allocation.
Maximum call stack depth for allocations	Data Collection <ul style="list-style-type: none"> Maximum call stack depth on allocation

Table 6-3. Error Detection Tab - API and OLE Error Checking

BoundsChecker 6	BoundsChecker 7.2
Check for API and OLE return code errors	Call Validation <ul style="list-style-type: none"> COM failure codes API failure codes
Check for OLE "Not Implemented" return code	Call Validation <ul style="list-style-type: none"> Check for COM "Not Implemented" return code.
Reset GetLastError value before each API call	No corresponding setting

Table 6-4. Error Detection Tab - Detailed Error Reporting

BoundsChecker 6	BoundsChecker 7.2
Report errors even if no source is available	Modules and Files <ul style="list-style-type: none"> • Show leaks and errors only if source code is available
Report errors caused by other errors	No corresponding setting. BoundsChecker 7.2 always has this version 6 feature enabled.
Event Rollups <ul style="list-style-type: none"> • Rollup duplicate errors • Rollup duplicate leaks • Suppress duplicate errors and leaks 	BoundsChecker 7.2 has a completely re-designed rollup mechanism designed to provide a high-level summary view along with rollups by category and by error within category.

General Settings

Table 6-5. General Settings

BoundsChecker 6	BoundsChecker 7.2
Report errors immediately	General <ul style="list-style-type: none"> • Display error and pause
Save these settings as the default for all new programs	Configuration File Management <ul style="list-style-type: none"> • Save As... • Internal User Defaults

Event Reporting Tab

Table 6-6. Event Reporting Tab

BoundsChecker 6	BoundsChecker 7.2
Collect and Report program event data	API Call Reporting <ul style="list-style-type: none">• Enable API call reporting COM Call Reporting <ul style="list-style-type: none">• Select Enable COM method call reporting on objects that are implemented in the selected modules• Select Report COM method events on objects implemented outside of the listed modules
Report messages	API Call Reporting <ul style="list-style-type: none">• Collect window messages
Report pointer data for API and OLE Methods	Data Collection <ul style="list-style-type: none">• Call parameter encoding depth: 2
Report hooks	API Call Reporting enabled <ul style="list-style-type: none">• Enable API call reporting (side effect)

Program Information Tab

Table 6-7. Program Information Tab

BoundsChecker 6	BoundsChecker 7.2
Working directory	General <ul style="list-style-type: none">• Working directory
Command line arguments	General <ul style="list-style-type: none">• Command line arguments
Source file search path	General <ul style="list-style-type: none">• Source file search path BoundsChecker 7.2 has a more extensive default setting.

Table 6-8. Error Suppression Tab

BoundsChecker 6	BoundsChecker 7.2
Available suppression libraries	<p>BoundsChecker 7.2 provides a new suppression mechanism designed to reduce the need for system level suppressions.</p> <p>Please refer to the online help for the System directories button in the Modules and files tab.</p> <p>Existing BoundsChecker 6 suppression libraries cannot be directly imported into BoundsChecker 7.2.</p> <p>However, BoundsChecker 7.2 introduces the concept of multiple suppression libraries. You can create multiple suppression libraries and access all suppression libraries at runtime. By doing so, you can create one suppression library per functional area, DLL, etc.</p> <p>Refer to the online documentation for suppressions for additional information.</p>

Table 6-9. Modules and Files Dialog Box

BoundsChecker 6	BoundsChecker 7.2
Modules and files dialog box	<p>BoundsChecker 7.2 provides a completely re-designed modules and files feature.</p> <p>The new mechanism enables you to select modules and files to exclude from your application. In addition, BoundsChecker 7.2 introduces the concept of system directories that are automatically ignored. System directories enable you to quickly exclude Microsoft or third-party libraries from analysis.</p> <p>Refer to the modules and files overview in this document as well as the online help.</p>

Other BoundsChecker 7.2 Capabilities

BoundsChecker 7.2 provides the following capabilities that can help you analyze your applications more effectively:

- ◆ Enable or disable various types of error detection to gather only the data you need.
- ◆ Modules and files support reduces unwanted processing or reporting on unrelated portions of an application.
- ◆ Instrument an application with FinalCheck but analyze it using only ActiveCheck.
- ◆ Improved summary and rollup features enable you to quickly locate critical information.
- ◆ Improved suppression and filtering logic can exclude or hide unwanted information.
- ◆ Graphical use-count analysis enables you to quickly correct difficult to find COM interface leaks.

- ◆ Performance and interoperability features to help you port your applications from native languages to the managed .NET environment.
- ◆ Cradle-to-grave monitoring enables you to monitor all phases of your application, including module loads, static constructors, static destructors and module unloads.
- ◆ Deadlock analysis monitors your application for synchronization errors.
- ◆ .NET method call reporting provides a mechanism to log method calls within the managed portions of your application.

Chapter 7

Analyzing Complex Applications



- ◆ About Complex Applications
- ◆ Image File Execution Options
- ◆ Analyzing Limited Parts of Your Program
- ◆ Deciding What to Monitor
- ◆ Analyzing Services
- ◆ Analyzing ActiveX controls using the Test Container
- ◆ Analyzing COM objects using MTS
- ◆ Analyzing Applications That Use COM+ 1.0 and 1.5
- ◆ Analyzing ISAPI filters under IIS 5.0
- ◆ Analyzing ISAPI Filters under IIS 6.0
- ◆ Frequently Asked Questions

This chapter provides information to help you use BoundsChecker more effectively when checking complex applications.

About Complex Applications

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

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

Complex applications can be divided into two groups:

- ◆ Large applications that contain many complex components

- ◆ Non-traditional applications such as 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 BoundsChecker, 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 BoundsChecker to perform the special debugging or analysis operations required to monitor these types of applications.

BoundsChecker Capabilities and Complex Applications

These BoundsChecker capabilities can help you analyze complex applications:

- ◆ Support for Image File Execution Options
- ◆ Ability to restrict the modules and files monitored by your application
- ◆ Ability to enable or disable the BoundsChecker log at run-time

Image File Execution Options

The Windows NT family of operating systems provide a feature called Image File Execution Options. You can use the Image File Execution Options registry key to associate BoundsChecker with an executable so that BoundsChecker will run whenever the executable runs.

This association starts up the specified debugger whenever the executable starts up. You can use this feature to specify BoundsChecker as the default debugger to analyze your application.

Note: You must be a member of the Administrators group to use BoundsChecker with Image File Execution Options.

An Example: Associating BoundsChecker with an Executable

To associate BoundsChecker as the debugger for an executable, create a registry key and a series of associated values in the following area of the registry:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\
Image File Execution Options

The procedure that follows shows how to create a registry key to make BoundsChecker to analyze the Notepad application:

- 1 Run the registry editor and locate the Image File Execution Options key in the registry.
- 2 Create a registry key called: notepad.exe.
- 3 Open the newly created notepad.exe key.
- 4 Create a **String Value** called **Debugger**.
- 5 Enter the path to bc7.exe as the **Value data**. If you installed BoundsChecker in the default location, the path is:
C:\Program Files\Compuware\BoundsChecker\bc7.exe
- 6 Start Notepad.
- 7 Windows NT will start the BoundsChecker standalone application.
- 8 In BoundsChecker, change the settings as necessary.
- 9 Click **Start** on the BoundsChecker toolbar.

More About bc7.exe Switches

You can add switches at the end of the Debugger registry value as you would when running bc7.exe from the command line:

- ◆ The /b switch starts BoundsChecker automatically, in batch mode, without the user interface
- ◆ The /c switch specifies a configuration file

Refer to the BoundsChecker command line help for additional switches.

Note: After you finish using BoundsChecker, either delete the notepad.exe registry key or rename it (for example, _notepad.exe) to prevent BoundsChecker from being started up every time you run the Notepad application.

Analyzing Limited Parts of Your Program

You can point BoundsChecker at a limited problem area within a large or complex application and ignore the rest of the application.

BoundsChecker 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 BoundsChecker logging.
- ◆ Add conditional code into your application to call `StartEvtReporting` and `StopEvtReporting`.

Note: `StartEvtReporting` and `StopEvtReporting` are BoundsChecker functions that you can call from inside your application to control the writing of data into the BoundsChecker log.

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 directory trees
- ◆ Exclude errors if source code is unavailable

See “[Using Modules and Files Settings](#)” on page 72.

Suppression and Filtering

There are two ways to hide the errors and events that BoundsChecker reports.

- ◆ **Suppression** prevents a specified type of error or event from being entered into the BoundsChecker log. To show a suppressed error, you need to remove the suppression instruction and re-run your application under BoundsChecker.
- ◆ **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 BoundsChecker 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 termination, many of the leaks you are trying to find will not be 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 BoundsChecker data collection engine to enable or disable BoundsChecker logging. The following sample code shows how to disable BoundsChecker logging around unwanted areas:

```
// The include file is located in:  
// C:\Program Files\Compuware\BoundsChecker\NMErptApi  
#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 `StartEventReporting` or `StopEvtReporting` API calls to prevent BoundsChecker from logging business-critical sections of an application. Examples might include password validation, serial number checking, or encryption routines.

Using Modules and Files Settings

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

- 1 Open your executable in BoundsChecker.
- 2 Disable all data collection.
 - ◇ In Visual C++ 6.0:
Select **Tools > Settings > BoundsChecker**
 - ◇ In BoundsChecker standalone:
Select **Tools > Settings > BoundsChecker**
 - ◇ In Visual Studio .NET:
Select **Tools > Options > BoundsChecker**

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 BoundsChecker.

BoundsChecker records all DLLs used by your application. Exercise the program in a way that causes all its DLLs to be loaded, then quit your application.
- 4 Open the BoundsChecker **Settings** or **Options** dialog box and select data collection settings.
- 5 Select **Modules and Files** in the **Settings** or **Options** dialog box.

BoundsChecker automatically lists all executables and DLLs used by your application except for files located in the system directories.
- 6 Review the list of modules and files. Clear any listed DLLs that do not pertain 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 directory, click **System directories** and add the directory to the list of excluded directories. If there is a particular file you want to include from a system directory, click **Add module** to add it to the list of monitored DLLs.

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.

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

See [Table 7-1](#) for a list of ways to use **Modules and Files** settings.

Table 7-1. Module and File Settings

When debugging...	Configure BoundsChecker to exclude...
An ActiveX control	All modules other than the DLL containing your ActiveX control including the ActiveX test container executable
A Windows NT 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
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 a 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 will affect 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 BoundsChecker to monitor your application.

As you decide what to monitor, remember that other parts of the program that 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, you will find your errors but you will not find 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 BoundsChecker to monitor your application by selecting **File > Open**. BoundsChecker will open your application and analyze 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 (TSTCON32.EXE) provided with Visual Studio.

If your application is invoked indirectly by a system action, more complex configuration options might be necessary. Examples of this include:

- ◆ Windows NT services

- ◆ Out-of-process COM servers

For example, if your application is a Windows NT service, create an Image File Execution Option registry key so the system will use BoundsChecker as the debugger for your application. (See “[Image File Execution Options](#)” on page 68.) By doing this, BoundsChecker will start up and will monitor your application whenever your application starts.

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 BoundsChecker for best results.

Analyzing Services

BoundsChecker can monitor Windows NT 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?

BoundsChecker 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

BoundsChecker has the following requirements for monitoring a service:

- ◆ The account being used to run BoundsChecker must have Administrative privileges.
- ◆ You must be able to access the registry to change the Image File Execution Options registry keys.

Also note that you may have additional problems if your application has tight timing requirements.

Analyzing a Service

Follow these steps to analyze your service with BoundsChecker:

- 1 Stop your service.
- 2 Build the Debug configuration of your service with symbols and no optimization (optionally with FinalCheck).
- 3 Create a BoundsChecker configuration file for your service.
 - a Open your service.
 - b Update the BoundsChecker settings appropriately for your service.
 - c Save your settings in a new configuration file.
 - d Quit BoundsChecker.
- 4 Create an Image File Execution Options entry for your service (for example, MyService.exe) and specify BoundsChecker as the Debugger.
 - a Use `bc7.exe /b sessionlog.DPbcl`.
 - b To use a custom configuration file (as described in [step 3](#), above), use the `/c` switch and specify the name of the configuration file. If you omit the `/c` switch, BoundsChecker will use the default configuration file stored in the same directory as your service.
 - c To change the working directory, use the `/w` switch.
- 5 Start your service from the Control Panel.

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 NT **Service Control Manager** will terminate your service. When using BoundsChecker, set `dwWait` to a large value such as 4 million.

Note: After you finish using BoundsChecker, restore the normal value for `dwWait`.

Interactive Debugging

To debug your service interactively with BoundsChecker, modify your service to interact with the desktop by selecting **Allow service to interact with desktop** in the **Log On** tab of the **Service Configuration** dialog box. You can then omit the `/b` switch from the BC7 command line.

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 BoundsChecker to check your service worker thread as if it were a Windows console program.

Custom Code to Turn the BoundsChecker Log On and Off

When dealing with a service that is not interactive, you can write custom code to turn BoundsChecker 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 71.

Common Service-related Issues

My 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.

My service starts and almost immediately terminates.

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

Another possible cause is that BoundsChecker cannot create the BoundsChecker log. In this case, specify a full path for the log file after the `/b` switch. Also, verify that BoundsChecker has a valid working directory. Use the `/w` switch to specify working directory.

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

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.

Also, BoundsChecker may have poisoned memory in your application, causing the crash. Disable the **Memory Tracking** feature in the BoundsChecker 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, `BoundsChecker` 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 will terminate 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 `BoundsChecker` 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 `BoundsChecker` with the Test Container:

- 1 Run `BoundsChecker`.
- 2 Select **File > Open** and choose the Test Container.
If you installed Visual Studio in the standard directory you will find the test container in one of the following locations:

```
C:\Program Files\Microsoft Visual Studio\Common\Tools1\
TestCon32.exe
```

```
C:\Program Files\Microsoft Visual Studio .NET\Common7\
Tools\TestCon32.exe
```
- 3 Bring up the **Modules and Files** settings.
- 4 De-select `TestCon32.exe`.
- 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 Optionally, select **Show leaks and errors only if source code is available**.

9 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, quit the Test Container.

During the run, BoundsChecker reports errors as they are detected. When you quit Test Container, BoundsChecker reports memory, resource and interface leaks that were not reported during the run.

Common Test Container Issues

Why is BoundsChecker reporting errors in *TestCon32.exe*?

By default, BoundsChecker 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 directories**. To prevent BoundsChecker from reporting errors on *TestCon32.exe*, exclude this executable from the list of modules to check.

Why isn't BoundsChecker COM Call Reporting logging calls to my object?

BoundsChecker logs methods only for COM interfaces that are known to BoundsChecker. Follow these steps to tell BoundsChecker about your ActiveX control:

- ◆ 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.
- ◆ Select **Generate NLB files dynamically** in the **Data Collection** settings to create new symbolic information for your COM object.

Why isn't BoundsChecker reporting COM interface leaks in my 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 COM Objects Using MTS

You can use BoundsChecker to monitor COM objects running under the Microsoft Transaction Server, MTX.EXE.

Follow these steps to use BoundsChecker with the Microsoft Transaction Server:

- 1 Shut down any existing server process using the MTS Explorer. Right-click **My Computer** and choose **Shutdown Server Process**.
 - 2 Start up BoundsChecker and open MTX.EXE.
 - 3 Open **Settings** (Visual C++ 6.0) or **Options** (Visual Studio .NET) dialog box and select **General**.
 - 4 Under **General**, enter `/p:PackageName` in the **Command line arguments** field where *PackageName* is the name of your MTS package.
- Note:** In this context, *PackageName* is a UUID (universally unique ID) of the form {EAE7E2CE-A7A5-11D1-B0F0-0040951082F7}.
- 5 In the **Settings** (Visual C++ 6.0) or **Options** (Visual Studio .NET) dialog box, select **Data Collection**.
 - 6 Under **NLB file directory**, specify the path to the appropriate working directory.
 - 7 To monitor COM activity in the process, make the appropriate changes in the **COM Call Reporting** and **COM Object Tracking** settings.
 - 8 Bring up the **Modules and Files** settings and clear MTX.EXE.
 - 9 Add the DLL that contains your MTS package into the list of modules and files.
 - 10 Click **Start** on the BoundsChecker menu.
 - 11 Run a program that exercises the MTS package.
 - 12 After you finish exercising the MTS package, wait for the MTX process to time out and quit to allow BoundsChecker to perform end of process error and leak detection.

Common MTS Issues

Why is BoundsChecker telling me about errors in the MTX.exe?

By default, BoundsChecker 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 directories. To prevent

BoundsChecker from reporting errors on MTX.exe, exclude this executable from the list of modules to check.

Why isn't BoundsChecker COM Call Reporting logging calls to my object?

BoundsChecker will only log methods for COM interfaces that are known to BoundsChecker. Perform the following steps to get BoundsChecker to log calls to your MTS package:

- ◆ 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.
- ◆ Select **Generate NLB files dynamically** in the **Data Collection** settings to create new symbolic information for your COM object.

Why isn't BoundsChecker reporting COM interface leaks in my object?

BoundsChecker will only report COM interface leak information if you select **Enable COM object tracking** and select COM classes to monitor in the **COM Object Tracking** settings.

To track only your own objects, scan the list of COM classes in the COM Object Tracking settings and individually select your classes. If you do not know which classes to select, select **All COM classes**.

Why does BoundsChecker appear to hang and not respond for a long time after I stop exercising the MTS package?

BoundsChecker is waiting for MTX.EXE to time out and terminate the process. When MTX.EXE terminates, BoundsChecker performs the final memory, resource and interface leak detection.

Is there a way to debug MTX.EXE using Image File Execution Options?

Yes. However, doing so requires you to create a custom configuration file with all the information described above. You would then create a key for MTX.EXE and specify Debugger value of:

```
BC7.EXE /b log-filename /c configuration-filename  
/w working-directory
```

If you use this method, include the `/p:PackageName` command line argument in the configuration file.

If multiple MTX.EXE processes need to start, they will all invoke BoundsChecker. If possible, use the method described in “[Image File Execution Options](#)” on page 68.

Analyzing Applications That Use COM+ 1.0 and 1.5

BoundsChecker can analyze COM components written for the COM+ 1.x environment. COM+ 1.0 was introduced in Windows 2000; COM+ 1.5 was introduced in and Windows XP.

For BoundsChecker to analyze COM components, you need to edit the **COM Component Services** settings to establish BoundsChecker as the debugger for the COM component.

Follow these steps to set up BoundsChecker 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**.
- 7 Change the **Debugger path** to point to BC7.exe. Provide the full path. If you chose the default path when you installed BoundsChecker, this path would be:

`C:\Program Files\Compuware\BoundsChecker\bc7.exe`

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

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

After you establish BoundsChecker 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 BoundsChecker starts up, select the settings you would like to use, then begin an error detection run.

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

- 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 BoundsChecker **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**.
 - 4 BoundsChecker will then perform the normal end-of-process error and leak detection.
 - 5 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

Why is BoundsChecker reporting errors in `dllhost.exe`?

By default, BoundsChecker 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 directories**. To prevent BoundsChecker from reporting errors on `dllhost.exe`, exclude this executable from the list of modules to check.

Why isn't BoundsChecker COM Call Reporting logging calls to my component?

BoundsChecker will log COM method calls only for interfaces that it can recognize. Perform these two tasks to generate an NLB file that describes the methods in your COM component:

- ◆ Select **Collect method events on objects that are implemented in the selected modules** in the **COM Call Reporting** settings to activate method logging.
- ◆ Select **Generate NLB files dynamically** in the **Data Collection** settings to create new symbolic information for your COM object.

Why isn't BoundsChecker reporting COM interface leaks in my component?

BoundsChecker will report 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**.

BoundsChecker appears to hang and not respond for a long time after I stop exercising my component.

BoundsChecker is waiting for `dllhost.exe` to time out and terminate the process. When `dllhost.exe` terminates, BoundsChecker will perform 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**.

Is there a way to debug `dllhost.exe` using Image File Execution Options?

Debugging `dllhost.exe` using Image File Execution Options is strongly discouraged. Given the number of components being created on Windows 2000 and Windows XP systems, 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 BoundsChecker.

Analyzing ISAPI Filters Under IIS 5.0

You can use BoundsChecker to analyze ISAPI filters within an IIS process. Follow these steps to monitor an ISAPI filter:

Follow these steps to analyze your ISAPI filter with BoundsChecker:

- 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 Modify the account used for IIS to be an account with Administrative rights.

Caution: Security Warning: Before performing this operation, move the system to a secure network segment.

- 4 Create a BoundsChecker configuration file for `inetinfo.exe`:
 - a Open `inetinfo.exe` in BoundsChecker.
 - b Select the Modules and files settings and uncheck all EXEs and DLLs.
 - c Add your ISAPI filter to the list of modules by using Add module.
 - d Update the remaining settings appropriately for your service.
 - e Save your settings.
 - f Quit BoundsChecker.
- 5 Create an Image File Execution Options entry for `inetinfo.exe` and specify BoundsChecker as the Debugger.
 - a Use `bc7.exe /b sessionlog.DPbcl`. (The `/b` switch runs `bc7.exe` in batch mode.)
 - b To use multiple configuration files, use the `/c configfile.DPbcc` switch and specify the name of the configuration file. If you omit the `/c` switch, the default configuration file stored in the same directory as your service will be used.
 - c To change the working directory use the `/w` switch.
- 6 To analyze your ISAPI filter interactively, do not include the `/b` switch in [step 5–a](#) and modify IIS to **Allow service to interact with the desktop** in the **Log On** tab of the **Service Configuration** dialog box.

Caution: Security Warning: If you modify the IIS service to interact with the desktop, you will also be required to use the Local System account. In that case, the logged in user must have Administrative rights. To prevent a security violation, move the system to a secure network segment before performing this step.

- 7 Start the Internet Information Server from the **Services** control panel.
- 8 Generate a series of HTTP requests to the IIS server to exercise your ISAPI filter.
- 9 After you finish exercising your ISAPI filter, use the **Service Control Panel** to stop the IIS Service.

10 BoundsChecker then performs end-of-process error and leak detection and writes out the log.

If you are using BoundsChecker from batch, run BoundsChecker, then open the saved BoundsChecker log file (file extension .DPBcl) and review it for errors.

11 After you finish debugging your ISAPI filter, remove the Image File Execution Option entry for `inetinfo.exe`.

Caution: Security Warning: If you altered the `inetinfo.exe` service settings to have greater privileges or to interact with the desktop you **MUST** restore the IIS service to the previous settings. Failure to perform this step may leave your system and network exposed to unwanted attack.

Caution: Security Warning: Do not leave *Allow service to interact with desktop* enabled after you have finished using BoundsChecker. Failure to disable this option may expose your system to a security risk.

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.

[Why does IIS startup and then hang?](#)

BoundsChecker requires Administrative rights to debug a service. If the account used does not have Administrator rights, IIS will either hang or terminate almost immediately with an error.

[Why does the BoundsChecker log contain so much unwanted 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, BoundsChecker automatically adds any DLLs that were dynamically loaded into the process to the list of modules and files. Check the **Modules and Files** settings and clear the unwanted DLLs.

[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.

Are there tips for debugging IIS interactively with BoundsChecker?

- ◆ You must be logged into an account with Administrative rights.
- ◆ You must enable **Allow service to interact with desktop**.
- ◆ If these suggestions do not solve the your problem, review Microsoft Technical Note 63: Debugging an ISAPI Application:
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsdebug/html/vxoriDebuggingISAPIApplication.asp>

Analyzing ISAPI Filters under IIS 6.0

You can use BoundsChecker 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 BoundsChecker, 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, you will run BoundsChecker against the `inetinfo.exe` executable.

Follow these steps to analyze your ISAPI filter:

Caution: Security Warning: Before performing this operation, move the system to a secure network segment.

- 1 Create a BoundsChecker configuration file for `inetinfo.exe`:
 - a Open `inetinfo.exe` in BoundsChecker.
You can find this file in
`%WINDIR%\System32\Inetsrv\inetinfo.exe`
 - b Open the **Settings** (standalone or Visual C++ 6.0 IDE) or **Options** (Visual Studio .NET) dialog box and select **Modules and files**.
 - c Under **Modules and Files**, de-select all EXEs and DLLs.
 - d Under **Modules and Files**, click **Add Module** to add your ISAPI filter to the list of modules.

- e Update the remaining settings or options as necessary for your ISAPI filter.
 - f Save your settings.
 - g Quit BoundsChecker
- 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 the IIS Admin Service and select the **Log On** tab.
 - 5 Select **Allow Service to Interact with Desktop** check box, then click **OK**.
 - 6 Repeat [step 4](#) and [step 5](#) for all processes that run under the IIS Admin process, for example, World Wide Web Publishing Service.
 - 7 Create an Image File Execution Options entry for `inetinfo.exe` and specify BoundsChecker as the debugger.
 - a Use `bc7.exe / b sessionlog.DPbcl`. (The `/b` switch runs `bc7.exe` in batch mode.)
 - b To use multiple configuration files, use the `/c` (configuration file.DPbcc) switch and specify the name of the configuration file. If you omit the `/c` switch, BoundsChecker will use the default configuration file stored in the same directory as your service.
 - c Use the `/w` switch to change the working directory.
 - 8 To analyze your ISAPI filter interactively, do not include the `/b` switch in [step 7- a](#).

Caution: Security Warning: If you modify the IIS service to interact with the desktop, you will also be required to use the Local System account. In that case, the logged in user must have Administrative rights. To prevent a security violation, move the system to a secure network segment before performing this step.

- 9 Start the World Wide Web Publishing service.
 BoundsChecker will automatically start up and monitor the `inetinfo.exe` process and your ISAPI filter.

- 10 Generate a series of HTTP requests to the web server to exercise your ISAPI filter.
- 11 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 BoundsChecker, both BoundsChecker and the `inetinfo.exe` process will be terminated and you will lose the data you have collected.
- 12 BoundsChecker will perform end-of-process error and leak detection, and will write the results to a log file.
If you are using BoundsChecker from batch, run BoundsChecker, then open the saved BoundsChecker log (.DPBcl file extension) and review it for errors.
- 13 After you finish debugging your ISAPI filter, remove the Image File Execution Option entry for `inetinfo.exe`.

Caution: Security Warning: If you altered the `inetinfo.exe` service settings to have greater privileges or to interact with the desktop you **MUST** restore the IIS service to the previous settings. Failure to perform this step may leave your system and network exposed to unwanted attack.

Caution: Security Warning: Do not leave Allow service to interact with desktop enabled after you have finished using BoundsChecker. Failure to disable this option may expose your system to a security risk.

IIS 6.0 Default Configuration

When running IIS 6.0 in the default configuration mode you will run BoundsChecker against the `W3WP.exe` executable.

Follow these steps to analyze your ISAPI filter:

Caution: Caution: Security Warning: Before performing this operation, move the system to a secure network segment.

- 1 Create a BoundsChecker configuration file for `W3WP.exe`:
 - a Open `W3WP.exe` in BoundsChecker.
You can find this file in `%WINDIR%\System32\Inetsrv\W3WP.exe`.
 - b Select Modules and files settings and uncheck all EXEs and DLLs.
 - c Add your ISAPI filter to the list of modules using Add module.

- d Update the remaining settings appropriately for your ISAPI filter.
 - e Save your settings.
 - f Quit BoundsChecker.
- 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 Open the **Services** dialog box in the Control Panel.
- 4 Select the **IIS Admin Service** and select the **Log On** tab.
- 5 Select **Allow Service to Interact with Desktop**, then click **OK**.
- 6 Repeat [step 4](#) and [step 5](#) for all processes that run under the IIS Admin process (for example, World Wide Web Publishing Service).
- 7 Create an Image File Execution Options entry for `W3WP.exe` and specify BoundsChecker as the debugger.
 - a Use `bc7.exe / b sessionlog.DPbcl`. (The `/b` switch runs `bc7.exe` in batch mode.)
 - b To use multiple configuration files, use the `/c configuration_file.DPbcc` switch and specify the name of the configuration file. If you omit the `/c` switch, the default configuration file stored in the same directory as your service will be used.
 - c Use the `/w` switch to change the working directory.
- 8 To analyze your ISAPI filter interactively, do not include the `/b` switch in [step 7- a](#).

Caution: Security Warning: If you modify the IIS service to interact with the desktop, you will also be required to use the Local System account. In that case, the logged in user must have Administrative rights. To prevent a security violation, move the system to a secure network segment before performing this step.

- 9 Start the World Wide Web Publishing service.
BoundsChecker will automatically start up and monitor the `W3WP.exe` process and your ISAPI filter.

- 10 Generate a series of HTTP requests to the web server to exercise your ISAPI filter.
- 11 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 W2WP.exe process.
- Note:** If you click the **Stop** button in BoundsChecker, both BoundsChecker and the W3WP.exe process will be terminated, and you will lose the data you have collected.
- 12 BoundsChecker will perform end-of-process error and leak detection and write out the log files.
If you are using BoundsChecker from batch, run BoundsChecker, then open the saved BoundsChecker log and review it for errors.
- 13 After you finish debugging your ISAPI filter, remove the Image File Execution Option entry for W3WP.exe.

Caution: Security Warning: If you altered the inetinfo.exe service settings to have greater privileges or to interact with the desktop you **MUST** restore the IIS service to the previous settings. Failure to perform this step may leave your system and network exposed to unwanted attack.

Caution: Security Warning: Do not leave Allow service to interact with desktop enabled after you have finished using BoundsChecker. Failure to disable this option may expose your system to a security risk.

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 inetninfo.exe.

See “[Common ISAPI Filter Issues](#)” on page 86.

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, BoundsChecker creates extremely large log (.DPBcl) files and will impact the performance of the IIS server.

Frequently Asked Questions

What is the difference between BoundsChecker ActiveCheck and FinalCheck or technologies?

BoundsChecker has two modes of operation:

- ◆ **ActiveCheck** - In this mode BoundsChecker will operate on any 32-bit Windows program and will intercept 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 BoundsChecker FinalCheck instrumentation logic. With FinalCheck instrumentation, BoundsChecker can watch every pointer fetch, store or indirect that occurs within the modules you instrument. BoundsChecker can also watch variables go in and out of scope.

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 will cause BoundsChecker 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 BoundsChecker 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), BoundsChecker performs a more detailed ActiveCheck analysis. Note that this feature can cause BoundsChecker to run as much as 20% slower.

How do I use the BoundsChecker 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 will increase the spacing between heap blocks and will provide an area between your blocks for BoundsChecker 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 BoundsChecker to validate each heap block whenever you make a call into a memory function. This will make your program run much slower but will isolate heap corruption to a localized area of your program, making it easier to track down.

What are the major differences between the 32-bit and 64-bit versions of BoundsChecker?

BoundsChecker 64 contains most, but not all, of the 32-bit features. The following features are not available in BoundsChecker 64:

- ◆ FinalCheck instrumentation
- ◆ Visual Studio IDE integration
- ◆ Support for the Microsoft .NET framework
- ◆ Support for common language runtime

Chapter 8

Working with User-Written Allocators



- ◆ Gathering Necessary Information
- ◆ Creating Entries in *UserAllocators.dat*
- ◆ How to Diagnose Errors in *UserAllocators.dat*

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

Introduction

BoundsChecker 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 sub-directory of the BoundsChecker installation. After you add user-written allocators to this file, BoundsChecker treats them like memory allocation routines provided with the operating system. If BoundsChecker detects a leak caused by a user-written allocator described in `UserAllocators.dat`, the user-written allocator will be 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 name of the allocation, deallocation, re-allocation and size function of your user-written allocator.
- 2 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.
- 3 Find out if your user-written memory allocator is statically linked to the application or is it provided in a separate module (DLL).
- 4 The name of the module (DLL) that contains your user-written allocator.
- 5 Any special assumptions made in your allocator such as zeroing memory on allocation or a user-written allocator that stores data in deallocated block.

Finding the Names of User-Written Allocators

To add records to `UserAllocators.dat` you need to provide the exact name of your allocate, deallocate, re-allocate 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`)
 - ◇ Re-allocation function (such as `realloc`)
 - ◇ Memory block size function (such as `_msize`)
- 2 Determine what the mangled version of the function name is by using either `DUMPBIN /EXPORTS` or by looking up the names in a linker map.

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 will always be 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 will look like this:

```

global operator new:          ??2SampleClass@@SAPAXI@Z
global operator delete:      ??3SampleClass@@SAXPAX@Z

```

If you are using the Microsoft Visual C++ compiler, you will always get `@@SAPAXI@Z` for global operator new and `@@SAXPAX@Z` for global operator delete.

If you are writing your own malloc / free replacement, search the map file for the mangled name of your routines and note the names.

If your user-written allocator can be called from a DLL, you can extract the same information from the DLL by using the DUMPBIN utility provided with the Platform SDK or Visual Studio. To extract the information, run the following command:

```
DUMPBIN /EXPORTS yourlibrary.dll > file.txt
```

Search file.txt for your user-written allocator, as described above.

Global Operator New in Visual Studio .NET 2003

BoundsChecker's handling of global_operator_new depends whether or not the module implementing it is the CRTL.

Code compiled with versions of Visual Studio prior to Visual Studio .NET 2003 called the version of global_operator_new in the C RunTime Library (when compiled against the DLL versions of the CRTL with the /MDd dynamic dll switch). With these older versions, the memory returned by global_operator_new is uninitialized. Boundschecker fills it with a known pattern to check for uninitialized variables.

Beginning with VS.NET 2003, the compiler creates the `global_operator_new` function in the module being compiled. This makes it difficult to know whether the user has initialized the memory to a known state and is depending on that state to be true. Filling the memory with a known pattern may cause problems with the application.

By default, BoundsChecker will not fill the memory returned by `global_operator_new` in modules other than the CRT DLLs with a pattern.

In order to check for uninitialized memory in code compiled with VS.NET 2003, you must add the following statement to `UserAllocators.dat` for each module (executable or DLL) to be checked:

```
Allocator  Module=MyModule.exe    Function=??2@YAPAXI@Z
MemoryType=MEM_NEW NumParams=1    Size=1    FILL    STATIC
```

This tells BoundsChecker to hook the implementation of `global_operator_new` in `MyModule.exe`, and fill it with the appropriate pattern to check for uninitialized memory. Replace `MyModule.exe` with the name of the appropriate executable and add a new line for each module in your application.

Examining Parameters of User-Written Allocator Functions

To describe a user-written allocator to BoundsChecker, you need to provide the following information about the function:

- ◆ Which allocation function parameter contains the number of bytes to be allocated?
- ◆ If your memory allocation function is a `calloc`-like function, which parameter contains the number of elements to be created?
- ◆ How does your allocation function return the newly allocated block?
- ◆ How do you pass a pointer to the block to the `msize`, re-allocation and deallocation routines?

Special Assumptions Made By User-Written Allocators about Memory

Normally, BoundsChecker will fill allocated memory with a fill pattern before returning the pointer to the user program and will “poison” the block of memory after it has been deallocated.

If your memory allocator pre-fills the block with special data, make note of this fact.

If your memory allocator assumes that it can write into the block after it has been deallocated, you must tell BoundsChecker not to “poison” the

block after it has been deallocated. 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.
- ◆ Your application assumes that deallocated blocks can continue to be referenced until the block has been re-allocated. (This 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 95, you can describe your allocator in `UserAllocators.dat`.

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

```
[ Record Type ] [ parameter 1 ] [ parameter 2 ] ...
```

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.

[Table 8-1](#) shows the Record Types that are currently defined.

Table 8-1. 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 BoundsChecker 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 8-2. Module types

Module Type	Description
Named module	An explicitly named module (DLL) that contains the user allocation function or method (for example, <code>foo.dll</code>).
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. In this case, you must provide debug symbols for the module and use the optional <code>STATIC</code> keyword to alert BoundsChecker to look for the function or method in the debug symbols. Note: Failure to include the <code>STATIC</code> keyword in the optional parameters for the record will prevent BoundsChecker from properly monitoring the user-written allocation function or method.

Table 8-2. Module types

Module Type	Description
*CRT	<p>This is a special case that enables you to reference a function, wherever it appears in your application. This case is used primarily to identify C or C++ runtime library functions.</p> <p>*CRT is not a macro; it covers the following 3 cases:</p> <ol style="list-style-type: none"> 1. The Microsoft C Runtime Library 2. The statically linked C runtime library 3. Any user function that has the same mangled name as anything that you are patching (for example, <code>global operator new.</code>)

Allocator Records

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

```
Allocator [Module] [Function] [MemoryType]
[NumParams] [Size] [Count] [Optional Parameters]
```

Table 8-3. Allocator Record Parameters

Parameter	Description
Allocator	The first parameter in the record must be the word <code>Allocator</code> to indicate that you are describing an allocation routine.
[Module]	The name of the module (DLL) that contains the user-written allocator.
[Function]	<p>The mangled name of the function that allocates blocks of memory in your user-written allocator.</p> <p>This parameter is case-sensitive.</p>

Table 8-3. Allocator Record Parameters

Parameter	Description
[MemoryType]	<p>Use this parameter to describe what type of memory is being allocated. BoundsChecker currently defines the following memory types:</p> <ul style="list-style-type: none"> • MEM_MALLOC Blocks of memory returned from routines such as <code>malloc</code>, <code>calloc</code>, <code>strdup</code>, and so on. Memory of this type is freed using a routine similar to the C runtime library <code>free</code> routine. • MEM_NEW Blocks of memory that are returned from operator <code>new</code> and are freed by operator <code>delete</code>. • 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. <p>BoundsChecker will verify 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, BoundsChecker will display a memory conflict error at runtime.</p>
[NumParams]	<p>The number of parameters passed to your function. This value must be between 1 and 32.</p> <p>Provide the proper number of parameters when describing a user-written allocator function. Failure to provide the correct value can cause unpredictable results.</p>
[Size]	<p>The parameter number that contains the size of the block to be allocated.</p>
[Count]	<p>This optional parameter describes <code>calloc</code>-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, BoundsChecker assumes that there is no count parameter.</p> <p>For more information on the use of this parameter, review the MSDN documentation for <code>calloc</code>.</p> <p>Note: If you specify a count parameter, you must also specify <code>NOFILL</code> if you are describing <code>calloc</code>-like functions.</p>

Table 8-3. Allocator Record Parameters

Parameter	Description
[Optional Parameters]	<p>You can include the following optional parameters at the end of the record:</p> <ul style="list-style-type: none"> FILL If this parameter is specified, BoundsChecker will fill the buffer returned with the BoundsChecker 'fill' character. <p>Note: If you do not specify FILL or NOFILL, the default is to fill the block.</p> NOFILL If this parameter is specified, BoundsChecker will not fill the buffer returned with the BoundsChecker 'fill' character. <p>If your user-written allocator initializes the block with data, specify NOFILL to avoid corrupting your data.</p> STATIC BoundsChecker will statically patch 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.

BoundsChecker assumes that the return value from your allocator function will be the address of the block of memory. If your function returns a value of NULL (0), BoundsChecker assumes that the allocation failed.

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. BoundsChecker 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 BoundsChecker 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. BoundsChecker 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.

BoundsChecker should not fill the buffer before returning it to the application program. Note that this record specifies `MEM_CUSTOM1` as the `MemoryType`. BoundsChecker verifies that the memory allocated with this function is deallocated by a routine that is also of type `MEM_CUSTOM1`. Using other deallocation routines will generate 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 and fourth parameters are internal to the application. The memory retrieved from the application has been pre-initialized so BoundsChecker should not fill the block.

```
Allocator Module = DataStore.dll Function = MyAlloc
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] [Function] [MemoryType]
[NumParams] [Address] [Optional Parameters]
```

Table 8-4. 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]	The name of the module (DLL) that contains the user-written allocator.
[Function]	The mangled name of the function that deallocates blocks of memory in your user-written allocator. This parameter is case-sensitive.
[MemoryType]	<p>This parameter provides a mechanism to describe what type of memory is being deallocated. BoundsChecker currently defines the following memory types:</p> <ul style="list-style-type: none"> • 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. <p>BoundsChecker will verify 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, BoundsChecker will display a memory conflict error at runtime.</p>
[NumParams]	<p>The number of parameters passed to your function. This value must be between 1 and 32.</p> <p>Provide the proper number of parameters when describing a user-written allocator function. Failure to provide the correct value can cause unpredictable results.</p>
[Address]	The parameter number that contains the pointer to the block being deallocated.

Table 8-4. Deallocator Record Parameters

Parameter	Description
[Optional Parameters]	<p>The following optional parameters may be included at the end of the record:</p> <ul style="list-style-type: none"> • POISON When a block of memory is deallocated, BoundsChecker will overwrite the block with the poison pattern. This is the default. • NOPOISON When you specify NOPOISON, BoundsChecker will 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 NOPOISON to avoid corrupting your data. • STATIC BoundsChecker will statically patch 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.

BoundsChecker 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. BoundsChecker 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. BoundsChecker should poison the memory before returning to the application program. When the block is deallocated, BoundsChecker verifies that the block was allocated by an allocator of type `MEM_CUSTOM1`. If the block was 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 `BoundsChecker`. 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] [Function] [MemoryType] [NumParams]
[Address] [Optional Parameters]
```

Note: if you omit a `QuerySize` record for a user-defined allocator, `BoundsChecker` will return an incorrect block size for that function.

Table 8-5. QuerySize Records

Parameter	Description
<code>QuerySize</code>	The first parameter in the record must be the word <code>QuerySize</code> to indicate that you are describing a size routine.
<code>[Module]</code>	The name of the module (DLL) that contains the user-written allocator.
<code>[Function]</code>	The mangled name of the function that returns the size of allocated blocks of memory in your user-written allocator. This parameter is case-sensitive.

Table 8-5. QuerySize Records

Parameter	Description
[MemoryType]	<p>This parameter provides a mechanism to describe what type of memory is being queried. BoundsChecker currently defines the following memory types:</p> <ul style="list-style-type: none"> • 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. <p>BoundsChecker will verify 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, BoundsChecker will display a memory conflict error at runtime.</p>
[NumParams]	<p>The number of parameters passed to your function. This value must be between 1 and 32.</p> <p>Provide the proper number of parameters when describing a user-written allocator function. Failure to provide the correct value can cause unpredictable results.</p>
[Address]	<p>The parameter number that contains the pointer to the block being queried.</p>
[Optional Parameters]	<p>The following optional parameter may be included at the end of the record:</p> <ul style="list-style-type: none"> • STATIC BoundsChecker will statically patch 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 is passed in 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] [Function] [MemoryType] [NumParams] [Address]
[Size] [Optional Parameters]
```

Table 8-6. Reallocator record parameters

Parameter	Description
Reallocator	The first parameter in the record must be the word Reallocator to indicate that you are describing a Re-allocation routine.
[Module]	The name of the module (DLL) that contains the user-written allocator.
[Function]	The mangled name of the function that re-allocates blocks of memory in your user-written allocator. This parameter is case-sensitive.

Table 8-6. Reallocator record parameters

Parameter	Description
[MemoryType]	<p>This parameter provides a mechanism to describe what type of memory is being re-allocated. BoundsChecker currently defines the following memory types:</p> <ul style="list-style-type: none"> • 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_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. <p>BoundsChecker will verify 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, BoundsChecker will display a memory conflict error at runtime.</p>
[NumParams]	<p>The number of parameters passed to your function. This value must be between 1 and 32.</p> <p>Provide the proper number of parameters when describing a user-written allocator function. Failure to provide the correct value can cause unpredictable results.</p>
[Address]	<p>The parameter number that contains the pointer to the block being reallocated.</p>
[Size]	<p>The parameter number that contains the size of the re-allocated block of memory.</p>

Table 8-6. Reallocator record parameters

Parameter	Description
[MemoryType]	<p>This parameter provides a mechanism to describe what type of memory is being re-allocated. BoundsChecker currently defines the following memory types:</p> <ul style="list-style-type: none"> • 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_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. <p>BoundsChecker will verify 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, BoundsChecker will display a memory conflict error at runtime.</p>
[NumParams]	<p>The number of parameters passed to your function. This value must be between 1 and 32.</p> <p>Provide the proper number of parameters when describing a user-written allocator function. Failure to provide the correct value can cause unpredictable results.</p>
[Address]	<p>The parameter number that contains the pointer to the block being reallocated.</p>
[Size]	<p>The parameter number that contains the size of the re-allocated block of memory.</p>

Table 8-6. Reallocator record parameters

Parameter	Description
[Optional Parameters]	<p>The following optional parameters may be included at the end of the record:</p> <ul style="list-style-type: none"> FILL If this parameter is specified, BoundsChecker will fill the buffer returned with the BoundsChecker 'fill' character. Note: If you do not specify FILL or NOFILL, the default is to fill the block. NOFILL If this parameter is specified BoundsChecker will not fill any additional bytes added to the end of the previous allocation with the BoundsChecker 'fill' character. If your user-written allocator initializes the block with data, specify NOFILL to avoid corrupting your data. STATIC BoundsChecker will statically patch 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.

BoundsChecker checks the return value from the re-allocation 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, BoundsChecker will fill 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 BoundsChecker should *not* fill additional memory in new blocks.

Note: BoundsChecker will ignore 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 BoundsChecker. 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 BoundsChecker memory tracking system. Follow this format:

```
Ignore [Module] [Function] [Optional Parameters]
```

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

Caution: If you create Ignore records, the BoundsChecker memory tracking system will no longer monitor memory allocated or freed from those APIs. As a result, BoundsChecker will no longer be 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 technical support for assistance.

Table 8-7. 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.

Table 8-7. Ignore record parameters

Parameter	Description
[Module]	The name of the module (DLL) that contains the function to be ignored.
[Function]	The mangled name of the function to be ignored by the memory tracking system. This parameter is case-sensitive.
[Optional Parameters]	<ul style="list-style-type: none"> • STATIC BoundsChecker will statically patch 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 Reallocator Records

The following examples show hypothetical ignore records.

Example 1 This example creates an ignore record that will cause BoundsChecker to monitor memory allocated by `GlobalAlloc` but will not see any requests to free the memory returned to the operating system using `GlobalFree`.

Note: This will cause BoundsChecker to report a large number of false memory leaks.

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

Example 2 This example tells BoundsChecker to ignore memory manipulated by the `GlobalAlloc` family of calls.

Note: This will cause BoundsChecker 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 BoundsChecker 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 BoundsChecker 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 technical support for assistance.

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` sub-directory of the `BoundsChecker` installation directory. If the file is deleted or made non-readable, `BoundsChecker` will report an error.

- ◆ **Parsing errors**

If `BoundsChecker` encounters errors while parsing the `UserAllocators.dat` file, `BoundsChecker` will log the errors in the `Errors` tab. If `Memory Tracking` or `Resource Tracking` are enabled when these `UserAllocators.dat` errors are encountered, these features will be disabled.

Token Parsing Errors

`BoundsChecker` will parse the file one line at a time using the following rules:

- 1 Blank lines and lines that start with semi-colons will be ignored.
- 2 The first token on the line must be a valid record type.
- 3 Any remaining tokens on the line must be separated by one or more spaces or tabs and must follow the rules for each record type.

Semantic Errors

Each record type will be 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 `BoundsChecker` to monitor calls to and from your user-written allocator.

If you did not properly describe the APIs to BoundsChecker, 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 NOPOSITION options to your description.

If you encounter an error and are unsure how to proceed please contact technical support for assistance. When you contact technical support, please provide the following information:

- 1 The version of BoundsChecker you are running
- 2 A copy of your `UserAllocators.dat` file
- 3 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 BoundsChecker to respond unpredictably when you analyze an application.

Chapter 9

Deadlock Analysis



- ◆ Background: Single and Multi-threaded Applications
- ◆ Deadlock - A Basic Definition
- ◆ Techniques for Avoiding Deadlocks
- ◆ Potential Deadlocks
- ◆ Other Synchronization Objects
- ◆ Additional Information

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

Newer applications can be multi-threaded. 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 will have 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 look like:

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 will be jumbled, as in the example below. 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 will prevent 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 didn't 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 will pause 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 will 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 will always appear correctly.

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

Note: In most cases you don't 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 will wait until C2 becomes available and will then be granted access to C2 so that 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 will 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 will become blocked waiting for another resource. Failure to release the object might cause a “deadly embrace” and will only make 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 124 lists MSDN resources and books that discuss synchronization objects.

Potential Deadlocks

BoundsChecker will report 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 9-1 illustrates the critical sections each thread requires to perform a given operation:

Table 9-1. Potential deadlock example: Threads and their required critical sections.

Thread	Critical Section
T1	C1, C2
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 BoundsChecker software contains sample code for a version of the Dining Philosophers. You can find it under:

```
...\BoundsChecker\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 will attempt 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, will then try to pick up the right chopstick. A philosopher with both left and right chopsticks will eat 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 will wait a few seconds and try again. A philosopher who succeeds will then attempt to pick up the right chopstick. If the right chopstick isn't available, the philosopher will wait a few seconds and then try again.

The problem occurs when all philosophers pick up the left chopstick at once. If this occurs, none of them will ever put down the left chopstick so they will 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.

BoundsChecker will notify you if a potential deadlock is detected long before the actual deadlock occurs. BoundsChecker will also provide detailed information describing how the actual deadlock will occur. 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 BoundsChecker 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 will be 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 BoundsChecker 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 118](#). 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

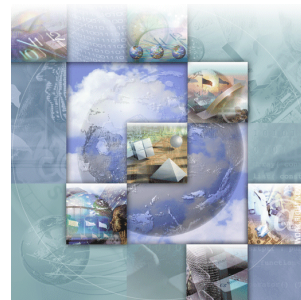
Debugging Applications for Microsoft .NET and Microsoft Windows, by John Robbins

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.

Index



A

ActiveX 74

- components 68

- debugging controls 73

Administrative privileges 77

Administrative rights 85, 86

B

bc7.com 13

bc7.exe 13, 69

BoundsChecker

- benefits 10

- features 9

- options 19

- settings 19

- workflow 10

BoundsChecker 6

- maximum mode 58

- normal mode 57

- settings 56

- user interface 55

- workflow 56

BoundsChecker64

- serial number 4

C

Call parameter encoding depth 49, 57, 58, 62

Call Validation 44, 46

CLR analysis 45

COM servers 75

COM usage 37

COM+ 82

- components 68

Complex applications 67

- analyzing 46

- debugging 73

Conditional code 70

Configuration File Management 41

Customer Service xi

D

Dangling pointers 53

Debugging environment 13

Default debugger 68

Default settings 43

dllhost.exe 82

dwWait 76

E

Errors

- leak 13

- memory 13

- pointer 13

- suppressing 26

F

Filter 26

Filters 70

FinalCheck [12](#), [13](#)

G

Guard bytes [38](#)

H

Hardware requirements [1](#)
 BoundsChecker64 [2](#)

I

IIS process [84](#)
Image File Execution Options [75](#), [76](#)
Instrumentation [12](#)
Interface leaks [47](#)
ISAPI filters [40](#), [68](#), [73](#), [84](#), [85](#)

L

Leak errors [13](#)
License file [2](#), [3](#), [4](#)
Locate in Transcript [18](#)
Log file [49](#)

M

Main window [17](#)
Managed code [39](#), [44](#), [45](#), [46](#)
master [4](#)
Master license file [4](#)
Memory
 errors [13](#)
 leaks [47](#)
 overrun [44](#)
 poisoning [38](#)
Memory and Resource Viewer [24](#)
memory leaks [24](#)
Memory Tracker [41](#)
Microsoft Transaction Server [80](#)
Modules and Files [41](#), [64](#), [70](#), [72](#)
 and complex applications [47](#)
 and reverse engineering [51](#)
MTX.EXE [80](#)
Multiprocessor application servers [52](#)

Multi-threaded applications [52](#)

N

Native code [39](#), [44](#), [45](#)

O

Options dialog box [19](#)

P

P/Invoke [45](#), [49](#)
PInvoke interop monitoring [45](#)
Pointers
 dangling [53](#)
 errors [13](#)
Poisoning memory [38](#), [51](#)

R

Resource leaks [40](#), [47](#)
resource leaks [24](#)
Resource Tracker [41](#)

S

Service control logic [77](#)
Settings [41](#)
 configuration [19](#)
 default [34](#)
 refining [34](#)
Settings dialog box [19](#)
Smart debugging [14](#)
Software complexity [7](#)
Software requirements [1](#)
 BoundsChecker64 [2](#)
StartEventReporting [72](#)
StopEvtReporting [72](#)
Summary tab [18](#)
Suppression [26](#), [63](#), [70](#)
 libraries [27](#)

T

Technical Support [xii](#)
Test container [74](#)

Third-party software [10](#), [34](#), [47](#), [64](#), [70](#)

Transactional applications [74](#)

W

Windows NT

- Service Control Manager [77](#)

- services [68](#), [74](#), [75](#)

- services, debugging [73](#)

Windows XP [82](#)

Worker thread [77](#)

