Intel® Inspector User Guide for Windows* OS

Contents

Intel® Inspector User Guide for Windows* OS

1

Intel® Inspector is a dynamic memory and threading error checking tool for users developing serial and multithreaded applications on Windows* and Linux* operating systems.

Many of our readers have demonstrated interest in the following User Guide content:

Typical Usage Scenarios

There are many ways to take advantage of the power and flexibility of the Intel Inspector. Workflows-Quick Paths to Maximizing Productivity offers three workflows to get you started.

Problem Type Reference

Intel Inspector groups detected issues into memory and threading categories. See Problem Type Reference for information on all the issues the Intel Inspector detects.

Command Line Interface Support

See Command Line Interface Support for information about the Intel Inspector `inspxe-cl` command tool.

Download Here

You can download and install the Intel Inspector as follows:

- as a standalone product
- as part of Intel HPC Toolkit

> **NOTE**
> Documentation for older versions of Intel Inspector are available for download only. For a list of documentation archives by product versions till 2023, see these pages:
>
> - Download Documentation for Intel® Parallel Studio XE
> - Download Documentation for Intel® System Studio
>
> For product documentation starting with version 2023, use the **Version** drop-down menu above to choose a version you need. To download this version of the document, click the **Download** button in the upper right corner of this window.

Start Here

Get Started with Intel Inspector

What's New

Intel Inspector Tutorials with samples installed with the product in the `/samples` folder

Get Started

- Get Started with Intel Inspector
- Dynamic Analysis vs. Static Analysis
- Key Terminology

- Sample Applications and Tutorials
- Workflows-Quick Paths to Maximizing Productivity

Dynamic Analysis vs. Static Analysis

Dynamic analysis is the testing and evaluation of an application during runtime.

Static analysis is the testing and evaluation of an application by examining the code without executing the application.

Many software defects that cause memory and threading errors can be detected both dynamically and statically. The two approaches are complementary because no single approach can find every error.

The primary advantage of dynamic analysis: It reveals subtle defects or vulnerabilities whose cause is too complex to be discovered by static analysis. Dynamic analysis can play a role in security assurance, but its primary goal is finding and debugging errors.

The primary advantage of static analysis: It examines all possible execution paths and variable values, not just those invoked during execution. Thus static analysis can reveal errors that may not manifest themselves until weeks, months or years after release. This aspect of static analysis is especially valuable in security assurance, because security attacks often exercise an application in unforeseen and untested ways.

Intel® Inspector generates dynamic analysis results to help you find and fix memory and threading errors.

See Also

Key Terminology

Workflows-Quick Paths to Maximizing Productivity

Key Terminology

analysis: A process during which the Intel Inspector performs collection and finalization.

code location: A fact the Intel Inspector observes at a source code location, such as a *write* code location. Previously called an *observation*.

collection: A process during which the Intel Inspector executes an application, identifies issues that may need handling, and collects those issues in a result.

false positive: A reported error that is not an error.

finalization: A process during which the Intel Inspector uses debug information from binary files to convert symbol information into filenames and line numbers, performs duplicate elimination (if requested), and forms problem sets.

problem: One or more occurrences of a detected issue, such as an uninitialized memory access. Multiple occurrences have the same call stack but a different thread or timestamp. You can view information for a problem as well as for each occurrence.

problem breakpoint: A breakpoint that halts execution when a memory or threading analysis detects a problem. In the Visual Studio* debugger, a problem breakpoint is indicated by a yellow arrow at the source line where execution halts.

problem set: A group of problems with a common problem type and a shared code location that might share a common solution, such as a problem set resulting from deallocating an object too early during application execution. You can view problem sets only after analysis is complete.

project: A compiled application; a collection of configurable attributes, including suppression rules and search directories; and a container for analysis results.

result: A collection of issues that may need handling.

suppression: An Intel Inspector productivity enhancement feature you can use to not collect result data that matches a rule you define.

target: An application the Intel Inspector inspects for errors.

See Also

Dynamic Analysis vs. Static Analysis
Workflows-Quick Paths to Maximizing Productivity

Sample Applications and Tutorials

Sample applications can help you learn basic Intel Inspector operations. They are installed as individual compressed files under the `samples` directory within the Intel Inspector installation directory. After you copy a sample application compressed file to a writable directory, use a suitable tool to extract the contents. Extracted contents include a short README that describes how to build the sample and fix issues. To load a sample into the Visual Studio* environment, double-click the `.sln` file.

Tutorials show you how to find and fix uninitialized memory access, memory leak, and data race errors using C++ and Fortran sample applications.

See Intel® Inspector Tutorials for more information.

NOTE

- Sample applications are non-deterministic.
- Sample applications are designed only to illustrate the Intel Inspector features and do not represent best practices for creating code.

Workflows-Quick Paths to Maximizing Productivity

There are many ways to take advantage of the power and flexibility of the Intel® Inspector.

Use the following workflows to help you maximize your productivity as quickly as possible. See links below for more details on each workflow.

Typical Workflows	Usage
Analysis	Use the graphical user interface (GUI) to: **1.** Inspect a running application for issues and collect a result. **2.** Investigate the result. **3.** Fix issues that should not remain in source code.
Reporting	Use the command line interface (CLI) to: **1.** Inspect a running application for issues and collect a result. **2.** Investigate the result.
Regression Testing	**1.** Create a gold standard: • Use the GUI (or CLI) to create a result from which you have successfully eliminated all real problems. • Use the CLI to create a suppression file to match any remaining problems you have determined you can ignore. **2.** Change your source code. **3.** Use the CLI (or GUI) to check for new problems: • Apply the suppression file when you run a new analysis. • Determine if the changed source code introduced new problems that do not match those in the suppression file.

Parent topic: Getting Started

Analysis Workflow

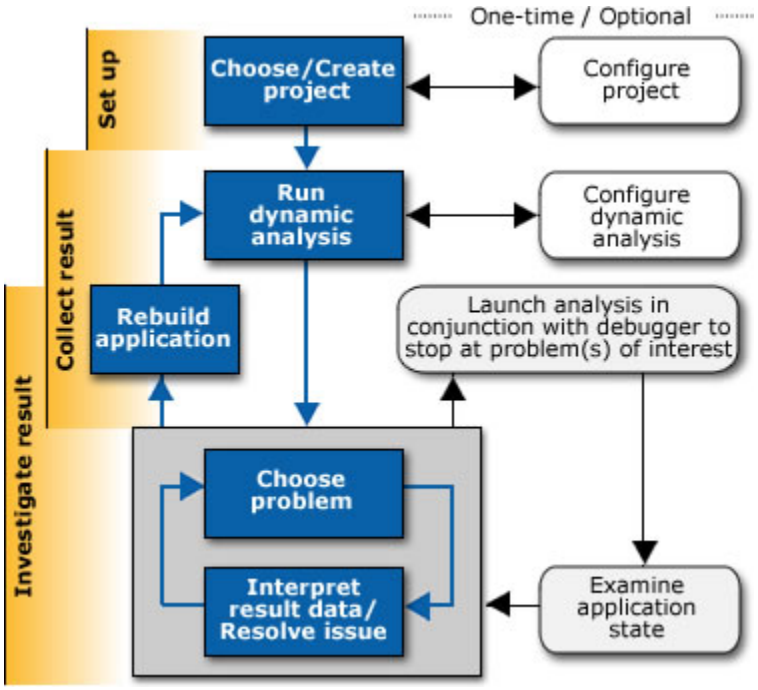
There are many ways to take advantage of the power and flexibility of the Intel Inspector.

Intel Inspector Help recommends this workflow to help maximize your productivity as quickly as possible when you work solely in the graphical user interface.

The following figure shows an overview of this workflow. To display more information about a workflow step:

- Position (hover) the mouse pointer to display a brief explanation.
- Click to display the associated topic.

NOTE
If clicking the workflow step does not display the associated topic, use the text links below the figure.



See Also

Build Your Application
Choose Problems
Choose Projects
Configure Analysis
Configure Projects
Interpret Result Data and Resolve Issues
Investigate Problems Using Interactive Debugging
Run Analysis
Workflows-Quick Paths to Maximizing Productivity

Reporting Workflow

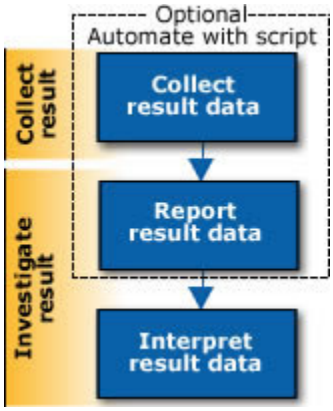
There are many ways to take advantage of the power and flexibility of the Intel Inspector.

Intel Inspector Help recommends this workflow to help maximize your productivity as quickly as possible when you work solely in the command line interface.

The following figure shows an overview of this workflow. To display more information about a workflow step:

- Position (hover) the mouse pointer to display a brief explanation.
- Click to display the associated topic.

NOTE
If clicking the workflow step does not display the associated topic, use the text links below the figure.



See Also

Collecting Result Data from the Command Line
Interpreting Result Data from the Command Line
Reporting Result Data from the Command Line
Workflows-Quick Paths to Maximizing Productivity

Regression Testing Workflow

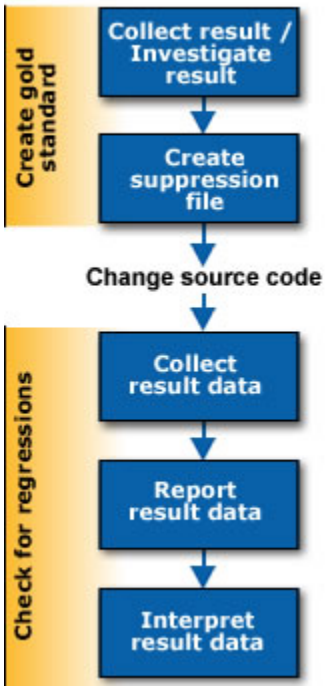
There are many ways to take advantage of the power and flexibility of the Intel Inspector.

Intel Inspector Help recommends this workflow to help maximize your productivity as quickly as possible when performing regression testing.

The following figure shows an overview of this workflow. To display more information about a workflow step:

- Position (hover) the mouse pointer to display a brief explanation.
- Click to display the associated topic.

NOTE
If clicking the workflow step does not display the associated topic, use the text links below the figure.



See Also

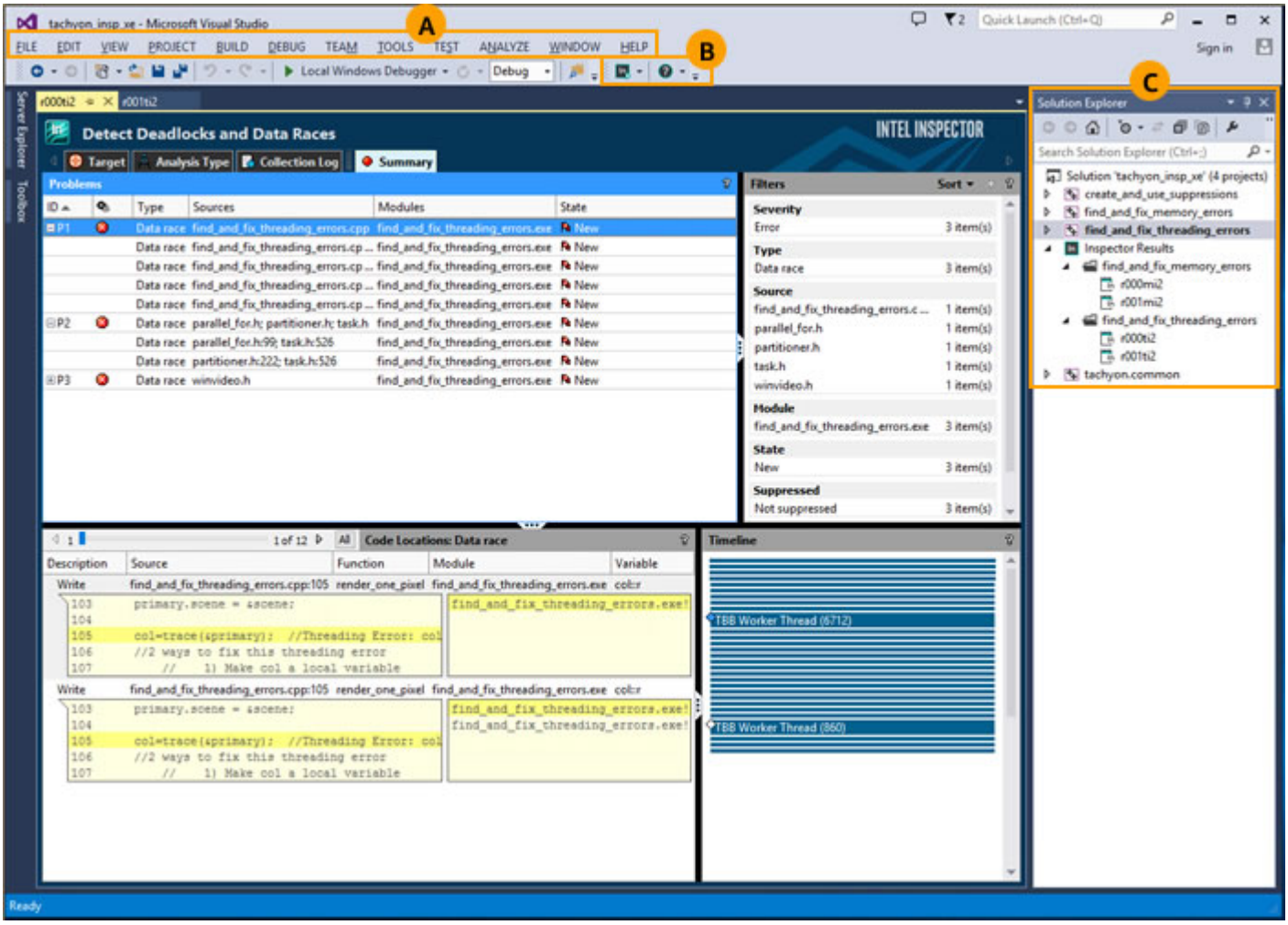
Workflows-Quick Paths to Maximizing Productivity
Tes for Regressions: Recommended Approach
Collect Results
Investigate Results
Creating a Suppress-All File from the Command Line
Collecting Result Data from the Command Line
Interpreting Result Data from the Command Line
Reporting Result Data from the Command Line

Before You Begin

- Visual Studio Integration
- Standalone Intel Inspector GUI
- Startup Tasks
  - Change Result and Result Directory Name Templates
  - Choose Baseline Results for Setting States
  - Display Application Output

Visual Studio* Integration

Intel® Inspector integrates into the Visual Studio* integrated development environment (IDE) and can be accessed from the menus, toolbar, and **Solution Explorer** in the following manner:



The menu, toolbar, and **Solution Explorer** offer different ways to perform many of the same functions.

A Use the **Tools > Intel Inspector [version]** menu to create analysis results; compare results; and import result archive files, results not associated with a project, and results from other Intel error-detection products into the current project.

B Use the **Intel Inspector** toolbar to create analysis results, compare results, configure projects, and open documentation resources.

C **Solution Explorer** context menus (right-click to open):

- Use the **Intel Inspector [version]** menu on the **Solution Explorer** project context menu to create analysis results and configure projects.
- Use the context menu on a result in the project result folder to open results, create analysis results, refinalize (re-resolve) results, export result archive files, and manage results.

Visual Studio 2022 Integration

Intel Inspector provides lightweight integration into Visual Studio 2022. You can launch a standalone Intel Inspector for your Visual Studio project as follows:

- Select the **Tools** > **Open Intel Inspector** menu item
- Click the

Open Intel Inspector toolbar icon
- Right-click the project entry in the **Solution Explorer** and select **Intel Inspector** > **Open Intel Inspector** from the context menu.

The standalone Intel Inspector graphical version opens inheriting the project properties of the target selected in Visual Studio.

In Visual Studio 2022, you can also open the documentation resources for Intel Inspector as follows:

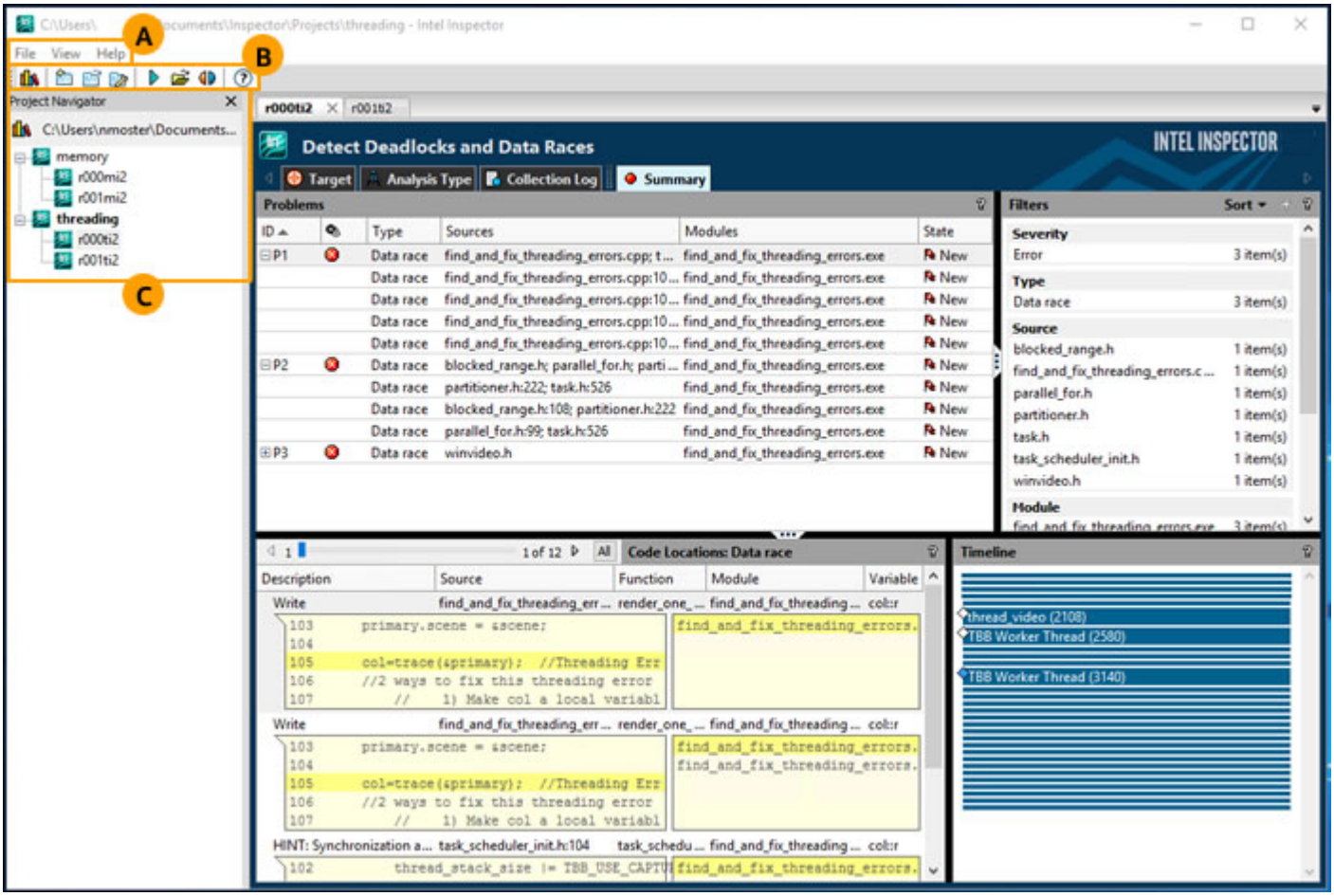
- Select the **Help** > **Intel Inspector** menu item and choose a required documentation format from the sub-menu.
- Click the drop-down control at the  toolbar icon and choose a documentation format.

See Also
View Results
Context Menus: Solution Explorer
Toolbar: Intel Inspector
Release Notes

Standalone Intel Inspector GUI

To access the standalone Intel® Inspector GUI, do one of the following:

- Run the `inspxe-gui` command.
- From the Windows* OS **Start** menu, choose **Intel Inspector [version]** .



The menu, toolbar, and **Project Navigator** offer different ways to perform many of the same functions.

A Use the menu to create projects and dynamic analysis results, import result archive files and results from other Intel error-detection products, open projects and results, compare results, configure projects, set various options, and access the product documentation.

B Use the toolbar to open the documentation; create, configure, and open projects; create dynamic analysis results; and open and compare results.

C Use the **Project Navigator**:

- **Tree** to see a hierarchical view of your projects and results based on the directory where the opened project resides.
- **Context menus** (right-click to open) to perform functions available from the menu and toolbar plus delete or rename a selected project or result, close all opened results, and copy various directory paths to the system clipboard.

See Also

View Results
Context Menus: Project Navigator
Toolbar: Intel Inspector

Startup Tasks

Before you start using the Intel® Inspector, determine if you want to change the default settings for:

About Choosing Baseline Results for Setting States

To avoid investigating the same issues over and over again, set the Intel Inspector to propagate state information from a baseline result the first time you open a new result.

Baseline Result Options

By default, the Intel Inspector propagates state information from the immediately previous result of the same analysis type. However, you may need to configure the Intel Inspector to propagate state information from a specific earlier result. This is most appropriate if your source code is undergoing branched development. You can also instruct the Intel Inspector to not propagate states from a baseline result, which sets the state of all issues in a new result to **New**.

Typical Usage Model

1. Use the default **Get problem states from previous result of same analysis type** baseline result option.
2. Run an analysis.
3. Use the filtering function to temporarily limit the displayed issues to only those that are **Not investigated**.
4. Set the state of each problem issue you investigate as:
   - **Confirmed** - Issue requires fixing but has not yet been fixed.
   - **Fixed** - Issue requires fixing and has been fixed.
   - **Not a Problem** - Issue does not require fixing.
   - **Deferred** - Delay further investigation.
5. The next time you rerun the analysis, verify the issues you expect to be fixed are indeed fixed.

Intel Inspector propagates state information from the baseline result when it determines an issue in a new result corresponds to an issue in the baseline result. For example, if you set the state for a problem in the baseline result to **Not a Problem**, the Intel Inspector sets the state for the corresponding problem in the new result to **Not a Problem**.

Caution
Intel Inspector may not recognize an issue as previously investigated when it propagates state information from a baseline result. This is more likely to happen when source code has undergone drastic changes between analysis runs.

Tip
Do not delete a result when you are finished interpreting it. You may need it to properly propagate state information to later results.

About Naming Results and Result Directories

You can control - at a cross-project level - how the Intel Inspector names future results and result directories. For example, you may choose a cryptic name for results you consider temporary and plan to delete, or you may choose a more descriptive name for long-term audit purposes.

About Displaying Application Output

When you run an analysis on an application, the Intel Inspector executes that application. You can send application output to the:

- Microsoft Visual Studio* output window
- **Collection Log** window
- Separate console window

Change Result and Result Directory Name Templates

You can control - at a cross-project level - how the Intel Inspector names future results and result directories. For example, you may choose a cryptic name for results you consider temporary and plan to delete, or you may choose a more descriptive name for long-term audit purposes.

To change the name template for future result and result directories:

1. From the:
   - Visual Studio* menu, choose **Tools** > **Options...**. In the **Options** dialog box, expand the **Intel Inspector <version>** folder and choose the **Result Location** page.

     NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

   - Standalone Intel Inspector GUI menu, choose **File** > **Options...**. In the **Options** dialog box, choose the **Result Location** page.
2. In the **Result name template** text box, supply a new naming convention.
3. Click the **OK** button.

Outcome: The Intel Inspector renames future results and result directories.

Caution
`@@@` represents the next available number. The name template must contain at least one but no more than eight @ symbols.

NOTE
When you change the name template, the Intel Inspector does not rename existing results or result directories.

See Also

Startup Tasks
Dialog Box: Options-Result Location
Intel Inspector Filenames and Locations

Choose Baseline Results for Setting States

To avoid investigating the same issues over and over again, set the Intel Inspector to propagate state information from a baseline result the first time you open a new result.

To choose a baseline result:

1. From the:
   - Visual Studio* menu, choose **Tools** > **Options...**. In the **Options** dialog box, expand the **Intel Inspector <version>** folder and choose the **State Management** page.

NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- Standalone Intel Inspector GUI menu, choose **File** > **Options...**. In the **Options** dialog box, choose the **State Management** page.

2. Select the appropriate radio button.
3. Choose the **OK** button.

Outcome: The first time you open a new result, the Intel Inspector propagates state information from the baseline result when it determines an issue in the new result corresponds to an issue in the baseline result.

See Also

Collaborate on Results
Startup Tasks
States
Dialog Box: Options-State Management

Display Application Output

When you run an analysis on an application, the Intel Inspector executes that application. You can send application output to the:

- Microsoft Visual Studio* output window
- **Collection Log** window
- Separate console window

To choose an output destination :

1.

- Visual Studio* menu, choose **Tools** > **Options...**. In the **Options** dialog box, expand the **Intel Inspector <version>** folder and choose the **General** page.

NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- Standalone Intel Inspector GUI menu, choose **File** > **Options...**. In the **Options** dialog box, choose the **General** page.

2. In the **Application output destination** group, select the appropriate radio button.
3. Choose the **OK** button.

Outcome: The next time you run an analysis, the application executes in the selected output destination.

NOTE
If you must interact with the application during execution, choose the separate console window option or use `stdin` redirection on the command line.

See Also

Startup Tasks
Dialog Box: Options-General

Set Up a Project

Build Your Application

Follow these guidelines to build applications that produce the most accurate and complete Intel Inspector analysis results:

- Use optimal compiler/link settings.
- Ensure applications create more than one thread before you run threading analyses.

About Using Optimal Compiler/Linker Settings

You can use the Intel® Inspector to analyze memory and threading errors in both debug and release modes of C++ and Fortran binaries; however, applications compiled/linked in debug mode using the following settings produce the most accurate and complete analysis results.

Compiler/Linker Property	Correct C/C++ Setting	Impact If Not Set Correctly
Debug information	Enabled (`/Zi` or `/ZI`)	Missing file/line information
Optimization	Disabled (`/Od`)	Incorrect file/line information
Dynamic runtime library	Selected (`/MD` or `/MDd`)	False positives or missing code locations
Basic runtime error checks	Disabled (do not use `/RTC`; **Default** option in Visual Studio* IDE)	False positives

Compiler/Linker Property	Correct Fortran Setting	Impact If Not Set Correctly
Debug information	Enabled (`/debug:full`)	Missing file/line information
Optimization	Disabled (`/Od`)	Incorrect file/line information
Dynamic runtime library	Selected (`/libs:dll/threads` or `libs:dll/threads/dbglibs`)	False positives or missing code locations
Basic runtime error checks	None (`/check:none`)	False positives

To learn more about options necessary to produce the most accurate and complete analysis results, see the following resources:

- Memory error analysis
- Threading error analysis

About Ensuring Applications Contain More Than One Thread

If you plan to run threading error analyses on systems with only one processor, you may need to take special steps to ensure applications create more than one thread:

Programming Model	Possible Strategies
oneAPI Threading Building Blocks (oneTBB)	When initializing an object of class `tbb::task_scheduler_init`, use the following function to force the creation of two or more threads: `initialize (int)`, where `int = 2` or higher. For more information, see http://www.threadingbuildingblocks.org/
Windows* API (Win32)	Use the following function to create as many threads as you need: `CreateThread` - Creates a thread to execute within the virtual address space of the calling process. For more information, see http://msdn.microsoft.com/en-us/library/
OpenMP* API	Use the following to force the creation of two or more threads: • `OMP_NUM_THREADS` runtime environment variable - Sets the maximum number of threads in the parallel region, unless overridden by an `omp_set_num_threads` function or `num_threads` clause. • `omp_set_num_threads` function - Sets the number of threads in subsequent parallel regions, unless overridden by a `num_threads` clause. • `num_threads` clause - Sets the number of threads in a thread team. For more information, see http://openmp.org/wp/openmp-specifications/

Configure Debug Mode for Applications in the Visual Studio* IDE

Before building applications in debug mode, configure debug mode with the optimal compiler and linker settings to produce the most accurate and complete Intel Inspector analysis results:

Compiler/Linker Property	Correct C/C++ Setting	Impact If Not Set Correctly
Debug information	Enabled (`/Zi` or `/ZI`)	Missing file/line information
Optimization	Disabled (`/Od`)	Incorrect file/line information
Dynamic runtime library	Selected (`/MD` or `/MDd`)	False positives or missing code locations
Basic runtime error checks	Disabled (do not use `/RTC`; **Default** option in Visual Studio* IDE)	False positives

Compiler/Linker Property	Correct Fortran Setting	Impact If Not Set Correctly
Debug information	Enabled (`/debug:full`)	Missing file/line information
Optimization	Disabled (`/Od`)	Incorrect file/line information
Dynamic runtime library	Selected (`/libs:dll/threads` or `libs:dll/threads/dbglibs`)	False positives or missing code locations
Basic runtime error checks	None (`/check:none`)	False positives

Configure Debug Mode for C/C++ Applications

1. Right-click the project in the **Solution Explorer** to display a context menu, then choose **Properties** to display the **Property Pages** dialog box.
2. In the **Configuration** drop-down list, choose **Debug**.
3. In the left pane, choose **Configuration Properties** > **C/C++**.
   - Choose **General** and verify the **Debug Information Format** field is set to **Program Database (/Zi)** or **Program Database for Edit & Continue (/ZI)**.
   - Choose **Optimization** and verify the **Optimization** field is set to **Disabled (/Od)**.
   - Choose **Code Generation**. Verify the **Runtime Library** field is set to **Multi-threaded DLL (/MD)** or **Multi-threaded Debug DLL (/MDd)** and the **Basic Runtime Checks** field is set to **Default**.
4. In the left pane, choose **Configuration Properties** > **Linker** > **Debugging** and verify the **Generate Debug Info** field is set to **Yes (/DEBUG)**.
5. Click the **OK** button to close the dialog box.

Configure Debug Mode for Fortran Applications

1. Right-click the project in the **Solution Explorer** to display a context menu, then choose **Properties** to display the **Property Pages** dialog box.
2. In the **Configuration** drop-down list, choose **Debug**.
3. In the left pane, choose **Configuration Properties** > **Fortran**.
   - Choose **Debugging** and verify the **Debug Information Format** field is set to **Full (/debug:full)**.
   - Choose **Optimization** and verify the **Optimization** field is set to **Disable (/Od)**.
   - Choose **Libraries** and verify the **Runtime Library** field is set to **Multithread DLL (/libs:dll/threads)** or **Debug Multithread DLL (libs:dll/threads/dbglibs)**.
   - Choose **Run-time** and verify the **Runtime Error Checking** field is set to **None (/check:none)**.
4. In the left pane, choose **Configuration Properties** > **Linker** > **Debugging** and verify the **Generate Debug Info** field is set to **Yes (/DEBUG)**.
5. Click the **OK** button to close the dialog box.

Set Up Applications to Build in Debug Mode in the Visual Studio* IDE

Prerequisite: Configure debug mode with the optimal compiler and linker settings to produce the most accurate and complete Intel Inspector analysis results:

Compiler/Linker Property	Correct C/C++ Setting	Impact If Not Set Correctly
Debug information	Enabled (`/Zi` or `/ZI`)	Missing file/line information
Optimization	Disabled (`/Od`)	Incorrect file/line information
Dynamic runtime library	Selected (`/MD` or `/MDd`)	False positives or missing code locations
Basic runtime error checks	Disabled (do not use `/RTC`; **Default** option in Visual Studio* IDE)	False positives

Compiler/Linker Property	Correct Fortran Setting	Impact If Not Set Correctly
Debug information	Enabled (`/debug:full`)	Missing file/line information
Optimization	Disabled (`/Od`)	Incorrect file/line information
Dynamic runtime library	Selected (`/libs:dll/threads` or `libs:dll/threads/dbglibs`)	False positives or missing code locations

Compiler/Linker Property	Correct Fortran Setting	Impact If Not Set Correctly
Basic runtime error checks	None (`/check:none`)	False positives

To set application to build in debug mode:

1. Right-click the project in the **Solution Explorer** to display a context menu, then choose **Properties** to display the **Property Pages** dialog box.
2. Click the **Configuration Manager** button.
3. In the **Active solution configuration** drop-down list, choose **Debug**.
4. Click the **Close** button to close the **Active solution configuration** dialog box.
5. Click the **OK** button to close the **Property Pages** dialog box.

Outcome: Whenever you build this application, it builds in debug mode.

Choose Projects

Intel Inspector is based on a project paradigm and requires that you create or open a project to enable analysis features.

Think of a project as a:

- Compiled application
- Collection of configurable attributes, including suppression rules and search directories
- Container for analysis results

Choose Projects in the StandaloneIntel Inspector GUI

Intel Inspector is based on a project paradigm and requires that you create or open a project to enable analysis features.

To choose an existing project:

Do one of the following:

- From the Standalone Intel Inspector GUI menu:
  - Choose **File** > **Open** > **Project...**. Browse to and select the appropriate `config.inspxeproj` file, then click the **OK** button.
  - Choose **File** > **Recent Projects**, then choose the appropriate `config.inspxeproj` file.
- From a command prompt: Supply the appropriate `config.inspxeproj` file as an argument to the `inspxe-gui` command. This action launches the Standalone Intel Inspector GUI if it is not already open.

Outcome:

- Intel Inspector displays the project name in the title bar.
- Intel Inspector features are enabled.

Tip
Some possible next steps include:

- Review or reconfigure project properties.
- Configure an analysis.
- Open an existing result.

See Also

Configure Analysis
Configure Projects
View Results
Intel Inspector Filenames and Locations

Choose Projects in the Visual Studio* IDE

Intel Inspector is based on a project paradigm and requires that you create or open a project to enable analysis features.

To choose a project:

1. Open an existing solution so that it appears in the Visual Studio* **Solution Explorer**.
2. Right-click the appropriate project in the **Solution Explorer** to display a context menu, then choose **Set as StartUp Project**.

Outcome: The Intel Inspector features are enabled.

Tip
Some possible next steps include:

- Clean and rebuild the application using recommended compiler/linker options.
- Review or reconfigure project properties.
- Configure an analysis.
- Open an existing result.

NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

See Also

Build Your Application

Create Projects in the StandaloneIntel Inspector GUI

Intel Inspector is based on a project paradigm and requires that you create or open a project to enable analysis features.

To create a new project:

1. Choose **File** > **New** > **Project...**.
2. In the **Create a Project** dialog box, supply a project directory name and location.
3. Click the **Create Project** button to display the **Target** tab on the **Project Properties** dialog box.
4. Do one of the following in the **Application** field:
   - Type the absolute path to an application.
   - Choose the  icon to browse to and select an application.
   - Click the field arrow to choose a previously selected application.

Tip
Recommended next step: Configure the project.

See Also

Configure Projects
Dialog Box: Create a Project
Dialog Box: Project Properties-Target
Intel Inspector Filenames and Locations

Create Solutions/Projects From Applications in the Visual Studio* IDE

Intel Inspector is based on a project paradigm and requires that you create or open a project to enable analysis features.

To create a project:

1. Choose **File > Open > Project/Solution**.
2. In the **Open Project** dialog box, browse to and select an application.

Outcome:

- The Visual Studio* IDE creates a solution/project.
- Intel Inspector features are enabled.

Tip
Recommended next step: Configure the project.

NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

Configure Projects

When you set basic target application properties to configure the Intel Inspector projects, make sure you choose small, representative data sets to produce the most accurate and complete analysis results at the least cost.

You can also:

- Set advanced target application properties.
- Manage suppression rules.
- Configure a project to search non-standard directories.

Set Basic Target Application Properties

Most basic application properties are not specific to Intel Inspector projects and their purpose should be self-evident. Some are inheritable from the Visual Studio* property pages of the Startup project (**Configuration Properties** > **Debugging**). The following are of special note:

Use This	To Do This
Microsoft* runtime environment drop-down list	• Automatically detect the Microsoft* runtime environment based on binary executable type (choose **Auto**). • Potentially speed up collection by excluding application managed code from inspection (choose **Native only**). • Potentially speed up collection by excluding application native code from inspection (choose **Managed only**). • Inspect all code, regardless of whether it is native or managed (choose **Mixed**).

Use This	To Do This
	NOTE • Microsoft .NET* 3.5 software support is deprecated in the Intel Inspector and will be removed after August, 2020. This deprecation does not apply to the Intel® VTune™ Profiler. See *Release Notes* for details. • During threading analysis, the Intel Inspector analyzes both native and managed code. • During memory analysis, the Intel Inspector analyzes only native code. If you choose **Mixed**, there is no analysis of the managed portion of your code. • During threading or memory analysis, the Intel Inspector supports interactive debugging only for pure native applications. If you choose **Mixed** or **Managed only**, the debug options are disabled.
Store result... radio buttons	Store results in the default project location or a custom location. **Tip** Consider storing results in - and linking to - a custom location if: • You have limited space in the project directory. • You customarily back up the project directory but do not want to back up results.
Result location field (read-only)	Verify the result location and current result directory name template.

Choose Small, Representative Data Sets

When you run an analysis, the Intel Inspector executes an application. Data set size and workload have a direct impact on application execution time and analysis speed.

For example, it takes longer to process a 1000x1000 pixel image than a 100x100 pixel image. A possible reason for the longer processing time: You may have loops with an iteration space of 1...1000 for the larger image, but only 1...100 for the smaller image. The exact same code paths may be executed in both cases. The difference is the number of times these code paths are repeated.

You may control analysis cost without sacrificing completeness by removing this kind of redundancy from your data set.

Instead of choosing large, repetitive data sets, choose small, representative data sets that fully create threads with minimal to moderate work per thread. *Minimal to moderate* means just enough work to demonstrate all the different behaviors a thread can perform.

Your objective: In as short a runtime period as possible, execute as many paths and the maximum number of tasks (parallel activities) as you can afford, while minimizing the redundant computation within each task to the bare minimum needed for good code coverage.

Data sets that run a few seconds are ideal. Create additional data sets to ensure all your code is inspected.

NOTE
If you plan to run threading analyses and your programming model uses dynamic scheduling, ensure sufficient work is available for assignment to more than one thread.

Set Advanced Target Application Properties

Advanced target application properties are specific to Intel Inspector projects.

Use This	To Do This
Suppressions radio buttons	• Not collect result data impacted by the suppression rules in files and directories specified in the **Suppressions** tab of the **Project Properties** dialog box (choose **Apply suppressions**). • Ignore all suppression rules (choose **Do not apply suppressions**).
Child application field	Inspect a file that is not the starting application. For example: Inspect an `.exe` file (identified in this field) called by a script (identified in the **Application** field).
Enable collection progress information checkbox	Display thread activity during collection to confirm the application is still executing.
Modules radio buttons, **Modify** button, and field	Potentially speed up collection by limiting application module(s) for inspection. You can limit by inclusion or exclusion. For example, you can: • Inspect specific modules and disable inspection of all other modules (click the **Include only the following module(s)** radio button and choose the modules). • Disable inspection of specific modules and inspect all other modules (click the **Exclude the following module(s)** radio button and choose the modules). **NOTE** By default, the Intel Inspector inspects all modules in the application.

Manage Suppression Rules

You (and your development team) will always know more about your code than the Intel Inspector can ever know. For example:

- Your team lead may know of error-prone third-party code.
- Your team architect may know specific code passages are correct as coded.
- You may know you are currently fixing a specific bug.

Suppressing known issues based on rules you define can improve your productivity by helping you focus on only those issues that currently require your attention.

You can manage the suppression rules applied to each project during analysis by adding information to or removing information from the **Suppressions** tab of the **Project Properties** dialog box.

Search Non-standard Directories

Intel Inspector is designed to search all standard directories for the supporting files necessary to execute an application during analysis and manage result data after analysis.

You can also configure the Intel Inspector to search non-standard directories. This can be useful when:

- The binary is built outside the Visual Studio* IDE.
- Specific PATH settings are required for correct operation.
- Source files reside somewhere other than where debug information indicates.
- Debug information files are not located with binary files.

Alert the Intel Inspector to search non-standard directories using the **Binary/Symbol Search** tab and the **Source Search** tab of the **Project Properties** dialog box.

NOTE
If you take full advantage of the solutions/projects paradigm to create application software in the Visual Studio* IDE, searching for non-standard directories should be unnecessary.

See Also

Binary/Symbol Search and Source Search Locations
Suppressions Support
Manage Suppression Rules
Search Non-standard Directories
Set Project Properties-Advanced
Set Target Application Properties - Basic
Intel Inspector Filenames and Locations

Binary/Symbol Search and Source Search Locations

When using the Standalone GUI:Intel Inspector:

- If you specify binary and symbol locations to search using the **Binary/Symbol Search** tab, they will be searched *in addition* to the default binary and symbol locations.
- If you specify source locations to search using the **Source Search** tab, they will be searched *in addition* to the default source search locations.

Binary/Symbol Search Locations

Intel Inspector searches binary and symbol files in default locations and in locations specified in the **Binary/Symbol Search** tab (if specified).

The following lists describe the order and default locations that are searched. As indicated below, some directory searches examine the specified directory and its subdirectories, while other searches do not examine its subdirectories.

The search order on **Windows* OS** systems is the following:

1. Search for binary and symbol files in the directories specified in the **Binary/Symbol Search** tab and their subdirectories (if enabled in the tab).
2. Search for symbol files in the directories near the related (corresponding) binary file(s) just found, such as a library:
   - Check in the directory of the corresponding binary file, using the corresponding name.
   - Check in the directory of the corresponding binary file, using a related name. For example, for `app.dll` where a file `app_x86.pdb` is present, also search for file `app.pdb`.
3. Search for files in Visual Studio project directories.

   For symbol files, also search using symbol server paths specified in the **Binary/Symbol Search** tab in the following notation: `srv*C:\localsymbols*http://msdl.microsoft.com/download/symbols` and/or provided in Visual Studio **Tools** > **Options** > **Debugging** > **Symbols**.
4. Search for binary files in this standard Windows OS system directory:

`%SYSTEMROOT%\system32\drivers` (subdirectories are not searched)

5. Search for symbol files in these standard Windows OS system directories:
   - All directories specified in the environment variable `_NT_SYMBOL_PATH` (subdirectories are not searched)
   - `srv*%SYSTEMROOT%\symbols` (symbol downstream or cache path)
   - `%SYSTEMROOT%\symbols\dll` (subdirectories are not searched)

The search order on **Linux* OS** systems is the following:

1. Search for binary and symbol files in the directories specified in the **Binary/Symbol Search** tab and their subdirectories (if enabled in the tab).
2. Search for binary files in directories from the collected result that provide an absolute path name. If the file name `vmlinux` is present, search these directories:
   - `/usr/lib/debug/lib/modules/`uname -r`/vmlinux`
   - `/boot/vmlinuz-`uname -r``
3. Search for symbol files in the directories near the related (corresponding) binary file(s) just found, such as a library:
   - Check in the directory of the corresponding binary file, using the corresponding name.
   - Check in the directory of the corresponding binary file, using a related name. For example, for `app.dll` where a file `app_x86.pdb` is present, also search for file `app.pdb`.
   - Search in the `.debug` subdirectory.
4. Search for binary files in these standard Linux OS system directories:
   - `/lib/modules` (subdirectories are not searched)
   - `/lib/modules/`uname -r`/kernel` (subdirectories are searched)
5. Search for symbol files in these standard Linux OS system directories:
   - `usr/lib/debug`(subdirectories are not searched)
   - `/usr/lib/debug` with appended path to the corresponding binary file, such as `/usr/lib/debug/usr/bin/ls.debug`

Source Search Locations

A limited set of default source locations are used in addition to the locations specified in the **Source Search** tab.

The following list describes the order and default locations that are searched. As indicated below, some directory searches examine the specified directory and its subdirectories, while other searches do not examine its subdirectories.

1. Search for source files in the directories specified in the **Source Search** tab.
2. Search for source files in directories from the collected result that provide an absolute path name.
3. **When using Microsoft Visual Studio***: Search for source files in Visual Studio project directories.
4. **On Linux OS systems**: Search for source files in these standard Linux locations (does not search subdirectories):

   `/usr/src`

   `/usr/src/linux-headers-`uname -r``

See Also

Search Non-standard Directories

Choose Data Sets

When you run an analysis, Intel® Inspector executes an application. Data set size and workload have a direct impact on application execution time and analysis speed.

There are three criteria, depending on which you can choose the smallest data sets available:

- Intel Inspector maintains shadow memory structure to keep extra information on allocated memory blocks and/or synchronization primitives used by the application. Depending on the selected analysis type, the size of those extra details can consume significant amount of process memory in addition to application memory usage. The system should have enough physical memory to run application with extra memory overhead.
- During threading analysis, Intel Inspector verifies correctness of every potentially shared memory access. If address which is being verified is simultaneously accessed by another thread, the checks will be serialized. The more such intersections at time, the more overhead analysis will add to the application. Reducing number of threads (e.g. by defining OMP_NUM_THREAD variable for OpenMP), might significantly reduce analysis time without significant loss in analysis quality.
- Intel Inspector intercepts those call for analysis and serialize it in many cases across threads. Reducing frequency of those events in the application will decrease analysis overhead in general and can improve performance of the application when it is run without analysis.

To choose a data set:

Do one of the following:

To Choose a Data Set	Do This
Before running an analysis	Visual Studio* IDE only: **1.** Right-click the project in the **Solution Explorer** to display a context menu, then choose **Properties > Configuration Properties > Debugging**. **2.** Edit **Command Arguments** to select the appropriate data set.
	1. Edit source code to select the appropriate data set. **2.** Rebuild the application.
During analysis	Interact with the application during execution to select the appropriate data set.

Manage Suppression Rules

You can manage the suppression rules applied to each project during Intel Inspector analysis by:

- Opening the Suppressions Tab of the Project Properties Dialog Box
- Adding a .sup file
- Adding a directory that contains (or will contain) .sup files
- Removing a .sup file or directory containing .sup files
- Removing a specific rule

> **NOTE**
> To apply suppression rules during analysis, select the **Apply Suppressions** radio button on the **Target** tab of the **Project Properties** dialog box.

Open the Suppressions Tab of the Project Properties Dialog Box

1. From the:

- Visual Studio* menu, choose **Project** > **Intel Inspector <version> Project Properties...**.

> **NOTE** In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- Standalone Intel Inspector GUI menu, choose **File** > **Project Properties...**.

2. In the **Project Properties** dialog box, choose the **Suppressions** tab.

Add a .sup File

1. Click the **Add File** button.
2. Browse to and choose a `.sup` file.
3. Click the **Open** button.

Outcome: The Intel Inspector updates the rules in the **Suppression rule list** region accordingly.

Add a Directory That Contains (or Will Contain) .sup Files

1. Click the **Add Directory** button.
2. Do one of the following:
   - Browse to and choose an existing directory.
   - Create a new directory to which you plan to save `.sup` files.
3. Click the **OK** button.

Outcome: The Intel Inspector:

- Displays the path to a `*.sup` file mask in the **Suppression files** region.
- Updates the rules in the **Suppression rule list** region accordingly.

NOTE
Intel Inspector does not search recursively in directories for files identified by a `*.sup` mask.

Remove a .sup File or Directory Containing .sup Files

1. Click a file or mask in the **Suppression files** region.
2. Click the **Remove** button directly below the **Suppressions files** region.

Outcome: The Intel Inspector updates the rules in the **Suppression rule list** region accordingly.

NOTE

- Removing a file or directory does not delete the file or directory from your file system.
- You cannot remove the project suppression directory.

Remove a Specific Rule

1. Click a rule in the **Suppression rule list** region.
2. If desired, click the **View...** button to view the stack frame(s) in the rule. When you are finished viewing, close the **View Stack** dialog box.
3. Click the **Remove** button directly below the **Suppression rule list** region.

Caution
Removing a rule from the list deletes the rule from your file system.

See Also

Suppressions Support
Dialog Box: Project Properties-Suppressions
Dialog Box: Project Properties-Target
Dialog Box: View Stack

Context Menus: Solution Explorer
Intel Inspector Filenames and Locations

Search Non-standard Directories

Intel Inspector is designed to search all standard directories for the supporting files necessary to execute an application during analysis and manage result data after analysis. You can also configure the Intel Inspector to search non-standard directories:

- Add a directory to the search list.
- Change the directory search order.
- Remove a directory from the search list.

> **NOTE**
> If you take full advantage of the solutions/projects paradigm to create application software in the Visual Studio* IDE, searching for non-standard directories should be unnecessary.

Add a Directory to the Search List

1. From the:
   - From the Visual Studio* menu, choose **Project** > **Intel Inspector <version> Project Properties...**.

   > **NOTE** In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

   - From the Standalone Intel Inspector GUI menu, choose **File** > **Project Properties...**.
2. In the **Project Properties** dialog box, choose the **Binary/Symbol Search** or **Source Search** tab.
3. In the **Search Directories** column, click the

   button to browse to and select a directory.
4. To add another search directory, click **Add new search location** in the **Search Directories** column to display another

   button.
5. Repeat steps 3 and 4 as necessary. When you are finished, click the **OK** button.

Outcome:

- During analysis, the Intel Inspector searches the listed and standard directories for the supporting files necessary to execute the application.
- After analysis, the Intel Inspector searches the listed and standard directories for the supporting files necessary to manage result data.

Change the Directory Search Order

1. Click the appropriate directory.
2. Click the

   or

   button.

Remove a Directory from the Search List

1. Click the appropriate directory.
2. Click the

button.

See Also

Binary/Symbol Search and Source Search Locations
Intel Inspector Filenames and Locations
Dialog Box: Project Properties-Binary/Symbol Search
Dialog Box: Project Properties-Source Search
Context Menus: Solution Explorer

Set Project Properties-Advanced

Advanced target application properties are specific to Intel Inspector projects and help you potentially speed up collection and improve productivity.

- Open the Target Tab of the Project Properties Dialog Box.
- Potentially speed up collection (include/exclude modules during analysis).
- Improve productivity (suppress known issues).

Open the Advanced Region on the Target Tab of the Project Properties Dialog Box for the Current Project

1. - From the Visual Studio* menu, choose **Project** > **Intel Inspector <version> Project Properties...**.

   NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

   - From the standalone Intel Inspector GUI menu, choose **File** > **Project Properties...**.
2. Click the

icon.

NOTE
Intel Inspector automatically opens the **Target** tab of the **Project Properties** dialog box when you create a project.

Potentially Speed up Collection

Click the **Include the following module(s)** or **Exclude the following module(s)** radio button, then identify the application (or child application) module(s) to include in or exclude from inspection:

1. Click the **Modify** button.
2. In the **Modules** column, click the

button to browse to a directory. Click the appropriate module to select a single module; use Shift+click or Ctrl+click to select multiple modules.
3. To identify modules in another directory, click **Add new line** in the **Modules** column to display another

button.

4. Repeat steps 2 and 3 as necessary. When you are finished, click the **OK** button.

Improve Productivity

Use the **Suppressions** radio buttons to improve your productivity by helping you focus on only those issues that currently require your attention.

To Do This	Do This
Not collect result data impacted by the suppression rules in files and folders specified in the **Suppressions** tab of the **Project Properties** dialog box.	Choose **Apply suppressions**.
Ignore all suppression rules.	Choose **Do not apply suppressions**.

See Also

Suppressions Support
Dialog Box: Project Properties-Target
Intel Inspector Filenames and Locations

Set Target Application Properties - Basic

Most basic target application properties are not specific to Intel Inspector projects and their purpose should be self-evident.

- Open the Target Tab of the Project Properties Dialog Box.
- Inherit Visual Studio* application properties.
- Set application execution parameters.
- Set application launch directory.
- Set user-defined environment variables.
- Save results to a custom location.

Open the Target Tab of the Project Properties Dialog Box for the Current Project

From the:

- Visual Studio* menu, choose **Project** > **Intel Inspector <version> Project Properties...**.

  NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- Standalone Intel Inspector GUI menu, choose **File** > **Project Properties...**.

  NOTE
  Intel Inspector automatically opens the **Target** tab of the **Project Properties** dialog box when you create a project in the Standalone Intel Inspector GUI.

Inherit Visual Studio* Project Properties

Select the **Inherit settings from Visual Studio* project** checkbox.

Outcome: The Intel Inspector:

- Populates all basic application properties from Visual Studio* property pages of the StartUp project (**Configuration Properties** > **Debugging**).
- Disables the basic settings section of the dialog box.

Set Application Execution Parameters

In the **Application parameters** field, do one of the following:

- Type directly in the field.
- Choose the **Modify** button to open an advanced editor, where you can transfer absolute paths to files or directories to prevent typographical errors.
- Click the field arrow to choose a previously applied set of application parameters.

Set the Application Launch Directory

In the **Working directory** field, do one of the following:

- Type the absolute path directly in the field.
- Choose the **Modify** button to browse to and select a directory.
- Click the field arrow to choose a previously applied directory.

Set User-defined Environment Variables

In the **User-defined environment variables** field:

1. Click the **Modify** button.
2. In the **Variables** column, click **Add new line** and supply a variable name.
3. In the corresponding **Values** column, supply a variable value.
4. Repeat steps 2 and 3 as necessary. When you are finished, click the **OK** button.

Save Results to a Custom Location

1. Choose the **Store result in (and create link file to) another directory** radio button.
2. Type the absolute path directly in the field or click the **Browse** button to browse to and select a directory.

See Also

Dialog Box: Project Properties-Target

Collect Results

Configure Analysis

Configure the Intel Inspector analyses using preset and custom analysis types.

You can also configure the Intel Inspector for interactive debugging during analysis.

About Using Analysis Types

Intel Inspector offers a range of preset memory and threading analysis types to help you control analysis scope and cost:

- Analysis types with the narrowest scope minimize the load on the system and the time and resources required to perform the analysis; however, they detect the narrowest set of errors and provide minimal details.
- Analysis types with the widest scope maximize the load on the system and the time and resources required to perform the analysis; however, they detect the widest set of errors and provide context and the maximum amount of detail for those errors.

Some settings in each preset analysis type are configurable. If the combination of settings in a preset analysis type almost meets your needs, try fine-tuning these configurable settings. For example:

- When you use preset memory error analysis types, you can choose to detect resource leaks, report still-allocated memory at application exit, and adjust stack frame depth.
- When you use preset threading error analyses types, you can choose to terminate on deadlock and adjust stack frame depth.

If the combination of analysis type settings in the preset analysis types does not meet your needs at all, try creating a new custom analysis type based on the currently selected analysis type.

Tip
Use analysis types iteratively. Start with a narrow scope to verify the application is set up correctly and set expectations for analysis duration. Widen the scope only if you need more answers and you can tolerate the increased cost.

About Configuring for Interactive Debugging During Analysis

Sometimes simply knowing the location of a problem is not enough. So the Intel Inspector provides the opportunity to investigate more deeply with an interactive debugging session during analysis.

When you run an interactive debugging session during analysis, the Intel Inspector halts execution at a detected problem. This is more efficient than simply setting a code breakpoint at a reported problem location because the code could execute thousands of times before the conditions that produced the problem occur.

Intel Inspector offers two configuration options:

- **Enable debugger when problem detected** - Allows investigation of every problem detected in an interactive debugging session. Typical usage scenario: You want to investigate the state of the application for every problem detected.
- **Select analysis start location with debugger** - Delays problem detection until a selected point in the application. Typical usage scenario: You configured the project to limit the application modules for inspection, but you need to narrow analysis focus even further. This option lets you choose an arbitrary location to turn on error detection, allowing faster execution until that point.

NOTE
You can also use the **Debug This Problem** function, a context menu option in a result **Problems** pane, to allow quick investigation of problems of interest. In this case, there is no need to manually configure for interactive debugging and rerun an analysis; the **Debug This Problem** function automatically launches a new analysis that is optimized to find the selected problems. Typical usage scenario: After reviewing a problem using result data provided by the Intel Inspector, you discover you need more application state information at the time the problem occurred, such as the contents of variables.

See Also

Investigate Problems Using Interactive Debugging
Memory Error Analysis Types
Threading Error Analysis Types
Manage Custom Analysis Types
Intel Inspector Filenames and Locations

Memory Error Analysis Types

Intel Inspector offers a range of preset memory analysis types to help you control analysis scope and cost:

- Analysis types with the narrowest scope minimize the load on the system and the time and resources required to perform the analysis; however, they detect the narrowest set of errors and provide minimal details.
- Analysis types with the widest scope maximize the load on the system and the time and resources required to perform the analysis; however, they detect the widest set of errors and provide context and the maximum amount of detail for those errors.

Tip

- Use analysis types iteratively. Start with a narrow scope to verify the application is set up correctly and set expectations for analysis duration. Widen the scope only if you need more answers and you can tolerate the increased cost.
- Estimated collection time may be 2 to 160 times longer than normal application execution time.
- Data set size and workload have a direct impact on application execution time and analysis speed.

Some settings in each preset analysis type are configurable. If the combination of settings in a preset analysis type almost meets your needs, try fine-tuning these configurable settings.

If the combination of analysis type settings in the preset analysis types does not meet your needs at all, try creating a new custom analysis type based on the currently selected analysis type.

Memory Error Configuration Settings for Each Preset Analysis Type

The following table shows the settings (in alphabetical order) for each preset analysis type.

- *N/A* means not applicable for the preset analysis type (which means any value displayed in the GUI is greyed out)
- *Configurable* means you can change the setting without creating a custom analysis type.
- Renamed/split/combined configuration settings show previous manifestations.

Configuration Settings / Preset Memory Error Analysis Types	Detect Leaks (Narrow Scope)	Detect Memory Problems	Locate Memory Problems (Wide Scope)
Analyze stack accesses	N/A	No	No (configurable)
Defer memory deallocation (previously called Byte limit before reallocation)	N/A	N/A	1 Mb
Detect invalid memory accesses (split from Detect invalid/uninitialized accesses)	No	Yes	Yes
Detect leaks at application exit (previously called Detect memory leaks upon application exit)	Yes (configurable)	Yes (configurable)	Yes (configurable)
Detect resource leaks	Yes (configurable)	Yes (configurable)	Yes (configurable)
Detect still-allocated memory at application exit (previously called Report still-allocated memory at application exit)	Yes (configurable)	Yes (configurable)	Yes (configurable)
Detect uninitialized memory reads (split from Detect invalid/uninitialized accesses)	N/A	No (configurable)	No (configurable)

Configuration Settings / Preset Memory Error Analysis Types	Detect Leaks (Narrow Scope)	Detect Memory Problems	Locate Memory Problems (Wide Scope)
Enable enhanced dangling pointer check	N/A	No	Yes
Enable guard zones	N/A	No	Yes
Enable memory growth detection (previously called Enable interactive memory growth detection)	No (configurable)	Yes (configurable)	Yes (configurable)
Enable on-demand leak detection (previously called Enable on-demand memory leak detection)	No (configurable)	Yes (configurable)	Yes (configurable)
Guard zone size (previously called Guard zone byte size)	N/A	N/A	32 bytes
Maximum number of leaks shown in result (previously called Maximum memory leaks)	100,000	100,000	100,000
Remove duplicates	Yes	Yes	Yes (configurable)
Revert to previous uninitialized memory algorithm (not recommended)	N/A	No (configurable)	No (configurable)
Stack frame depth	8 (configurable)	8 (configurable)	16 (configurable)

Memory Error Analysis Type Configuration Setting Descriptions

The following table describes the purpose, usefulness, and *cost* (low, medium, high, or proportional in terms of time and resources) for each analysis type configuration setting. (The settings are listed in alphabetical order.)

Setting	Purpose, Usefulness, and Cost
Analyze stack accesses	Available only if **Detect invalid memory accesses** is selected. Select to analyze invalid and uninitialized accesses to thread stacks. Selecting is useful when: • You want as thorough an analysis as possible. • An application calls `alloca()`. High **cost**. Recommendation: • Select the first time you analyze an application and periodically thereafter. • Select to analyze automatic variables.

Setting	Purpose, Usefulness, and Cost
Defer memory deallocation (previously called Byte limit before reallocation)	Available only if **Detect invalid memory accesses** and **Enable enhanced dangling pointer check** are selected. Select to have the Intel Inspector prevent freed memory blocks from immediately returning to the pool of available memory. Selecting is useful for discovering if an application tries to use memory after freeing it. High **cost** if an application is performing many allocations/deallocations. Recommendation: Select to improve analysis quality if the cost is not too high.
Detect invalid memory accesses (split from Detect invalid/ uninitialized accesses)	Select to detect problems where a read or write instruction references memory that is logically or physically invalid. Selecting is useful to ensure an application accesses only valid memory. Medium **cost**. Recommendation: Select. **NOTE** May change application behavior by initializing memory that may normally be uninitialized. If your application reads this normally uninitialized memory, it may: • Simply miscalculate a value. • Treat the memory as a pointer, deference it, and crash during analysis.
Detect leaks at application exit (previously called Detect memory leaks upon application exit)	Select to report typical memory leaks in which the application allocates a memory block, never releases it, and doesn't keep a pointer to the block (e.g. **unreachable memory blocks**). Selecting is useful when an application: • Runs out of memory. • Appears to be using more memory than expected. Extremely low **cost** – especially if used only with **Remove duplicates** selected. Recommendation: Select.
Detect resource leaks	Select to detect open kernel and GDI handles when the application ends. For example, the application may open a file, get its handle, but never close or release that handle until it stops running. On Windows, GDI resources are limited, and the application may experience some drawing issues if it uses more than ~10,000 of these types of handles at once (pen, bitmap, brush, etc.) Selecting is useful if you see constant growth in acquired kernel and GDI objects in the system task manager for your process. Low **cost**. Recommendation: Select unless you want to focus on memory problems exclusively.
Detect still-allocated memory at application exit (previously called	Available only if **Detect leaks at application exit** is selected. Select to report typical memory leaks in which the application allocates a memory block, doesn't deallocate it, but a valid variable still holds a pointer to that block when the application ends (e.g. **reachable memory blocks**).

Setting	Purpose, Usefulness, and Cost
Report still-allocated memory at application exit)	**Cost** is proportional to the number of memory blocks still allocated when the application stops executing. Recommendation: Select to investigate memory growth.
Detect uninitialized memory reads (split from Detect invalid/ uninitialized accesses)	Available only if **Detect invalid memory accesses** is selected. Select to detect problems where a read instruction accesses an uninitialized memory location. Selecting is useful when an application: • Exhibits unexpected behavior. • Shows evidence of uninitialized values in computations. High **cost**. Recommendation: Deselect. **NOTE** May change application behavior by initializing memory that may normally be uninitialized. If your application reads this normally uninitialized memory, it may: • Simply miscalculate a value. • Treat the memory as a pointer, deference it, and crash during analysis.
Enable enhanced dangling pointer check	Available only if **Detect invalid memory accesses** is selected. Select to detect if an application is trying to access memory after it was logically freed. May be higher **cost** if an application is performing many allocations/deallocations, and the **Defer memory deallocation** list value is smaller than the amount of memory the application allocates. Recommendation: Select when an application exhibits unexpected behavior you suspect may be caused by a dangling pointer. **NOTE** • Changes application behavior by intercepting calls to deallocators and deferring actual deallocation. • Enhanced dangling pointer check is not supported for `new`/`delete` and `new[]`/`delete[]` allocation/deallocation pairs.
Enable guard zones	Available only if **Detect invalid memory accesses** is selected. Use in conjunction with **Guard zone size** to show offset information if the Intel Inspector detects memory use beyond the end of an allocated block. Selecting is useful when: • An application exhibits unexpected behavior. • You need more context about heap allocations to interpret **Invalid memory access** problems.

Setting	Purpose, Usefulness, and Cost
	Cost is proportional to number of allocations. Recommendation: Select unless: • Intel Inspector runs out of memory. • An application becomes destabilized. **NOTE** • May change application behavior. • Increases the amount of memory the Intel Inspector uses. • Intel Inspector creates guard zones only at the end of allocated space, not at the start of allocated space.
Enable memory growth detection (previously called Enable interactive memory growth detection)	Select to enable buttons in the GUI that let you send commands during application execution. This will show you a list of **reachable and unreachable memory blocks** for a time segment. Selecting is useful for modeling memory usage patterns and ensuring a transactional application deallocates all memory allocations after a transaction completes. Use in conjunction with the **Reset Growth Tracking** and **Measure Growth** buttons during analysis. Low **cost**.
Enable on-demand leak detection (previously called Enable on-demand memory leak detection)	Select to enable buttons in the GUI that let you send "leak" commands. This will show you a list of **unreachable memory blocks** for a time segment. Selecting is useful for checking for memory leaks in an application that never exits, or in only the portion of an application for which you are responsible. Use in conjunction with the **Reset Leak Tracking** and **Find Leaks** buttons during analysis. **Cost** is proportional to the number of allocations.
Guard zone size (previously called Guard zone byte size)	Available only if **Detect invalid memory accesses** and **Enable guard zones** are selected. Use in conjunction with **Enable guard zones** to set the number of bytes beyond the allocated block of memory the Intel Inspector reserves to identify **Invalid memory access** problems related to the allocation. Setting is useful when: • An application exhibits unexpected behavior. • You need more context about heap allocations to interpret **Invalid memory access** problems. **Cost** is proportional to number of allocations. Recommendation: Set unless: • Intel Inspector runs out of memory. • An application becomes destabilized.

Setting	Purpose, Usefulness, and Cost
	NOTE • May change application behavior. • Increases the amount of memory the Intel Inspector uses. • Intel Inspector creates guard zones only at the end of allocated space, not at the start of allocated space.
Maximum number of leaks shown in result (previously called Maximum memory leaks)	Use to set the maximum number of leaks the Intel Inspector shows in a result after analysis is complete. A zero setting shows all detected memory leaks. **Cost** is proportional to the number of leaks. Recommendation: Use the default value unless you want an exhaustive list of all leaks. **Tip** Even the default value can generate an unmanageable number of leaks. Consider sorting the displayed memory leaks by **Object Size**, fixing the largest leaks, and then re-inspecting your application. Or use the on-demand leak detection feature to narrow your focus and eat the elephant one bite at a time.
Remove duplicates	Deselect to show all occurrences of a detected problem in the **Code Locations** pane. Deselecting is: • Useful when you need to fully visualize all threads and problem occurrences in relation to time • Low **cost** in terms of time; however, the number of duplicate errors could crowd out the number of unique errors. Recommendation: Select.
Revert to previous uninitialized memory algorithm (not recommended)	Available only if **Detect uninitialized memory reads** is selected. The current algorithm for detecting uninitialized memory reads decreases false positives but increases analysis time and memory overhead. Select to use the previous version of the algorithm. Recommendation: Deselect.
Stack frame depth	Use to provide more or less call stack context for detected errors. A high setting is useful when analyzing highly object-oriented applications. A higher number does not significantly impact **cost**. Recommendation: Use only as large a value as an application requires to display complete call paths.

See Also

Investigate Problems Using Interactive Debugging
Threading Error Analysis Types
Manage Custom Analysis Types

Pane: Analysis Type-Custom
Intel Inspector Filenames and Locations

Threading Error Analysis Types

Intel Inspector offers a range of preset threading analysis types to help you control analysis scope and cost:

- Analysis types with the narrowest scope minimize the load on the system and the time and resources required to perform the analysis; however, they detect the narrowest set of errors and provide minimal details.
- Analysis types with the widest scope maximize the load on the system and the time and resources required to perform the analysis; however, they detect the widest set of errors and provide context and the maximum amount of detail for those errors.

Tip

- Use analysis types iteratively. Start with a narrow scope to verify the application is set up correctly and set expectations for analysis duration. Widen the scope only if you need more answers and you can tolerate the increased cost.
- Estimated collection time may be 2 to 320 times longer than normal application execution time.
- Data set size and workload have a direct impact on application execution time and analysis speed.

Some settings in each preset analysis type are configurable. If the combination of settings in a preset analysis type almost meets your needs, try fine-tuning these configurable settings.

If the combination of analysis type settings in the preset analysis types does not meet your needs at all, try creating a new custom analysis type based on the currently selected analysis type.

Threading Error Configuration Settings for Each Preset Analysis Type

The following table shows the settings (in alphabetical order) for each preset analysis type:

- *N/A* means not applicable for the preset analysis type (which means any value displayed in the GUI is greyed out)
- *Configurable* means you can change the setting without creating a custom analysis type.
- Renamed/split/combined configuration settings show previous manifestations.

Configuration Settings / Preset Threading Error Analysis Types	Detect Deadlocks (Narrow Scope)	Detect Deadlocks and Data Races	Locate Deadlocks and Data Races (Wide Scope)
Cross-thread stack access detection	Hide problems/ Hide warnings	Hide problems/ Show warnings	Hide problems/ Show warnings if Scope=Normal, Hide problems/ Hide warnings if Scope=Extremely thorough
Detect data races	No	Yes	Yes
Detect data races on stack (previously called Detect data races on stack accesses	N/A	No	No if Scope=Normal, Yes if Scope=Extremely thorough
Detect deadlocks (split from Detect lock hierarchy violations and deadlocks)	Yes	Yes	Yes

Configuration Settings / Preset Threading Error Analysis Types	Detect Deadlocks (Narrow Scope)	Detect Deadlocks and Data Races	Locate Deadlocks and Data Races (Wide Scope)
Detect lock hierarchy violations (split from Detect lock hierarchy violations and deadlocks)	Yes	Yes	Yes
Filter guaranteed atomics	N/A	No	No
Race analysis byte granularity (previously called Memory access byte granularity)	N/A	4 bytes	2 bytes if Scope=Normal, 1 byte if Scope=Extremely thorough
Remove duplicates	Yes	Yes	Yes (configurable)
Save stack on first access	N/A	No	Yes
Save stack on lock creation	Yes	Yes	Yes
Save stack on memory allocation (previously called Save stack on allocation)	N/A	No	Yes
Scope	N/A	N/A	Normal (configurable)
Stack frame depth	1 (configurable)	1 (configurable)	16 (configurable)
Terminate on deadlock	No (configurable)	No (configurable)	No (configurable)
Use maximum resources	No	No	No (configurable)

Threading Error Analysis Type Configuration Setting Descriptions

The following table describes the purpose, usefulness, and *cost* (low, medium, high, or proportional in terms of time and resources) for each analysis type configuration setting. (The settings are listed in alphabetical order.)

Setting	Purpose, Usefulness, and Cost
Cross-thread stack access detection	Use to set the alert mechanism for when a thread accesses stack memory of another thread. The alert mechanism helps you decide if this is an issue that requires handling. All options are low **cost** if **Detect data races** is selected. Recommendation: • Use **Hide problems/Hide warnings** if using an OpenMP* or oneAPI Threading Building Blocks (oneTBB) programming model; or if cross-thread stack accesses are anticipated. Also select **Detect races on stack**. • Use **Hide problems/Show warnings** if cross-thread stack accesses are not anticipated. Also deselect **Detect data races on stack**.

Setting	Purpose, Usefulness, and Cost
	• Use **Show problems/Hide warnings** if cross-thread stack accesses are not anticipated but a previous analysis indicated they exist and you are not using an OpenMP* or oneAPI Threading Building Blocks (oneTBB) programming model. Also deselect **Detect data races on stack**.
Detect data races	Select to detect problems where multiple threads access the same memory location without proper synchronization and at least one access is a write. Selecting is useful when you suspect data races that are not yet evident. High **cost**. Recommendation: Select. Consider also deselecting **Use maximum resources** to reduce cost.
Detect data races on stack (previously called Detect data races on stack accesses)	Available only if **Detect data races** is selected. Select to detect data races for variables allocated on the stack. Selecting is useful when threads in an application share variables from the stack and you suspect data races on the variables. High **cost**. Recommendation: Deselect. If you select, consider also deselecting **Use maximum resources** to reduce cost.
Detect deadlocks	Select to detect problems where two or more threads are waiting for the other to release resources, but none of the threads releases the resources. Thus no thread can proceed. Selecting is useful when you want to troubleshoot the location of a deadlock. Low **cost**.
Detect lock hierarchy violations	Select to detect problems where the acquisition hierarchy order of multiple synchronization objects in one thread differs from the acquisition hierarchy order in another thread, and could cause a deadlock under certain conditions. Selecting is useful when an application has complicated synchronization and it is hard to verify correctness. Low **cost** unless an application has a significant number of locks.
Filter guaranteed atomics	Available only if **Detect data races** is selected. Select to ignore data races on guaranteed atomic operations on the Intel® P6 processor family or newer. For details, please refer to the *Guaranteed Atomic Operations* section of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. Selecting is useful if you observe many false data race reports on simple operations like Load or Store to shared variables and the size of those variables is less than the processor cache line size. Do not select this option if you develop cross-platform code that should work on other architectures. Selecting this option might also hide true-races on variables that were properly aligned during this run but might not be aligned in general, e.g., if it just works "by chance".

Setting	Purpose, Usefulness, and Cost
	Low **cost**. Recommendations: Use this option with caution only if you observe many false reports on simple memory operations.
Race analysis byte granularity (previously called Memory access byte granularity)	Available only if **Detect data races** is selected. Use to set the size of the smallest memory block the Intel Inspector considers a single block of memory when determining if non-synchronized accesses to a memory block constitute a data race. Selecting is useful to control memory consumption during analysis for some applications. High **cost** when set to 1 byte. Recommendation: Set to 4 unless you continually see data races based on safe access to smaller memory blocks. If so, reset to 1.
Remove duplicates	Deselect to show all occurrences of a detected problem in the **Code Locations** pane. Deselecting is: • Useful when you need to fully visualize all threads and problem occurrences in relation to time • Low **cost** in terms of time; however, the number of duplicate errors could crowd out the number of unique errors. Recommendation: Select.
Save stack on first access	Available only if **Detect data races** is selected. Select to show as much information as possible on all threads involved in a data race. Selecting is useful when investigating complex data race problems. High **cost**. Recommendation: Deselect on initial analysis runs. Select only when you need the maximum information and context about all threads involved in a data race to solve the problem.
Save stack on lock creation	Select to show creation information on synchronization objects involved in deadlocks, lock hierarchy violations, and data races. Selecting is useful when acquisition stacks are not sufficient to understand the problem. Low **cost**.
Save stack on memory allocation (previously called Save stack on allocation)	Available only if **Detect data races** is selected. Select to identify the allocation site of dynamically allocated memory objects involved in data races. Medium **cost**.

Setting	Purpose, Usefulness, and Cost
	Recommendation: Select when you need to identify the object hierarchy of low-level objects involved in data races. For example: If object R is involved in a data race and is instantiated within objects O1, O2, and O3, the allocation call stack can help you identify which encapsulating object is not properly protecting access to object R.
Stack frame depth	Use to provide more or less call stack context for detected errors. A high setting is useful when analyzing highly object-oriented applications. A higher number does not significantly impact **cost** with one exception: Choosing a higher number plus selecting **Save stack on first access** increases **cost**. Recommendation: Use only as large a value as an application requires to display complete call paths.
Terminate on deadlock	Available only if **Detect deadlocks** is selected. Select to stop analysis and application execution if the Intel Inspector detects a deadlock. Selecting is useful when running your application as part of a kernel or unit testing suite. Low **cost**. Recommendation: Deselect. Instead, use the corresponding knob in the command line interface to perform kernel or unit testing in a nightly scenario. If the Intel Inspector identifies a deadlock, decide if it is appropriate to continue analysis.
Use maximum resources	Select to potentially find more problems. High **cost**. Recommendation: Deselect to run a quicker analysis that should find most of your data race and cross-thread stack access problems. Once you have found and fixed these problems, select to get more complete analysis coverage of possible data race and cross-thread stack access problems.

See Also

Investigate Problems Using Interactive Debugging
Memory Error Analysis Types
Manage Custom Analysis Types
Pane: Analysis Type-Custom
Intel Inspector Filenames and Locations

Manage Custom Analysis Types

If the combination of analysis type settings in the Intel Inspector preset analysis types does not meet your needs, try creating new custom analysis types based on the currently selected analysis type.

Open the Analysis Type window :

- Visual Studio* menu, choose **Tools > Intel Inspector*version* > New Analysis...**.

> **NOTE** In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- Standalone Intel Inspector GUI menu, choose **File** > **New** > **Analysis...**.

Create a New Custom Analysis Type

1. Do one of the following:

To Do This	Do This
Create a custom analysis type based on a preset memory analysis type.	1. Choose **Memory Error Analysis** from the Analysis Type drop-down. 2. Use the configuration slider to choose a preset memory analysis type. 3. Click the **Copy** button to display a **Custom Analysis** dialog box.
Create a custom analysis type based on a preset threading error analysis type.	1. Choose **Threading Error Analysis** from the Analysis Type drop-down. 2. Use the configuration slider to choose a preset threading analysis type. 3. Click the **Copy** button to display a **Custom Analysis** dialog box.
Create a custom analysis type based on a custom analysis type.	1. Choose **Custom Analysis Types** from the Analysis Type drop-down. 2. Choose a custom analysis type from the custom analysis type selection box. 3. Click the **Copy** button to display a **Custom Analysis** dialog box.

2. Supply identifying information to distinguish this custom analysis type from other analysis types.
3. In the **Properties** region, select or deselect checkboxes and choose values in the drop-down lists as appropriate. When you are finished, click the **OK** button.

NOTE
Some checkboxes and drop-down lists are dependent on the selection of other checkboxes. For example, for custom memory analysis types:

- The **Enable guard zones** checkbox is disabled unless you first select the **Detect invalid/ uninitialized accesses** checkbox.
- The **Guard zone byte size** drop-down list is disabled unless you first select the **Enable guard zones** checkbox.

Outcome: The Intel Inspector creates a new custom analysis type.

Change a Custom Analysis Type

1. Choose **Custom Analysis Types** from the Analysis Type drop-down.
2. Choose the custom analysis type from the custom analysis type selection box.
3. Click the **Edit** button. Select or deselect checkboxes and change values in drop-down lists as appropriate. When you are finished, click the **OK** button.

Delete a Custom Analysis Type

1. Choose **Custom Analysis Types** from the Analysis Type drop-down.
2. Choose the custom analysis type from the custom analysis type selection box.
3. Click the **Delete** button.
4. Click **Yes** in the confirmation dialog box.

See Also

Dialog Box: Custom Analysis
Pane: Analysis Type-Custom

Toolbar: Intel Inspector
Intel Inspector Filenames and Locations

Run Analysis

When you run an analysis, the Intel® Inspector:

- Executes an application.
- Identifies issues that may need handling.
- Collects those issues in a result.
- Converts symbol information into filenames and line numbers.
- Applies suppression rules.
- Performs duplicate elimination.
- Forms problem sets.
- Depending on your analysis configuration options, may launch an interactive debugging session.

See Also

Configure Analysis
Run Memory Error and Threading Error Analyses
Stop Analysis

Run Memory Error and Threading Error Analyses

When you run an analysis, the Intel Inspector executes an application, optionally launches a debugger depending on your analysis configuration options, identifies issues that may need handling, collects those issues in a result, converts symbol information into filenames and line numbers, applies suppression rules, performs duplicate elimination, and forms problem sets.

Run a Newly Configured Memory Error or Threading Error Analysis

On the **Command** toolbar on the **Analysis Type** window, click the **Start** button.

Outcome: The Intel Inspector starts executing and inspecting the application for issues, and collects those issues in a result.

Rerun a Memory Error or Threading Error Analysis

To Do This		Do This
Run another analysis...	Use a recently used analysis type.	From the: • Visual Studio* menu, choose **Tools** > **Intel Inspector <version>**. Then choose a recently used analysis. • Intel Inspector standalone GUI menu, choose **File** > **New**. Then choose a recently used analysis.
Run another analysis...	Use the same analysis type as that in a specific result.	In the Visual Studio* **Solution Explorer**: **1.** Right-click the result in the project result folder to display a context menu. **2.** Choose **Re-inspect**.

NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

See Also
Choose Projects
Hot Keys
Context Menus: Solution Explorer
Toolbar: Command

Stop Analysis

On the **Command** toolbar on the **Collection Log**, **Summary**, or **Sources** window, click the **Stop** button.

Outcome: The application stops executing and the Intel Inspector:

- Stops inspecting the application for issues.
- Finalizes the result collected thus far.
- Displays the result collected thus far.

See Also
Hot Keys
Toolbar: Command

Examine Result Data During Analysis

You can start managing the Intel Inspector analysis results before analysis (collection and finalization) is complete; however, some features are not available until after analysis is complete, such as:

- **Problem sets** - During analysis, the Intel Inspector displays detected problems in the order detected. Intel Inspector does not group problems into problem sets or prioritize problem sets into a to-do list until after analysis is complete.
- **States** - You cannot determine if an issue in the new result corresponds to an issue in a project baseline result until after analysis is complete.
- **Filters** - You cannot temporarily limit the items in the **Problems** pane until after analysis is complete.

You also have the following options while the Intel Inspector performs an analysis:

- On-demand leak detection
- Memory growth detection

About On-demand Memory Leak Detection

A **Memory leak** problem occurs when a block of memory is allocated, never deallocated, and not reachable (there is no pointer available to deallocate the block). As a result, the block of memory cannot be:

- Used by the application (because the application has no reference to it)
- Freed (again because the application has no reference to it to pass to the free routine)

An application that leaks enough memory may run out of address space and fail, or cause the system to run out of swap space, resulting in system-wide instability or performance issues.

Intel Inspector customarily displays memory leaks at the end of an analysis run when an application exits; however, you can also use the Intel Inspector on-demand memory leak detection feature to gather memory leak information while an application is running. This is useful if:

- An application does not terminate (such as a server process).
- You want memory leak information, but you do not want to wait for an application to terminate.
- You want to determine if memory is leaked during a specific interval of application execution, or during a specific user action.
- You want to discard information about allocations performed during initialization as a way of filtering out allocations that are not currently of interest.

Tip
For more precision, consider using the GUI-based on-demand memory leak detection functions in tandem with APIs for custom memory allocation.

About Memory Growth Detection

A **Memory growth** problem occurs when a block of memory is allocated, but not deallocated, within a specific time segment during application execution.

Intel Inspector can measure memory growth to help you ensure an application uses no more memory than expected. This includes:

- Memory an application has allocated and still needs for future calculations
- Memory an application has allocated and no longer needs, but has not deallocated
- Memory an application has allocated and then leaked

Tip
For more precision, consider using the GUI-based memory growth detection functions in tandem with APIs for custom memory allocation.

See Also

Edit Source Code
Find Memory Leaks On Demand
Measure Memory Growth
Selecting Problems
APIs for Custom Memory Allocation

Find Memory Leaks On Demand

Intel Inspector distinguishes among **Memory leak**, **Memory not deallocated**, and **Memory growth** problem types in the following manner:

- **Memory leak** problems occur when a block of memory is allocated, never deallocated, and not reachable (there is no pointer available to deallocate the block). Severity level =  (**Error**).
- **Memory not deallocated** problems occur when a block of memory is allocated, never deallocated, but still reachable at application exit (there is a pointer available to deallocate the block). Severity level =  (**Warning**).
- **Memory growth** problems occur when a block of memory is allocated, but not deallocated, within a specific time segment during application execution. Severity level =  (**Warning**).

Intel Inspector customarily displays memory leaks at the end of an analysis run when an application exits; however, you can also use the Intel Inspector on-demand memory leak detection feature to gather memory leak information while an application is running. This is useful if:

- An application does not terminate (such as a server process).
- You want memory leak information, but you do not want to wait for an application to terminate.
- You want to determine if memory is leaked during a specific interval of application execution, or during a specific user action.
- You want to discard information about allocations performed during initialization as a way of filtering out allocations that are not currently of interest.

To find memory leaks on demand:

Prerequisites:

- Select the **Enable on-demand leak detection** checkbox when configuring a memory error analysis.
- Run the memory error analysis.

Steps:

1. If desired, click the **Reset Leak Tracking** button on the **Command** toolbar to mark the start point of the time period during which you want to find memory leaks. (The default start point is the application start.)

   Caution
   When you click this button, the Intel Inspector discards earlier allocation data and tracks leaks only in new allocations. Any new memory leaks on memory allocated prior to the button click do not appear in the result.

2. When desired, click the **Find Leaks** button to find memory leaks in the current allocation set.
   Outcome: The Intel Inspector generates a **Memory leak** problem in the **Summary** window for any memory leak detected during the time period.
3. Do one or more of the following:

To Do This	Do This
Review the detected memory leaks.	Click the **Summary** button to check for **Memory leak** problems.
Continue finding memory leaks in the current allocation set.	Repeat step 2.
Discard earlier allocation data and search for memory leaks only in new allocations.	Repeat steps 1 and 2.

4. When you are finished, exit the application to end analysis, or simply let the analysis finish.
   Outcome: The Intel Inspector finalizes and displays the result.

   Tip
   For more precision, consider using the GUI-based on-demand memory leak detection commands in tandem with APIs for custom memory allocation.

See Also

Toolbar: Command
APIs for Custom Memory Allocation

Measure Memory Growth

Intel Inspector distinguishes among **Memory leak**, **Memory not deallocated**, and **Memory growth** problem types in the following manner:

- **Memory leak** problems occur when a block of memory is allocated, never deallocated, and not reachable (there is no pointer available to deallocate the block). Severity level = 
  (**Error**).
- **Memory not deallocated** problems occur when a block of memory is allocated, never deallocated, but still reachable at application exit (there is a pointer available to deallocate the block). Severity level = 
  (**Warning**).
- **Memory growth** problems occur when a block of memory is allocated, but not deallocated, within a specific time segment during application execution. Severity level =

(**Warning**).

Intel Inspector can measure memory growth to help you ensure an application uses no more memory than expected. This includes:

- Memory an application has allocated and still needs for future calculations
- Memory an application has allocated and no longer needs, but has not deallocated
- Memory an application has allocated and then leaked

NOTE
Intel Inspector does not perform a reachability analysis on detected **Memory growth** problems.

To measure memory growth:

Prerequisites:

- Select the **Enable memory growth detection** checkbox when configuring a memory error analysis.
- Run the memory error analysis.

Steps:

1. If desired, click the **Reset Growth Tracking** button on the **Command** toolbar to mark the start point of the time period during which you want to measure memory growth. (The default start point is the application start.)
   Outcome: The Intel Inspector marks the memory usage graph on the **Collection Log** pane with a reset request icon.
2. When desired, click the **Measure Growth** button to mark the end point of the time period and measure how much memory has grown since the start point.
   Outcome: The Intel Inspector marks the memory usage graph on the **Collection Log** pane with a measurement request icon and generates a **Memory Growth** problem that contains a code location for the start marker, for the end marker, and for any block of memory that was allocated, but not deallocated, during the time period.
3. Do one or more of the following:

To Do This	Do This
Review memory growth.	Click the **Summary** button to check the **Memory growth** problem for any code locations other than a start and end marker.
Continue measuring memory growth from the previous start point.	Repeat step 2.
Measure memory growth from a new start point.	Repeat steps 1 and 2.

4. When you are finished measuring memory growth, exit the application to end analysis.
   Outcome: The Intel Inspector finalizes and displays the result.

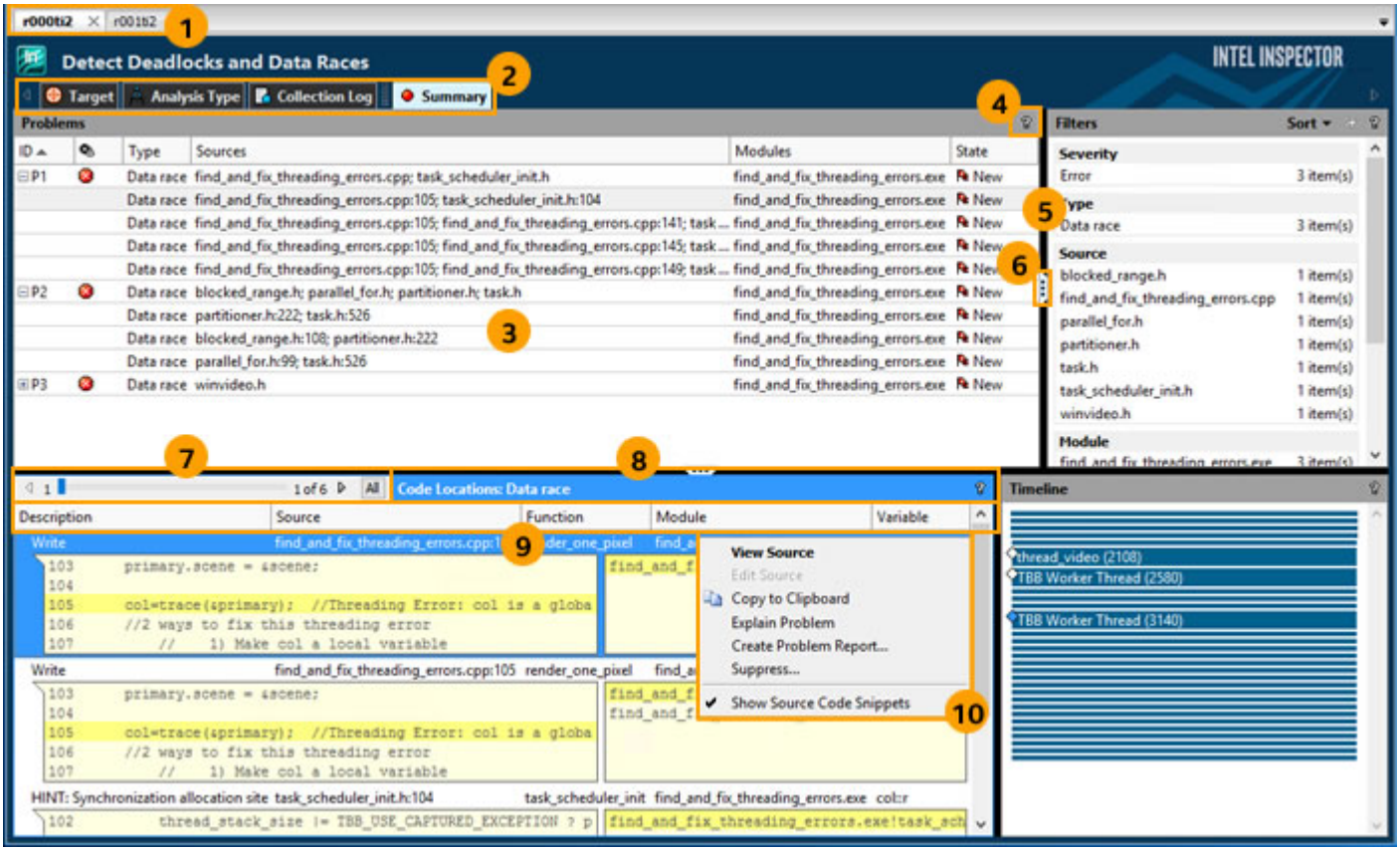
Tip
For more precision, consider using the GUI-based memory growth detection commands in tandem with APIs for custom memory allocation.

See Also

Toolbar: Command
APIs for Custom Memory Allocation

Investigate Results

View Results



1 Use result tab names to distinguish among results.

2 Click buttons on the navigation toolbar to change window views.

3 Use window panes to view and manage result data.

4 Click

buttons to display help pages that describe how to use window panes.

5 Drag window pane borders to resize window panes.

6 Click

,

,

, and

controls to show/hide window panes.

7 Click window pane data controls to adjust result data within the pane (and possibly in adjacent panes).

8 Use title bars to identify window panes.

9 Data column headers - Drag to reposition the data column; drag the left or right border to resize the data column; click to sort results in ascending or descending order by column data.

10 Right-click data in window panes to display context menus that provide access to key capabilities.

Tip
In the Standalone Intel Inspector GUI, press F11 to toggle on and off a full-screen view of result tab data. You can also:

- Choose **View** > **Full Screen** to toggle on full-screen view.
- Press ESC to toggle off full-screen view.

See Also

Visual Studio* Integration
Standalone Intel Inspector GUI
Open Results

Manage Help Snippet Displays

Intel Inspector offers a help snippet that:

- Explains how to use various Intel Inspector windows.
- Offers links to more help information.

To display the help snippet:

Click the

Intel Inspector icon in the result tab.

To close the help snippet:

Click the **X** button on the help snippet window.

See Also

View Results

Open Results

Intel Inspector analysis results are available during and immediately after analysis is complete.

You can also open previously collected analysis results.

To open a previously collected result:

To Do This	Do This
Open a result from the Visual Studio* IDE.	Do one of the following: • In the **Solution Explorer**, double-click the result in the project result folder. • Choose **File** > **Open**. Browse to and select the appropriate `*.inspxe` file, then click the **OK** button. • Choose **File** > **Recent Files**, then choose the appropriate `*.inspxe` file. **NOTE** In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.
Open a result from the Standalone Intel Inspector GUI.	Do one of the following: • In the **Project Navigator** pane, double-click the result in the `[project]` folder. • Choose **File** > **Open** > **Result...**. Browse to and select the appropriate `*.inspxe` file, then click the **OK** button. • Choose **File** > **Recent Results**, then choose the appropriate `*.inspxe` file.
Open a result from the file system.	Select the appropriate `*.inspxe` file. This action launches the Standalone Intel Inspector GUI if it is not already open.

Outcome: The Intel Inspector displays a progress indicator as it loads the result and then displays the **Summary** window.

See Also

View Results
Toolbar: Intel Inspector
Context Menus: Project Navigator
Context Menus: Solution Explorer

Show Result Summaries

To Do This	Do This
Show a summary of the currently displayed Intel Inspector result.	From the Visual Studio* menu, choose **Tools > Intel Parallel Inspector > Result Summary...**. **NOTE** In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.
Show a summary for a result not currently displayed.	Right-click the result in the `Inspector Results/[project]` folder in the **Solution Explorer** to display a context menu, then choose **Result Summary**.

See Also

Context Menu: Solution Explorer

Choose Problems

During analysis, the Intel Inspector displays problems in the order detected in the **Problems** pane.

After analysis is complete, the Intel Inspector:

- Groups problems detected during analysis into problem sets (but still provides visibility into individual problems and problem occurrences).
- Prioritizes the items in the **Problems** pane.
- Offers filtering to help you focus on those items that require your attention. For example, you can temporarily limit the items displayed in the **Problems** pane by:
  - Inclusion
  - Exclusion
  - A set of source files

About Prioritization

Intel Inspector assigns problems a severity level: **Critical** (

), **Error** (

) and **Warning** (

). The severity indicates the seriousness of the defect.

Problem sets inherit the severity of the problems they contain. The Intel Inspector prioritizes problem sets first by severity and then by number of code locations. Problem sets containing more code locations precede those containing fewer.

Think of this prioritized list of items on the **Problems** pane as a *to-do* list. Start at the top and work down.

About Filtering by Inclusion

Use the Intel Inspector filtering function to temporarily limit the items in the **Problems** pane to only those problem sets containing problems that meet specific criteria.

For example, you can filter the list to show only those problem sets containing problems...

- With **Investigated**=**Not investigated** to show only problem sets with a state of **New**, **Not fixed**, or **Regression** (that is, hide the problem sets you have already investigated)
- Or with **State=Not fixed** and **Severity**=**Error**
- Or with **State=Not fixed** and **Severity**=**Error** and **Type**=**Data Race**
- Or in a specific file

About Filtering by Exclusion

Use the Intel Inspector filtering function to temporarily set aside occurrences of the same, possibly false positive, items en masse in the **Problems** pane. For example:

1. In the **Investigated** category in the **Filters** pane on the **Summary** window, choose **Not investigated** to display only problems with a state of **New**, **Not fixed**, or **Regression**.

   NOTE
   If this is the first result of this analysis type, or if you chose the **Do not get problem states from another result** option on the **State Management** page of the product **Options** dialog box during startup, all problems have a state of **New**.

2. Click the **Sort** drop-down list in the **Filters** pane and choose **Sort by Item Count** to sort remaining problems by count in descending numerical order.
3. In the **Type** category in the **Filters** pane, choose the problem type with the highest item count to display only those problems in the **Problems** pane.
4. Click on any problem in the **Problems** pane, then press Ctrl+A to select all problems.
5. Right-click any of the selected problems to display a context menu, then choose **Change State** > **Not a Problem** to change the state of all selected problems.
6. In the **Type** category, click the **All** button to deselect problem criteria.
7. Repeat steps 3 - 6 for each problem type with many occurrences.
8. Start working the remaining problems.

Be aware that when you temporarily set aside problems en masse, you may inadvertently set aside a problem of interest. For example, if you are not responsible for problems in source file A and you:

1. Use the filtering function to select all problems in source file A.
2. Set the state for all problems in source file A to **Not a problem**.

Some of these problems may touch more than one source file. When you filter on a source file, you filter on all problems intersecting that source file. The fact that a problem has some intersection with a file for which you are not responsible does not mean the problem does not concern you.

However if you are responsible for source files A, B, and C, you can filter first to source file A, then to source file B, then to source file C. You may see some problems more than once, but you will ultimately investigate all problems in files for which you are responsible.

About Filtering by a Set of Source Files

Use the Intel Inspector filtering function to temporarily limit the items in the **Problems** pane to those from a set of source files.

Deselecting Filter Criteria

To Do This	Do This
Deselect the filter criterion for a specific category.	In the category header in the Intel Inspector**Filters** pane, click the **All** button.
Deselect all filter criteria.	In the **Filters** pane, click the  button.

Outcome: The Intel Inspector readjusts:

- Category presentation to the previous or original state
- Items displayed in the **Problems** pane

See Also

Pane: Filters
Pane: Problems

Refining the Source File Set

Use the **Refine Source File Set...** option on the Intel Inspector**Filters** pane context menu to temporarily limit the items displayed in the **Problems** pane to those from a set of source files.

Refine the Source File Set

1. Right-click anywhere in the **Filters** pane to display a context menu, then choose **Refine Source File Set...** to display a **Refine Source File Set** dialog box.

2. In the **Search substring** field, enter any portion of a file path to use as a criterion for selecting a file set.
3. When the filenames in the **Matching files** region match the desired file set, click the **OK** button.

Outcome:

- In the **Filters** pane, the Intel Inspector:
  - Adds a **Search Source Substring** category containing the search substring.
  - Displays the names of the source files that match the source substring in the **Source** category.
- In the **Problems** pane, displays only those items that match the source file criteria.

> **NOTE**
> In the **Source** category in the **Filters** pane, the Intel Inspector also displays the name of any other source files mentioned in the items still listed in the **Problems** pane.

Clear the Source File Filter

Do one of the following:

- Right-click anywhere in the **Filters** pane to display a context menu, then choose **Clear Source File Filter**.
- In the **Search Source Substring** category header in the **Filters** pane, click the **All** button.

See Also

Dialog Box: Refine Source File Set
Pane: Filters
Context Menus: Summary Window Panes

Selecting Filter Criteria

After analysis is complete, you can filter (temporarily limit) the items in the Intel Inspector**Problems** pane to focus on those issues that require your attention.

Select Filter Criteria

1. In the **Filters** pane, click a filter criterion in a category.
2. If appropriate, click a criterion in another category. Repeat until you identify all appropriate criteria.

Outcome: The Intel Inspector limits data accordingly in:

- The categories in the **Filters** pane
- The **Problems** pane

> **NOTE**
> Filters are persistent within a result, including when you reopen the result later.

Example

The following example:

- Identifies a filtering objective and describes the steps necessary to accomplish it.
- Shows simulations of **Filters** panes.
- Refers to associated **Problems** panes but does not show them.

Objective	Filters Pane Simulations
Display only those problem sets containing problems that match these criteria:	Severity

- **Severity=Error**
- **Description=Deadlock**
- **Source=a.cpp**

Process

Starting point: The simulation (shown at right) of a **Filters** pane on the **Summary** window.

NOTE
This simulation shows only some of the possible categories you can use as filter criteria.

Error	16 items
Warning	4 items
Remark	5 items
Type	
Deadlock	4 items
Data race	12 items
Potential deadlock	4 items
Thread information	5 items
Source	
a.cpp	15 items
b.c	4 items
c.cpp	6 items

Notice the number of problem sets containing problems with a **Severity** of **Error**: 16.

In the **Severity** category, click **Error.**

Severity	**All**
Error	16 items
Type	
Deadlock	4 items
Data race	12 items
Source	
a.cpp	6 items
b.c	4 items
c.cpp	6 items

Intel Inspector limits the data displayed in both the **Filters** pane (simulated at right) and the **Problems** pane (not shown here) to only problem sets containing problems that match the **Severity=Error** criterion.

Notice:

- The only displayed criterion in the **Severity** category is **Error**.
- The **Severity** category header changes color and now contains an **All** button (to deselect the **Error** criterion).
- The criteria and/or item counts in other categories are decreased.

In the **Type** category, click **Deadlock.**

Severity	**All**
Error	4 items
Type	**All**
Deadlock	4 items
Source	
a.cpp	2 items
c.cpp	2 items

Intel Inspector limits the data displayed in the **Filters** pane (simulated at right) and the associated **Problems** pane (not shown here) to only problem sets containing problems that match the **Severity=Error** and **Type=Deadlock** criteria.

In the **Source** category, click `a.cpp`.

Intel Inspector limits the data displayed in the **Filters** pane (simulated at right) and the associated **Problems** pane (not shown here) to only problem sets containing problems that match the **Severity=Error** and **Type=Deadlock** and **Source=a.cpp** criteria.

Severity	**All**
Error	2 items

Type	**All**
Deadlock	2 items

Source	**All**
a.cpp	2 items

See Also

States
Pane: Filters
Pane: Problems

Selecting Problems

Do one of the following in the Intel Inspector**Problems** pane:

To Do This	Do This
Show information in the **Problems** pane for each problem in a problem set.	Click the  icon for the problem set. (Available only after analysis is complete.)
Show information in the **Code Locations** pane for a single item in the **Problems** pane.	Click the item.
Show information in the **Code Locations** pane for multiple items in the **Problems** pane.	Ctrl+Click the items.
Show information in the **Code Locations** pane for a range of items in the **Problems** pane.	Click the item at the start of the range and Shift+Click the item at the end of the range.
Show information in the **Code Locations** pane for all currently displayed items the **Problems** pane.	Ctrl+A.
Show a **Sources** window focused on the selected item.	Double-click the item.

See Also

Pane: Filters
Pane: Problems

Interpret Result Data and Resolve Issues

Use the following Intel® Inspector features to enhance your productivity:

Objective	Feature	During Analysis / After Analysis Completes
Interpret result data	Explain Problem help	• During analysis

Objective	Feature	During Analysis / After Analysis Completes
		• After analysis is complete
Focus only on those issues that require your attention	Severity levels	• During analysis • After analysis is complete
	States	After analysis is complete
	Suppression rules	After analysis is complete
Resolve issues	Direct access to a default editor	• During analysis • After analysis is complete

Severity Levels

Intel Inspector severity levels denote the seriousness of a defect:

-  (**Critical**) is more serious than  (**Error**).
-  (**Error**) is more serious than  (**Warning**).
-  (**Remark**) denotes additional information to help you understand the context of an analysis.

Critical severity is used for **Unhandled application exception** defects that usually cause the application undergoing analysis to crash.

Severity level is a property of a problem type. All problems in a problem set have the same severity level.

States

To avoid investigating issues over and over again, save your investigative conclusions by assigning states to the issues you investigate.

State information is persistent within the Intel Inspector result, including when you reopen the result.

Intel Inspector also propagates state information from an older result to a newer result when you take advantage of various baseline result options.

- Available States
- Typical Usage Model
- States Propagation
- States and Filters
- States Hierarchy

Available States

Intel Inspector offers the following states:

Filter Criterion	Set By	State	Meaning
Not investigated	Intel Inspector	 **Regression**	The issue requires more investigation because it was marked as **Fixed** in the baseline result, but still appears.

Filter Criterion	Set By	State	Meaning
		New	The issue did not appear in the baseline result, or there is no older result from which the Intel Inspector can propagate state information.
	Intel Inspector or User	**Not Fixed**	The issue appeared in the baseline result and still requires investigation.
Investigated	User	**Confirmed**	The issue requires fixing but has not yet been fixed. **Tip** Report **Confirmed** issues in a bug tracking system.
		Fixed	The issue requires fixing and has been fixed.
		Not a problem	The issue does not require fixing.
		Deferred	You are postponing further investigation on an issue that may or may not require fixing.

Typical Usage Model

1. Use the default **Get problem states from previous result of same analysis type** baseline result option.
2. Run an analysis.
3. Use the filtering function to temporarily limit the displayed issues to only those that are **Not investigated**.
4. Set the state of each problem issue you investigate as:
   - **Confirmed** - Issue requires fixing but has not yet been fixed.
   - **Fixed** - Issue requires fixing and has been fixed.
   - **Not a Problem** - Issue does not require fixing.
   - **Deferred** - Delay further investigation.
5. The next time you rerun the analysis, verify the issues you expect to be fixed are indeed fixed.

State Propagation

Intel Inspector propagates state information from the baseline result when it determines an issue in a new result corresponds to an issue in the baseline result. For example, if you set the state for a problem in the baseline result to **Not a Problem**, the Intel Inspector sets the state for the corresponding problem in the new result to **Not a Problem**.

The only exception: If you set an issue state to **Fixed** in the baseline result but the issue still appears in the new result, the Intel Inspector sets the issue state to **Regression** in the new result.

Caution
Intel Inspector may not recognize an issue as previously investigated when it propagates state information from a baseline result. This is more likely to happen when source code has undergone drastic changes between analysis runs.

States and Filters

Use the Intel Inspector filtering to focus on issues in specific states. For example, in the **Filters** pane on the **Summary** window:

- In the **Investigated** category, choose **Not investigated** to reduce clutter by displaying only issues with a state of **New**, **Not fixed**, or **Regression**.
- In the **State** category, choose **Confirmed** to review issues that are - or should be - reported in your bug tracking system.
- In the **State** category, choose **New** to concentrate on issues introduced since the last analysis run.
- In the **State** category, choose **Regression** to verify all issues marked as **Fixed** in the baseline were successfully fixed.
- In the **State** category, choose **Not a problem** to double-check earlier conclusions that no fix is needed.

States Hierarchy

When you change the state of a problem set, the Intel Inspector responds by similarly changing the state of all problems currently in the problem set.

In scenarios where a problem set contains problems with different states (such as when source code changes introduce a new instance of a problem previously set to a non-new state), the Intel Inspector assigns the most severe state to the problem set according to this hierarchy:

1. **Regression** (most severe state)
2. **New**
3. **Not Fixed**
4. **Confirmed**
5. **Fixed**
6. **Not a Problem**
7. **Deferred** (least severe state)

See Also

Collaborate on Results
Change States
Choose Baseline Results for Setting States
Merge States Across Results
Selecting Filter Criteria

Access the Explain Problem Help

Use the Intel Inspector*Explain Problem* help to understand, in generic terms, the error associated with the selection.

To access Explain Problem Help:

1. Do one of the following:
   - On the **Summary** window, right-click any problem in the **Problems** pane or any code location in the **Code Locations** pane.
   - On the **Sources** window, right-click anywhere in the **Source** tab or **Disassembly** tab.
2. Choose **Explain Problem**.

Outcome: The Intel Inspector opens a help window that provides the following information:

- Problem description
- Code location relationship diagram example(s)
- Code location descriptions
- Examples
- Possible correction strategies

See Also

Context Menus: Summary Window Panes
Context Menus: Sources Window Panes

Change States

Change the Intel Inspector problem state to help you focus on only those problems that require your attention.

To change problem state:

1. Right-click a problem or problem set in the **Problems** pane to display a context menu.
2. Choose one of the following:
   - **Change State** > **Confirmed** if the problem(s) requires fixing but has not yet been fixed
   - **Change State** > **Fixed** if the problem(s) requires fixing and has been fixed
   - **Change State** > **Not a problem** if the problem(s) does not require fixing
   - **Change State** > **Deferred** if you plan to postpone further investigation of this problem(s) at this time

See Also

States
Pane: Problems
Context Menus: Summary Window Panes

Customize Frame Presentation

1. Select a problem or code location to open the Intel Inspector**Sources** window.
2. In the **Call Stack** tab, right-click to display a context menu with the following options:

Choose This Option	To Do This
Show Module Names (**Call Stack** tab only)	Show this information: • Module • Function • Source
Show Sources	Show this information: • Function • Source • Source line
Show Two Lines	View frame information more easily. **NOTE** The **Show Two Lines** option is enabled on the**Call Stack** tab only when both the **Show Module Names** and **Show Sources** options are checked.
Show Horizontal Scrollbar	

Choose This Option	To Do This
Copy to Clipboard	Copy the frames to the system clipboard.

See Also

Pane: Code and Stack
Context Menus: Sources Window Panes

Customize Data Columns

To Do This	Do This
Hide a specific Intel Inspector data column.	Right-click the data column header to display a context menu, then choose **Hide Column**.
Show a specific data column	Right-click any data column header to display a context menu, then choose **Show Column** > **<appropriate column>** to display the previously hidden data column to the right of currently displayed data columns.
Show all data columns	Right-click any data column header to display a context menu, then choose **Show All Columns** to display previously hidden data columns to the right of currently displayed data columns.
Reposition a data column.	Drag the data column header to the desired position in the pane.
Resize a data column.	Drag the left or right border of the data column header.
Sort results in ascending or descending order by column data.	Click the data column header. (Available only after analysis is complete.)

See Also

Pane: Code Locations
Pane: Problems
Context Menus: Summary Window Panes

Edit Source Code

You can access source code directly from Intel Inspector windows to resolve issues.

Caution
Editing the source file may desynchronize source file line numbers and result line number references. If your edits change line numbering, consider rebuilding the application and running another analysis. If you plan to examine a large number of issues before running another analysis, consider editing a copy of the source file.

To Do This	Do This
Edit the source code for the primary code location in a problem or problem set.	In the **Problems** pane on the **Summary** window, right-click the problem or problem set to display a context menu, then choose **Edit Source**.

To Do This	Do This
Edit the source code for any code location in a problem.	Do one of the following: • In the **Code Locations** pane on the **Summary** window, right-click the code location to display a context menu, then choose **Edit Source**. • Double-click anywhere in the **Source** tab on the **Sources** window.
Edit call stack source code.	Double-click the call stack frame in the **Call Stack** tab on the **Sources** window. **NOTE** This action also displays associated code in the **Source** tab on the **Sources** window.

Outcome: The source file opens in your default editor.

See Also

Pane: Code and Stack
Pane: Code Locations
Context Menus: Sources Window Panes
Context Menus: Summary Window Panes

Investigate Problems Using Interactive Debugging

Sometimes simply knowing the location of a problem is not enough. So the Intel Inspector provides the opportunity to investigate more deeply with an interactive debugging session during analysis.

When you run an interactive debugging session during analysis, the Intel Inspector halts execution at a selected or detected problem. This is more efficient than simply setting a code breakpoint at a reported problem location because the code could execute thousands of times before the conditions that produced the problem occur.

During the interactive debugging session, use the normal Visual Studio* debugger actions to examine memory, set code breakpoints, and continue execution. Only the use of data breakpoints is not supported.

Use one of the following ways to initiate an interactive debugging session during analysis so that you see the state of your application code instead of the state of Intel Inspector analysis code:

Objective	Process Summary	Usage
Allow quick investigation of problem(s) of interest.	In a result, choose a problem(s) in the **Summary** window, right-click to display a context menu, then choose **Debug This Problem**. Outcome: The Intel Inspector: • Launches a new analysis, of the same type, optimized to find the selected problem(s). • Halts application execution when it detects the selected problem(s) and opens a debugging session.	Typical scenario: After reviewing a problem using result data provided by the Intel Inspector, you discover you need more application state information at the time the problem occurred, such as the contents of variables. The **Debug This Problem** function returns you to the point of the problem, halting execution in a debugging session so you can examine memory and other state information at any point in the call stack.
Delay problem detection until a	In the Visual Studio* debugger, set a code breakpoint to halt application execution prior to where you want analysis to begin. In the Intel Inspector, select the **Select analysis start**	Typical scenario: You configured the project to limit the application modules for inspection, but you need to narrow analysis focus even further. This option

Objective	Process Summary	Usage
selected point in the application.	**location with debugger** radio button when configuring an analysis and start the analysis. Wait for the debugging session to open and execution to halt at the code breakpoint. Choose **Debug** > **Continue with Inspector Analysis** to turn on analysis and resume execution. Outcome: Execution halts for each problem detected after you turn on analysis and resume execution. You cannot turn off analysis again.	lets you choose an arbitrary location to turn on error detection, allowing faster execution until that point. **Tip** Because debugging in conjunction with a threading analysis significantly slows execution: • Use this option only if the area targeted for threading analysis has a very short execution time. • For best performance, use the **Debug This Problem** function to selectively investigate threading issues identified during a non-debug run.
Allow investigation of every detected problem.	In the Intel Inspector, select the **Enable debugger when problem detected** radio button when configuring an analysis. Outcome: When you run the analysis, the Intel Inspector halts application execution when it detects the first problem and launches a debugging session. Execution halts for each detected problem when you resume execution using normal debugger commands.	Typical scenario: You want to investigate the state of the application for every problem detected. **Tip** Because of analysis speed issues, this option is not recommended when detecting data races. It is faster to use the **Debug This Problem** function to selectively investigate threading problems identified during a non-debug run.

Tips

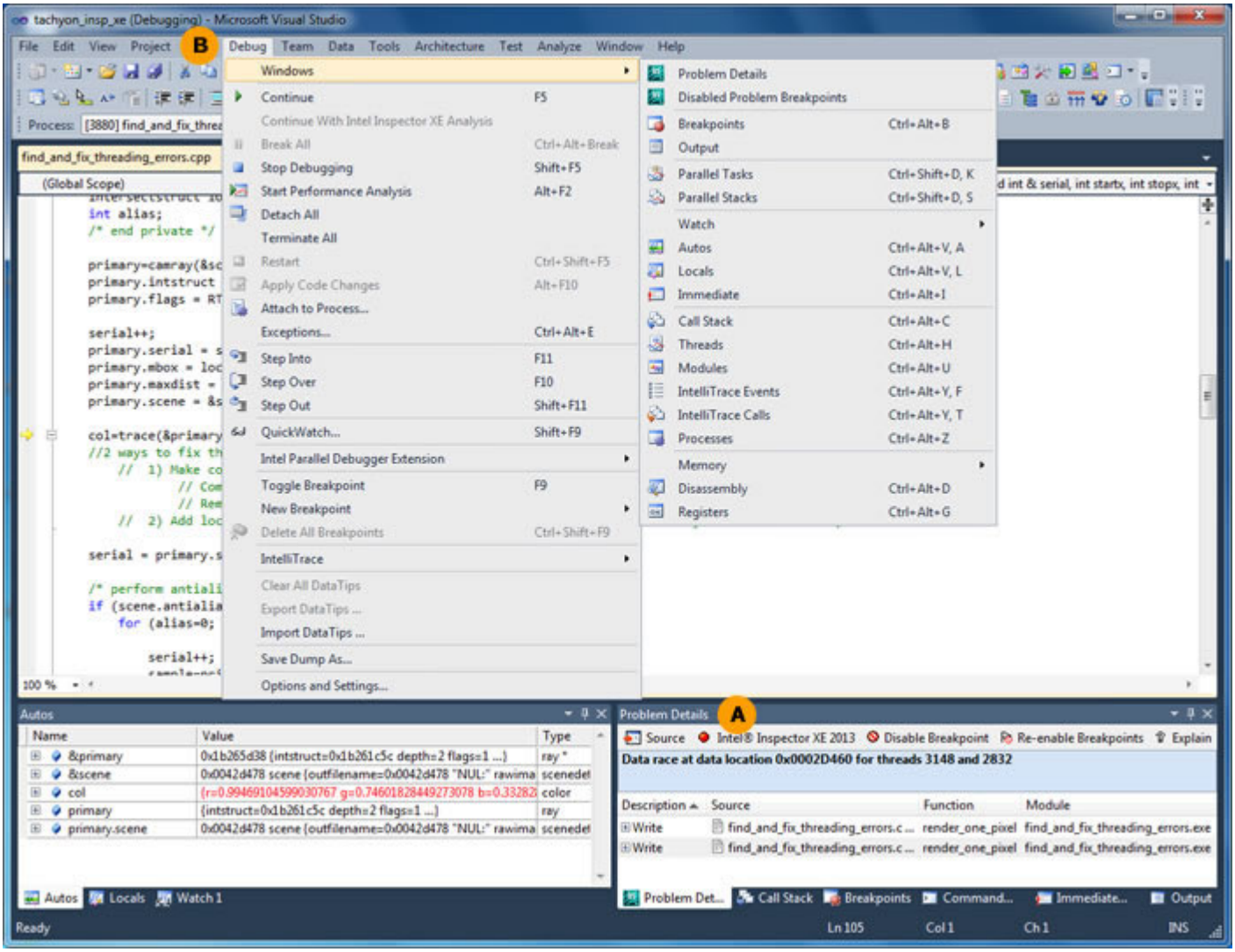
- Intel Inspector supports interactive debugging only for pure native applications.
- Intel Inspector does not offer interactive debugging for the **Detect Leaks** (`mi1`) analysis type because memory and resource leaks are determined after an application terminates and therefore cannot be used to halt execution during analysis. However, you can perform a standard debugger attach to a process launched under this analysis type.
- Do not recompile your application after generating a result if you plan to use the result **Debug This Problem** function. Intel Inspector cannot guarantee finding the same problem(s) when the binary changes; consequently, it checks for a binary change and reports a warning to prevent rerunning an analysis that may not stop at the selected problem(s).
- When you use the **Debug This Problem** function, the problem list in the new result may differ from the problem list in the source result. Intel Inspector automatically adjusts the debugging session analysis to return to the selected problem(s) more quickly; however the analysis adjustments do not correspond to individual problem types. Consequently, the Intel Inspector may detect and report additional problems, but it will break only for a selected problem(s).

See Also

Configure Analysis
Configure Projects
Intel Inspector Debug Extensions in the Visual Studio* IDE
Disable and Re-enable Problem Breakpoints
Stop Analyses During Interactive Debugging

Intel Inspector Debug Extensions in the Visual Studio* IDE

Intel Inspector functionality appears in the Visual Studio* debugger in the following manner:



A The **Problem Details** pane appears automatically when a problem breakpoint occurs. Use this pane to view the same problem details found in the Intel Inspector result, duplicated within the debugger workspace for convenient referencing; and to disable and re-enable problem breakpoints.

B Use the **Debug** > **Windows** > **Problem Details** menu option to reopen the Intel Inspector**Problems Details** pane.

Use the **Debug** > **Continue with Intel Inspector <version> Analysis** menu option in conjunction with the **Select analysis start location with debugger** analysis configuration option to start checking for errors at a specific point in an application.

NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

Disable and Re-enable Problem Breakpoints

To maximize your productivity, disable breaking on detected problems that execute multiple times.

NOTE

- You cannot disable breaking on a detected problem until execution halts at that problem at least once.
- Disabled problem breakpoints are not persistent between analysis runs.

Disable a Problem Breakpoint

When the problem is displayed in the Intel Inspector**Problem Details** pane, click the **Disable Breakpoint** button.

Re-enable a Problem Breakpoint

At any time when data is displayed in the **Problem Details** pane:

1. Click the **Re-enable Breakpoints** button to display a **Disabled Problem Breakpoints** dialog box.
2. Select the problem breakpoint and click the **Re-enable** button.

See Also

Dialog Box: Disabled Problem Breakpoints
Pane: Problem Details

Stop Analyses During Interactive Debugging

Do one of the following to stop analysis during an interactive debugging session:

- Exit the debugging session.
- Click the **Intel Inspector** button on the **Problem Details** pane to change focus to the new Intel Inspector result **Sources** window, then click the **Stop** button on the **Command** toolbar.

Outcome: The application stops executing and the Intel Inspector:

- Stops inspecting the application for issues.
- Finalizes the result collected thus far.
- Displays the result collected thus far.

See Also

Toolbar: Command

Collaborate on Results

Intel Inspector offers a variety of productivity features to help teammates share their interpretation, investigation, and resolution efforts, including features to:

- Export result information in plain text format.
- Merge state information across results.

About Exporting Result Information

Use the **Create Problem Report** option on **Summary** window context menus to export result data in plain text format so you can distribute it to teammates in other media, such as email.

The problem report provides more data than the **Copy to Clipboard** option available on various context menus. For example:

If You Select This	The Report Groups Data by Problem Set and Contains This
One or more code locations	• Source code snippet and call stack information for the selected code location(s) • Summary information for the parent problem and problem set
One or more problems	• Source code snippet and call stack information for all code locations in the selected problem(s) • Summary information for the parent problem set
One or more problem sets	Source code snippet and call stack information for all code locations in all problems in the selected problem set

The problem report is also more versatile than the **Copy to Clipboard** option because you can send all or some of the data to the system clipboard or to a text file.

About Merging State Information Across Results

By default, the Intel Inspector propagates state information from a baseline result (the immediately previous result of the same analysis type) when you open a result for the first time.
In addition, you can use the **Merge States...** option on the **Problems** pane context menu to merge state information from any result in any project into the currently open result.
The state merge operation updates in the currently open result the state assigned to any issues that appear in both results. The state of issues that appear only in the currently open result remain unchanged. Issues that appear only in the selected result are not added to the open result.
Combining the efforts of multiple team members working the same result is an excellent way to take advantage of the state merge feature.

See Also

State Merge
States
Choose Baseline Results for Setting States
Merge States Across Results
Report Result Data

State Merge

By default, the Intel Inspector propagates state information from a baseline result (the immediately previous result of the same analysis type) when you open a result for the first time.
In addition, you can use the **Merge States...** option on the **Problems** pane context menu to merge state information from any result in any project into the currently open result.
The state merge operation updates in the currently open result the state assigned to any issues that appear in both results. The state of issues that appear only in the currently open result remain unchanged. Issues that appear only in the selected result are not added to the open result.
Combining the efforts of multiple team members working the same result is an excellent way to take advantage of the state merge feature.

That is, team members agree upon which result issues each will investigate, make a private copy of the result, investigate their assigned issues and set issue state accordingly, and then merge their state changes back into a master copy. In this way, all team members can work independently but still combine their efforts into a single result.

For example: The state of an issue in the master result is **New**. Your teammate investigated the issue in a private copy of the result and set the issue state to **Not a Problem**. When you merge state information from your teammate's result (selected result) into your master result (open result), the Intel Inspector sets the issue state to **Not a Problem**.

Problem State/Selected Result	Problem State/Open Result						
	Not Fixed	New	Regression	Not a Problem	Deferred	Confirmed	Fixed
Not Fixed	Not Fixed	New	Regression	Not a Problem	Deferred	Confirmed	Fixed
New	Not Fixed	New	Regression	Not a Problem	Deferred	Confirmed	Fixed
Regression	Regression	Regression	Regression	Not a Problem	Deferred	Confirmed	Fixed
Not a Problem		Not a Problem	Not a Problem	Not a Problem	Deferred	Confirmed	Fixed
Deferred	Deferred	Deferred	Deferred	Deferred	Deferred	Confirmed	Fixed
Confirmed	Confirmed	Confirmed	Confirmed	Confirmed	Confirmed	Confirmed	Fixed
Fixed	Fixed	Fixed	Fixed	Fixed	Fixed	Fixed	Fixed

NOTE

- State information changes to the currently open result are not reversible. You cannot *roll back* a merge operation.
- There is no link between the two results after a state merge operation is complete.
- You can still manually change the state of any issue in a merged result.
- A merged result can serve as the baseline result, in which case the Intel Inspector propagates state (and note) information to the new result.

See Also

States
Choose Baseline Results for Setting States
Merge States Across Results

Merge States Across Results

By default, the Intel Inspector propagates state information from a baseline result (the immediately previous result of the same analysis type) when you open a result for the first time.
In addition, you can use the **Merge States...** option on the **Problems** pane context menu to merge state information from any result in any project into the currently open result.
The state merge operation updates in the currently open result the state assigned to any issues that appear in both results. The state of issues that appear only in the currently open result remain unchanged. Issues that appear only in the selected result are not added to the open result.
Combining the efforts of multiple team members working the same result is an excellent way to take advantage of the state merge feature.

To merge states across results:

1. In the **Problems** pane, right-click any problem.
2. From the context menu, choose **Merge States...** to display a **Merge States** dialog box.
3. Browse to and select any result in any project, then click the **Merge** button.

Outcome: The Intel Inspector updates in the currently open result the state assigned to any issues that appear in both results in the following manner:

Problem State/Selected Result	Problem State/Open Result						
	Not Fixed	New	Regression	Not a Problem	Deferred	Confirmed	Fixed
Not Fixed	Not Fixed	New	Regression	Not a Problem	Deferred	Confirmed	Fixed
New	Not Fixed	New	Regression	Not a Problem	Deferred	Confirmed	Fixed
Regression	Regression	Regression	Regression	Not a Problem	Deferred	Confirmed	Fixed
Not a Problem	Not a Problem	Not a Problem	Not a Problem	Not a Problem	Deferred	Confirmed	Fixed
Deferred	Deferred	Deferred	Deferred	Deferred	Deferred	Confirmed	Fixed
Confirmed	Confirmed	Confirmed	Confirmed	Confirmed	Confirmed	Confirmed	Fixed
Fixed	Fixed	Fixed	Fixed	Fixed	Fixed	Fixed	Fixed

See Also

State Merge
States
Choose Baseline Results for Setting States
Dialog Box: Merge States
Pane: Problems
Context Menus: Summary Window Panes

Report Result Data

Use the **Create Problem Report** option on Intel Inspector**Summary** window context menus to export result data in plain text format so you can distribute it to teammates in other media, such as email.

> **Tip**
> This functionality provides more data and versatility than the **Copy to Clipboard** option available on various context menus.

To report result data:

1. In the **Summary** window, right-click one or more code locations, problems, or problem sets.
2. From the context menu, choose **Create Problem Report...** to display a **Problem Report** dialog box containing result data in plain text format for the selected item(s).
3. Do one of the following:
   - Click the **Save to File** button, then identify a filename and location to export all the result data in the **Problem Report** dialog box to a text file.
   - Deselect the **Include call stacks** checkbox to exclude call stack information, click the **Save to File** button, then identify a filename and location to export the remaining result data to a text file.
   - Click the **Copy to Clipboard** button to copy all the result data in the **Problem Report** dialog box to the system clipboard.
   - Select result data in the window, then click the **Copy to Clipboard** button to copy the selected result data to the system clipboard.

See Also

Dialog Box: Problem Report
Pane: Code Locations
Pane: Problems
Context Menus: Summary Window Panes

Export and Import Files

Create result archive files to easily:

- Move Intel Inspector results between machines or even between operating systems. For example: You can export an archive file for a result produced on a Linux* machine and import it to a Windows* machine.
- Share results for collaboration purposes. For example: You can combine the efforts of multiple team members working a subset of assigned issues in the same result.
- Send results to the Intel© support team for troubleshooting purposes.

You can choose to include or exclude source files when you create result archive files.

When you import a result archive file into the current project, the Intel Inspector:

1. Extracts the result archive.
2. Names the extracted result `import@@@`, where `import` is a constant, and `@@@` represents the next available number.
3. Saves the extracted result to an `import@@@` directory of the same name.

You can also import a result not associated with a project or a result from another Intel© error-detection product into the current project. When you import such results, the Intel Inspector:

1. Makes a copy of the result.
2. Converts the copy to the Intel Inspector result format.
3. Names the new result `import@@@`, where `import` is a constant, and `@@@` represents the next available number.
4. Saves the new result to an `import@@@` directory of the same name.

Tip
Refinalize the imported result to apply current project symbol information. (Use the **Re-resolve** option on the **Solution Explorer** or **Project Navigator** pane context menu.)

See Also

Export Archives
Import Archives and Other Files
Intel Inspector Filenames and Locations

Export Archives

You can choose to include or exclude source files when you create Intel Inspector result archive files.

To export a result:

1. From the:
   - Visual Studio* **Solution Explorer**, right-click a result to display a context menu, then choose **Export Result...**.

   NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

   - Standalone Intel Inspector**Project Navigator** pane, right-click a result to display a context menu, then choose **Export Result...**.
2. In the **Export Result** dialog box, choose a result archive file destination.
3. Deselect the **Include source files** checkbox if desired.
4. Click the **Export** button.

Outcome: The Intel Inspector shows progress as it creates the result archive file and displays warnings if applicable.

See Also

Dialog Box: Export Result
Context Menus: Project Navigator
Context Menus: Solution Explorer

Import Archives and Other Files

You can import into the current project:

- Intel Inspector result archive files
- Intel Inspector results not associated with a project
- Results from other Intel® error-detection products

To import a file:

1. From the:
   - Visual Studio* menu, choose **Tools> Intel Inspector*version*> Import...**.

     NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

   - Standalone Intel Inspector GUI menu, choose **File** > **Import Result...**.
2. In the **Import Result** pane, choose an import file.
3. Click the **Import** button.

Outcome: If the import file is a result archive file, the Intel Inspector:

1. Extracts the result archive.
2. Names the extracted result `import@@@`, where `import` is a constant, and `@@@` represents the next available number.
3. Saves the extracted result to an `import@@@` directory of the same name.

If the import file is a result not associated with a project or a result from another Intel® error-detection product, the Intel Inspector:

1. Makes a copy of the result.
2. Converts the copy to the Intel Inspector result format.
3. Names the new result `import@@@`, where `import` is a constant, and `@@@` represents the next available number.
4. Saves the new result to an `import@@@` directory of the same name.

Tip

- Refinalize the imported result to apply current project symbol information. (Use the **Re-resolve** option on the **Solution Explorer** or **Project Navigator** pane context menu.)
- Change the cross-project result name template to change the name of the imported result.
- Change basic project properties to save the imported result to a custom location.

See Also

Pane: Import Result
Intel Inspector Filenames and Locations

Compare Results

The information in an Intel Inspector results comparison is most useful when you compare two results of the same analysis type from the same project, such as two **Detect Leaks** (`mi1`) results.

To compare two results:

1. From the:
   - Visual Studio* menu, choose **Tools> Intel Inspector> Compare...**.

   NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

   - Standalone Intel Inspector GUI menu, choose **File** > **Open** > **Results Comparison...**.
2. In the **Compare Results** pane, choose two `*.inspxe` files.
3. On the **Command** toolbar, click the **Compare** button.

Outcome: The Intel Inspector displays a result that looks similar to a standard analysis result with the following exceptions:

- Result tab name - The Intel© Inspector shows the names of the results selected for comparison in the following manner: **Results Comparison: "<Result 1 name>" - "<Result 2 name>"**
- **State** information - The Intel Inspector shows:
  - **Result 1** or **Result 2** if the issue was detected in only one of the results
  - **Both** if the issue was detected in both results

NOTE
Intel Inspector does not save result comparisons. The information disappears if you close the result tab, open another project, close the Standalone Intel Inspector GUI, or close the Visual Studio* IDE.

See Also

Pane: Compare Results
Toolbar: Command

Test for Regressions

The recommended approach for regression testing with the Intel© Inspector involves these basic steps:

1. Create a gold standard:
   - Use the GUI (or the `inspxe-cl` command) to create a result from which you have successfully eliminated all real problems.
   - Use the `inspxe-cl` command to create a suppression file to match any remaining problems you have determined you can ignore.
2. Change your source code.
3. Use the `inspxe-cl` command (or the GUI) to check for new problems:
   - Apply the suppression file when you run a new analysis.
   - Determine if the changed source code introduced new problems that do not match those in the suppression file.

Intel Inspector also offers alternative ways to test for regressions.

Tes for Regressions: Recommended Approach

The recommended approach for regression testing with the Intel Inspector involves these basic steps:

1. Create a gold standard:
   - Use the GUI (or the `inspxe-cl` command) to create a result from which you have successfully eliminated all real problems.
   - Use the `inspxe-cl` command to create a suppression file to match any remaining problems you have determined you can ignore.
2. Change your source code.
3. Use the `inspxe-cl` command (or the GUI) to check for new problems:
   - Apply the suppression file when you run a new analysis.
   - Determine if the changed source code introduced new problems that do not match those in the suppression file.

Create a Result

In the GUI (or the `inspxe-cl` command tool if you prefer), create a result from which you have successfully eliminated all real problems. Any problem remaining in the result is a positive you have determined you can ignore. For example:

- You may have **Data race** problem caused by more than one thread accessing a variable without synchronization; however, you know there are other timing factors that make it impossible for the **Data race** to occur.
- You may have an issue in a third-party library that is of no interest to you.

Create one result for each data set you plan to test, or one result for your entire application. You can use a result created with the same analysis type you plan to use for the new analysis or with the widest analysis type available.

Create a Suppression File

In the `inspxe-cl` command tool, use the `create-suppression-file` action to generate a suppress-all file for all remaining problems.

For example: The following `inspxe-cl` command creates a `mySup` suppression file with rules that suppress every error in the `r002ti2` result.

```
$ inspxe-cl -create-suppression-file mySup -result-dir r002ti2
```

Create a suppress-all file for each data set you plan to test and store them all in one directory. If you prefer to use the GUI to perform the remainder of this recommended approach, use the **Suppressions** tab in the **Project Properties** dialog box to identify that directory.

> **Caution**
> Applying many suppression files or large suppressions files during analysis may slow down finalization, so make sure you remove obsolete suppression files from the directory and generate new suppression files when previously known issues are eliminated.

Change Your Source Code

Use your development environment to edit your source code.

Suppression files remain effective for changed source code with the following caveats:

- Intel Inspector identifies source code lines relative to the start of a function, not as absolute values. So you can insert and delete lines of code anywhere in source files without rendering a suppression file ineffective except inside a function where a problem is marked for suppression.
- Adding or deleting lines in a function prior to the location of a suppressed problem causes a problem to no longer match the suppression rule. Therefore the problem appears as a new problem in subsequent analysis runs.

> **Tip**
> Intel Inspector-created suppression files contain (usually multiple) narrow rules that include line numbers (to not accidentally suppress extra problems). Consider handwriting your own suppression files to create fewer rules with wider reach - or at least rules that do not include line numbers.

Run a New Analysis

In the `inspxe-cl` command tool (or the GUI if you prefer), run a new analysis with the suppress-all file.

> **Tip**
> You can incrementally apply multiple suppress-all files if your gold standard result changes.

For example: The following `inspxe-cl` command uses the `suppression-file` action-option to apply all suppression files in the `C:\My Inspector Results\suppressions` directory when collecting a new result for the `myApp` application.

```
$ inspxe-cl -collect ti2 -suppression-file "C:\My Inspector Results\suppressions" -- myApp
```

Determine if the Changed Source Code Introduced New Problems

When analysis is complete, investigate the result to check for new problems:

- In the `inspxe-cl` command tool: Check the automatically generated `inspxe-cl.txt` Summary report.
- In the GUI: Check the **Problems** pane.

Any new problems are regressions.

Automate Your Regression Testing Process

Intel Inspector must execute your code path to find errors in it, so you should consider running the Intel Inspector on multiple versions of your code, on different workloads that stress different code paths, as well as on corner cases. However, multiple Intel Inspector analysis runs can take time.

It may be more efficient to automate: Batch process multiple analysis runs with different data sets passed in as arguments - possibly overnight - and have the computer do the work for you.

The simplest automation scenario involves adding the following steps to a script:

1. Set up the `inspxe-cl` command environment.
2. Invoke the `inspxe-cl` command to run an analysis and apply the suppress-all file(s).
3. Determine if the changed source code introduced new problems by checking the returned exit code in the automatically generated `inspxe-cl.txt` Summary report when analysis is complete. If the reported exit code equals 0, there are no new problems detected in unsuppressed code locations. If the reported exit code does not equal 0, consider sending the Summary report to a log file and email the log file to an engineer for further triage.
4. Repeat steps 2 and 3 as necessary.

See Also

Test for Regressions: Other Approaches

Test for Regressions: Other Approaches

Intel Inspector also offers alternative ways to test for regressions.

View Problem States to Check for New Problems

When you run an analysis to create a new result, the Intel Inspector automatically:

- Propagates state information from a baseline (older) result to the new result.
- Sets the state in the new result of all problems that do not appear in the baseline result to **New**.

Take advantage of this *free* information for regression testing purposes.

In the GUI:

1. Establish a baseline result using the **Options-State Management** dialog box.
2. Run a new analysis.
3. When analysis is complete, use the **Filters** pane to temporarily limit the items displayed in the **Problems** pane to those with a state of **New.**

In the `inspxe-cl` command tool:

1. Run a new analysis against a baseline result.

   For example: The following `inspxe-cl` command collects a new result for the `MyApp` application against a `dataset1_baseline` baseline result and writes the new result to the `myRes002mi3` directory in the current working directory.

```
$ inspxe-cl -c mi3 -baseline-result ./dataset1_baseline -r myRes002mi3 -- ./myApp
```

2. When analysis is complete, generate a Status report for the new result to determine if there are any new problems.

   For example: The following `inspxe-cl` command generates a Status report for the `myRes002mi3` result.

```
$ inspxe-cl -report status -result-dir myRes002mi3
```

3. If there are new problems, generate a Problems report containing only the new problems.

   For example: The following `inspxe-cl` command generates a Problems report of new problems detected in the `myRes002mi3` result.

```
$ inspxe-cl -report problems -filter state=new -result-dir myRes002mi3
```

NOTE

- By default, the Intel Inspector establishes a baseline result from the immediately previous result of the same analysis type.
- You can use a baseline result created with the same analysis type you plan to use for the new analysis or with the widest analysis type available.
- Use one baseline result per data set you plan to test, or one baseline result for your entire application.
- You can also automate these `inspxe-cl` command tool steps.

View Results Side-by-Side to Check for Differences

You can also create a result or generate a report that compares two results.

In the GUI:

1. Use the comparison feature to identify a previous result and a new result, and create a temporary result that shows the following state information for each detected problem:
   - **Result 1** or **Result 2** if the problem appears in only one result
   - **Both** if the problem appears in both results
2. Use the **Filters** pane to temporarily limit the items displayed in the **Problems** pane to those that appear in only the new result. For example: If **Result 1** is a previous result and **Result 2** is the new result, filter for problems with a state of **Result 2**.

In the `inspxe-cl` command tool, generate a Summary report for two results simultaneously to produce a differences report.

For example: The following `inspxe-cl` command generates a differences report for two previously collected results: `myRes001mi3` and `myRes002mi3`.

```
$ inspxe-cl -report summary -result-dir myRes001mi3 -result-dir myRes002mi3
```

NOTE
The information in a results comparison is most useful when you compare two results of the same analysis type from the same project.

See Also

Tes for Regressions: Recommended Approach
States
Choose Baseline Results for Setting States
Collect Results
Compare Results
Selecting Filter Criteria
Collecting Result Data from the Command Line
Reporting Result Data from the Command Line
baseline-result action-option

Suppressions Support

You (and your development team) will always know more about your code than the Intel Inspector can ever know. For example:

- Your team lead may know of error-prone third-party code.

- Your team architect may know specific code passages are correct as coded.
- You may know you are currently fixing a specific bug.

Suppressing known issues based on rules you define can improve your productivity by helping you focus on only those issues that currently require your attention.

Intel® Inspector offers a variety of ways to take advantage of the power of suppression rules:

You Can Do This	Using This	Helpful Information
Define suppression rules	Intel Inspector GUI **Tip** This is the recommended way to define suppressions.	• Typical Suppressions Usage Model Using the GUI • Suppression Rule Range in the GUI • Suppression Rule Examples in the GUI • Defining Suppression Rules in the GUI • Deleting Suppression Rules Applied to Marked Result Data in the GUI
	Intel Inspector command line interface	• Creating Suppression Files • create-suppression-file
	Text editor	• Typical Suppressions Usage Models Using Handwritten Suppression Files • Suppression Rule Syntax in Text Format • Suppression Rule Examples in Text Format
Edit suppression rules/files	Text editor	• Suppression Rule Syntax in Text Format • Suppression Rule Examples in Text Format
Convert third-party suppression files to the Intel Inspector suppression file format	Intel Inspector command line interface	• Third-party Suppression Files • Typical Usage Models Using Third-party Suppression Files • Converting Third-Party Suppression Files • convert-suppression-file
Apply suppression rules/files	Intel Inspector GUI	Setting Project Properties-Advanced
	Intel Inspector command line interface	Applying Suppression Files
Review/remove suppression rules/files	Intel Inspector GUI	Managing Suppression Rules

Suppressions Support Using the GUI

Typical Suppressions Usage Model Using the GUI

The following is the typical usage model for handling GUI-created suppression rules in the Intel Inspector:

1. In the **Target** tab of the **Project Properties** dialog box, set **Suppressions** to **Apply suppressions**.
2. Do one of the following:
   - Collect a new result.
   - Open an existing result (collected using the Intel Inspector GUI or command line interface).

3. Define a suppression rule based on one or more code locations in a problem. Save the new suppression rule to the `default.sup` file or create a new `.sup` file in the default suppression directory for the project.

   Outcome: In the **Code Locations** and **Problems** panes, the Intel Inspector marks (strikes through) all result data *potentially* impacted by the rules in the file. (It may take some time for all marks to appear. The marks remain in the result even if you close and then reopen the result.)
4. Review the marked data to assess the effectiveness and potential reach of the rule. If necessary, delete the rule and start over, or edit the suppression file in a text editor.
5. Repeat steps 3 and 4 as necessary.
6. Run a new analysis using the Intel Inspector GUI or command line interface.

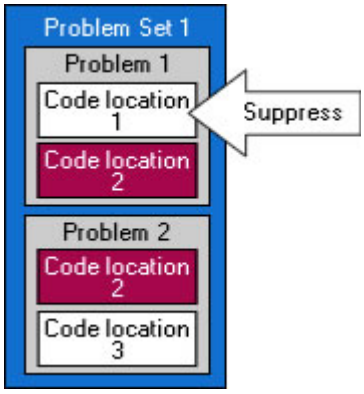
   Outcome: The Intel Inspector does not collect result data impacted by the new suppression rule(s) and the rules in any other files and directories identified in the **Suppressions** tab of the **Project Properties** dialog box.

See Also

Interpret Result Data and Resolve Issues
Suppression Rule Reach in the GUI
Suppressions Support
Defining Suppression Rules in the GUI
Deleting Suppression Rules Applied to Marked Result Data in the GUI
Manage Suppression Rules
Set Project Properties-Advanced
Intel Inspector Filenames and Locations

Suppression Rule Reach in the GUI

When you define a suppression rule using the Intel Inspector**Create Suppression** dialog box, the Intel Inspector marks all result data on the **Summary** window that is *potentially* impacted by the rule. For example:

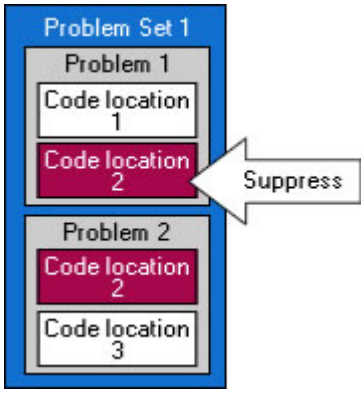


Example 1

Suppress *Problem 1* by creating a suppression rule based on *Code Location 1*.

Intel Inspector responds by marking anything that touches *Problem 1*:

- *Code Location 1*
- *Code Location 2*
- *Problem Set 1*



Example 2

Suppress *Problem 1* by creating a suppression rule based on *Code Location 2*. *Code Location 2* is also part of *Problem 2*.

Intel Inspector responds by marking anything that touches both *Problem 1* and *Problem 2*:

- *Code Location 1*
- *Code Location 2*
- *Code Location 3*
- *Problem Set 1*

See Also

Interpret Result Data and Resolve Issues
Suppressions Support
Defining Suppression Rules in the GUI
Manage Suppression Rules
Intel Inspector Filenames and Locations

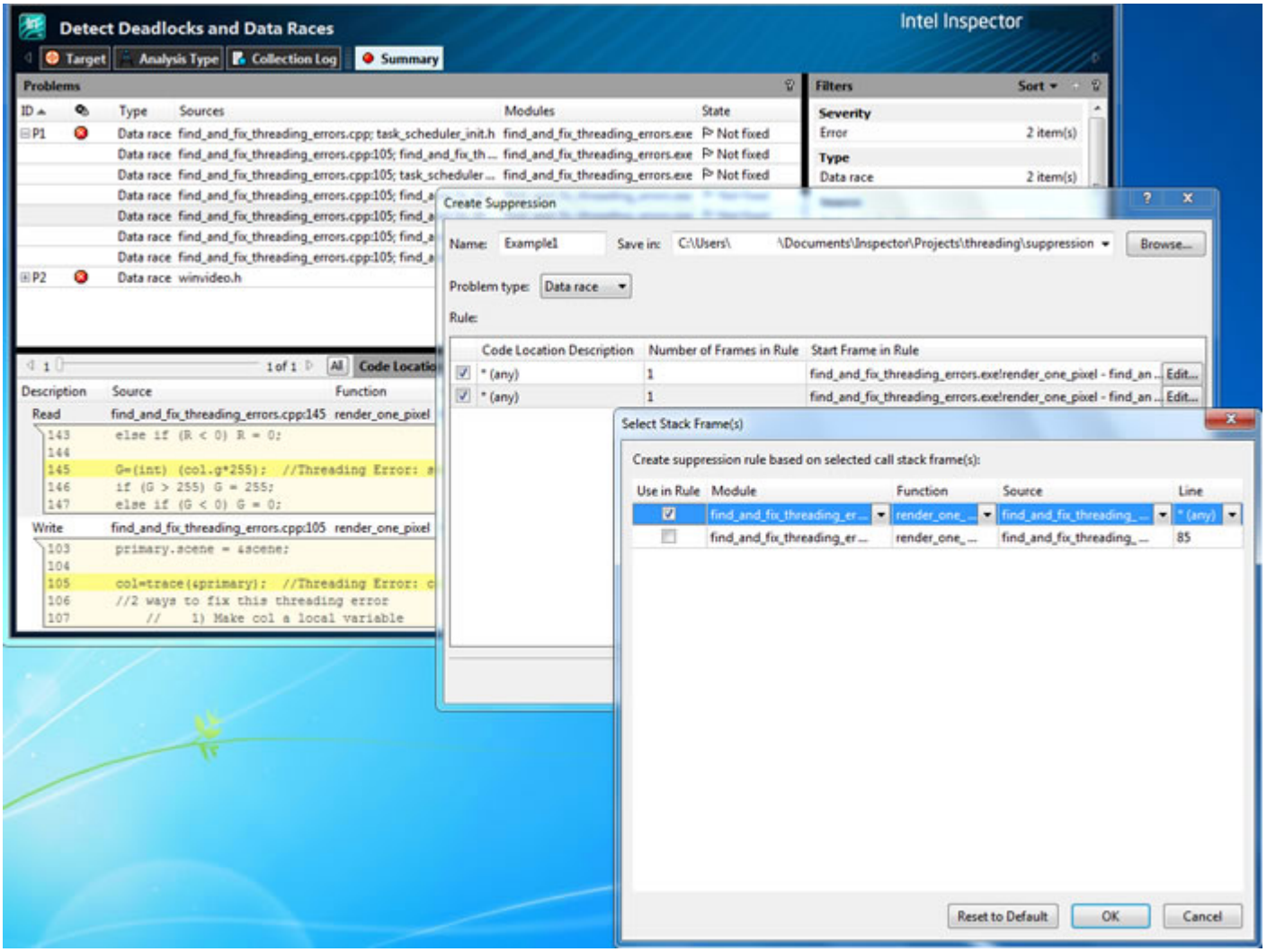
Suppression Rule Examples in the GUI

Tip

- Although you are ultimately trying to suppress problems, the Intel Inspector*vehicle* for defining a suppression rule is one or more code locations.
- Narrow rules suppress a limited number of relevant problems; wider rules suppress a greater number of relevant problems.
- Every rule applied during analysis adds processing time.
- The goal: Suppress the greatest number of relevant problems with the fewest number of rules.
- To review rules to be applied during analysis, check the **Suppressions** tab of the **Project Properties** dialog box.
- To apply rules during analysis, select the **Apply Suppressions** radio button on the **Target** tab of the **Project Properties** dialog box.
- A code location may be part of multiple problems; therefore, multiple rules may suppress the same code location, or a rule created to suppress one problem may partially impact another problem.

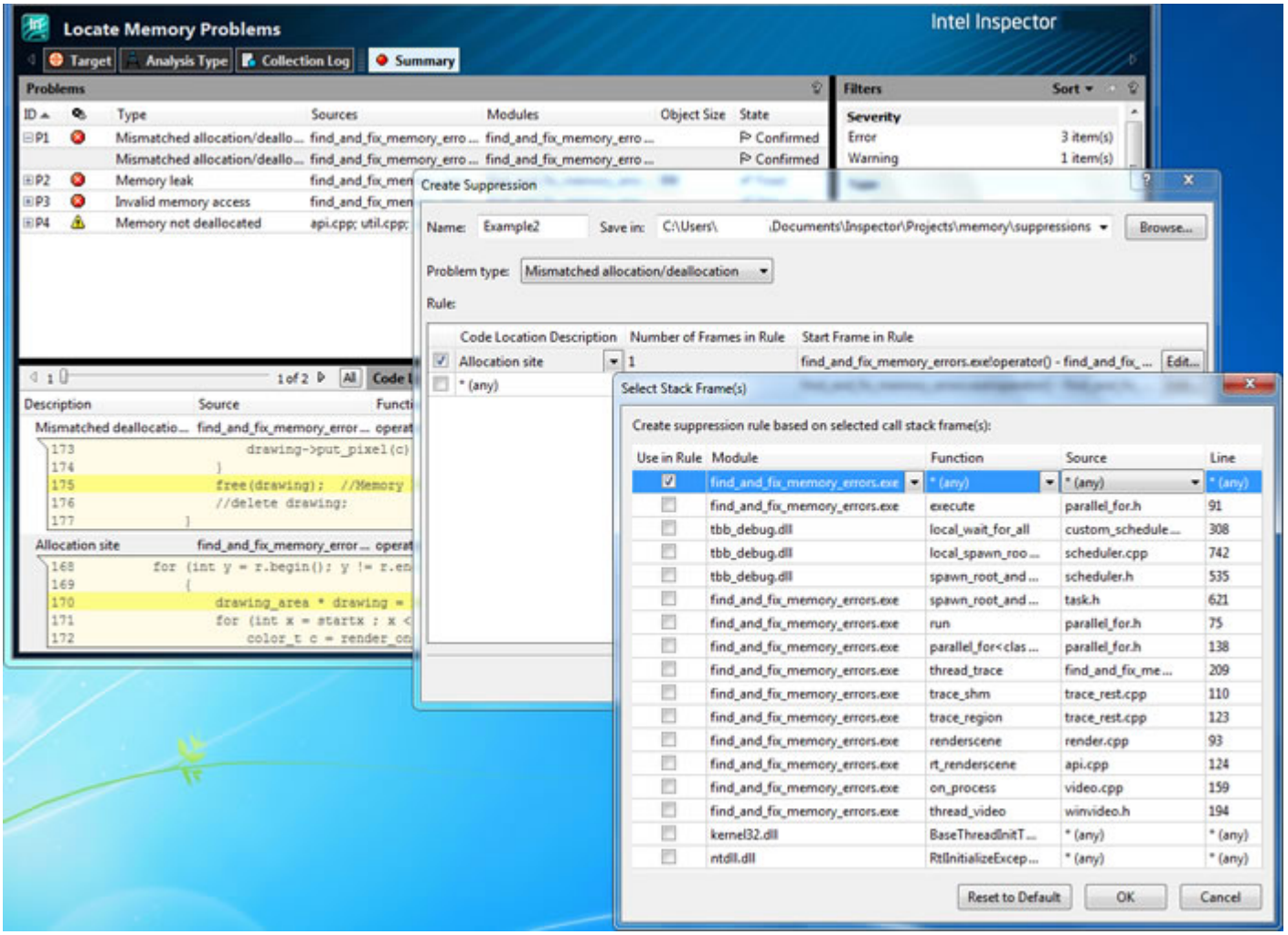
Suppression Rule Example 1

Suppression rule Example1 suppresses any **Data race** problem between **Read** and **Write** code locations with the following characteristics as the last-called frame in the stack: `find_and_fix_threading_errors.exe` module, `render_one_pixel` function, and `find_and_fix_threading_errors.cpp` source file.



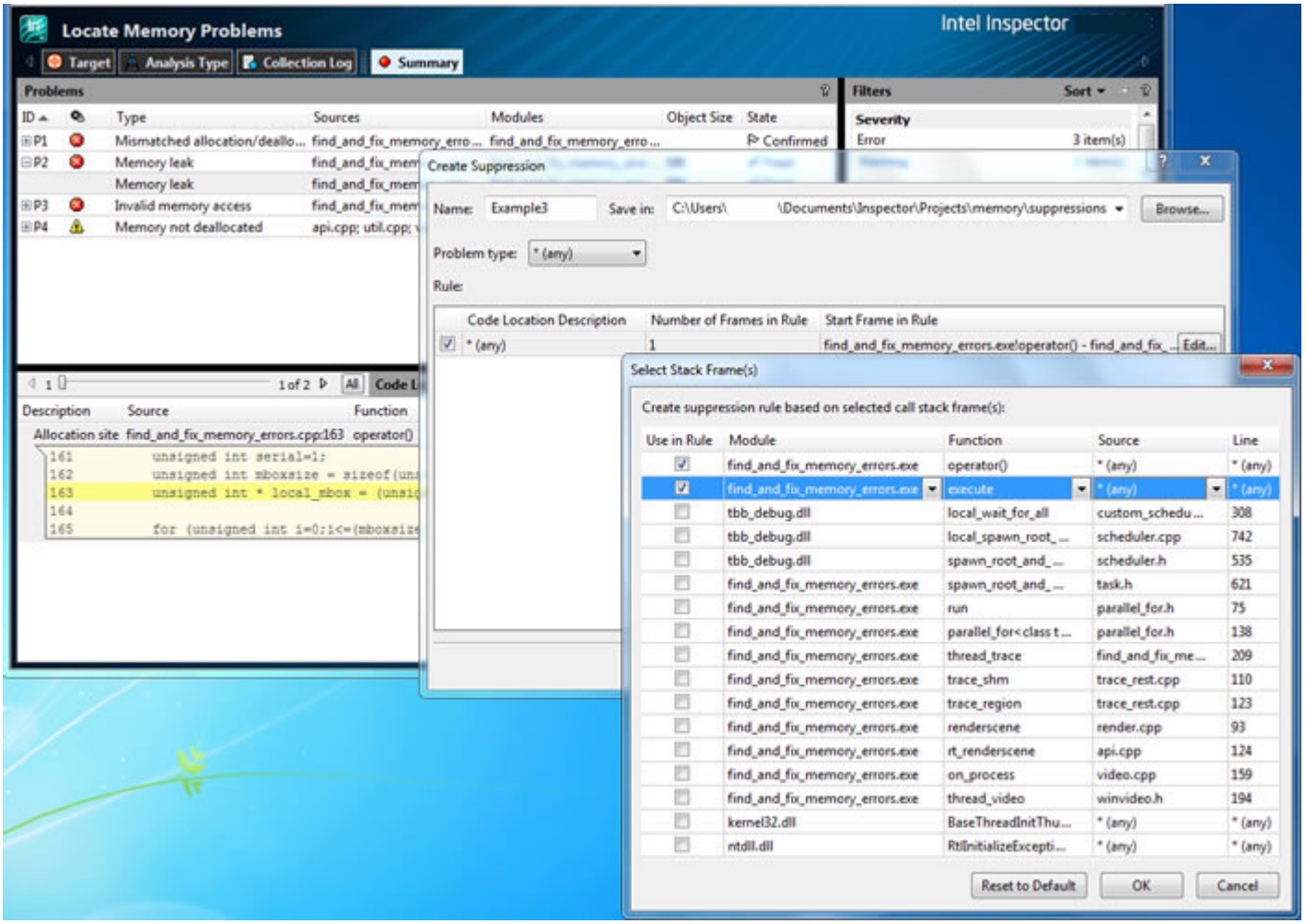
Suppression Rule Example 2

Suppression rule Example2 suppresses any problem with an **Allocation site** code location whose last-called frame in the stack is in the `find_and_fix_memory_errors.exe` module.



Suppression Rule Example 3

Suppression rule Example3 suppresses any problem where the last-called frame in the stack is in the `operator()` function of the `find_and_fix_memory_errors.exe` module and the end of the stack path is `execute` calling `operator()`.



See Also

Interpreting Result Data and Resolving Issues
Suppressions Support
Defining Suppression Rules in the GUI
Managing Suppression Rules
Intel Inspector Filenames and Locations

Defining Suppression Rules in the GUI

Prerequisite: Apply suppressions when you configure a project.

To create a suppression rule to help you focus during future analysis runs on only those issues that require your attention:

1. Choose problems to suppress during future analysis runs.
2. Identify the new rule by name and storage location.
3. Widen the default rule reach to suppress the greatest number of relevant problems.
4. Save the rule.

Tip

- Although you are ultimately trying to suppress problems, the Intel Inspector*vehicle* for defining a suppression rule is one or more code locations.
- Narrow rules suppress a limited number of relevant problems; wider rules suppress a greater number of relevant problems.
- Every rule applied during analysis adds processing time.
- The goal: Suppress the greatest number of relevant problems with the fewest number of rules.
- To review rules to be applied during analysis, check the **Suppressions** tab of the **Project Properties** dialog box.
- To apply rules during analysis, select the **Apply Suppressions** radio button on the **Target** tab of the **Project Properties** dialog box.
- A code location may be part of multiple problems; therefore, multiple rules may suppress the same code location, or a rule created to suppress one problem may partially impact another problem.
- See suppression rule examples for more information.

1. Choose Problems to Suppress During Future Analysis Runs

1. In the **Code Locations** pane, right-click a code location for an occurrence of the problem you want to suppress during future analysis runs.
2. In the context menu, choose **Suppress...**.

Outcome: The Intel Inspector displays the **Create Suppression** dialog box. The **Rule** grid in the center of the dialog box shows the right-clicked code location and any related code locations in the specific occurrence of the problem.

2. Identify the new rule by name and storage location

1. In the **Name** text box, type a short description to distinguish the new rule from other rules.
2. If you want to store the new rule somewhere besides the default location, browse to and choose a filename and/or location.

Tip
Choose a non-default location only to make the rule easily accessible to others.

3. Widen the default rule reach to suppress the greatest number of relevant problems

By default, the Intel Inspector drafts a rule that suppresses problems based on:

- A specific problem type
- All the code locations in the problem occurrence
- A specific module, function, source, and line in the last-called stack frame in each code location.

To widen the rule reach, try one or more of the following - do not suppress problems by:

- A specific problem type - In the **Problem type** drop-down list, click the drop-down arrow and choose *** (any)**.
- A specific code location - In the **Rule** grid, deselect the code location checkbox.
- A specific code location description - In the **Code Location Description** drop-down list, click the drop-down arrow and choose *** (any)**.

NOTE
There are three **Code Location Description** possibilities: **Allocation site**, **Deallocation site**, and *** (any)**. Memory error example: If you right-click a **Mismatched allocation site** code location for an occurrence of a **Mismatched allocation/deallocation** problem, the Intel Inspector displays one code location marked **Allocation site** and one code location marked *** (any)**; you can click the drop-down arrow to change **Allocation site** to *** (any)**. Threading error example: If you right-click a **Read** code location for an occurrence of a Read/Write **Data race** problem, the Intel Inspector displays two code locations marked *** (any)** and does not provide drop-down arrows.

- A specific stack frame module, function, source, and line number -

  1. Click a code location **Edit** button to open the **Select Stack Frame(s)** dialog box.
  2. Notice you can select a different stack frame or multiple stack frames as the rule focus.
  3. In the **Module**, **Function**, **Source**, and/or **Line** drop-down list for each selected stack frame, click the drop-down arrow and choose *** (any)**

Tip

- A single code location with all but one characteristic set to *** (any)** is ideal for widening a rule to suppress the greatest number of relevant problems.
- Suppression rules are more robust if you specify a stack with multiple frames instead of frames with line numbers. (Because line numbers can be altered by code insertions or deletions, suppressions may be rendered ineffective by even minor code maintenance. Stack-based suppressions require larger code changes to invalidate them, such as changes to function call sequences.)

4. Save the Rule

Click the **Create** button.

Outcome

Outcome:

- Intel Inspector saves the new suppression rule in a `.sup` file.
- In the **Code Locations** and **Problems** panes, the Intel Inspector marks (strikes through) all result data *potentially* impacted by the rule. (It may take some time for all marks to appear.) The marks remain in the result even if you close and then reopen the result.

Tip
Review the marked data to assess the effectiveness and potential reach of the rule. If necessary, delete the rule and start over, or edit the suppression file in a text editor.

- During the next analysis run, if **Suppressions** in the **Target** tab of the **Project Properties** dialog box is set to:
  - **Apply suppressions** - The Intel Inspector does not collect result data impacted by the suppression rules in files and directories specified in the **Suppressions** tab of the **Project Properties** dialog box.
  - **Do not apply suppressions** - The Intel Inspector ignores all suppression rules.

See Also

Interpret Result Data and Resolve Issues
Suppressions Support
Pane: Code Locations
Dialog Box: Create Suppression
Dialog Box: Project Properties-Suppressions

Dialog Box: Project Properties-Target
Dialog Box: Select Stack Frame(s)
Dialog Box: View Stack
Context Menus: Summary Window Panes
Intel Inspector Filenames and Locations

Deleting Suppression Rules Applied to Marked Result Data in the GUI

Prerequisite: Marked result data in the Intel Inspector GUI from a new suppression rule(s).

To delete a new rule:

1. In the **Code Locations** pane, right-click marked result data.
2. In the context menu, choose **Do Not Suppress** to display the **Delete Suppressions** dialog box.
3. Select a rule to delete.
4. If desired, click the **View...** button to view the stack frame(s) in the rule. When you are finished viewing, close the **View Stack** dialog box.
5. Click the **Remove** button.

Outcome: In the **Code Locations** and **Problems** panes, the Intel Inspector unmarks result data accordingly.

See Also

Interpret Result Data and Resolve Issues
Suppressions Support
Manage Suppression Rules
Dialog Box: Delete Suppressions
Context Menus: Summary Window Panes
Context Menus: Sources Window Panes
Dialog Box: View Stack

Suppressions Support Using Handwritten Suppression Rules

Typical Suppressions Usage Models Using Handwritten Suppression Files

There are two typical usage models for handling handwritten suppression files in the Intel Inspector. They differ depending on how you want to confirm the effectiveness of the files:

- Apply the handwritten file to an existing result, check the resulting strikethrough marks, then continue editing the file in the text editor as necessary.
- Apply the handwritten file to a new result, check the result for problems that should have but did not vanish, then continue editing the file in the text editor as necessary.

> **NOTE**
> These usage models also apply to editing suppression files already created using the Intel Inspector GUI or command line interface.

Apply the File to an Existing Result

1. Create the handwritten file using a text editor. Save the handwritten file to the default suppression directory for the project. Use the default `.sup` extension.
2. In the Intel Inspector GUI, in the **Target** tab of the **Project Properties** dialog box, set **Suppressions** to **Apply suppressions**.
3. Open an existing result for the project (collected using the Intel Inspector GUI or command line interface).

Outcome: In the **Code Locations** and **Problems** panes, the Intel Inspector marks (strikes through) all result data *potentially* impacted by the rules in the handwritten file. (It may take some time for all marks to appear. The marks remain in the result even if you close and then reopen the result.)

4. Review the marked data to assess the effectiveness and potential reach of the rules in the handwritten file. If necessary, continue editing the handwritten file in the text editor.
5. Repeat steps 3 and 4 as necessary.
6. Run a new analysis using the Intel Inspector GUI or command line interface.

   Outcome: The Intel Inspector does not collect result data impacted by suppression rules in the handwritten file and the rules in other files and directories identified in the **Suppressions** tab of the **Project Properties** dialog box.

Apply the File to a New Result

1. Create the handwritten file using a text editor. Save the handwritten file to the default suppression directory for the project. Use the default `.sup` extension.
2. In the Intel Inspector GUI, in the **Target** tab of the **Project Properties** dialog box, set **Suppressions** to **Apply suppressions**.
3. Run a new analysis using the Intel Inspector GUI or command line interface.

   Outcome: The Intel Inspector does not collect result data impacted by suppression rules in the handwritten file and the rules in other files and directories identified in the **Suppressions** tab of the **Project Properties** dialog box).
4. Review the remaining problems in the result to assess the effectiveness and potential reach of the rules in the handwritten file. If necessary, continue editing the handwritten file in the text editor.
5. Repeat steps 3 and 4 as necessary.

See Also

Interpret Result Data and Resolve Issues
Suppression Rule Reach in the GUI
Suppression Rule Syntax in Text Format
Suppressions Support
Manage Suppression Rules
Set Project Properties-Advanced
Intel Inspector Filenames and Locations

Suppression Rule Syntax in Text Format

Tip

- Although you are ultimately trying to suppress problems, the Intel Inspector*vehicle* for defining a suppression rule is one or more code locations.
- Narrow rules suppress a limited number of relevant problems; wider rules suppress a greater number of relevant problems.
- Every rule applied during analysis adds processing time.
- The goal: Suppress the greatest number of relevant problems with the fewest number of rules.
- To review rules to be applied during analysis, check the **Suppressions** tab of the **Project Properties** dialog box.
- To apply rules during analysis, select the **Apply Suppressions** radio button on the **Target** tab of the **Project Properties** dialog box.
- A code location may be part of multiple problems; therefore, multiple rules may suppress the same code location, or a rule created to suppress one problem may partially impact another problem.
- See suppression rule examples for more information.

Syntax

Each suppression rule consists of an optional name, optional problem type, optional allocation/deallocation site code location, and a stack with required and optional information in the following syntax:

```
Suppression = {
  Name = "<Rule name>";
  Type = { <problem-type> }
  Stacks = {
       allocation|deallocation = {
       mod=<module-name>, func=<function-name>, src=<source-filename>,
       line=<absolute-line-number>, func_line=<relative-function-line-number>;
    }
  }
}
```

For example: This rule suppresses any **Memory not deallocated** problem where the last-called frame in the problem stack is in the `my_alloc` function from the `alloc.c` source file.

```
Suppression = {
    Type = { reachable_memory_leak }
    Stacks = {
        {
            func=my_alloc, src=alloc.c;
        }
    }
}
```

Tip

- Keywords values are case sensitive. For example: `Type = { reachable_memory_leak }` is valid; `Type = { Reachable_memory_leak }` is invalid.
- Semicolons are optional except for at the end of the line (or set of lines) that describes a specific stack frame.
- Insert comments using the pound sign (`#`) as the first non-space character on a line.

Type Keyword

Use `Type` to indicate a problem must match a specific problem **Type** to be suppressed.

You can identify one `Type` per rule or omit `Type` entirely. If you omit `Type`, the Intel Inspector matches any problem **Type**.

Tip
Omit `Type` to widen the reach of a rule.

Valid `Type` values include the following:

Type Descriptions	Corresponding Type Values
Cross-thread stack access	cross_thread_stack_access
Data race	datarace
Deadlock	deadlock
GDI resource leak	gdi_handle_leak

Type Descriptions	Corresponding Type Values
Incorrect memcpy call	invalid_call
Invalid deallocation	invalid_memory_unmap_notmapped
Invalid memory access	invalid_memory_access
Invalid partial memory access	invalid_memory_access_partial
Kernel resource leak	kernel_handle_leak
Lock hierarchy violation	potential_deadlock
Memory growth	intermediate_memory_leak
Memory leak	unreachable_memory_leak
Memory not deallocated	reachable_memory_leak
Mismatched allocation/deallocation	invalid_deallocation_mapped
Mismatched allocation/deallocation	mismatched_deallocation
Mismatched allocation/deallocation	invalid_memory_unmap_allocated
Missing allocation	invalid_deallocation
Uninitialized memory access	uninitialized_memory_access
Uninitialized partial memory access	uninitialized_memory_access_partial

Stacks Keyword

Use `Stacks` to indicate a problem must match a specific stack to be suppressed.

Each stack frame must contain at least one of the following: `mod`, `func`, `src`, or an ellipsis (`...`). You cannot include `line` or `func_line` without `src`. If you supply both `line` and `func_line`, the latter overrides the former. You can use the asterisk (`*`) wildcard when identifying modules, functions, and source files. Enclose function names that contain special symbols, such as commas, equation marks, and blanks, in quotation marks.

Use an ellipsis (`...`) to wildcard a set of frames (zero or more). Use a triple-bang (`!!!`) to match one or more unresolved frames, such as those for which a function name cannot be determined. (This maintains backward-compatibility with *best location* in previous Intel Inspector versions.) By default, matching starts with the last-called frame in the stack. If you want to match a frame somewhere in the stack, insert an ellipsis as the first frame.

Tip

- Each additional frame in the stack narrows the reach of a rule.
- Each additional keyword in a frame narrows the reach of a rule.
- Suppression rules are more robust if you specify a stack with multiple frames instead of frames with line numbers. (Because line numbers can be altered by code insertions or deletions, suppressions may be rendered ineffective by even minor code maintenance. Stack-based suppressions require larger code changes to invalidate them, such as changes to function call sequences.)

Allocation and Deallocation Keywords

Use `allocation` or `deallocation` to indicate a frame must match an **Allocation site** or **Deallocation site** code location respectively for a problem to be suppressed.

You can include an `allocation` or `deallocation` keyword for each frame or omit the keywords entirely. If you omit the keywords, the Intel Inspector matches any code location in the frame.

Tip
Omit `allocation` and `deallocation` to widen the reach of a rule.

See Also

Interpret Result Data and Resolve Issues
Suppressions Support
Manage Suppression Rules
Intel Inspector Filenames and Locations

Suppression Rule Examples in Text Format

Tip

- Although you are ultimately trying to suppress problems, the Intel Inspector*vehicle* for defining a suppression rule is one or more code locations.
- Narrow rules suppress a limited number of relevant problems; wider rules suppress a greater number of relevant problems.
- Every rule applied during analysis adds processing time.
- The goal: Suppress the greatest number of relevant problems with the fewest number of rules.
- To review rules to be applied during analysis, check the **Suppressions** tab of the **Project Properties** dialog box.
- To apply rules during analysis, select the **Apply Suppressions** radio button on the **Target** tab of the **Project Properties** dialog box.
- A code location may be part of multiple problems; therefore, multiple rules may suppress the same code location, or a rule created to suppress one problem may partially impact another problem.

Suppression Rule Example 1

Suppression rule Example1 suppresses any problem where the last-called frame in the stack is in the `m.so` module.

```
Suppression = {
    Name = "Example1";
    Stacks = {
        {
            mod=m.so;
        }
    }
}
```

Suppression Rule Example 2

Suppression rule Example2 suppresses any **Data race** problem with one code location in the `a.out` module, `update_x` function and another code location in the `a.out` module, `update_y` function.

```
Suppression = {
    Name = "Example2";
    Type = { datarace }
```

```
    Stacks = {
        {
            mod=a.out, func=update_x;
        }
        {
            mod=a.out, func=update_y;
        }
    }
}
```

Suppression Rule Example 3

Suppression rule Example3 suppresses any **Memory not deallocated** problem where the last-called frame in the stack is an **Allocation site** code location in the `my_alloc` function from the `alloc.c` source file.

```
Suppression = {
    Name = "Example3";
    Type = { reachable_memory_leak }
    Stacks = {
        allocation = {
            func=my_alloc, src=alloc.c;
        }
    }
}
```

Suppression Rule Example 4

Suppression rule Example4 suppresses any **Uninitialized memory access** where the last-called frame in the stack is in the `_itoa_word` function and the stack path is `main` calling `printf` calling `vfprintf` calling `_itoa_word`.

```
Suppression = {
    Name = "Example4";
    Type = { uninitialized_memory_access }
    Stacks = {
        {
            func=_itoa_word;
            func=vfprintf;
            func=printf;
            func=main;
        }
    }
}
```

Suppression Rule Example 5

Suppression rule Example5 suppresses any **Memory leak** problem where the last-called frame in the stack is in the `malloc` function and the stack path is `main` calling `ccc` calling `ddd` calling `malloc`, possibly through a series of interim function calls.

```
Suppression = {
    Name = "Example5";
    Type = { unreachable_memory_leak }
    Stacks = {
        {
            func=malloc;
            ...;
            func=ddd;
            ...;
```

```
            func=ccc;
            ...;
            func=main;
        }
    }
}
```

Suppression Rule Example 6

Suppression rule Example6 suppresses any problem with an **Allocation site** code location in the `my_alloc` function from the `alloc.c` source file anywhere in the stack.

```
Suppression = {
    Name = "Example6";
    Stacks = {
        allocation = {
            ...;
            func=my_alloc, src=alloc.c;
        }
    }
}
```

See Also

Interpret Result Data and Resolve Issues
Suppressions Support
Manage Suppression Rules
Intel Inspector Filenames and Locations

Suppressions Support Using Third-party Suppression Rules

Typical Suppressions Usage Models Using Third-party Suppression Files

There are two typical usage models for handling third-party suppression files in the Intel Inspector. They differ depending on how you want to confirm the effectiveness of the converted files:

- Apply the converted file to an existing result, check the resulting strikethrough marks, then hand-edit the converted file in a text editor as necessary.
- Apply the converted file to a new result, check the result for problems that should have but did not vanish, then hand-edit the converted file in a text editor as necessary.

Apply the Converted File to an Existing Result

1. In the Intel Inspector command line interface, use the `convert-suppression-file` action to convert an IBM Rational Purify* or Valgrind* suppression file to the Intel Inspector suppression file format. Save the converted file to the default suppression directory for the project. Use the default `.sup` extension.
2. In the Intel Inspector GUI, in the **Target** tab of the **Project Properties** dialog box, set **Suppressions** to **Apply suppressions**.
3. Open an existing result for the project (collected using the Intel Inspector GUI or command line interface).

   Outcome: In the **Code Locations** and **Problems** panes, the Intel Inspector marks (strikes through) all result data *potentially* impacted by the rules in the converted file. (It may take some time for all marks to appear. The marks remain in the result even if you close and then reopen the result.)
4. Review the marked data to assess the effectiveness and potential reach of the rules in the converted file. If necessary, edit the converted file in a text editor.
5. Repeat steps 3 and 4 as necessary.

6. Run a new analysis using the Intel Inspector GUI or command line interface.

   Outcome: The Intel Inspector does not collect result data impacted by suppression rules in the converted file and the rules in other files and directories identified in the **Suppressions** tab of the **Project Properties** dialog box.

Apply the Converted File to a New Result

1. In the Intel Inspector command line interface, use the `convert-suppression-file` action to convert an IBM Rational Purify* or Valgrind* suppression file to the Intel Inspector suppression file format. Save the converted file to the default suppression directory for the project. Use the default `.sup` extension.
2. In the Intel Inspector GUI, in the **Target** tab of the **Project Properties** dialog box, set **Suppressions** to **Apply suppressions**.
3. Run a new analysis using the Intel Inspector GUI or command line interface.

   Outcome: The Intel Inspector does not collect result data impacted by suppression rules in the converted file and the rules in other files and directories identified in the **Suppressions** tab of the **Project Properties** dialog box).
4. Review the remaining problems in the result to assess the effectiveness and potential reach of the rules in the converted file. If necessary, edit the converted file in a text editor.
5. Repeat steps 3 and 4 as necessary.

See Also

Interpret Result Data and Resolve Issues
Suppression Rule Reach in the GUI
Suppressions Support
Third-party Suppression Files
Converting Third-Party Suppression Files from the Command Line Convert suppression files from IBM Rational* PurifyPlus*, Valgrind* or older versions of Intel® Inspector to a format compatible with Intel Inspector SP1 and above.
Manage Suppression Rules
Set Project Properties-Advanced
Intel Inspector Filenames and Locations

Third-party Suppression Files

The command line interface `convert-suppression-file` action converts suppression files from the following products to the current Intel Inspector suppression file format:

- Third-party software
  - IBM Rational* PurifyPlus* suppression files (Linux* OS only)
  - Valgrind* suppression files (Linux OS only)
- Previous versions of the Intel Inspector suppression files (Linux and Windows* OS)

During conversion, the Intel Inspector:

1. Reads the suppression file to internal structures.
2. Converts third-party product messages/errors to Intel Inspector problem types.
3. Skips suppression rules that cannot be converted.
4. Writes the converted data to an Intel Inspector suppression file.
5. Reports all errors encountered during processing.

Caution
Even with a completely successful conversion, you may still need to fine-tune the contents of the converted suppression file using a text editor.

PurifyPlus Message to Intel Inspector Problem Type Mapping

PurifyPlus messages map to Intel Inspector problem **Types** in the following manner during conversion:

PurifyPlus Message	Message Description	Intel Inspector Analog
ABR	Array Bounds Read	invalid_memory_access
ABW	Array Bounds Write	invalid_memory_access
ABWL	Late Array Bounds Write	invalid_memory_access
BRK	Misuse of BRK or SBRK (Using BRK or SBRK directly to allocate memory)	None
BSR	Beyond Stack Read	invalid_memory_access
BSW	Beyond Stack Write	invalid_memory_access
COR	Core Dump Imminent	None
FFM	Freeing Freed Memory	invalid_deallocation
FIM	Freeing Invalid Memory	invalid_deallocation
FIU	File Descriptors In Use	None
FMM	Freeing Mismatched Memory	invalid_deallocation
FMR	Free Memory Read (Read from heap memory that has already been freed)	invalid_memory_access
FMW	Free Memory Write (Write to heap memory that has already been freed)	invalid_memory_access
FMWL	Free Memory Write Late (Write to heap memory that has already been freed)	invalid_memory_access
FNH	Freeing Non Heap Memory	invalid_deallocation
FUM	Freeing Unallocated Memory	invalid_deallocation
IPR	Invalid Pointer Read	invalid_memory_access
IPW	Invalid Pointer Write	invalid_memory_access
MAF	Malloc Failure	None
MIU	Memory In-Use	None
MLK	Memory Leak	unreachable_memory_leak
MRE	Malloc Reentrancy Error	Not applicable
MSE	Memory Segment Error	invalid_memory_access
NPR	Null Pointer Read (SEGV signal)	invalid_memory_access
NPW	Null Pointer Write (SEGV signal)	invalid_memory_access
PAR	Bad Parameter	Not applicable

PurifyPlus Message	Message Description	Intel Inspector Analog
PLK	Potential Memory Leak - Heap memory that potentially might be leaked (program has pointers only to the middle of the region)	Not applicable
SBR	Stack Array Bounds Read (concerns local variables) - Only generated on SPARC; not on Linux* or Windows* OS	None
SBW	Stack Array Bounds Write (concerns local variables) - Only generated on SPARC; not on Linux or Windows OS	None
SIG	Signal	None
SOF	Stack Overflow	None
UMC	Uninitialized Memory Copy	uninitialized_memory_access
UMR	Uninitialized Memory Read	uninitialized_memory_access
WPF	Watchpoint Free	Not applicable
WPM	Watchpoint Malloc	Not applicable
WPN	Watchpoint Entry	Not applicable
WPR	Watchpoint Read	Not applicable
WPW	Watchpoint Write	Not applicable
WPX	Watchpoint Exit	Not applicable
ZPR	Zero Page Read (read from a bad pointer)	invalid_memory_access
ZPW	Zero Page Write (write to a bad pointer)	invalid_memory_access

Valgrind Error to Intel Inspector Problem Type Mapping

Valgrind errors map to Intel Inspector problem **Types** in the following manner during conversion:

Valgrind Error	Problem Description	Intel Inspector Analog
AddrN	Invalid memory access	invalid_memory_access
ValueN	Uninitialized memory access	uninitialized_memory_access
Cond	Use of an uninitialized CPU condition code	Not applicable
Jump	Jump to an unaddressable location error	Not applicable
Param	Invalid system call parameter error	invalid_call
Overlap	Src/dest overlap in memcpy or similar function	invalid_call
Free	Freeing error or mismatched deallocation	invalid_deallocation, mismatched_deallocation, invalid_deallocation_mapped

Valgrind Error	Problem Description	Intel Inspector Analog
Leak	Memory leak	unreachable_memory_leak

NOTE
Valgrind software detects uninitialized memory problems differently than the Intel Inspector. This difference impacts the stacks each product detects, which impacts the corresponding suppression files.

See Also
Suppression Rule Syntax in Text Format
Suppressions Support
Converting Third-Party Suppression Files Using the CLI
Intel Inspector Filenames and Locations

API Support

Intel® Inspector provides API support that allows you to:

- Gather semantic information related to your synchronization constructs.
- Identify the semantics of your `malloc`-like heap management functions.
- Specify which parts of your application should be analyzed.

Applications or modules linked to the static API library do not have a runtime dependency on a dynamic library, so they can be executed independently of Intel Inspector and other Intel studio tools.

NOTE
- There are no Java or .NET APIs. If you need runtime environment support, you can use a JNI, or C/C++ function call from the managed code. If the collector causes significant overhead or data storage, you can pause the analysis to reduce the overhead.
- For information on Windows* OS API support, see the Appendix.

Using C/C++ and Fortran APIs

The default Intel Inspector installation path, `<install-dir>`, is below `C:\Program Files (x86)\Intel\oneAPI\inspector\latest` (on certain systems, the directory is `Program Files`).

1. Specify this file in your code:
   - For C/C++, include `<install-dir>\include\ittnotify.h`
   - For Fortran, use `<install-dir>\include\<ia32|intel64>\ittnotify.mod`
2. Insert `__itt_*` notifications in appropriate places in your code.
3. Link to this file:
   - For C/C++, link to `<install-dir>\<lib32|lib64>\libittnotify.lib`
   - For Fortran, link to `<install-dir>\<lib32|lib64>\libittnotify.lib`

Conditional Compilation for Release Versions

For best performance in the release version of your code, use conditional compilation to turn off all annotations. To eliminate all `__itt_*` functions from your code during compilation of the release version, define the macro `INTEL_NO_ITTNOTIFY_API` before including `ittnotify.h`.

You can also define this macro to remove the static library during the linking stage.

APIs for Custom Synchronization

While the Intel Inspector supports a significant portion of the Windows* OS and POSIX* APIs, it is often useful to define your own synchronization constructs. Any specially built constructs that you create are not normally tracked by the Intel Inspector; however, the Intel Inspector supports synchronization APIs to help you gather semantic information related to your custom synchronization constructs.

Synchronization constructs may generally be modeled as a series of signals. One thread, or many threads, may wait for a signal from another group of threads before proceeding with some action. Synchronization APIs track when a thread begins waiting for a signal and when the signal occurs.

Using User-Defined Synchronization APIs in Your Code

Use This in C/C++ Code	Use This in Fortran Code	To Do This
`void __itt_sync_acquired ( void *addr)`	`subroutine itt_sync_acquired(addr) integer(kind=itt_ptr), intent(in), value :: addr end subroutine itt_sync_acquired`	Tell the Intel Inspector that the code received a signal on the specified synchronization object.
`void __itt_sync_releasing ( void *addr)`	`subroutine itt_sync_releasing(addr) integer(kind=itt_ptr), intent(in), value :: addr end subroutine itt_sync_releasing`	Tell the Intel Inspector that the code is about to send a signal on the specified synchronization object.
`void __itt_sync_destroy ( void *addr)`	`subroutine itt_sync_destroy(addr) integer(kind=itt_ptr), intent(in), value :: addr end subroutine itt_sync_destroy`	Tell the Intel Inspector that the synchronization object will not be used again, so the Intel Inspector can dispose of bookkeeping information associated with this object.

The `addr` parameter is simply a value that uniquely identifies the synchronization object to be modeled. Unique values allow the Intel Inspector to track distinct custom synchronization objects. To use the same custom object to protect access in different parts of your code, use the same `addr` parameter around each.

Since each custom synchronization construct may involve any number of synchronization objects, each synchronization object must be triggered off a unique memory handle, which the synchronization APIs will use to track the object. You can track any number of synchronization objects at one time using synchronization APIs, as long as each object uses a unique memory pointer. You can think of this as modeling objects similar to the `WaitForMultipleObjects` function in the Windows* OS API. You can create more complex synchronization constructs from a group of synchronization objects.

API Usage Tips

Follow these guidelines to properly insert synchronization APIs within your code:

- Insert an `acquired` API immediately **after** your code stops waiting for a synchronization object.
- Insert a `releasing` API immediately **before** the code signals that it no longer holds a synchronization object.

If you place the synchronization APIs improperly, the Intel Inspector may report threading problems where there are none or fail to detect real threading problems.

Usage Example: User-Defined Synchronized Critical Section

The following code snippets show how to create a critical section construct that can be tracked with synchronization APIs:

C/C++ Example

```
#include <ittnotify.h>

CSEnter(MyCriticalSection * cs)
{
  while(cs->LockIsUsed)
  {
    if(cs->LockIsFree)
    {
    // Code to acquire the lock goes here
    __itt_sync_acquired((void *) cs);
    }
  }
}
CSLeave(MyCriticalSection *cs)
{
if(cs->LockIsMine)
    {
        __itt_sync_releasing((void *) cs);
        // Code to release the lock goes here
    }
}
```

Fortran Example

```
use ittnotify

subroutine CSEnter(cs)

  integer cs
  while(LockIsUsed(cs) .ne. 1)
if(LockIsFree(cs) .eq. 1)

    ! Code to acquire the lock goes here
    call itt_sync_acquired(LOC(cs))
    end if
  enddo
end subroutine

subroutine CSLeave(integer cs)
{
    integer cs
    if(LockIsMine(cs) .eq. 1)
        call itt_sync_releasing(LOC(cs));
        ! Code to release the lock goes here
    end if
end subroutine
```

Note the following when looking at this simple critical section example:

- The `acquired` API is placed immediately after the code obtains the user lock.
- The `releasing` API is placed before the code releases the user lock. This ensures another thread does not call the `acquired` API before the Intel Inspector realizes this thread has released the lock.

Usage Example: User-Level Synchronized Barrier

Higher-level constructs, such as barriers, are also easy to model using synchronization APIs. The following code snippets show how to create a barrier construct that can be tracked using synchronization APIs:

C/C++ Example

```
#include <ittnotify.h>

Barrier()
{
    teamflag = false;
    __itt_sync_releasing((void *) &counter);
    InterlockedIncrement(&counter);
    //use the atomic increment primitive
      appropriate to your OS and compiler

    if( counter == thread_count )
    {
        __itt_sync_acquired((void *)
&counter);
        __itt_sync_releasing((void *)
&teamflag);
```

Fortran Example

```
use ittnotify

subroutine barrier()
    common /x/ teamflag, counter, thread_count
    integer teamflag
    integer thread_count
    integer counter
    teamflag = 0
    call itt_sync_releasing(LOC(counter))
    !atomically update counter here
    !use the atomic increment primitive
      !appropriate to your OS and compiler

    If ( counter .eq. thread_count ) then
        call itt_sync_acquired(LOC(counter))
        call itt_sync_releasing(LOC(teamflag))
```

C/C++ Example	Fortran Example

`counter = 0;`
`teamflag = true;`
 	
`}`
`else`
`{`
`Wait for team flag`
`__itt_sync_acquired((void *)`
`&teamflag);`
`}`
`}`
	`counter = 0`
	`teamflag = 1`
	
	`else`
	`!Wait for team flag`
	`call itt_sync_acquired(LOC(teamflag))`
	`end if`
	`end subroutine`

Note the following when looking at this example:

- There are two synchronization objects in this barrier code. The `counter` object is used to do a gather-like signaling from all the threads to the final thread indicating that each thread has entered the barrier. Once the last thread hits the barrier, it uses the `teamflag` object to signal all the other threads that they may proceed.
- As each thread enters the barrier, it calls the `releasing` API to tell the Intel Inspector it is about to signal the last thread by incrementing `counter`.
- The last thread to enter the barrier calls the `acquired` API to tell the Intel Inspector it was successfully signaled by all the other threads.
- The last thread to enter the barrier then calls the `releasing` API to tell the Intel Inspector it is going to signal the barrier completion to all the other threads by setting `teamflag`.
- Finally, before leaving the barrier, each thread calls the `acquired` API to tell the Intel Inspector it successfully received the end-of-barrier signal.

APIs for Custom Memory Allocation

Intel Inspector provides a set of APIs to help it identify the semantics of your `malloc`-like heap management functions. Annotating your code with these APIs reduces the number of false positives the Intel Inspector reports when analyzing your code.

Usage Tips

Follow these guidelines when using the memory allocation APIs:

- Create *wrapper* functions for your routines, and put the `__itt_heap_*_begin` and `__itt_heap_*_end` calls in these functions.
- Allocate a unique domain for each pair of `allocate`/`free` functions when calling `__itt_heap_function_create`. This allows the Intel Inspector to verify a matching `free` function is called for every `allocate` function call.
- Annotate the beginning and end of every `allocate` function and `free` function.
- Call all function pairs from the same stack frame, otherwise the Intel Inspector assumes an exception occurred and the allocation attempt failed.
- Do not call an `end` function without first calling the matching `begin` function.

Using Memory Allocation APIs in Your Code

Use This	To Do This

`typedef void* __itt_heap_function;`
 	
`__itt_heap_function __itt_heap_function_create( const __itt_char* name, const __itt_char* domain );`
	Declare a handle type to match `begin` and `end` calls and `domains`.
	• *name* = Name of the function you want to annotate.
	• *domain* = String identifying a matching set of functions. For example, if there are three functions that all work with `my_struct`, such as `alloc_my_structs`, `free_my_structs`, and `realloc_my_structs`, pass the same domain to all three `__itt_heap_function_create()` calls.
	
	NOTE
	Parameters of type `__itt_char` follow the Windows* OS unicode convention. If UNICODE is defined when compiling on a Windows* OS, `__itt_char` is `wchar_t`; otherwise it is `char`.

`void __itt_heap_allocate_begin( __itt_heap_function h, size_t size, int initialized );`
 	
`void __itt_heap_allocate_end( __itt_heap_function h, void** addr, size_t size, int initialized );`
	Identify allocation functions.
	• *h* = Handle returned when this function's name was passed to `__itt_heap_function_create()`.
	• *size* = Size in bytes of the requested memory region.
	• *initialized* = Flag indicating if the memory region will be initialized by this routine.
	• *addr* = Pointer to the address of the memory region this function has allocated, or 0 if the allocation failed.

`void __itt_heap_free_begin( __itt_heap_function h, void* addr );`
 	
`void __itt_heap_free_end( __itt_heap_function h, void* addr );`
	Identify deallocation functions.
	• *h* = Handle returned when this function's name was passed to `__itt_heap_function_create()`.
	• *addr* = Pointer to the address of the memory region this function is deallocating.

`void __itt_heap_reallocate_begin( __itt_heap_function h, void* addr, size_t new_size, int initialized );`
 	
`void __itt_heap_reallocate_end( __itt_heap_function h, void* addr, void** new_addr, size_t new_size, int initialized );`
	Identify reallocation functions.
	Note that `itt_heap_reallocate_end()` must be called after the attempt even if no memory is returned. Intel Inspector assumes C-runtime `realloc` semantics.
	• *h* = Handle returned when this function's name was passed to `__itt_heap_function_create()`.
	• *addr* = Pointer to the address of the memory region this function is reallocating. If *addr* is NULL, the Intel Inspector treats this as if it is an allocation.
	• *new_addr* = Pointer to a pointer to hold the address of the reallocated memory region.
	• *size* = Size in bytes of the requested memory region. If *new_size* is 0, the Intel Inspector treats this as if it is a deallocation.

Use This	To Do This

`void __itt_heap_internal_access_begin( void );`
 	
`void __itt_heap_internal_access_end( void );`
	Identify functions related to private heap management, such as queries for remaining heap size and validation of heap state.
	
	This function tells the Intel Inspector that this memory access is intentional and should not be flagged.
	
	Call `__itt_heap_internal_access_begin()` before accessing the underlying heap management internals and `__itt_heap_internal_access_end()` after the access is finished

`void __itt_heap_record_memory_growth_begin( void );`
 	
`void __itt_heap_record_memory_growth_end( void );`
	Produce reports of memory growth that occurs between calls to `__itt_heap_record_memory_growth_begin()` and `__itt_heap_record_memory_growth_end()`
	
	The report identifies any memory allocated but not freed between the two calls. Any memory allocated but not freed prior to the first call to `__itt_heap_record_memory_growth_begin()` is not reported. Any memory allocated but not freed after the final call to`__itt_heap_record_memory_growth_end()` is reported when collection is complete.

Usage Example: Heap Allocation

```
#include <ittnotify.h>

void* user_defined_malloc(size_t size);
void user_defined_free(void *p);
void* user_defined_realloc(void *p, size_t s);

__itt_heap_function my_allocator;
__itt_heap_function my_reallocator;
__itt_heap_function my_freer;

void* my_malloc(size_t s)
{
    void* p;

    __itt_heap_allocate_begin(my_allocator, s, 0);
    p = user_defined_malloc(s);
    __itt_heap_allocate_end(my_allocator, &p, s, 0);

    return p;
}


void my_free(void *p)
{
    __itt_heap_free_begin (my_freer, p);
    user_defined_free(p);
    __itt_heap_free_end (my_freer, p);
}

void* my_realloc(void *p, size_t s)
```

```
{
    void *np;

    __itt_heap_reallocate_begin (my_reallocator, p, s, 0);
    np = user_defined_realloc(p, s);
    __itt_heap_reallocate_end(my_reallocator, p, &np, s, 0);

    return(np);
}

// Make sure to call this init routine before any calls to
// user defined allocators.
void init_itt_calls()
{
    my_allocator = __itt_heap_function_create("my_malloc", "mydomain");
    my_reallocator = __itt_heap_function_create("my_realloc", "mydomain");
    my_freer = __itt_heap_function_create("my_free", "mydomain");
}

void test_size_of_held_memory(void *p)
{
    size_t s=0;

    // This will tell Intel Inspector that this memory access
    // is intentional, and should not be flagged.
    __itt_heap_internal_access_begin();

#ifdef TARGET_WINDOWS
    s = _msize(p);
#endif

    __itt_heap_internal_access_end();
}
// Now use my_alloc, my_free, my_realloc in place of the user defined
// functions.
```

Usage Example: Heap Growth

```
#include <ittnotify.h>

void ProcessTransaction(TransactionContext x)
{
   ...
   char* m = (char*) malloc(128); // Memory leak Inspector will report
   ...
   return;
}

// In this example, a leak report will be generated for each transaction in
// the for loop.
void WaitForTransactions()
{
   ...
   for (;;)
   {
      __itt_heap_record_memory_growth_begin();
      TransactionContext x = WaitForTransaction(); // Transaction end-point
      ProcessTransaction(x);
```

```
        __itt_heap_record_memory_growth_end();
    }
}
```

Memory Allocation APIs for On-demand Memory Leak Detection and Memory Growth Detection

These APIs support on-demand memory leak detection and memory growth detection features in the GUI, and analogous memory leak reporting and growth detection capabilities in the CLI.

NOTE
Mask values can be added or OR'ed together to reset both values with a single call.

Use This in C/C++ Code	Use This in Fortran Code	To Do This
`void __itt_heap_reset_detection( unsigned int mask);`	`subroutine itt_heap_reset_detection(mask) integer, intent(in), value:: mask end subroutine itt_heap_reset_detection`	Reset the starting point for on-demand leak detection and/or memory growth reporting. For C/C++, the mask is: • `__itt_heap_leaks` to reset the leak detection starting point • `__itt_heap_growth` to reset the memory growth starting point. For Fortran, the mask is: • `itt_heap_leaks` to reset the leak detection starting point • `itt_heap_growth` to reset the memory growth starting point. If `__itt_heap_record()` is called without a prior call to `__itt_heap_reset_detection()`, the program's start is used for the starting point.
`void __itt_heap_record(unsigned int mask);`	`subroutine itt_heap_record(mask) integer, intent(in), value:: mask end subroutine itt_heap_record`	Record current data about memory leaks and/or memory growth for inclusion in the result. For C/C++, the mask is: • `__itt_heap_leaks` to detect and record information about memory leaks • `__itt_heap_growth` to detect and record information about memory growth For Fortran, the mask is: • `itt_heap_leaks` to detect and record information about memory leaks

Use This in C/C++ Code	Use This in Fortran Code	To Do This
		• `itt_heap_growth` to detect and record information about memory growth If you are using the GUI for collection, this data is displayed immediately. If you are using the CLI for collection, the data is available after the result is finalized.

`__itt_heap_record_memory_growth_begin()` and `__itt_heap_record_memory_growth_end()` are aliases for `__itt_heap_reset_detection(__itt_heap_growth)` and `__itt_heap_record(__itt_heap_growth)`, respectively.

Usage Example: On-demand Leak Reporting (C++)

In this example, the goal is to measure the memory leaked by the functions `pipeline_stage1()` , and `pipeline_stage2()` . We also want to examine the memory growth exhibited by the function `pipeline_stage2()` . Strictly speaking, the call to `__itt_heap_record()` after `pipeline_stage1()` finishes is not necessary, because any leaks that occur in pipeline stage 1 area also reported by the call to `__itt_heap_record()` that follows `pipeline_stage2()`. This example also demonstrates finer-grained leak and growth detection than that available via the CLI and GUI.

```
#include <ittnotify.h>

void ProcessPipeline()
{
    __itt_heap_reset_detection(__itt_heap_leaks);  // Start measuring memory leaks here
    pipeline_stage1();                             // Run pipeline stage 1
    __itt_heap_record(__itt_heap_leaks);           // Report leaks in stage 1

    // Now process stage 2 of the pipeline - measure growth and leaks there.
    // We reset the growth starting point for stage 2.
    // There's no need to reset the leak start point, since it follows stage 1.
    __itt_heap_reset_detection(__itt_heap_growth); // Reset growth starting point for stage 2
    pipeline_stage2();
    __itt_heap_record(__itt_heap_leaks|__itt_heap_growth);// Report leaks and growth
}
```

Usage Example: On-demand Leak Detection (Fortran)

In this example, the goal is to measure the memory leaked by the functions `pipeline_stage1()` , and `pipeline_stage2()` . We also want to examine the memory growth exhibited by the function `pipeline_stage2()` . Strictly speaking, the call to `itt_heap_record()` after `pipeline_stage1()` finishes is not necessary, because any leaks that occur in pipeline stage 1 are also reported by the call to`itt_heap_record()` that follows `pipeline_stage2()`. This example also demonstrates finer-grained leak and growth detection than that available via the CLI and GUI.

```
#include <ittnotify.h>

subroutine process_pipeline()
    ! Start looking for leaks starting with pipeline stage 1.
    call itt_heap_reset_detection(itt_heap_leaks);
    pipeline_stage1();
    call itt_heap_record(itt_heap_leaks);
```

```
    ! Now process stage 2 of the pipeline - measure growth and leaks there.
    ! We reset the growth starting point for stage 2.
    ! There's no need to reset the leak start point, since it follows stage 1.
    call itt_heap_reset_baseline(itt_heap_growth);
    pipeline_stage2();
    call itt_heap_record(itt_heap_leaks+itt_heap_growth);
}
```

APIs for Collection Control

Intel Inspector provides a set of collection control APIs to help you specify which parts of your application should be included in, or excluded from, analysis.

To support independence of implementation between library implementers and library users, the collection control APIs allow nesting by pushing and popping state information.

There are two kinds of collection control APIs:

- Time-oriented APIs to control which time periods of thread execution to analyze or not analyze
- Class-oriented APIs to control which data objects to analyze or not analyze

Using Time-oriented Collection Control APIs in Your Code

Use This in C/C++ Code	Use This in Fortran Code	To Do This
`void __itt_suppress_push( unsigned int etype)`	`subroutine itt_suppress_push(mask)  integer, intent(in), value :: mask end subroutine itt_suppress_push`	Tell the Intel Inspector to stop analyzing for errors on the current thread. **C/C++** *etype* is: • `__itt_suppress_memory_errors` to stop analyzing for memory errors • `__itt_suppress_threading_errors` to stop analyzing for threading errors • `__itt_suppress_memory_errors|_itt_suppress_threading_errors` to stop analyzing for memory or threading errors **Fortran** *mask* is: • `itt_suppress_memory_errors` to stop analyzing for memory errors • `itt_suppress_threading_errors` to stop analyzing for threading errors • `itt_suppress_all_errors` to stop analyzing for memory or threading errors

Use This in C/C++ Code	Use This in Fortran Code	To Do This
`void __itt_suppress_pop ( void)`	`subroutine itt_suppress_pop() end subroutine itt_suppress_pop`	Tell the Intel Inspector to undo the action corresponding to the most recent matching nested `push` call. `Push` calls nest and are additive, so the Intel Inspector does not resume analyzing for: • Memory errors until there are an equal number of `pop` calls corresponding to `push` calls for memory errors • Threading errors until there are an equal number of `pop` calls corresponding to `push` calls for threading errors

For errors that might involve two threads, the state the other thread was in at the time the memory operation happened is more important than the current thread state. For example:

1. Thread A writes to variable X, then makes a `push` call.
2. Thread B reads variable X.

Even though thread A is currently suppressed, neither thread was suppressed at the time the conflicting accesses occurred, so the race is reported.

Conversely:

1. Thread A suppresses, writes to variable Y, then unsuppresses.
2. Thread B reads variable Y.

Even though both threads are currently unsuppressed, the error is not reported because thread A was suppressed at the time the conflicting accesses occurred.

Using Object-oriented Collection Control APIs in Your Code

Use This in C/C++ Code	Use This in Fortran Code	To Do This
`void __itt_suppress_mark_range( __itt_suppress_mode_t mode, unsigned int etype, void * address, size_t size);`	`subroutine itt_suppress_mark_range( action, mask, addr, size) integer, intent(in), value :: action integer, intent(in), value :: mask integer(kind=itt_ptr), intent(in), value :: addr integer(kind=itt_ptr), intent(in), value :: size end subroutine itt_suppress_mark_range`	**C/C++** Tell the Intel Inspector to mark the memory range defined by *address* and *size* with the flags *mode* and *etype*. *mode* is `__itt_suppress_range` or `__itt_unsuppress_range`. *etype* is: • `__itt_suppress_memory_errors` to stop analyzing for memory errors • `__itt_suppress_threading_errors` to stop analyzing for threading errors

Use This in C/C++ Code	Use This in Fortran Code	To Do This
		• `__itt_suppress_memory_errors | __itt_suppress_threading_errors` to stop analyzing for memory or threading errors *addr* is the address of the first byte to suppress. *size* is the number of bytes to suppress. **Fortran** Tell the Intel Inspector to mark the memory range defined by *addr* and *size* with the flags *action* and *mask*. *action* is `itt_suppress_range` or `itt_unsuppress_range`. *mask* is: • `itt_suppress_memory_errors` to stop analyzing for memory errors • `itt_suppress_threading_errors` to stop analyzing for threading errors • `itt_suppress_all_errors` to stop analyzing for memory or threading errors *addr* is the address of the first byte to suppress. *size* is the number of bytes to suppress.
`void __itt_suppress_clear_range( __itt_suppress_mode_t mode, unsigned int etype, void * address, size_t size);`	`subroutine itt_suppress_clear_range( action, mask, addr, size) integer, intent(in), value :: action integer, intent(in), value :: mask integer(kind=itt_ptr), intent(in), value :: addr integer(kind=itt_ptr), intent(in), value :: size end subroutine itt_suppress_clear_range`	**C/C++** Tell the Intel Inspector to clear the previously marked range defined by *address* and *size* with the flags *mode* and *etype*. *mode* is `__itt_suppress_range` or `__itt_unsuppress_range`. *etype* is: • `__itt_suppress_memory_errors` to stop analyzing for memory errors • `__itt_suppress_threading_errors` to stop analyzing for threading errors • `__itt_suppress_memory_errors | __itt_suppress_threading_errors` to stop analyzing for memory or threading errors *addr* is the address of the first byte to not suppress.

Use This in C/C++ Code	Use This in Fortran Code	To Do This
		size is the number of bytes to not suppress. **Fortran** Tell the Intel Inspector to clear the previously marked range defined by *addr* and *size* with the flags *action* and *mask*. *action* is `itt_suppress_range` or `itt_unsuppress_range`. *mask* is: • `itt_suppress_memory_errors` to stop analyzing for memory errors • `itt_suppress_threading_errors` to stop analyzing for threading errors • `itt_suppress_all_errors` to stop analyzing for memory or threading errors *addr* is the address of the first byte to not suppress. *size* is the number of bytes to not suppress.

When the Intel Inspector detects an error, it checks the set of marked ranges and, using the smallest enclosing range, filters out the error if the *mode/action* is `suppress_range` and the *etype/mask* matches the detected error.

The largest supported range is 2 GB.

The default *mode/action* for all of memory is `unsuppress_range`. You can change the default by creating a range with address NULL and size 0.

You can nest ranges only if the new range is a subset of existing ranges.

Two calls to `mark_range` that specify the same range and intersecting *etype/mask* may not specify both `suppress_range` and `unsuppress_range` as the *mode/action*.

Usage Example: Time-oriented Collection Control

C/C++ Example	Fortran Example

`#include <ittnotify.h>`
 	
`...`
`#pragma omp parallel`
`  __itt_suppress_push(`
`  __itt_suppress_threading_errors);`
`  /* Any threading errors here will be`
`  ignored by the calling thread.`
`  In this case, each thread in the`
`region */`
	`use ittnotify`
	
	`...!$omp parallel`
	`  call itt_suppress_push(`
	`  itt_suppress_threading_errors)`
	`  !Any threading errors here will be`
	`  !ignored by the calling thread.`
	`  !In this case, each thread in the`
	`region`
	
	`  call itt_suppress_pop()`

C/C++ Example	Fortran Example

`__itt_suppress_pop();`
`/* Any threading errors here will be seen by Inspector*/`
`}`
	`! Any threading errors here will be !seen by Inspector`
	
	`!$omp end parallel`

Usage Example: Object-oriented Collection Control

C/C++ Example	Fortran Example

`#include <ittnotify.h>`
 	
`int variable_to_watch;`
`int other_variable;`
`…`
`//change the default mode by using NULL and 0 as address and size`
`__itt_suppress_mark_range(__itt_suppress_range, __itt_suppress_threading_errors, NULL, 0);`
`//ensure we see errors on variable_to_watch`
`__itt_suppress_mark_range(__itt_unsuppress_range, __itt_suppress_threading_errors, &variable_to_watch, sizeof(variable_to_watch));`
`#pragma omp parallel`
`…`
`variable_to_watch++; //race will be reported`
`other_variable++; //race will not be reported`
`}`
`//clear the record for all of memory`
`__itt_suppress_clear_range(__itt_suppress_range, __itt_suppress_threading_errors, NULL, 0);`
`//clear the record for variable_to_watch`
`__itt_suppress_clear_range(__itt_unsuppress_range, __itt_suppress_threading_errors, &variable_to_watch, sizeof(variable_to_watch));`
`//mark the range for other_variable so we don't see errors`
`__itt_suppress_mark_range(__itt_suppress_range, __itt_suppress_threading_errors, &other_variable,`
	`use ittnotify`
	
	`common /x/ variable_to_watch, other_variable`
	`integer variable_to_watch`
	`integer other_variable`
	`…`
	`!change the default mode by using 0 and 0 !as address and size`
	`call itt_suppress_mark_range(itt_suppress_range, itt_suppress_threading_errors, 0, 0)`
	`!ensure we see errors on variable_to_watch`
	`call itt_suppress_mark_range(itt_unsuppress_range, itt_suppress_threading_errors, LOC(variable_to_watch), 4)`
	`!$omp parallel`
	`…`
	`variable_to_watch = variable_to_watch + 1`
	`!race will be reported`
	`other_variable = other_variable + 1`
	`!race will not be reported`
	`!$omp end parallel`
	`!clear the record for all of memory`
	`call itt_suppress_clear_range(itt_suppress_range, itt_suppress_threading_errors, 0, 0);`
	`!clear the record for variable_to_watch`
	`call itt_suppress_clear_range(itt_unsuppress_range, itt_suppress_threading_errors, LOC(variable_to_watch), 4)`
	`!mark the range for other_variable !so we don't see errors`
	`call itt_suppress_mark_range(itt_suppress_range, itt_suppress_threading_errors, LOC(other_variable),`

C/C++ Example	Fortran Example

sizeof(other_variable));
#pragma omp parallel
…
variable_to_watch++; //race will be reported
other_variable++; //race not be reported
}
	4)
	!$omp parallel
	…
	variable_to_watch = variable_to_watch + 1
	!race will be reported
	other_variable = other_variable + 1
	!race not be reported
	!$omp end parallel

Troubleshooting

Troubleshooting Anti-virus Software Issues

Symptoms

When you run an Intel Inspector analysis, anti-virus software:

- Reports attempts to introduce and execute unsafe code.
- Possibly terminates application execution.

Details

Intel Inspector uses dynamic binary instrumentation technology to identify issues and collect results. Some anti-virus software products load a small runtime agent to monitor for the presence of malicious software. These agents may incorrectly identify dynamic binary instrumentation technology as malicious software.

Possible Correction Strategies

Check anti-virus software documentation for a configuration mechanism to treat the application and the Intel Inspector as trusted applications.

Troubleshooting Application Crashes

Symptoms

The application crashes when you run an Intel Inspector analysis.

Details

Both the application and the Intel Inspector run in the same address space. This means:

- A bug that can cause the application to crash while running without the Intel Inspector can cause the application to crash while running with the Intel Inspector.
- A bug that does not manifest itself and does not cause the application to crash while running without the Intel Inspector may manifest itself and cause the application to crash while running with the Intel Inspector because of parallel non-deterministic scheduling or techniques used to detect incorrect memory accesses.

Possible Correction Strategies

- Use a conventional debugger to debug the crash.

- Run a memory error analysis to help debug memory-related crashes before running a threading error analysis.
- If the application crashes during analysis and the Intel Inspector reports errors before the application crashes, interpret the result data to determine if any of the reported errors caused the crash. A likely culprit should be near the bottom of the **Collection Log** pane.
- Contact the Intel Customer Support team by email. Include a description of the steps leading up to the problem, the generated problem report, and the corresponding result directory.

See Also

Troubleshooting Out-of-memory Conditions

Troubleshooting Internal Thread Suspension Attempt

Symptoms

Intel Inspector displays *Detected an attempt to suspend an internal thread*.

Details

Intel Inspector displays this message when the target application tries to suspend all threads, causing Intel Inspector thread suspension as well.

Correction Strategies

To avoid this error:

1. Disable the **Enable collection progress information** checkbox in the **Target** tab of the **Project Properties** dialog box.
2. Disable the **Enable interactive memory growth detection** checkbox in the **Analysis Type** pane.
3. Rerun the analysis.

Also avoid running an interactive debugging session during analysis.

See Also

Configure Projects
Memory Error Analysis Types

Troubleshooting No Problems Detected

Symptoms

In the **Problems** pane, the Intel Inspector displays no problem sets.

> **NOTE**
> This may be valid result that requires no corrective action.

Possible Correction Strategies

If this result is unexpected, do one or more the following:

- When you configure the project:
  - Choose a different data set.
  - Increase the number of threads used.
- When you configure the project:

  - Choose a wider preset analysis type. For example, if you ran a threading error analysis using the **ti2** preset analysis type, run a new analysis using the **ti3** preset analysis type.
  - Choose the **Do not apply suppressions** radio button in the **Advanced** region on the **Target** tab of the **Project Properties** dialog box to ignore all suppression rules.
- If you ran a memory error analysis, run a threading error analysis. If you ran a threading error analysis, run a memory error analysis.

See Also

Configure Analysis
Configure Projects

Troubleshooting No Symbolic Information

Symptoms

In the **Sources** window, the Intel Inspector displays no source code for any code locations in the problem set.

Details

Intel Inspector cannot display source code for viewing and editing unless the application has symbols available.

If the Intel Inspector detects no symbols for a location, it sets the call stack and code pane to the first location where it can find symbols.

If the Intel Inspector cannot find any location in the call stack with symbols, it provides the module name and relative virtual address (RVA) for the location.

Possible Correction Strategies

- When you compile and link an application on Windows* systems:
  - Enable the debug information compiler option.
  - Set the linker option to generate debug information.
- Configure the project to search non-standard directories.
- Copy the symbol files, such as `*.pdb` files, to a location where the Intel Inspector can find them.

See Also

Configure Projects

Troubleshooting OpenMP* Technology Issues

Symptoms

Intel Inspector collects false positives with unrecognized code locations in your OpenMP* application.

Possible Correction Strategies

- Link with dynamic OpenMP* libraries instead of static ones.
- If you are linking to the oneAPI Math Kernel Library (oneMKL), link with the dynamic libraries instead of the static ones.
- Use Intel compiler products to compile and link your application.
- If using Microsoft* compiler products, link with the OpenMP* Compatibility Library included in the Intel® Compiler products.

Troubleshooting Out-of-memory Conditions

Symptoms

- Intel Inspector displays an out-of-memory message.
- Intel Inspector stops application execution and collection (with or without reporting an out-of-memory message).
- In the **Windows Task Manager** window, the memory used by the application approaches the process limit before application execution stops.

Details

Both the application and the Intel Inspector run in the same address space.

This means the memory footprint of the application running with the Intel Inspector is larger than the memory footprint of the application running without the Intel Inspector, because the Intel Inspector itself consumes memory to detect errors in the application.

Intel Inspector exhibits the symptoms above if the total memory consumed by application and the Intel Inspector is close to or exceeds the total process virtual space.

Possible Correction Strategies

Reduce the amount of work the Intel Inspector must perform during inspection. For example:

- Use a smaller data set.
- Inspect the application multiple times and exclude a different set of modules each time.
- Use `OMP_NUM_THREADS` to limit the number of threads when inspecting OpenMP* applications. In most cases, `OMP_NUM_THREADS=2` is sufficient.

Troubleshooting Tamper-resistance Issues

Symptoms

When you run an Intel Inspector analysis, the application:

- Reports symptoms of unauthorized use.
- Disables features.
- Terminates execution.

Details

Intel Inspector uses dynamic binary instrumentation technology to identify issues and collect results. An application may implement runtime checks to ensure it is not used in unintended ways. Typically such checks ensure you do not tamper with license checking, digital rights management (DRM), or other security-related features. An application implementing such checks may incorrectly identify dynamic binary instrumentation technology as evidence of tampering.

Possible Correction Strategies

If possible, turn off tamper-resistance features during application development when the Intel Inspector is used. Otherwise, build the application in such a way that it is possible to debug by attaching a debugger, setting breakpoints, and stepping through the code.

Troubleshooting Unexpected Application Behavior During Memory Error Analyses

Symptoms

The application exhibits unexpected behavior, or even crashes, during Intel Inspector memory error analyses that include enhanced dangling pointer detection.

Details

Under certain circumstances, enhanced dangling pointer detection may result in failed memory allocation calls that would normally succeed.

Intel Inspector performs dangling pointer detection by intercepting calls to deallocators and deferring the actual deallocation. Ordinarily, when an application frees memory, that memory goes back into the pool of available memory and is often reused by an allocation call soon after it is freed. This makes it difficult to distinguish between a valid use of the new memory block and an invalid attempt to access the memory referred to by the freed block.

Because the Intel Inspector defers deallocation of the memory block, the memory is not immediately returned to the pool of available memory. The block of memory becomes logically invalid to the application because the application freed it. Therefore, if the application subsequently tries to access this memory, the Intel Inspector is certain the access is invalid and reports an error.

In most circumstances, all this behavior is transparent to the application. The operating system uses new blocks of memory to fulfill application requests and everything functions as expected as long as the pool of available memory is large enough to accommodate new allocation requests. The Intel Inspector limits the total size of the memory it holds for dangling pointer detection to maintain this expected behavior.

However, you may encounter situations where the amount of memory available for a specific type of allocation is more limited than the Intel Inspector allows for. For example, if an application repeatedly allocates and frees memory from a fixed size heap, the heap may eventually be entirely consumed and you may see unexpected problems.

Heap Example

```
void doSomething()
{
    HANDLE fixedHeap = HeapCreate( 0, 0x10000, 0x10000 );
    for ( int i = 0; i < 20; i++ )
    {
        int *p = (int*)HeapAlloc( fixedHeap, HEAP_ZERO_MEMORY, 0x1000 );
        if( p )
            printf( "HeapAlloc succeeded: 0x%p\n", p );
        else
        {
            printf( "HeapAlloc failed\n" );
            break;
        }
        HeapFree( fixedHeap, 0, p );
    }
    HeapDestroy( fixedHeap );
}
```

This code creates a heap with a fixed size and then loops 20 times allocating and freeing memory from that heap. When run without the Intel Inspector, the call to `HeapAlloc` is likely to return the same pointer every time through the loop. However, when you run a memory error anaysis, the call to `HeapFree` is intercepted and the memory is not returned to the heap. Therefore, each iteration through the loop results in a unique pointer, and the call to `HeapAlloc` fails when `i` equals 16.

Possible Correction Strategies

Work around the problem by temporarily changing the heap to one that can grow. For example, set the maximum size parameter to `0` when calling `HeapCreate`.

Caution
Sample Code Caveats

User Interface Reference

Context Menus: Problem Details Pane

Access key capabilities from the Intel® Inspector**Problem Details** pane context menu.

Do This	To Do This
Right-click a code location.	• **Source** - Open the corresponding source file in the Visual Studio* editor at the line indicated in the problem description. • **Intel Inspector** - Change focus to the corresponding Intel Inspector result **Sources** window. • **Copy to Clipboard** - Copy the selected code location to the system clipboard. • **Explain Problem** - Explain, in generic terms, the error associated with the code.

See Also

Investigate Problems Using Interactive Debugging

Context Menus: Project Navigator

Access key capabilities from the Intel Inspector **Project Navigator** context menus (Standalone Intel Inspector GUI only).

Do This	To Do This
Right-click the directory in the **Project Navigator** banner.	• **New Project...** - Open the **Create a Project** dialog box, where you can browse to or create a directory in which the Intel Inspector will create an Intel Inspector project (`config.inspxeproj`). • **Open Project From New Location** - Open the **Select Project** dialog box, where you can browse to a directory containing Intel Inspector projects. • **Copy Path to Clipboard** - Copy this directory path to the system clipboard.

Right-click a project in the **Project Navigator**.	• **Open Project** - Open the Intel Inspector project.
	• **Close Project** - Close the current project and any opened results.
	• **New Analysis...** - Open the **Analysis Type** window, where you can:
	  • Choose a preset memory or threading error analysis type.
	  • Configure a custom analysis type.
	• **<Recent Analysis Type>** - Rerun a recent analysis.
	• **Close All Results** - Close all opened results for this project.
	• **Delete Project** - Immediately delete the selected project (and associated results) from the **Project Navigator** and file system.

Do This	To Do This

	• **Rename Project** - Rename the selected project:
	  • Immediately in the **Project Navigator**
	  • In the file system after you close the project or exit the Intel Inspector
	
	NOTE
	The corresponding project directory on your file system is not renamed.
	
	• **Copy Project Path to Clipboard** - Copy the file path for this project to the system clipboard.
	• **Project Properties...** - Open the **Project Properties** dialog box, where you can review or change current project properties.

Right-click a result in the **Project Navigator**.	• **Open Result** - Open the Intel Inspector result.
	• **Export Result...** - Create an easy-to-share result archive file.
	• **Re-inspect** - Run a new analysis using the same analysis type as that in the selected result.
	• **Re-resolve [and Open]** - Refinalize (and open if not already opened) the result using a more complete set of symbol information.
	
	NOTE
	• When you run an analysis using the GUI, finalization occurs after the result is generated and when the result is first opened in the standalone GUI or in the Microsoft Visual Studio* IDE.
	• Intel Inspector retrieves symbol information using the directories identified in the **Binary/Symbol Search** and **Source Search** tabs of the **Project Properties** dialog box as well as default search locations and algorithms.
	• Refinalizing may change problem set, code location, and stack information because the Intel Inspector discards previously resolved symbol information and performs symbol resolution again.
	
	• **Delete Result** - Immediately delete the selected result from the **Project Navigator** and file system.
	• **Rename Result** - Rename the selected result:
	  • Immediately in the **Project Navigator**
	  • In the file system after you close the result or project, or exit the Intel Inspector
	
	NOTE
	The corresponding result directory in the file system is not renamed.
	
	• **Copy Result Path to Clipboard** - Copy the file path for this result to the system clipboard.

See Also

Choose Projects
Configure Analysis
Configure Projects
Export and Import Files
Run Analysis
View Results

Binary/Symbol Search and Source Search Locations

Context Menus: Solution Explorer

Access key capabilities from the Intel Inspector**Solution Explorer** context menus (Visual Studio* IDE only).

NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

Do This	To Do This

Right-click the project folder to access the following options from the **Intel Inspector <version>** submenu.	• **<Recent Analysis Type>** - Run another analysis using this recent analysis type.
	• **New Analysis...** - Display the **Analysis Type** window, where you can:
	  • Choose and, if necessary, fine-tune a preset memory or threading error analysis type.
	  • Configure a custom analysis type.
	• **Project Properties...** - Display the **Project Properties** dialog box, where you can review or change current project properties.

Right-click a result.	• **Open** - Display the selected result. • **Re-inspect** - Run another analysis using the same analysis type as that in the selected result. • **Re-resolve [and Open]** - Refinalize (and open if not already opened) the result using a more complete set of symbol information. **NOTE** • When you run an analysis using the GUI, finalization occurs after the result is generated and when the result is first opened in the standalone GUI or in the Microsoft Visual Studio* IDE. • Intel Inspector retrieves symbol information using the directories identified in the **Binary/Symbol Search** and **Source Search** tabs of the **Project Properties** dialog box as well as default search locations and algorithms. • Refinalizing may change problem set, code location, and stack information because the Intel Inspector discards previously resolved symbol information and performs symbol resolution again. • **Export Result...** - Create an easy-to-share archive result file. • **Remove** - Remove the selected result from the project and delete it from your file system. • **Rename** - Rename the selected result in the project and on your file system. **NOTE** The corresponding result directory on your file system is not renamed.

See Also

Configure Analysis
Configure Projects
Export and Import Files
Run Analysis

View Results
Binary/Symbol Search and Source Search Locations

Context Menus: Sources Window Panes

Access key capabilities from the Intel Inspector **Sources** window pane context menus.

Do This	To Do This
Right-click anywhere in the **Source** tab.	• **Edit Source** - Edit source code in your default editor. • **Copy to Clipboard** - Copy the selected code to the system clipboard. • **Explain Problem** - Explain, in generic terms, the error associated with the code.
Right-click anywhere in the **Disassembly** tab.	• **Copy to Clipboard** - Copy the selected code to the system clipboard. • **Explain Problem** - Explain, in generic terms, the error associated with the code.
Right-click anywhere in the **Call Stack** tab.	• **Show Module Names**, **Show Sources**, **Show Two Lines**, and **Show Horizontal Scrollbar** - Customize frame presentation. • **Copy to Clipboard** - Copy the frames to the system clipboard.

See Also

Interpret Result Data and Resolve Issues

Context Menus: Summary Window Panes

Access key capabilities from the Intel Inspector **Summary** window pane context menus.

Do This	To Do This
Right-click a data row in the **Problems** pane.	• **View Source** - Display a **Sources** window focused on the selected problem(s) (default action when you double-click a data row in the **Problems** pane). • **Edit Source** - Open the source file for the selected problem(s) in your default editor. • **Copy to Clipboard** - Copy a summary of the selected problem(s) to the system clipboard. • **Explain Problem** - Explain, in generic terms, the error associated with the code. • **Create Problem Report...** - Export details of the selected problem(s) to plain text format. (Enabled only after analysis is complete.) • **Debug This Problem** - Launch a new analysis in conjunction with a debugger to stop at problems of interest. (Visual Studio* IDE only). (Enabled only after analysis is complete.) • **Change State** - Change the state of the selected problem(s) to **Not Fixed**, **Confirmed**, **Fixed**, **Not a problem**, or **Deferred**. (Enabled only after analysis is complete.) • **Merge States...** - Merge state information from another result; update in this result the state assigned to any problems that appear in both results. (Enabled only after analysis is complete.)
Right-click in the **Filters** pane.	• **Refine Source File Set...**- Temporarily limit the items in the **Problems** pane to those from a set of source files. • **Clear Source File Filter** - Deselect the source file filter criterion. • **Explain Problem** - Explain, in generic terms, the error associated with the problem type. (Displayed only for criteria in the **Type** category).

Do This	To Do This

Right-click a code location in the **Code Locations** pane.	• **View Source** - Display a **Sources** window focused on the selected problem occurrence. (Default action when you double-click a data row in the **Code Locations** pane.)
	• **Edit Source** - Open the source file for the selected code location in your default editor.
	• **Copy to Clipboard** - Copy a summary of the selected code location to the system clipboard.
	• **Explain Problem** - Explain, in generic terms, the error associated with the code.
	• **Create Problem Report...** - Export details of the selected code location to plain text format. (Enabled only after analysis is complete.)
	• **Suppress...** or **Do Not Suppress...** -
	  • Create a rule to suppress problems associated with the selected code location during the next analysis run. (Intel Inspector marks - with strikethroughs - problems *potentially* impacted during the next analysis run.)
	  • Or delete a rule for a marked code location. (Strikethroughs disappear.)
	  (Enabled only after analysis is complete.)
	  NOTE
	  You can also delete suppression rules using the **Suppressions** tab of the **Project Properties** dialog box.
	• **Show Source Code Snippets** - Display or not display several source code lines surrounding each code location. (The yellow highlight marks the code location source line.)

Right-click a marker in the **Timeline** pane.	**View Source** - Display a **Sources** window focused on the selected problem occurrence.
Right-click a data column header.	• **Hide Column** - Hide the selected data column. • **Show All Columns** - Show all previously hidden data columns. • **Show Column** - Show a specific, previously hidden data column.

See Also

Choose Problems
Collaborate on Results
Configure Projects
Interpret Result Data and Resolve Issues
Investigate Problems Using Interactive Debugging

Dialog Box: Corresponding inspxe-cl Command Options

To access this Intel Inspector dialog box: On the **Command** toolbar, click the **Command Line** button.

Use this dialog box to review - and, if desired, copy to the clipboard - the command to perform the same analysis using the Intel Inspector command-line interface (`inspxe-cl` command).

Use This	To Do This
Text box	Review the complete `inspxe-cl` command to perform the same analysis.
Copy to Clipboard button	Make the complete `inspxe-cl` command available in other scenarios.

See Also

Toolbar: Command
Command Syntax
inspxe-cl Actions, Options and Arguments
Command Line Interface Support

Dialog Box: Create a Project

To access this Intel Inspector dialog box: From the Standalone Intel Inspector GUI menu, choose **File** > **New** > **Project...**

Use this dialog box to create a new Standalone Intel Inspector GUI project.

Use This	To Do This

Project name field	Create or choose a directory to contain:
	• A project file that identifies:
	  • A compiled application for inspection
	  • A collection of configurable attributes surrounding the compiled application
	• Results
	• Suppression rules

Location field and **Browse** button	Choose or create a directory to contain the project directory.

See Also

Choose Projects
Intel Inspector Filenames and Locations

Dialog Box: Create Suppression

To access this Intel Inspector dialog box: In the **Code Locations** pane, right-click a code location in a problem you want to suppress during future analysis runs. In the context menu, choose **Suppress...**.

Use this dialog box to define a suppression rule to help you focus during future analysis runs on only those issues that require your attention.

Tip

- Although you are ultimately trying to suppress problems, the Intel Inspector*vehicle* for defining a suppression rule is one or more code locations.
- Narrow rules suppress a limited number of relevant problems; wider rules suppress a greater number of relevant problems.
- Every rule applied during analysis adds processing time.
- The goal: Suppress the greatest number of relevant problems with the fewest number of rules.
- To review rules to be applied during analysis, check the **Suppressions** tab of the **Project Properties** dialog box.
- To apply rules during analysis, select the **Apply Suppressions** radio button on the **Target** tab of the **Project Properties** dialog box.
- A code location may be part of multiple problems; therefore, multiple rules may suppress the same code location, or a rule created to suppress one problem may partially impact another problem.
- See suppression rule examples for more information.

Use This	To Do This
Name text box	Add a short description to distinguish the new rule from other rules.
Save in drop-down list and **Browse** button	• Save the new rule in the default location. • Save the new rule in another location. **Tip** Choose a non-default location only to make the rule easily accessible to others.
Problem type drop-down list	• Suppress problems during analysis by **Problem type** (keep specific **Problem type**). Outcome: Rules with narrower reach. • Do not suppress problems by **Problem type** (click drop-down arrow and choose ***(any)**). Outcome: Rules with wider reach.
Code Location Description checkboxes	• Suppress problems during analysis by a code location (select checkbox). Outcome: Rules with narrower reach. • Do not suppress problems by a code location (deselect checkbox). Outcome: Rules with wider reach. **Tip** A single code location with all but one characteristic set to *** (any)** is ideal for widening a rule to suppress the greatest number of relevant problems.
Code Location Description drop-down lists	• Suppress problems during analysis by **Code Location Description** (keep specific **Code Location Description**). Outcome: Rules with narrower reach. • Do not suppress problems by **Code Location Description** (click drop-down arrow and choose ***(any)**). Outcome: Rules with wider reach. **NOTE** There are three **Code Location Description** possibilities: **Allocation site**, **Deallocation site**, and *** (any)**. Memory error example: If you right-click a **Mismatched allocation site** code location for an occurrence of a **Mismatched allocation/deallocation** problem, the Intel Inspector displays one code location marked **Allocation site** and one code location marked *** (any)**; you can click the drop-down arrow to change **Allocation site** to *** (any)**. Threading error example: If you right-click a **Read** code location for an occurrence of a Read/Write **Data race** problem, the Intel Inspector displays two code locations marked *** (any)** and does not provide drop-down arrows.
Number of Frames in Rule fields	Identify the current number of stack frames that are the focus of the rule.
Start Frame in Rule fields	Identify the last-called stack frame that is currently the focus of the rule.
Edit... buttons	• Review stack frames for the code location. • Choose a different stack frame as the focus of the rule. • Choose multiple stack frames as the focus of the rule.

Use This	To Do This
	Tip • Each additional frame in the stack narrows the reach of a rule. • Suppression rules are more robust if you specify a stack with multiple frames instead of frames with line numbers. (Because line numbers can be altered by code insertions or deletions, suppressions may be rendered ineffective by even minor code maintenance. Stack-based suppressions require larger code changes to invalidate them, such as changes to function call sequences.)
Create button	• Create the new rule. • In the **Problems** and **Code Locations** panes, mark (strike through) all result data *potentially* impacted by the new rule. **NOTE** It may take some time for all strikethroughs to appear.

See Also

Interpret Result Data and Resolve Issues
Suppressions Support
Dialog Box: Project Properties-Suppressions
Dialog Box: Project Properties-Target
Dialog Box: Select Stack Frame(s)
Intel Inspector Filenames and Locations

Dialog Box: Custom Analysis

(One way) To access this Intel Inspector dialog box:

- Click the **Copy** button on the **Analysis Type-Memory Errors** or **Analysis Type-Threading Errors** pane.
- Click the **Copy** button or **Edit** button on the **Analysis Type-Custom** pane.

Use this dialog box to create or edit custom analyses.

Use This	To Do This
Analysis name field	View or change the default name that distinguishes this custom analysis type from other analysis types when you use the graphical user interface.
Description field	View or change the default detailed information that distinguishes this custom analysis type from other analysis types.
Command-line name field	View or change the default identifier that distinguishes this custom analysis type from other analysis types when you use the command-line interface.
Analysis identifier field	View or change the `{at}` identifier that distinguishes results produced by this custom analysis type from other results. For example: `ti2c`, where: • `ti2` represents the preset analysis type on which the custom analysis type is based • `c` represents custom

Configuration Settings for Custom Analysis Types Based on Memory Error Analysis Types

Use the **Properties** region of the dialog box to configure the custom analysis type.

The following table describes the purpose, usefulness, and *cost* (low, medium, high, or proportional in terms of time and resources) for each configuration setting in a custom analysis type based on a memory error analysis type. (The settings are listed in alphabetical order.)

Setting	Purpose, Usefulness, and Cost
Analyze stack accesses	Available only if **Detect invalid memory accesses** is selected. Select to analyze invalid and uninitialized accesses to thread stacks. Selecting is useful when: • You want as thorough an analysis as possible. • An application calls `alloca()`. High **cost**. Recommendation: • Select the first time you analyze an application and periodically thereafter. • Select to analyze automatic variables.
Defer memory deallocation (previously called Byte limit before reallocation)	Available only if **Detect invalid memory accesses** and **Enable enhanced dangling pointer check** are selected. Select to have the Intel Inspector prevent freed memory blocks from immediately returning to the pool of available memory. Selecting is useful for discovering if an application tries to use memory after freeing it. High **cost** if an application is performing many allocations/deallocations. Recommendation: Select to improve analysis quality if the cost is not too high.
Detect invalid memory accesses (split from Detect invalid/ uninitialized accesses)	Select to detect problems where a read or write instruction references memory that is logically or physically invalid. Selecting is useful to ensure an application accesses only valid memory. Medium **cost**. Recommendation: Select. **NOTE** May change application behavior by initializing memory that may normally be uninitialized. If your application reads this normally uninitialized memory, it may: • Simply miscalculate a value. • Treat the memory as a pointer, deference it, and crash during analysis.
Detect leaks at application exit (previously called Detect memory leaks upon application exit)	Select to report typical memory leaks in which the application allocates a memory block, never releases it, and doesn't keep a pointer to the block (e.g. **unreachable memory blocks**). Selecting is useful when an application: • Runs out of memory. • Appears to be using more memory than expected. Extremely low **cost** – especially if used only with **Remove duplicates** selected.

Setting	Purpose, Usefulness, and Cost
	Recommendation: Select.
Detect resource leaks	Select to detect open kernel and GDI handles when the application ends. For example, the application may open a file, get its handle, but never close or release that handle until it stops running. On Windows, GDI resources are limited, and the application may experience some drawing issues if it uses more than ~10,000 of these types of handles at once (pen, bitmap, brush, etc.) Selecting is useful if you see constant growth in acquired kernel and GDI objects in the system task manager for your process. Low **cost**. Recommendation: Select unless you want to focus on memory problems exclusively.
Detect still-allocated memory at application exit (previously called Report still-allocated memory at application exit)	Available only if **Detect leaks at application exit** is selected. Select to report typical memory leaks in which the application allocates a memory block, doesn't deallocate it, but a valid variable still holds a pointer to that block when the application ends (e.g. **reachable memory blocks**). **Cost** is proportional to the number of memory blocks still allocated when the application stops executing. Recommendation: Select to investigate memory growth.
Detect uninitialized memory reads (split from Detect invalid/ uninitialized accesses)	Available only if **Detect invalid memory accesses** is selected. Select to detect problems where a read instruction accesses an uninitialized memory location. Selecting is useful when an application: • Exhibits unexpected behavior. • Shows evidence of uninitialized values in computations. High **cost**. Recommendation: Deselect. **NOTE** May change application behavior by initializing memory that may normally be uninitialized. If your application reads this normally uninitialized memory, it may: • Simply miscalculate a value. • Treat the memory as a pointer, deference it, and crash during analysis.
Enable enhanced dangling pointer check	Available only if **Detect invalid memory accesses** is selected. Select to detect if an application is trying to access memory after it was logically freed. May be higher **cost** if an application is performing many allocations/deallocations, and the **Defer memory deallocation** list value is smaller than the amount of memory the application allocates. Recommendation: Select when an application exhibits unexpected behavior you suspect may be caused by a dangling pointer.

Setting	Purpose, Usefulness, and Cost
	NOTE • Changes application behavior by intercepting calls to deallocators and deferring actual deallocation. • Enhanced dangling pointer check is not supported for `new`/`delete` and `new[]`/`delete[]` allocation/deallocation pairs.
Enable guard zones	Available only if **Detect invalid memory accesses** is selected. Use in conjunction with **Guard zone size** to show offset information if the Intel Inspector detects memory use beyond the end of an allocated block. Selecting is useful when: • An application exhibits unexpected behavior. • You need more context about heap allocations to interpret **Invalid memory access** problems. **Cost** is proportional to number of allocations. Recommendation: Select unless: • Intel Inspector runs out of memory. • An application becomes destabilized. **NOTE** • May change application behavior. • Increases the amount of memory the Intel Inspector uses. • Intel Inspector creates guard zones only at the end of allocated space, not at the start of allocated space.
Enable memory growth detection (previously called Enable interactive memory growth detection)	Select to enable buttons in the GUI that let you send commands during application execution. This will show you a list of **reachable and unreachable memory blocks** for a time segment. Selecting is useful for modeling memory usage patterns and ensuring a transactional application deallocates all memory allocations after a transaction completes. Use in conjunction with the **Reset Growth Tracking** and **Measure Growth** buttons during analysis. Low **cost**.
Enable on-demand leak detection (previously called Enable on-demand memory leak detection)	Select to enable buttons in the GUI that let you send "leak" commands. This will show you a list of **unreachable memory blocks** for a time segment. Selecting is useful for checking for memory leaks in an application that never exits, or in only the portion of an application for which you are responsible. Use in conjunction with the **Reset Leak Tracking** and **Find Leaks** buttons during analysis. **Cost** is proportional to the number of allocations.

Setting	Purpose, Usefulness, and Cost
Guard zone size (previously called Guard zone byte size)	Available only if **Detect invalid memory accesses** and **Enable guard zones** are selected. Use in conjunction with **Enable guard zones** to set the number of bytes beyond the allocated block of memory the Intel Inspector reserves to identify **Invalid memory access** problems related to the allocation. Setting is useful when: • An application exhibits unexpected behavior. • You need more context about heap allocations to interpret **Invalid memory access** problems. **Cost** is proportional to number of allocations. Recommendation: Set unless: • Intel Inspector runs out of memory. • An application becomes destabilized. **NOTE** • May change application behavior. • Increases the amount of memory the Intel Inspector uses. • Intel Inspector creates guard zones only at the end of allocated space, not at the start of allocated space.
Maximum number of leaks shown in result (previously called Maximum memory leaks)	Use to set the maximum number of leaks the Intel Inspector shows in a result after analysis is complete. A zero setting shows all detected memory leaks. **Cost** is proportional to the number of leaks. Recommendation: Use the default value unless you want an exhaustive list of all leaks. **Tip** Even the default value can generate an unmanageable number of leaks. Consider sorting the displayed memory leaks by **Object Size**, fixing the largest leaks, and then re-inspecting your application. Or use the on-demand leak detection feature to narrow your focus and eat the elephant one bite at a time.
Remove duplicates	Deselect to show all occurrences of a detected problem in the **Code Locations** pane. Deselecting is: • Useful when you need to fully visualize all threads and problem occurrences in relation to time • Low **cost** in terms of time; however, the number of duplicate errors could crowd out the number of unique errors. Recommendation: Select.

Setting	Purpose, Usefulness, and Cost
Revert to previous uninitialized memory algorithm (not recommended)	Available only if **Detect uninitialized memory reads** is selected. The current algorithm for detecting uninitialized memory reads decreases false positives but increases analysis time and memory overhead. Select to use the previous version of the algorithm. Recommendation: Deselect.
Stack frame depth	Use to provide more or less call stack context for detected errors. A high setting is useful when analyzing highly object-oriented applications. A higher number does not significantly impact **cost**. Recommendation: Use only as large a value as an application requires to display complete call paths.

Configuration Settings for Custom Analysis Types Based on Threading Error Analysis Types

Use the **Properties** region of the dialog box to configure the custom analysis type.

The following table describes the purpose, usefulness, and *cost* (low, medium, high, or proportional in terms of time and resources) for each configuration setting in a custom analysis type based on a threading error analysis type. (The settings are listed in alphabetical order.)

Setting	Purpose, Usefulness, and Cost
Cross-thread stack access detection	Use to set the alert mechanism for when a thread accesses stack memory of another thread. The alert mechanism helps you decide if this is an issue that requires handling. All options are low **cost** if **Detect data races** is selected. Recommendation: • Use **Hide problems/Hide warnings** if using an OpenMP* or oneAPI Threading Building Blocks (oneTBB) programming model; or if cross-thread stack accesses are anticipated. Also select **Detect races on stack**. • Use **Hide problems/Show warnings** if cross-thread stack accesses are not anticipated. Also deselect **Detect data races on stack**. • Use **Show problems/Hide warnings** if cross-thread stack accesses are not anticipated but a previous analysis indicated they exist and you are not using an OpenMP* or oneAPI Threading Building Blocks (oneTBB) programming model. Also deselect **Detect data races on stack**.
Detect data races	Select to detect problems where multiple threads access the same memory location without proper synchronization and at least one access is a write. Selecting is useful when you suspect data races that are not yet evident. High **cost**. Recommendation: Select. Consider also deselecting **Use maximum resources** to reduce cost.

Setting	Purpose, Usefulness, and Cost
Detect data races on stack (previously called Detect data races on stack accesses)	Available only if **Detect data races** is selected. Select to detect data races for variables allocated on the stack. Selecting is useful when threads in an application share variables from the stack and you suspect data races on the variables. High **cost**. Recommendation: Deselect. If you select, consider also deselecting **Use maximum resources** to reduce cost.
Detect deadlocks	Select to detect problems where two or more threads are waiting for the other to release resources, but none of the threads releases the resources. Thus no thread can proceed. Selecting is useful when you want to troubleshoot the location of a deadlock. Low **cost**.
Detect lock hierarchy violations	Select to detect problems where the acquisition hierarchy order of multiple synchronization objects in one thread differs from the acquisition hierarchy order in another thread, and could cause a deadlock under certain conditions. Selecting is useful when an application has complicated synchronization and it is hard to verify correctness. Low **cost** unless an application has a significant number of locks.
Filter guaranteed atomics	Available only if **Detect data races** is selected. Select to ignore data races on guaranteed atomic operations on the Intel® P6 processor family or newer. For details, please refer to the *Guaranteed Atomic Operations* section of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. Selecting is useful if you observe many false data race reports on simple operations like Load or Store to shared variables and the size of those variables is less than the processor cache line size. Do not select this option if you develop cross-platform code that should work on other architectures. Selecting this option might also hide true-races on variables that were properly aligned during this run but might not be aligned in general, e.g., if it just works "by chance". Low **cost**. Recommendations: Use this option with caution only if you observe many false reports on simple memory operations.
Race analysis byte granularity (previously called Memory access byte granularity)	Available only if **Detect data races** is selected. Use to set the size of the smallest memory block the Intel Inspector considers a single block of memory when determining if non-synchronized accesses to a memory block constitute a data race. Selecting is useful to control memory consumption during analysis for some applications. High **cost** when set to 1 byte.

Setting	Purpose, Usefulness, and Cost
	Recommendation: Set to 4 unless you continually see data races based on safe access to smaller memory blocks. If so, reset to 1.
Remove duplicates	Deselect to show all occurrences of a detected problem in the **Code Locations** pane. Deselecting is: • Useful when you need to fully visualize all threads and problem occurrences in relation to time • Low **cost** in terms of time; however, the number of duplicate errors could crowd out the number of unique errors. Recommendation: Select.
Save stack on first access	Available only if **Detect data races** is selected. Select to show as much information as possible on all threads involved in a data race. Selecting is useful when investigating complex data race problems. High **cost**. Recommendation: Deselect on initial analysis runs. Select only when you need the maximum information and context about all threads involved in a data race to solve the problem.
Save stack on lock creation	Select to show creation information on synchronization objects involved in deadlocks, lock hierarchy violations, and data races. Selecting is useful when acquisition stacks are not sufficient to understand the problem. Low **cost**.
Save stack on memory allocation (previously called Save stack on allocation)	Available only if **Detect data races** is selected. Select to identify the allocation site of dynamically allocated memory objects involved in data races. Medium **cost**. Recommendation: Select when you need to identify the object hierarchy of low-level objects involved in data races. For example: If object R is involved in a data race and is instantiated within objects O1, O2, and O3, the allocation call stack can help you identify which encapsulating object is not properly protecting access to object R.
Stack frame depth	Use to provide more or less call stack context for detected errors. A high setting is useful when analyzing highly object-oriented applications. A higher number does not significantly impact **cost** with one exception: Choosing a higher number plus selecting **Save stack on first access** increases **cost**. Recommendation: Use only as large a value as an application requires to display complete call paths.
Terminate on deadlock	Available only if **Detect deadlocks** is selected. Select to stop analysis and application execution if the Intel Inspector detects a deadlock.

Setting	Purpose, Usefulness, and Cost
	Selecting is useful when running your application as part of a kernel or unit testing suite. Low **cost**. Recommendation: Deselect. Instead, use the corresponding knob in the command line interface to perform kernel or unit testing in a nightly scenario. If the Intel Inspector identifies a deadlock, decide if it is appropriate to continue analysis.
Use maximum resources	Select to potentially find more problems. High **cost**. Recommendation: Deselect to run a quicker analysis that should find most of your data race and cross-thread stack access problems. Once you have found and fixed these problems, select to get more complete analysis coverage of possible data race and cross-thread stack access problems.

See Also
Configure Analysis
Intel Inspector Filenames and Locations

Dialog Box: Delete Suppressions

To access this Intel Inspector dialog box: In the **Code Locations** pane, right-click marked result data you do not want to suppress in future analysis runs. In the context menu, choose **Do Not Suppress...**.

Use this dialog box to delete suppression rules applied to marked result data.

Use This	To Do This
Suppression rule list	Review all rules applied to the selected marked data.
Grid checkboxes	Choose a rule for deletion (select).
Rule description	Review rule details.
View button	View the stack frame(s) in the rule. When you are finished viewing, close the **View Stack** dialog box.
Remove button	Delete the checked rules in the rule list (click).

See Also
Interpret Result Data and Resolve Issues
Suppressions Support
Dialog Box: View Stack

Dialog Box: Disabled Problem Breakpoints

To access this Intel Inspector dialog box: Click the **Disable Breakpoint** button on the **Problem Details** pane.

Use this dialog box to re-enable problem breakpoints.

Use This	To Do This
Re-enable Breakpoint button	Re-enable the selected breakpoint (click).
Data row(s)	• View a list of currently disabled breakpoints. • Select a disabled breakpoint.

See Also

Investigate Problems Using Interactive Debugging

Dialog Box: Export Result

To access this Intel Inspector dialog box: From the:

- Visual Studio* **Solution Explorer**, right-click a result to display a context menu, then choose **Export Result...**.

  NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- Standalone Intel Inspector**Project Navigator** pane, right-click a result to display a context menu, then choose **Export Result...**.

Use this dialog box to create an easy-to-share result archive file.

Use This	To Do This
Save to field and **Browse** button	• Save the result archive file in the default location. • Save the result archive file in another location.
Include source files checkbox	• Include source files in the result archive file (select). • Exclude source files from the result archive file (deselect).
Export button	Create the result archive file. **NOTE** Intel Inspector shows progress as it creates the result archive file and displays warnings if applicable.

See Also

Export and Import Files

Intel Inspector Filenames and Locations

Dialog Box: Merge States

To access this Intel Inspector dialog box: In the **Problems** pane, right-click any problem to display a context menu, then choose **Merge States...**.

Use this dialog box to merge state information from any result in any project into the currently open result.

Use This	To Do This
Result path field and **Browse** button	Browse to and select any result in any project.

Use This	To Do This
Merge button	Update in the currently open result the state assigned to any issues that appear in both results. **NOTE** Intel Inspector does not expand the set of issues in the currently open result. Issues that appear only in the currently open result remain unchanged.

See Also

Collaborate on Results

Dialog Box: Options-General

To access this Intel Inspector dialog box:

- From the Microsoft Visual Studio* menu, choose **Tools** > **Options...**. In the **Options** dialog box, expand the **Intel Inspector <version>** folder and choose the **General** page.

NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- From the Standalone Intel Inspector GUI menu, choose **File** > **Options...**. In the **Options** dialog box, choose the **General** page.

Use this dialog box to configure the Intel Inspector behavior regarding the application.

Use This	To Do This
Application output destination radio buttons	Choose the application output destination: • Microsoft Visual Studio* output window • **Collection Log** window • Separate console window **NOTE** If you must interact with the application during execution, choose the separate console window option or use `stdin` redirection on the command line.

See Also

Startup Tasks

Dialog Box: Options-Result Location

To access this Intel Inspector dialog box:

- From the Visual Studio* menu, choose **Tools** > **Options...**. In the **Options** dialog box, expand the **Intel Inspector <version>** folder and choose the **Result Location** page.

NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- From the Standalone Intel Inspector GUI menu, choose **File** > **Options...**. In the **Options** dialog box, choose the **Result Location** page.

Use this dialog box to control - at a cross-project level - how the Intel Inspector names results and results directories.

Use This	To Do This
Result name template text box	Change the result and result directory name template.

See Also

Startup Tasks

Intel Inspector Filenames and Locations

Dialog Box: Options-State Management

To access this Intel Inspector dialog box:

- From the Visual Studio* menu, choose **Tools** > **Options...**. In the **Options** dialog box, expand the **Intel Inspector <version>** folder and choose the **State Management** page.

> **NOTE** In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- From the Standalone Intel Inspector GUI menu, choose **File** > **Options...**. In the **Options** dialog box, choose the **State Management** page.

Intel Inspector uses a *difference processor* to set problem states in future results based on problem states in a baseline result. Use this dialog box to choose a baseline result.

Use This	To Do This
Get problem results from previous result of same analysis type radio button	Focus on only those issues that require your attention by setting problem state in future results to: • **New** for all problems not present in the baseline result • **Not Fixed** for all problems marked **New** in the baseline result • **Regression** for all problems marked **Fixed** in the baseline result If the Intel Inspector cannot find a suitable previous result of the same analysis type, it sets problem state for all problems in future results to **New**.
Get problem states from a specific result radio button, path field, and **Browse** button	Set problem states in future results the same as above when your source code is undergoing branched development and you need to configure the difference processor to compare against a more appropriate previous result.
Do not get problem states from another result radio button	Set problem state for all problems in future results to **New**.

See Also

Interpret Result Data and Resolve Issues

Startup Tasks

States

Dialog Box: Problem Report

To access this Intel Inspector dialog box: In the **Summary** window, right-click one or more code locations, problems, or problem sets to display a context menu, then choose **Create Problem Report...**.

Use this dialog box to export result data in plain text format so you can distribute it to teammates in other media, such as email.

Use This	To Do This
Result data grouped by problem set	• View result data for the selected code location(s), problem(s), or problem set(s). • Select result data to copy to the system clipboard.
Include call stacks checkbox	• Include call stack information in the report (select). • Exclude call stack information from the report (deselect).
Save to File button	Save the result data (with or without call stack information) to a text file.
Copy to Clipboard button	Copy the selected result data to the system clipboard.

See Also

Collaborate on Results

Dialog Box: Project Properties-Binary/Symbol Search

(One way) To access this Intel Inspector dialog box:

- From the Visual Studio* menu, choose **Project** > **Intel Inspector <version> Project Properties...**. In the **Project Properties** dialog box, choose the **Binary/Symbol Search** tab.

  NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- From the Standalone Intel Inspector GUI menu, choose **File** > **Project Properties...**. In the **Project Properties** dialog box, choose the **Binary/Symbol Search** tab.

Use this dialog box to configure the Intel Inspector to search non-standard directories for the supporting files necessary to execute the application during collection and manage result data after collection.

NOTE
If you take full advantage of the solutions/projects paradigm to create application software in the Visual Studio* IDE, searching for specific directories should be unnecessary.

Use This	To Do This
Search Directories	• Search non-standard directories. • View non-standard directories currently in the search list.
 button	Browse for directories to include in the search list.
 and	Change the search order of the selected directory.

Use This	To Do This
 buttons	
 button	Remove the selected directory from the search list.

See Also

Configure Projects

Binary/Symbol Search and Source Search Locations

Dialog Box: Project Properties-Source Search

(One way) To access this Intel Inspector dialog box:

- From the Visual Studio* menu, choose **Project** > **Intel Inspector <version> Project Properties...**. In the **Project Properties** dialog box, choose the **Source Search** tab.

> **NOTE** In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- From the Standalone Intel Inspector GUI menu, choose **File** > **Project Properties...**. In the **Project Properties** dialog box, choose the **Source Search** tab.

Use this dialog box to configure the Intel Inspector to search non-standard directories for the supporting files necessary to execute the application during collection and manage result data after collection.

> **NOTE**
> If you take full advantage of the solutions/projects paradigm to create application software in the Visual Studio* IDE, searching for specific directories should be unnecessary.

Use This	To Do This
Search Directories	• Search non-standard directories. • View non-standard directories currently in the search list.
 button	Browse for directories to include in the search list.
 and  buttons	Change the search order of the selected directory.
 button	Remove the selected directory from the search list.

See Also

Configure Projects

Binary/Symbol Search and Source Search Locations

Dialog Box: Project Properties-Suppressions

(One way) To access this Intel Inspector dialog box:

- From the Visual Studio* menu, choose **Project** > **Intel Inspector <version> Project Properties...**. In the **Project Properties** dialog box, choose the **Suppressions** tab.

  NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- From the Standalone Intel Inspector GUI menu, choose **File** > **Project Properties...**. In the **Project Properties** dialog box, choose the **Suppressions** tab.

Use this dialog box to manage the suppression rules applied to each project during analysis.

NOTE
To apply suppression rules during analysis, select the **Apply Suppressions** radio button on the **Target** tab of the **Project Properties** dialog box.

Use This	To Do This
Suppression files region	• Review the files and directories containing rules. • Add files that contains rules (click the **Add File** button). • Add directories that contain (or will contain) files with rules (click the **Add Directory** button). • Remove files and directories from the list of those to be applied during future analysis runs for this project (click the files and directories, then click the **Remove** button directly below the region). **NOTE** • Removing a file or directory from the list does not delete the file or directory from your file system. • You cannot remove the project suppression directory. • Intel Inspector does not search recursively in directories for files identified by a `*.sup` mask.
Suppression rule list region	• Review the rules in the files and directories identified in the **Suppression files** region. • View the stack frame(s) in a rule (click the **View...** button). When you are finished viewing, close the **View Stack** dialog box. • Remove rules from the list of those to be applied during future analysis runs for this project (click the rules, then click the **Remove** button directly below the region). **Caution** Removing a rule from the list deletes the rule from your file system.

See Also

Configure Projects
Suppressions Support
Dialog Box: Project Properties-Target

Dialog Box: View Stack

Dialog Box: Project Properties-Target

(One way) To access this Intel Inspector dialog box:

- From the Visual Studio* menu, choose **Project** > **Intel Inspector <version> Project Properties...**.

> **NOTE** In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- From the Standalone Intel Inspector GUI menu, choose **File** > **Project Properties...**.

Use this dialog box to:

- Choose the project application.
- Set basic target application properties, such as application parameters, working directory, and environment variables.
- Set advanced target application properties, such as applying suppressions and including or excluding modules during analysis.
- Review current target application properties.

Use This	To Do This
Inherit settings from Visual Studio* project checkbox and field	Visual Studio* IDE only: • Populate all basic settings from the Visual Studio* property pages of the StartUp project (**Configuration Properties** > **Debugging**). • Disable the basic settings section of the dialog box.
Microsoft* runtime environment drop-down list	• Automatically detect the Microsoft* runtime environment based on binary executable type (choose **Auto**). • Potentially speed up collection by excluding application managed code from inspection (choose **Native only**). • Potentially speed up collection by excluding application native code from inspection (choose **Managed only**). • Inspect all code, regardless of whether it is native or managed (choose **Mixed**). **NOTE** • Microsoft .NET* 3.5 software support is deprecated in the Intel Inspector and will be removed after August, 2020. This deprecation does not apply to the Intel® VTune™ Profiler. See *Release Notes* for details. • During threading analysis, the Intel Inspector analyzes both native and managed code. • During memory analysis, the Intel Inspector analyzes only native code. If you choose **Mixed**, there is no analysis of the managed portion of your code. • During threading or memory analysis, the Intel Inspector supports interactive debugging only for pure native applications. If you choose **Mixed** or **Managed only**, the debug options are disabled.

Use This	To Do This
Store result... radio buttons, path fields, **Browse** button, and **Result location** read-only field	• Save results in result directories in the default location (select the **Store result in the project directory** radio button). • Save results in result directories in a custom location (select the **Store result in (and create link file to) another directory** radio button, and identify the custom location). **NOTE** Change the cross-project result name template to the change the result name shown in the **Result location** field.
Suppressions radio buttons	• Not collect result data impacted by the suppression rules in files and directories specified in the **Suppressions** tab of the **Project Properties** dialog box (choose **Apply suppressions**). • Ignore all suppression rules (choose **Do not apply suppressions**).
Child application field	Inspect a file that is not the starting application. For example: Inspect an `.exe` file (identified in this field) called by a script (identified in the **Application** field).
Enable collection progress information checkbox	Display thread activity during collection to confirm the application is still executing.
Modules radio buttons, **Modify** button, and field	Potentially speed up collection by limiting application module(s) for inspection. You can limit by inclusion or exclusion. For example, you can: • Inspect specific modules and disable inspection of all other modules (click the **Include only the following module(s)** radio button and choose the modules). • Disable inspection of specific modules and inspect all other modules (click the **Exclude the following module(s)** radio button and choose the modules). **NOTE** By default, the Intel Inspector inspects all modules in the application.

See Also

Choose Projects
Configure Projects
Startup Tasks
Dialog Box: Project Properties-Suppressions
Intel Inspector Filenames and Locations

Dialog Box: Refine Source File Set

To access this Intel Inspector dialog box: Right-click anywhere in the **Filters** pane to display a context menu, then choose **Refine Source File Set...**.

Use this dialog box to temporarily limit displayed issues to those from a set of source files.

Use This	To Do This
Search substring field	Enter any portion of a file path to use as a criterion for selecting a file set.

Use This	To Do This
	NOTE Starting and ending wildcard characters are unnecessary; however, you can use wildcard characters within the substring. For example: • The substring `i?e` displays the `video.cpp` and `asctime.c` files in the **Matching files** list. • The substring `i*i` displays the `winvideo.h`, `find_and_fix_threading_errors.cpp`, and `task_scheduler_init.h` files in the **Matching files** list.
Matching files list	View the names of all source files available as filter criteria: • Before you enter a search substring • As you enter a search substring

See Also
Choose Problems

Dialog Box: Select Stack Frame(s)

To access this Intel Inspector dialog box: In the **Create Suppressions** dialog box, click the **Edit...** button.

Use this dialog box while defining a suppression rule to:

- View stack frames.
- Choose a different stack frame as the rule focus.
- Choose multiple stack frames as the rule focus.

Tip
Suppression rules are more robust if you specify a stack with multiple frames instead of frames with line numbers. (Because line numbers can be altered by code insertions or deletions, suppressions may be rendered ineffective by even minor code maintenance. Stack-based suppressions require larger code changes to invalidate them, such as changes to function call sequences.)

Use This	To Do This
Use in Rule checkboxes	• Suppress problems during analysis based on stack frame characteristics (select checkbox). Outcome: Rules with narrower reach. • Do not suppress problems based on stack frame characteristics (deselect checkbox). Outcome: Rules with wider reach.
Grid row drop-down lists	• Suppress problems during analysis based on specific stack frame characteristics (keep specific **Module**, **Function**, **Source**, and/or **Line** data). Outcome: Rules with narrower reach. • Do not suppress problems based on specific stack frame characteristics (click drop-down arrow and choose ***(any)**). Outcome: Rules with wider reach.

See Also
Interpret Result Data and Resolve Issues
Suppressions Support
Dialog Box: Create Suppression

Dialog Box: View Stack

To access this Intel Inspector dialog box:

- In the **Delete Suppressions** dialog box, click the **View...** button.
- In the **Suppressions** tab of the **Project Properties** dialog box, click the **View...** button.

Use this dialog box while deleting a suppression rule to view the stack frame(s) in the rule.

See Also

Interpret Result Data and Resolve Issues
Suppressions Support
Deleting Suppression Rules Applied to Marked Result Data in the GUI
Manage Suppression Rules
Dialog Box: Delete Suppressions
Dialog Box: Project Properties-Suppressions

Hot Keys

Use hot keys to quickly perform various tasks in the Standalone Intel Inspector GUI:

Press This	To Do This
Ctrl+N	Open the **Analysis Type** window, where you can: • Choose and, if necessary, fine-tune a preset memory or threading error analysis type. • Configure a custom analysis type.
Ctrl+Shift+N	Open the **Create a Project** dialog box, where you can browse to or create a directory the Intel Inspector will populate with a new `config.inspxeproj` file.
Ctrl+Alt+N	Open the **Import Result** window, where you can import result archive files, results not associated with a project, and results from other Intel® error-detection products into the current project. **Tip** Refinalize the imported result to apply current project symbol information. (Use the **Re-resolve** option on the **Solution Explorer** or **Project Navigator** pane context menu.)
Ctrl+O	Open a **Select Result** dialog box, where you can browse to and choose an existing `*.inspxe` file.
Ctrl+Shift+O	Open a **Select Project** dialog box, where you can browse to and choose an existing `config.inspxeproj` file.
Ctrl+Alt+O	Open the **Compare Results** window, where you can compare two results; and identify issues that exist in one but not the other, or that still exist in both.
Ctrl+P	Open the **Project Properties** dialog box, where you can review or change project properties.
F11	Toggle on and off a full-screen view of result tab data. You can also: • Choose **View** > **Full Screen** to toggle on full-screen view. • Press ESC to toggle off full-screen view.

See Also

Choose Projects
Compare Results
Configure Analysis
Configure Projects
Export and Import Files
Run Analysis
View Results
Intel Inspector Filenames and Locations

Pane: Analysis Type-Custom

Before Analysis

(One way) To access this Intel Inspector pane:

- From the Visual Studio* menu, choose **Tools > Intel Inspector*version* > New Analysis...**. Then choose **Custom Analysis Types** from the Analysis Type drop-down list.

> **NOTE** In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- From the Standalone Intel Inspector GUI menu, choose **File** > **New** > **Analysis...**. Then choose **Custom Analysis Types** from the Analysis Type drop-down list.

Use this pane on the **Analysis Type** window to:

- Choose and, if necessary, fine-tune a custom analysis type.
- Configure a custom analysis to investigate problems in an interactive debugging session.

During Analysis

To access this pane: Click the **Analysis Type** button on the **Navigation** toolbar.

Use this pane to review the analysis type settings for this analysis run.

After Analysis Is Complete

To access this pane: Click the **Analysis Type** button on the **Navigation** toolbar.

Use this pane to:

- Review the analysis type settings for this analysis run.
- Re-inspect - run another analysis using the same analysis type settings.

Use This	To Do This
Analysis Type drop-down list	Switch to another category of analysis types.
Custom analysis type selection box	Choose an existing custom analysis type.
Copy button	Create a new custom analysis type based on the currently selected analysis type.
Edit button	Modify this custom analysis type.
Delete button	Delete this custom analysis type.

Use This	To Do This
Analyze without debugger radio button	Select to run an analysis without launching an interactive debugging session. Useful for investigating all types of memory and threading problems. **Tip** You can later use the **Debug This Problem** function from within a result to launch a new analysis in conjunction with a debugger to stop at problems of interest. The rerun analysis is automatically focused to find the selected problems, making it return to the problems more quickly. The **Debug This Problem** function is the recommended method for investigating threading errors in an interactive debugging session.
Enable debugger when problem detected radio button	Select to allow investigation of every problem detected in an interactive debugging session. Useful for investigating all types of memory problems except memory and resource leaks.
Select analysis start location with debugger radio button	Select to allow investigation of problems in a particular area of code during an interactive debugging session. Typical scenario: You need to check for errors in a specific section of application code, but the Intel Inspector does not provide the appropriate granularity to analyze only that code section. This option quickly takes you near that execution point in a debugging session. Useful for investigating all types of memory problems except memory and resource leaks. **Tip** You can also use this option to investigate all types of threading errors, but it may be quicker to use the **Debug This Problem** function in a result if you plan to investigate a single threading problem.

Configuration Settings for Custom Analysis Types Based on Memory Error Analysis Types

The following table describes the purpose, usefulness, and *cost* (low, medium, high, or proportional in terms of time and resources) for each configuration setting in a custom analysis type based on a memory error analysis type. (The settings are listed in alphabetical order.)

Setting	Purpose, Usefulness, and Cost
Analyze stack accesses	Available only if **Detect invalid memory accesses** is selected. Select to analyze invalid and uninitialized accesses to thread stacks. Selecting is useful when: • You want as thorough an analysis as possible. • An application calls `alloca()`. High **cost**. Recommendation: • Select the first time you analyze an application and periodically thereafter. • Select to analyze automatic variables.

Setting	Purpose, Usefulness, and Cost
Defer memory deallocation (previously called Byte limit before reallocation)	Available only if **Detect invalid memory accesses** and **Enable enhanced dangling pointer check** are selected. Select to have the Intel Inspector prevent freed memory blocks from immediately returning to the pool of available memory. Selecting is useful for discovering if an application tries to use memory after freeing it. High **cost** if an application is performing many allocations/deallocations. Recommendation: Select to improve analysis quality if the cost is not too high.
Detect invalid memory accesses (split from Detect invalid/ uninitialized accesses)	Select to detect problems where a read or write instruction references memory that is logically or physically invalid. Selecting is useful to ensure an application accesses only valid memory. Medium **cost**. Recommendation: Select. **NOTE** May change application behavior by initializing memory that may normally be uninitialized. If your application reads this normally uninitialized memory, it may: • Simply miscalculate a value. • Treat the memory as a pointer, deference it, and crash during analysis.
Detect leaks at application exit (previously called Detect memory leaks upon application exit)	Select to report typical memory leaks in which the application allocates a memory block, never releases it, and doesn't keep a pointer to the block (e.g. **unreachable memory blocks**). Selecting is useful when an application: • Runs out of memory. • Appears to be using more memory than expected. Extremely low **cost** – especially if used only with **Remove duplicates** selected. Recommendation: Select.
Detect resource leaks	Select to detect open kernel and GDI handles when the application ends. For example, the application may open a file, get its handle, but never close or release that handle until it stops running. On Windows, GDI resources are limited, and the application may experience some drawing issues if it uses more than ~10,000 of these types of handles at once (pen, bitmap, brush, etc.) Selecting is useful if you see constant growth in acquired kernel and GDI objects in the system task manager for your process. Low **cost**. Recommendation: Select unless you want to focus on memory problems exclusively.
Detect still-allocated memory at application exit (previously called	Available only if **Detect leaks at application exit** is selected. Select to report typical memory leaks in which the application allocates a memory block, doesn't deallocate it, but a valid variable still holds a pointer to that block when the application ends (e.g. **reachable memory blocks**).

Setting	Purpose, Usefulness, and Cost
Report still-allocated memory at application exit)	**Cost** is proportional to the number of memory blocks still allocated when the application stops executing. Recommendation: Select to investigate memory growth.
Detect uninitialized memory reads (split from Detect invalid/ uninitialized accesses)	Available only if **Detect invalid memory accesses** is selected. Select to detect problems where a read instruction accesses an uninitialized memory location. Selecting is useful when an application: • Exhibits unexpected behavior. • Shows evidence of uninitialized values in computations. High **cost**. Recommendation: Deselect. **NOTE** May change application behavior by initializing memory that may normally be uninitialized. If your application reads this normally uninitialized memory, it may: • Simply miscalculate a value. • Treat the memory as a pointer, deference it, and crash during analysis.
Enable enhanced dangling pointer check	Available only if **Detect invalid memory accesses** is selected. Select to detect if an application is trying to access memory after it was logically freed. May be higher **cost** if an application is performing many allocations/deallocations, and the **Defer memory deallocation** list value is smaller than the amount of memory the application allocates. Recommendation: Select when an application exhibits unexpected behavior you suspect may be caused by a dangling pointer. **NOTE** • Changes application behavior by intercepting calls to deallocators and deferring actual deallocation. • Enhanced dangling pointer check is not supported for `new`/`delete` and `new[]`/`delete[]` allocation/deallocation pairs.
Enable guard zones	Available only if **Detect invalid memory accesses** is selected. Use in conjunction with **Guard zone size** to show offset information if the Intel Inspector detects memory use beyond the end of an allocated block. Selecting is useful when: • An application exhibits unexpected behavior. • You need more context about heap allocations to interpret **Invalid memory access** problems.

Setting	Purpose, Usefulness, and Cost
	Cost is proportional to number of allocations. Recommendation: Select unless: • Intel Inspector runs out of memory. • An application becomes destabilized. **NOTE** • May change application behavior. • Increases the amount of memory the Intel Inspector uses. • Intel Inspector creates guard zones only at the end of allocated space, not at the start of allocated space.
Enable memory growth detection (previously called Enable interactive memory growth detection)	Select to enable buttons in the GUI that let you send commands during application execution. This will show you a list of **reachable and unreachable memory blocks** for a time segment. Selecting is useful for modeling memory usage patterns and ensuring a transactional application deallocates all memory allocations after a transaction completes. Use in conjunction with the **Reset Growth Tracking** and **Measure Growth** buttons during analysis. Low **cost**.
Enable on-demand leak detection (previously called Enable on-demand memory leak detection)	Select to enable buttons in the GUI that let you send "leak" commands. This will show you a list of **unreachable memory blocks** for a time segment. Selecting is useful for checking for memory leaks in an application that never exits, or in only the portion of an application for which you are responsible. Use in conjunction with the **Reset Leak Tracking** and **Find Leaks** buttons during analysis. **Cost** is proportional to the number of allocations.
Guard zone size (previously called Guard zone byte size)	Available only if **Detect invalid memory accesses** and **Enable guard zones** are selected. Use in conjunction with **Enable guard zones** to set the number of bytes beyond the allocated block of memory the Intel Inspector reserves to identify **Invalid memory access** problems related to the allocation. Setting is useful when: • An application exhibits unexpected behavior. • You need more context about heap allocations to interpret **Invalid memory access** problems. **Cost** is proportional to number of allocations. Recommendation: Set unless: • Intel Inspector runs out of memory. • An application becomes destabilized.

Setting	Purpose, Usefulness, and Cost
	NOTE • May change application behavior. • Increases the amount of memory the Intel Inspector uses. • Intel Inspector creates guard zones only at the end of allocated space, not at the start of allocated space.
Maximum number of leaks shown in result (previously called Maximum memory leaks)	Use to set the maximum number of leaks the Intel Inspector shows in a result after analysis is complete. A zero setting shows all detected memory leaks. **Cost** is proportional to the number of leaks. Recommendation: Use the default value unless you want an exhaustive list of all leaks. **Tip** Even the default value can generate an unmanageable number of leaks. Consider sorting the displayed memory leaks by **Object Size**, fixing the largest leaks, and then re-inspecting your application. Or use the on-demand leak detection feature to narrow your focus and eat the elephant one bite at a time.
Remove duplicates	Deselect to show all occurrences of a detected problem in the **Code Locations** pane. Deselecting is: • Useful when you need to fully visualize all threads and problem occurrences in relation to time • Low **cost** in terms of time; however, the number of duplicate errors could crowd out the number of unique errors. Recommendation: Select.
Revert to previous uninitialized memory algorithm (not recommended)	Available only if **Detect uninitialized memory reads** is selected. The current algorithm for detecting uninitialized memory reads decreases false positives but increases analysis time and memory overhead. Select to use the previous version of the algorithm. Recommendation: Deselect.
Stack frame depth	Use to provide more or less call stack context for detected errors. A high setting is useful when analyzing highly object-oriented applications. A higher number does not significantly impact **cost**. Recommendation: Use only as large a value as an application requires to display complete call paths.

Configuration Settings for Custom Analysis Types Based on Threading Error Analysis Types

The following table describes the purpose, usefulness, and *cost* (low, medium, high, or proportional in terms of time and resources) for each configuration setting in a custom analysis type based on a threading error analysis type. (The settings are listed in alphabetical order.)

Setting	Purpose, Usefulness, and Cost
Cross-thread stack access detection	Use to set the alert mechanism for when a thread accesses stack memory of another thread. The alert mechanism helps you decide if this is an issue that requires handling. All options are low **cost** if **Detect data races** is selected. Recommendation: • Use **Hide problems/Hide warnings** if using an OpenMP* or oneAPI Threading Building Blocks (oneTBB) programming model; or if cross-thread stack accesses are anticipated. Also select **Detect races on stack**. • Use **Hide problems/Show warnings** if cross-thread stack accesses are not anticipated. Also deselect **Detect data races on stack**. • Use **Show problems/Hide warnings** if cross-thread stack accesses are not anticipated but a previous analysis indicated they exist and you are not using an OpenMP* or oneAPI Threading Building Blocks (oneTBB) programming model. Also deselect **Detect data races on stack**.
Detect data races	Select to detect problems where multiple threads access the same memory location without proper synchronization and at least one access is a write. Selecting is useful when you suspect data races that are not yet evident. High **cost**. Recommendation: Select. Consider also deselecting **Use maximum resources** to reduce cost.
Detect data races on stack (previously called Detect data races on stack accesses)	Available only if **Detect data races** is selected. Select to detect data races for variables allocated on the stack. Selecting is useful when threads in an application share variables from the stack and you suspect data races on the variables. High **cost**. Recommendation: Deselect. If you select, consider also deselecting **Use maximum resources** to reduce cost.
Detect deadlocks	Select to detect problems where two or more threads are waiting for the other to release resources, but none of the threads releases the resources. Thus no thread can proceed. Selecting is useful when you want to troubleshoot the location of a deadlock. Low **cost**.
Detect lock hierarchy violations	Select to detect problems where the acquisition hierarchy order of multiple synchronization objects in one thread differs from the acquisition hierarchy order in another thread, and could cause a deadlock under certain conditions. Selecting is useful when an application has complicated synchronization and it is hard to verify correctness. Low **cost** unless an application has a significant number of locks.

Setting	Purpose, Usefulness, and Cost
Filter guaranteed atomics	Available only if **Detect data races** is selected. Select to ignore data races on guaranteed atomic operations on the Intel® P6 processor family or newer. For details, please refer to the *Guaranteed Atomic Operations* section of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. Selecting is useful if you observe many false data race reports on simple operations like Load or Store to shared variables and the size of those variables is less than the processor cache line size. Do not select this option if you develop cross-platform code that should work on other architectures. Selecting this option might also hide true-races on variables that were properly aligned during this run but might not be aligned in general, e.g., if it just works "by chance". Low **cost**. Recommendations: Use this option with caution only if you observe many false reports on simple memory operations.
Race analysis byte granularity (previously called Memory access byte granularity)	Available only if **Detect data races** is selected. Use to set the size of the smallest memory block the Intel Inspector considers a single block of memory when determining if non-synchronized accesses to a memory block constitute a data race. Selecting is useful to control memory consumption during analysis for some applications. High **cost** when set to 1 byte. Recommendation: Set to 4 unless you continually see data races based on safe access to smaller memory blocks. If so, reset to 1.
Remove duplicates	Deselect to show all occurrences of a detected problem in the **Code Locations** pane. Deselecting is: • Useful when you need to fully visualize all threads and problem occurrences in relation to time • Low **cost** in terms of time; however, the number of duplicate errors could crowd out the number of unique errors. Recommendation: Select.
Save stack on first access	Available only if **Detect data races** is selected. Select to show as much information as possible on all threads involved in a data race. Selecting is useful when investigating complex data race problems. High **cost**. Recommendation: Deselect on initial analysis runs. Select only when you need the maximum information and context about all threads involved in a data race to solve the problem.

Setting	Purpose, Usefulness, and Cost
Save stack on lock creation	Select to show creation information on synchronization objects involved in deadlocks, lock hierarchy violations, and data races. Selecting is useful when acquisition stacks are not sufficient to understand the problem. Low **cost**.
Save stack on memory allocation (previously called Save stack on allocation)	Available only if **Detect data races** is selected. Select to identify the allocation site of dynamically allocated memory objects involved in data races. Medium **cost**. Recommendation: Select when you need to identify the object hierarchy of low-level objects involved in data races. For example: If object R is involved in a data race and is instantiated within objects O1, O2, and O3, the allocation call stack can help you identify which encapsulating object is not properly protecting access to object R.
Stack frame depth	Use to provide more or less call stack context for detected errors. A high setting is useful when analyzing highly object-oriented applications. A higher number does not significantly impact **cost** with one exception: Choosing a higher number plus selecting **Save stack on first access** increases **cost**. Recommendation: Use only as large a value as an application requires to display complete call paths.
Terminate on deadlock	Available only if **Detect deadlocks** is selected. Select to stop analysis and application execution if the Intel Inspector detects a deadlock. Selecting is useful when running your application as part of a kernel or unit testing suite. Low **cost**. Recommendation: Deselect. Instead, use the corresponding knob in the command line interface to perform kernel or unit testing in a nightly scenario. If the Intel Inspector identifies a deadlock, decide if it is appropriate to continue analysis.
Use maximum resources	Select to potentially find more problems. High **cost**. Recommendation: Deselect to run a quicker analysis that should find most of your data race and cross-thread stack access problems. Once you have found and fixed these problems, select to get more complete analysis coverage of possible data race and cross-thread stack access problems.

See Also

Configure Analysis
Investigate Problems Using Interactive Debugging
Memory Error Analysis Types
Threading Error Analysis Types
Intel Inspector Filenames and Locations

Pane: Analysis Type-Memory Errors

Before Analysis

(One way) To access this Intel Inspector pane:

- From the Visual Studio* menu, choose **Tools> Intel Inspector*version*> New Analysis...**. Then choose **Memory Error Analysis** from the Analysis Type drop-down list.

  NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- From the Standalone Intel Inspector GUI menu, choose **File** > **New** > **Analysis...**. Then choose **Memory Error Analysis** from the Analysis Type drop-down list.

Use this pane on the **Analysis Type** window to:

- Choose and, if necessary, fine-tune a preset memory error analysis type.
- Configure a memory error analysis to investigate problems in an interactive debugging session.

Tip

- If the combination of analysis type settings in the preset memory error analysis types does not meet your needs at all, try creating a custom memory error analysis type.
- Intel Inspector does not offer interactive debugging for the **Detect Leaks** analysis type because memory and resource leaks are determined after an application terminates and therefore cannot be used to halt execution during analysis. However, you can perform a standard debugger attach to a process launched under this analysis type.

During Analysis

To access this pane: Click the **Analysis Type** button on the **Navigation** toolbar.

Use this pane to review the analysis type settings for this analysis run.

After Analysis Is Complete

To access this pane: Click the **Analysis Type** button on the **Navigation** toolbar.

Use this pane to:

- Review the analysis type settings for this analysis run.
- Re-inspect - run another analysis using the same analysis type settings.

Use This	To Do This
Analysis Type drop-down list	Switch to another category of analysis types.
Configuration slider	Choose a preset analysis type (drag slider).
Analysis Time Overhead gauge	Quickly estimate the time it may take to collect a result using various preset analyses. Time is expressed in relation to normal application execution time.

Use This	To Do This
	For example, 2x - 20x is 2 to 20 times longer than normal application execution time. If normal application execution time is 5 seconds, estimated collection time is 10 to 100 seconds.
Memory Overhead gauge 	The **Memory Overhead** gauge helps you quickly estimate the memory the Intel Inspector may consume to detect errors using this preset analysis type. Memory is expressed in blue-filled bars. **NOTE** The gauge does not show memory used by the running application during analysis.
Copy button	Create a new custom analysis type based on the currently selected analysis type.
Analyze without debugger radio button (**Detect Memory Problems** and **Locate Memory Problems** analysis types only)	Select to run an analysis without launching an interactive debugging session. Useful for investigating all types of memory and threading problems. **Tip** You can later use the **Debug This Problem** function from within a result to launch a new analysis in conjunction with a debugger to stop at problems of interest. The rerun analysis is automatically focused to find the selected problems, making it return to the problems more quickly. The **Debug This Problem** function is the recommended method for investigating threading errors in an interactive debugging session.
Enable debugger when problem detected radio button (**Detect Memory Problems** and **Locate Memory Problems** analysis types only)	Select to allow investigation of every problem detected in an interactive debugging session. Useful for investigating all types of memory problems except memory and resource leaks.
Select analysis start location with debugger radio button (**Detect Memory Problems** and **Locate Memory Problems** analysis types only)	Select to allow investigation of problems in a particular area of code during an interactive debugging session. Typical scenario: You need to check for errors in a specific section of application code, but the Intel Inspector does not provide the appropriate granularity to analyze only that code section. This option quickly takes you near that execution point in a debugging session. Useful for investigating all types of memory problems except memory and resource leaks.

Configurable Memory Error Analysis Type Settings

The following list shows the configurable settings (in alphabetical order) for preset memory analysis types. *Configurable* means you can change the setting without creating a custom analysis type:

- **Analyze stack accesses** checkbox (configurable for the **Locate Memory Problems** analysis type only)

- **Detect leaks at application exit** checkbox
- **Detect resource leaks** checkbox
- **Detect still-allocated memory at application exit** checkbox
- **Enable memory growth detection** checkbox
- **Enable on-demand leak detection** checkbox
- **Remove duplicates** checkbox (configurable for the **Locate Memory Problems** analysis type only)
- **Stack frame depth** drop-down list

Memory Error Analysis Type Settings

Use the **Details** region to review current analysis type settings. The following table describes the purpose, usefulness, and *cost* (low, medium, high, or proportional in terms of time and resources) for each memory error analysis type configuration setting. (The settings are listed in alphabetical order.)

Setting	Purpose, Usefulness, and Cost
Analyze stack accesses	Available only if **Detect invalid memory accesses** is selected. Select to analyze invalid and uninitialized accesses to thread stacks. Selecting is useful when: • You want as thorough an analysis as possible. • An application calls `alloca()`. High **cost**. Recommendation: • Select the first time you analyze an application and periodically thereafter. • Select to analyze automatic variables.
Defer memory deallocation (previously called Byte limit before reallocation)	Available only if **Detect invalid memory accesses** and **Enable enhanced dangling pointer check** are selected. Select to have the Intel Inspector prevent freed memory blocks from immediately returning to the pool of available memory. Selecting is useful for discovering if an application tries to use memory after freeing it. High **cost** if an application is performing many allocations/deallocations. Recommendation: Select to improve analysis quality if the cost is not too high.
Detect invalid memory accesses (split from Detect invalid/ uninitialized accesses)	Select to detect problems where a read or write instruction references memory that is logically or physically invalid. Selecting is useful to ensure an application accesses only valid memory. Medium **cost**. Recommendation: Select. **NOTE** May change application behavior by initializing memory that may normally be uninitialized. If your application reads this normally uninitialized memory, it may: • Simply miscalculate a value. • Treat the memory as a pointer, deference it, and crash during analysis.

Setting	Purpose, Usefulness, and Cost
Detect leaks at application exit (previously called Detect memory leaks upon application exit)	Select to report typical memory leaks in which the application allocates a memory block, never releases it, and doesn't keep a pointer to the block (e.g. **unreachable memory blocks**). Selecting is useful when an application: • Runs out of memory. • Appears to be using more memory than expected. Extremely low **cost** – especially if used only with **Remove duplicates** selected. Recommendation: Select.
Detect resource leaks	Select to detect open kernel and GDI handles when the application ends. For example, the application may open a file, get its handle, but never close or release that handle until it stops running. On Windows, GDI resources are limited, and the application may experience some drawing issues if it uses more than ~10,000 of these types of handles at once (pen, bitmap, brush, etc.) Selecting is useful if you see constant growth in acquired kernel and GDI objects in the system task manager for your process. Low **cost**. Recommendation: Select unless you want to focus on memory problems exclusively.
Detect still-allocated memory at application exit (previously called Report still-allocated memory at application exit)	Available only if **Detect leaks at application exit** is selected. Select to report typical memory leaks in which the application allocates a memory block, doesn't deallocate it, but a valid variable still holds a pointer to that block when the application ends (e.g. **reachable memory blocks**). **Cost** is proportional to the number of memory blocks still allocated when the application stops executing. Recommendation: Select to investigate memory growth.
Detect uninitialized memory reads (split from Detect invalid/ uninitialized accesses)	Available only if **Detect invalid memory accesses** is selected. Select to detect problems where a read instruction accesses an uninitialized memory location. Selecting is useful when an application: • Exhibits unexpected behavior. • Shows evidence of uninitialized values in computations. High **cost**. Recommendation: Deselect. **NOTE** May change application behavior by initializing memory that may normally be uninitialized. If your application reads this normally uninitialized memory, it may: • Simply miscalculate a value. • Treat the memory as a pointer, deference it, and crash during analysis.

Setting	Purpose, Usefulness, and Cost
Enable enhanced dangling pointer check	Available only if **Detect invalid memory accesses** is selected. Select to detect if an application is trying to access memory after it was logically freed. May be higher **cost** if an application is performing many allocations/deallocations, and the **Defer memory deallocation** list value is smaller than the amount of memory the application allocates. Recommendation: Select when an application exhibits unexpected behavior you suspect may be caused by a dangling pointer. **NOTE** • Changes application behavior by intercepting calls to deallocators and deferring actual deallocation. • Enhanced dangling pointer check is not supported for `new`/`delete` and `new[]`/`delete[]` allocation/deallocation pairs.
Enable guard zones	Available only if **Detect invalid memory accesses** is selected. Use in conjunction with **Guard zone size** to show offset information if the Intel Inspector detects memory use beyond the end of an allocated block. Selecting is useful when: • An application exhibits unexpected behavior. • You need more context about heap allocations to interpret **Invalid memory access** problems. **Cost** is proportional to number of allocations. Recommendation: Select unless: • Intel Inspector runs out of memory. • An application becomes destabilized. **NOTE** • May change application behavior. • Increases the amount of memory the Intel Inspector uses. • Intel Inspector creates guard zones only at the end of allocated space, not at the start of allocated space.
Enable memory growth detection (previously called Enable interactive memory growth detection)	Select to enable buttons in the GUI that let you send commands during application execution. This will show you a list of **reachable and unreachable memory blocks** for a time segment. Selecting is useful for modeling memory usage patterns and ensuring a transactional application deallocates all memory allocations after a transaction completes. Use in conjunction with the **Reset Growth Tracking** and **Measure Growth** buttons during analysis. Low **cost**.

Setting	Purpose, Usefulness, and Cost
Enable on-demand leak detection (previously called Enable on-demand memory leak detection)	Select to enable buttons in the GUI that let you send "leak" commands. This will show you a list of **unreachable memory blocks** for a time segment. Selecting is useful for checking for memory leaks in an application that never exits, or in only the portion of an application for which you are responsible. Use in conjunction with the **Reset Leak Tracking** and **Find Leaks** buttons during analysis. **Cost** is proportional to the number of allocations.
Guard zone size (previously called Guard zone byte size)	Available only if **Detect invalid memory accesses** and **Enable guard zones** are selected. Use in conjunction with **Enable guard zones** to set the number of bytes beyond the allocated block of memory the Intel Inspector reserves to identify **Invalid memory access** problems related to the allocation. Setting is useful when: • An application exhibits unexpected behavior. • You need more context about heap allocations to interpret **Invalid memory access** problems. **Cost** is proportional to number of allocations. Recommendation: Set unless: • Intel Inspector runs out of memory. • An application becomes destabilized. **NOTE** • May change application behavior. • Increases the amount of memory the Intel Inspector uses. • Intel Inspector creates guard zones only at the end of allocated space, not at the start of allocated space.
Maximum number of leaks shown in result (previously called Maximum memory leaks)	Use to set the maximum number of leaks the Intel Inspector shows in a result after analysis is complete. A zero setting shows all detected memory leaks. **Cost** is proportional to the number of leaks. Recommendation: Use the default value unless you want an exhaustive list of all leaks. **Tip** Even the default value can generate an unmanageable number of leaks. Consider sorting the displayed memory leaks by **Object Size**, fixing the largest leaks, and then re-inspecting your application. Or use the on-demand leak detection feature to narrow your focus and eat the elephant one bite at a time.
Remove duplicates	Deselect to show all occurrences of a detected problem in the **Code Locations** pane.

Setting	Purpose, Usefulness, and Cost
	Deselecting is: • Useful when you need to fully visualize all threads and problem occurrences in relation to time • Low **cost** in terms of time; however, the number of duplicate errors could crowd out the number of unique errors. Recommendation: Select.
Revert to previous uninitialized memory algorithm (not recommended)	Available only if **Detect uninitialized memory reads** is selected. The current algorithm for detecting uninitialized memory reads decreases false positives but increases analysis time and memory overhead. Select to use the previous version of the algorithm. Recommendation: Deselect.
Stack frame depth	Use to provide more or less call stack context for detected errors. A high setting is useful when analyzing highly object-oriented applications. A higher number does not significantly impact **cost**. Recommendation: Use only as large a value as an application requires to display complete call paths.

See Also

Configure Analysis
Investigate Problems Using Interactive Debugging
Memory Error Analysis Types
Intel Inspector Filenames and Locations

Pane: Analysis Type-Threading Errors

Before Analysis

(One way) To access this Intel Inspector pane:

- From the Visual Studio* menu, choose **Tools> Intel Inspector*version*> New Analysis...** . Then choose **Threading Error Analysis** from the Analysis Type drop-down list.

  NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- From the Standalone Intel Inspector GUI menu, choose **File** > **New** > **Analysis...**. Then choose **Threading Error Analysis** from the Analysis Type drop-down list.

Use this pane on the **Analysis Type** window to:

- Choose and, if necessary, fine-tune a preset threading error analysis type.
- Configure a threading error analysis to investigate problems in an interactive debugging session.

Tip
If the combination of analysis type settings in the preset threading error analysis types does not meet your needs at all, try creating a custom threading error analysis type.

During Analysis

To access this pane: Click the **Analysis Type** button on the **Navigation** toolbar.

Use this to review the analysis type settings for this analysis run.

After Analysis Is Complete

To access this pane: Click the **Analysis Type** button on the **Navigation** toolbar.

Use this pane to:

- Review the analysis type settings for this analysis run.
- Re-inspect - run another analysis using the same analysis type settings.

Use This	To Do This
Analysis Type drop-down list	Switch to another category of analysis types.
Configuration slider	Choose a preset analysis type (drag slider).
Analysis Time Overhead gauge 	Quickly estimate the time it may take to collect a result using various preset analysis types. Time is expressed in relation to normal application execution time. For example, 2x - 20x is 2 to 20 times longer than normal application execution time. If normal application execution time is 5 seconds, estimated collection time is 10 to 100 seconds.
Memory Overhead gauge 	The **Memory Overhead** gauge helps you quickly estimate the memory the Intel Inspector may consume to detect errors using this preset analysis type. Memory is expressed in blue-filled bars. **NOTE** The gauge does not show memory used by the running application during analysis.
Copy button	Create a new custom analysis type based on the currently selected analysis type.
Analyze without debugger radio button	Select to run an analysis without launching an interactive debugging session. Useful for investigating all types of memory and threading problems. **Tip** You can later use the **Debug This Problem** function from within a result to launch a new analysis in conjunction with a debugger to stop at problems of interest. The rerun analysis is automatically focused to find the selected problems, making it return to the problems more quickly. The **Debug This Problem** function is the recommended method for investigating threading errors in an interactive debugging session.

Use This	To Do This
Enable debugger when problem detected radio button	Select to allow investigation of every problem detected in an interactive debugging session. Useful for investigating all types of memory problems except memory and resource leaks.
Select analysis start location with debugger radio button	Select to allow investigation of problems in a particular area of code during an interactive debugging session. Typical scenario: You need to check for errors in a specific section of application code, but the Intel Inspector does not provide the appropriate granularity to analyze only that code section. This option quickly takes you near that execution point in a debugging session. Useful for investigating all types of memory problems except memory and resource leaks. **Tip** You can also use this option to investigate all types of threading errors, but it may be quicker to use the **Debug This Problem** function in a result if you plan to investigate a single threading problem.

Configurable Threading Error Analysis Type Settings

The following list shows the configurable settings (in alphabetical order) for preset threading analysis types. *Configurable* means you can change the setting without creating a custom analysis type.

- **Remove duplicates** checkbox (configurable for the **Locate Deadlocks and Data Races** analysis type only)
- **Scope** drop-down list (configurable for the **Locate Deadlocks and Data Races** analysis type only)
- **Stack frame depth** drop-down list
- **Terminate on deadlock** checkbox
- **Use maximum resources** checkbox (configurable for the **Locate Deadlocks and Data Races** analysis type only)

Threading Error Analysis Type Settings

Use the **Details** region to review current analysis type settings. The following table describes the purpose, usefulness, and *cost* (low, medium, high, or proportional in terms of time and resources) for each threading error analysis type configuration setting. (The settings are listed in alphabetical order.)

Setting	Purpose, Usefulness, and Cost
Cross-thread stack access detection	Use to set the alert mechanism for when a thread accesses stack memory of another thread. The alert mechanism helps you decide if this is an issue that requires handling. All options are low **cost** if **Detect data races** is selected. Recommendation: • Use **Hide problems/Hide warnings** if using an OpenMP* or oneAPI Threading Building Blocks (oneTBB) programming model; or if cross-thread stack accesses are anticipated. Also select **Detect races on stack**. • Use **Hide problems/Show warnings** if cross-thread stack accesses are not anticipated. Also deselect **Detect data races on stack**. • Use **Show problems/Hide warnings** if cross-thread stack accesses are not anticipated but a previous analysis indicated they exist and you are not using an OpenMP* or oneAPI Threading Building Blocks (oneTBB) programming model. Also deselect **Detect data races on stack**.

Setting	Purpose, Usefulness, and Cost
Detect data races	Select to detect problems where multiple threads access the same memory location without proper synchronization and at least one access is a write. Selecting is useful when you suspect data races that are not yet evident. High **cost**. Recommendation: Select. Consider also deselecting **Use maximum resources** to reduce cost.
Detect data races on stack (previously called Detect data races on stack accesses)	Available only if **Detect data races** is selected. Select to detect data races for variables allocated on the stack. Selecting is useful when threads in an application share variables from the stack and you suspect data races on the variables. High **cost**. Recommendation: Deselect. If you select, consider also deselecting **Use maximum resources** to reduce cost.
Detect deadlocks	Select to detect problems where two or more threads are waiting for the other to release resources, but none of the threads releases the resources. Thus no thread can proceed. Selecting is useful when you want to troubleshoot the location of a deadlock. Low **cost**.
Detect lock hierarchy violations	Select to detect problems where the acquisition hierarchy order of multiple synchronization objects in one thread differs from the acquisition hierarchy order in another thread, and could cause a deadlock under certain conditions. Selecting is useful when an application has complicated synchronization and it is hard to verify correctness. Low **cost** unless an application has a significant number of locks.
Filter guaranteed atomics	Available only if **Detect data races** is selected. Select to ignore data races on guaranteed atomic operations on the Intel® P6 processor family or newer. For details, please refer to the *Guaranteed Atomic Operations* section of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. Selecting is useful if you observe many false data race reports on simple operations like Load or Store to shared variables and the size of those variables is less than the processor cache line size. Do not select this option if you develop cross-platform code that should work on other architectures. Selecting this option might also hide true-races on variables that were properly aligned during this run but might not be aligned in general, e.g., if it just works "by chance". Low **cost**. Recommendations: Use this option with caution only if you observe many false reports on simple memory operations.

Setting	Purpose, Usefulness, and Cost
Race analysis byte granularity (previously called Memory access byte granularity)	Available only if **Detect data races** is selected. Use to set the size of the smallest memory block the Intel Inspector considers a single block of memory when determining if non-synchronized accesses to a memory block constitute a data race. Selecting is useful to control memory consumption during analysis for some applications. High **cost** when set to 1 byte. Recommendation: Set to 4 unless you continually see data races based on safe access to smaller memory blocks. If so, reset to 1.
Remove duplicates	Deselect to show all occurrences of a detected problem in the **Code Locations** pane. Deselecting is: • Useful when you need to fully visualize all threads and problem occurrences in relation to time • Low **cost** in terms of time; however, the number of duplicate errors could crowd out the number of unique errors. Recommendation: Select.
Save stack on first access	Available only if **Detect data races** is selected. Select to show as much information as possible on all threads involved in a data race. Selecting is useful when investigating complex data race problems. High **cost**. Recommendation: Deselect on initial analysis runs. Select only when you need the maximum information and context about all threads involved in a data race to solve the problem.
Save stack on lock creation	Select to show creation information on synchronization objects involved in deadlocks, lock hierarchy violations, and data races. Selecting is useful when acquisition stacks are not sufficient to understand the problem. Low **cost**.
Save stack on memory allocation (previously called Save stack on allocation)	Available only if **Detect data races** is selected. Select to identify the allocation site of dynamically allocated memory objects involved in data races. Medium **cost**. Recommendation: Select when you need to identify the object hierarchy of low-level objects involved in data races. For example: If object R is involved in a data race and is instantiated within objects O1, O2, and O3, the allocation call stack can help you identify which encapsulating object is not properly protecting access to object R.
Stack frame depth	Use to provide more or less call stack context for detected errors. A high setting is useful when analyzing highly object-oriented applications.

Setting	Purpose, Usefulness, and Cost
	A higher number does not significantly impact **cost** with one exception: Choosing a higher number plus selecting **Save stack on first access** increases **cost**. Recommendation: Use only as large a value as an application requires to display complete call paths.
Terminate on deadlock	Available only if **Detect deadlocks** is selected. Select to stop analysis and application execution if the Intel Inspector detects a deadlock. Selecting is useful when running your application as part of a kernel or unit testing suite. Low **cost**. Recommendation: Deselect. Instead, use the corresponding knob in the command line interface to perform kernel or unit testing in a nightly scenario. If the Intel Inspector identifies a deadlock, decide if it is appropriate to continue analysis.
Use maximum resources	Select to potentially find more problems. High **cost**. Recommendation: Deselect to run a quicker analysis that should find most of your data race and cross-thread stack access problems. Once you have found and fixed these problems, select to get more complete analysis coverage of possible data race and cross-thread stack access problems.

See Also
Configure Analysis
Investigate Problems Using Interactive Debugging
Threading Error Analysis Types
Intel Inspector Filenames and Locations

Pane: Application Output

To access this pane: The Intel Inspector displays this pane on the **Collection Log** window during analysis.

If you set the Intel Inspector to send application output to the Collection Log window, use this pane to view that output during analysis.

See Also
Examine Result Data During Analysis
Startup Tasks

Pane: Code and Stack

To access this Intel Inspector pane: In the **Summary** window, double-click result data to display the **Sources** window. Use this pane on the **Sources** window to:

- Explore the relationship among code locations for a problem occurrence.
- Examine a snapshot of source code and low-level operations at the time of analysis.
- Edit source code to resolve issues.

Use This	To Do This
Code location region title	Review code location summary and thread information.
Code location region control	Collapse or expand the code location region (click  or ).
Source tab	• Explore source code surrounding the code location. (The blue highlight marks the code location source line.) • Edit source code in your default editor (double-click).
Disassembly tab	Examine the low-level operations involved in the problem.
Source tab context menu	• **Edit Source** - Edit source code in your default editor. • **Copy to Clipboard** - Copy the selected code to the system clipboard. • **Explain Problem** - Explain, in generic terms, the error associated with the code.
Disassembly tab context menu	• **Copy to Clipboard** - Copy the selected code to the system clipboard. • **Explain Problem** - Explain, in generic terms, the error associated with the code.
Call Stack tab	Display call stack frame code in the **Source** tab (click).
Call Stack tab context menu	• **Show Module Names**, **Show Sources**, **Show Two Lines**, and **Show Horizontal Scrollbar** - Customize frame presentation. • **Copy to Clipboard** - Copy the frames to the system clipboard.
Tab border	Resize the pane (drag).

See Also

Interpret Result Data and Resolve Issues

Pane: Code Locations

To access this Intel Inspector pane: Click the **Summary** button on the **Navigation** toolbar.

During Analysis

Use this pane on the **Summary** window to:

- View code locations for selected problems.
- Choose problem occurrences of interest to display in the **Sources** window.
- Edit corresponding source code in your default editor.

After Analysis is Complete

The Intel Inspector displays this pane after analysis is complete and when you open a result. Use this pane on the **Summary** window to:

- View code locations for selected problems.
- Choose problems or problem occurrences of interest to display in the **Sources** window.
- Edit corresponding source code in your default editor.
- Export result data in plain text format.
- Define suppression rules and mark all data potentially impacted by the rules during future analysis runs.

- Delete suppression rules applied to marked data.

Tip
The number of problem occurrences available for display in this pane depends on how you configure the analysis. For future analysis runs:

- Deselect the **Remove duplicates** checkbox on the **Analysis Type** window to report all occurrences of a detected problem. Deselecting is:
  - Useful when you need to fully visualize all threads and problem occurrences in relation to time
  - Low cost in terms of performance; however, the number of detected problem occurrences could be significantly higher and may require more system resources for storage and processing
- Select the **Remove duplicates** checkbox to report only a representative occurrence of each detected problem.

Use This	To Do This
Pane title	Verify the type of the selected problems.

Pane controls	• Review code location, surrounding source code snippet, call stack, and timeline information for all code locations in:
	  • An occurrence of the selected problems (move the slider control, click  or  , or press Ctrl+Left Arrow or Ctrl+Right Arrow).
	  • All occurrences of the selected problems (click the **All** button to toggle on and off).
	• Hide/show this pane (click  or ).

Code location region	• Review summary information for a code location in a problem occurrence. • Review a source code snippet surrounding a code location. (The yellow highlight marks the code location source line.) • Review the call stack for a problem occurrence. • Identify the thread associated with a code location (click to color the code location marker blue in the **Timeline** pane). • Choose a problem or problem occurrence of interest to display in the **Sources** window (double-click).
Data column header	• Reposition the data column (drag). • Resize the data column (drag left or right border). • Sort code locations in ascending or descending order by column data (click). (Enabled only after analysis is complete.)
Pane border	Resize the pane (drag).
Context menu	• **View Source** - Display a **Sources** window focused on the selected problem occurrence. (Default action when you double-click a data row in the **Code Locations** pane.) • **Edit Source** - Open the source file for the selected code location in your default editor. • **Copy to Clipboard** - Copy a summary of the selected code location to the system clipboard.

Use This	To Do This

	• **Explain Problem** - Explain, in generic terms, the error associated with the code.
	• **Create Problem Report...** - Export details of the selected code location to plain text format. (Enabled only after analysis is complete.)
	• **Suppress...** or **Do Not Suppress...** -
	  • Create a rule to suppress problems associated with the selected code location during the next analysis run. (Intel Inspector marks - with strikethroughs - problems *potentially* impacted during the next analysis run.)
	  • Or delete a rule for a marked code location. (Strikethroughs disappear.)
	  (Enabled only after analysis is complete.)
	  NOTE
	  You can also delete suppression rules using the **Suppressions** tab of the **Project Properties** dialog box.
	• **Show Source Code Snippets** - Display or not display several source code lines surrounding each code location. (The yellow highlight marks the code location source line.)

See Also

Collaborate on Results
Configure Analysis
Configure Projects
Interpret Result Data and Resolve Issues

Pane: Collection Log

During Analysis

To access this pane: The Intel Inspector displays this pane on the **Collection Log** window during analysis. If you navigate away from the pane, click the **Collection Log** button on the **Navigation** toolbar to return.

Use this window to:

- Confirm the application is still executing (thread activity columns are resizable).
- Check memory consumption of the target application plus the Intel Inspector.
- Check for memory growth measurement requests and resets.
- View finalization progress.
- View suppression summary statistics.

After Analysis Is Complete

To access this pane: Click the **Collection Log** button on the **Navigation** toolbar.

Use this pane to ensure all key analysis milestones completed successfully.

See Also

Examine Result Data During Analysis

Pane: Collector Messages

During Analysis

To access this pane: The Intel Inspector displays this pane on the **Collection Log** window during analysis.

Use this pane to review analysis informational, warning, and error messages.

After Analysis Is Complete

To access this pane: Click the **Collection Log** button on the **Navigation** toolbar.

Use this pane to review analysis informational, warning, and error messages.

See Also

Examine Result Data During Analysis

Pane: Compare Results

(One way) To access this Intel Inspector pane:

- From the Visual Studio* menu, choose **Tools> Intel Inspector*version*> Compare...** .

> **NOTE** In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- From the Standalone Intel Inspector GUI menu, choose **File** > **Open** > **Results Comparison...**.

Use this pane on the **Compare Results** window to compare two results; and identify issues that exist in one but not the other, or that still exist in both.

Use This	To Do This
Result fields, field arrows, and **Browse** buttons	Choose two results to compare.

Pane: Filters

To access this pane: The Intel Inspector displays this pane after analysis is complete and when you open an existing result. Use this pane on the **Summary** window to focus on those issues that require your attention by temporarily limiting the items displayed in the **Problems** pane.

Use This	To Do This
Pane controls	• Sort all filter criteria by name in ascending alphabetical order or by count in descending numerical order (click **Sort** drop-down list).

Use This	To Do This
	NOTE You cannot change the order in which filter categories are presented. • Deselect all filter criteria (click the  button). • Select filter criteria (click filter criterion). • Deselect filter criteria for this category (click the **All** button). • Show/hide this pane (click  and ).
Category header	Review categories for which you can apply a filter criterion, such as **Type** and **State**. **NOTE** You can apply only one criterion per category; however, you can work multiple categories simultaneously.
N items	Identify the number of problem sets that match this criterion.
Pane border	Resize the pane (drag).
Context menu	• **Refine Source File Set...** - Temporarily limit displayed items to those from a set of source files. • **Clear Source File Filter** - Deselect the source file filter criterion. • **Explain Problem** - Explain, in generic terms, the error associated with the problem type. (Displayed only for criteria in the **Type** category).

See Also

Choose Problems

Pane: Import Result

To access this Intel Inspector pane :

- From Visual Studio* menu, choose **Tools > Intel Inspector*version* > Import...**.

NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- From the Standalone Intel Inspector GUI menu, choose **File** > **Import Result...**.

Use this pane on the **Import Result** window to import the following into the current project:

- Intel Inspector result archive files
- Intel Inspector results not associated with a project
- Results from other Intel© error-detection products

Tip
Refinalize the imported result to apply current project symbol information. (Use the **Re-resolve** option on the **Solution Explorer** or **Project Navigator** pane context menu.)

Use This	To Do This
Result field, field arrow, and **Browse** button	Choose a file to import.

See Also

Configure Projects
Export and Import Files
Startup Tasks
Intel Inspector Filenames and Locations

Pane: Launch Application

During Analysis

To access this Intel Inspector pane: Click the **Target** button on the **Navigation** toolbar.

Use this pane on the **Target** window to review the project properties for this analysis run.

After Analysis Is Complete

To access this pane: Click the **Target** button on the **Navigation** toolbar.

Use this pane on the **Target** window to:

- Review the project properties for this analysis run.
- Re-inspect - run another analysis using the same analysis type as that in the current result.

Use This	To Do This
Inherit settings from Visual Studio* project checkbox	Determine if all basic project properties came from Visual Studio* property pages of the StartUp project (**Configuration Properties** > **Debugging**).
Microsoft* runtime environment drop-down list	Determine if any source code was excluded from inspection depending on whether it was native or managed code. **NOTE** Microsoft .NET* 3.5 software support is deprecated in the Intel Inspector and will be removed after August, 2020. This deprecation does not apply to the Intel® VTune™ Profiler. See *Release Notes* for details.
Child application field	Identify the actual inspected file (if different from the application identified in the **Application** field).

Use This	To Do This
Suppressions radio buttons	Determine if suppression rules in `.sup` files were applied.
Modules radio buttons, **Modify** button, and field	Determine if a specific application module(s) was included in or excluded from inspection.

See Also

Configure Projects

Pane: Problem Details

To display this Intel Inspector pane, run an Intel Inspector analysis configured to use interactive debugging.

Use this pane to:

- View details of problems detected by the Intel Inspector that resulted in breakpoints. This is the same data found in the Intel Inspector result, duplicated within the debugger workspace for convenient referencing.
- Disable and re-enable problem breakpoints.

Use This	To Do This
Source button	Open the corresponding source file in the Visual Studio* editor at the line indicated in the problem description (click).
Intel Inspector button	Change focus to the corresponding Intel Inspector result **Sources** window (click).
Disable Breakpoint button	Not break again for this problem type at this location (click).
Re-enable Breakpoints button	• View disabled problem breakpoints. • Re-enable selected breakpoints (click, then select in **Disabled Problem Breakpoints** dialog box).
Explain Problem button	Explain, in generic terms, the error associated with the selection (click).
[problem] at [memory location] for [thread]	Review summary information for the detected problem.
Data row(s)	• Review code locations in the detected problem. • Open the corresponding source file in the Visual Studio* editor at the line indicated in the problem description (double-click).
Context menu	• **Source** - Open the corresponding source file in the Visual Studio* editor at the line indicated in the problem description. • **Intel Inspector** - Change focus to the corresponding Intel Inspector result **Sources** window. • **Copy to Clipboard** - Copy the selected code location to the system clipboard. • **Explain Problem** - Explain, in generic terms, the error associated with the code.

See Also

Investigate Problems Using Interactive Debugging

Pane: Problems

To access this Intel Inspector pane: Click the **Summary** button on the **Navigation** toolbar.

During Analysis

Use this pane on the **Summary** window to:

- View problems displayed in the order detected.
- Choose problems of interest to display in the **Sources** window.
- Edit corresponding source code in your default editor.

After Analysis Is Complete

Intel Inspector displays this pane after analysis is complete and when you open an existing result. Use this pane on the **Summary** window to:

- View prioritized problems.
- Choose problems of interest to display in the **Sources** window.
- Edit corresponding source code in your default editor.
- Change problem state to help you focus on only those problems that require your attention during this and future analysis runs.
- Report result data in plain text format.
- Merge state information from a result with an overlapping set of problems into the currently open result.
- Launch a new analysis in conjunction with a debugger to stop at problems of interest.

Use This	To Do This
Pane controls	Hide/show adjacent panes (click  ,  , or).

Data rows	• During analysis:
	  • Review summary information for problems displayed in the order detected.
	  • Show code location information for selected problems (click).
	  • Choose problems of interest to display in the **Sources** window (double-click).
	• After analysis is complete:
	  • Review summary information for prioritized sets of detected problems.
	  • Review summary information for each detected problem in a problem set (click ) .
	  • Show code location information for an occurrence of the selected problems (click).
	  • Choose problems of interest to display in the **Sources** window (double-click).

	Quickly determine problem severity.

Use This	To Do This
,  ,  , and  icons	
,  , and  icons (after analysis is complete)	Quickly determine problem state.
Data column header	• Reposition the data column (drag). • Resize the data column (drag left or right border). • Sort grid data in ascending or descending order by column data (click). (Enabled only after analysis is complete.) **NOTE** Multiple file names in the **Sources** column are presented in alphabetical order. Sorting on this data may be unproductive.
Pane border	Resize the pane (drag).
Context menu	• **View Source** - Display a **Sources** window focused on the selected problem(s) (default action when you double-click a data row in the **Problems** pane). • **Edit Source** - Open the source file for the selected problem(s) in your default editor. • **Copy to Clipboard** - Copy a summary of the selected problem(s) to the system clipboard. • **Explain Problem** - Explain, in generic terms, the error associated with the code. • **Create Problem Report...** - Export details of the selected problem(s) to plain text format. (Enabled only after analysis is complete.) • **Debug This Problem** - Launch a new analysis in conjunction with a debugger to stop at problems of interest. (Visual Studio* IDE only). (Enabled only after analysis is complete.) • **Change State** - Change the state of the selected problem(s) to **Not Fixed**, **Confirmed**, **Fixed**, **Not a problem**, or **Deferred**. (Enabled only after analysis is complete.) • **Merge States...** - Merge state information from another result; update in this result the state assigned to any problems that appear in both results. (Enabled only after analysis is complete.)

See Also

Choose Problems
Collaborate on Results
Investigate Problems Using Interactive Debugging
Severity Levels
States

Pane: Project Navigator

(One way) To access this Intel Inspector pane: Click the  icon on the Intel Inspector toolbar. Use this pane to:

- See a hierarchical view of your projects and results based on the directory where the opened project resides.
- Perform functions available from the menu and toolbar plus:
  - Delete a selected project or result.
  - Rename a selected project or result.
  - Close all opened results.
  - Copy various directory paths to the system clipboard.
  - Refinalize (re-resolve) - and open if not already opened - a result using a more complete set of symbol information.

Use This	To Do This
Menu or toolbar commands	Populate the **Project Navigator** project hierarchy based on the directory where the currently opened project resides.
Title bar	Move the **Project Navigator** (drag). Dock the **Project Navigator** (drag to a window edge).
Directory context menu	• **New Project...** - Open the **Create a Project** dialog box, where you can browse to or create a directory in which the Intel Inspector will create an Intel Inspector project (`config.inspxeproj`). • **Open Project From New Location** - Open the **Select Project** dialog box, where you can browse to a directory containing Intel Inspector projects. • **Copy Path to Clipboard** - Copy this directory path to the system clipboard.
Project	Open the project (double-click). **NOTE** Opening a project closes the currently opened project.

Project context menu	• **Open Project** - Open the Intel Inspector project.
	• **Close Project** - Close the current project and any opened results.
	• **New Analysis...** - Open the **Analysis Type** window, where you can:
	  • Choose a preset memory or threading error analysis type.
	  • Configure a custom analysis type.
	• **<Recent Analysis Type>** - Rerun a recent analysis.
	• **Close All Results** - Close all opened results for this project.
	• **Delete Project** - Immediately delete the selected project (and associated results) from the **Project Navigator** and file system.
	• **Rename Project** - Rename the selected project:
	  • Immediately in the **Project Navigator**
	  • In the file system after you close the project or exit the Intel Inspector
	NOTE
	The corresponding project directory on your file system is not renamed.

Use This	To Do This
	• **Copy Project Path to Clipboard** - Copy the file path for this project to the system clipboard. • **Project Properties...** - Open the **Project Properties** dialog box, where you can review or change current project properties.
Result	Open the result (double-click). **NOTE** Opening a result opens the associated project if it is not already open.

Result context menu	• **Open Result** - Open the Intel Inspector result.
	• **Export Result...** - Create an easy-to-share result archive file.
	• **Re-inspect** - Run a new analysis using the same analysis type as that in the selected result.
	• **Re-resolve [and Open]** - Refinalize (and open if not already opened) the result using a more complete set of symbol information.
	
	NOTE
	• When you run an analysis using the GUI, finalization occurs after the result is generated and when the result is first opened in the standalone GUI or in the Microsoft Visual Studio* IDE.
	• Intel Inspector retrieves symbol information using the directories identified in the **Binary/Symbol Search** and **Source Search** tabs of the **Project Properties** dialog box as well as default search locations and algorithms.
	• Refinalizing may change problem set, code location, and stack information because the Intel Inspector discards previously resolved symbol information and performs symbol resolution again.
	
	• **Delete Result** - Immediately delete the selected result from the **Project Navigator** and file system.
	• **Rename Result** - Rename the selected result:
	  • Immediately in the **Project Navigator**
	  • In the file system after you close the result or project, or exit the Intel Inspector
	
	NOTE
	The corresponding result directory in the file system is not renamed.
	
	• **Copy Result Path to Clipboard** - Copy the file path for this result to the system clipboard.

See Also

Choose Projects
Configure Analysis
Configure Projects
Run Analysis
View Results
Binary/Symbol Search and Source Search Locations

Pane: Timeline

To access this Intel Inspector pane: Click the **Summary** button on the **Navigation** toolbar. Use this pane on the **Summary** window to graphically visualize the relationship between threads and the code locations for the problem occurrences displayed in the **Code Locations** pane.

Use This	To Do This
Pane controls	• Show code location information in text form (hover the mouse pointer over a ◇ or ◆ marker). **Tip** There is a marker for every code location in the problem occurrence displayed in the **Code Locations** pane. If a marker is hidden behind another, the Intel Inspector shows text information for both the shown and hidden code locations. • Show/hide this pane (click  or ).

Thread (blue) bars	• Review the number of threads used during application execution.
	• Review relative thread duration.
	• Distinguish a thread with a code location in the **Code Locations** pane (thick bar) from a thread without such a code location (thin bar).
	• Identify the name of a thread with a code location in the **Code Locations** pane. Intel Inspector tries to provide meaningful thread names in the following forms:
	  • Thread ID
	  • And thread name or function called to start the thread

◆ marker	Identify the thread with the code location highlighted in the **Code Locations** pane.
◇ marker(s)	Identify the thread with code locations not highlighted in the **Code Locations** pane.
Pane border	Resize the pane (drag).
Context menu	**View Source** - Display a **Sources** window focused on the selected problem occurrence.

See Also

Interpret Result Data and Resolve Issues

Toolbar: Command

Use to perform key Intel Inspector tasks.

Use This	On The Command Toolbar on This Window	To Do This
Close button	All	Stop inspecting the application for issues, finalize the result collected thus far, but close the new result tab. (You can open and display the result at a later date.)
Command Line... button	**Analysis Type** window	Open the **Corresponding inspxe-cl Command Options** dialog box, where you can review - and, if desired, copy to the clipboard - the command to perform the same analysis using the Intel Inspector command line interface (`inspxe-cl` command).
Compare button	**Compare** window	Start comparing two results; and identifying issues that exist in one but not the other, or that still exist in both.
Elapsed Time message	**Collection Log**, **Sources**, and **Summary** windows	Review analysis duration.
Find Leaks button	**Collection Log**, **Sources**, and **Summary** windows during memory error analysis	Search for new memory leaks since the analysis started (default) or since the last reset. **NOTE** • To enable this button: Select the **Enable on-demand memory leak detection** checkbox when configuring a memory error analysis. • To reset the start point for finding memory leaks: Click the **Reset Leak Tracking** button during analysis. • To review detected memory leaks: Click the **Summary** button.
Import button	**Import** window	Import result archive files, results not associated with a project, and results from other Intel® error-detection products into the current project. **Tip** Refinalize the imported result to apply current project symbol information. (Use the **Re-resolve** option on the **Solution Explorer** or **Project Navigator** pane context menu.)
Measure Growth button	**Collection Log**, **Sources**, and **Summary**	Measure memory growth since the analysis started (default) or since the last reset.

Use This	On The Command Toolbar on This Window	To Do This
	windows during memory error analysis	**NOTE** • To enable this button: Select the **Enable interactive memory growth detection** checkbox when configuring a memory error analysis. • To reset the start point for measuring memory growth: Click the **Reset Growth Tracking** button during analysis. • To review memory growth: Click the **Summary** button.
Project Properties... button	**Analysis Type** window	Open the **Project Properties** dialog box, where you can review or change current project properties.
Re-inspect button	**Target**, **Analysis Type**, and **Collection Log** windows	Run another analysis using the same analysis type as that in the current result.
Reset Growth Tracking button	**Collection Log**, **Sources**, and **Summary** windows during memory error analysis	Reset the start point for measuring memory growth. The default start point is the application start. **NOTE** • To enable this button: Select the **Enable interactive memory growth detection** checkbox when configuring a memory error analysis. • To measure memory growth: Click the **Measure Growth** button during analysis. • To review memory growth: Click the **Summary** button.
Reset Leak Tracking button	**Collection Log**, **Sources**, and **Summary** windows during memory error analysis	Reset the start point for finding memory leaks. The default start point is the application start. **Caution** When you click this button, the Intel Inspector discards earlier allocation data and tracks leaks only in new allocations. Any new memory leaks on memory allocated prior to the button click do not appear in the result. **NOTE** • To enable this button: Select the **Enable on-demand memory leak detection** checkbox when configuring a memory error analysis. • To find memory leaks in new allocations: Click the **Find Leaks** button during analysis. • To review detected memory leaks: Click the **Summary** button.

Use This	On The Command Toolbar on This Window	To Do This
Start button	**Analysis Type** window	Start executing and inspecting an application for issues, and collect those issues in a result.
Stop button	**Collection Log**, **Sources**, and **Summary** windows	Stop inspecting the application for issues, finalize the result collected thus far, open a new result tab, and display the result.

See Also

Command Syntax
inspxe-cl Actions, Options and Arguments
Compare Results
Configure Projects
Examine Result Data During Analysis
Run Analysis
Export and Import Files
Command Line Interface Support

Toolbar: Intel Inspector

Use to perform key Intel Inspector tasks.

Use This Icon		To Do This
Visual Studio* IDE	**Standalone GUI**	
		Open the **Analysis Type** window for a new or recent analysis type, where you can: • Choose and, if necessary, fine-tune a preset or custom analysis type. • Start the process of creating a custom analysis type if the combination of analysis type settings in the preset analysis types does not meet your needs. • Configure an analysis to investigate issues in an interactive debugging session.

		Open the **Project Navigator**, where you can:
		• See a hierarchical view of your projects and results based on the directory where the opened project resides.
		• Perform functions available from the menu and toolbar plus:
		  • Delete a selected project or result.
		  • Rename a selected project or result.
		  • Close all opened results.
		  • Copy various directory paths to the system clipboard.

		Open the **Create a Project** dialog box, where you can browse to or create a directory in which the Intel Inspector will create an Intel Inspector project (`config.inspxeproj`).
		Open the **Select Project** dialog box, where you can browse to and choose a project (`config.inspxeproj`).

Use This Icon		To Do This
Visual Studio* IDE	**Standalone GUI**	
		Open the **Project Properties** dialog box, where you can review or change project properties.
		Open the **Analysis Type** window, where you can: • Choose and, if necessary, fine-tune a preset or custom analysis type. • Start the process of creating a custom analysis type if the combination of analysis type settings in the preset analysis types does not meet your needs. • Configure an analysis to investigate issues in an interactive debugging session.
		Open the **Select Result** dialog box, where you can browse to and choose a result (`*.inspxe`).
		Open the **Compare Results** window, where you can compare two results to identify issues that exist in one but not the other, or that exist in both.
		Access resources such as *Help*, *Getting Started* documentation, videos, and articles.

NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

See Also

Choosing Projects
Comparing Results
Configuring Results
Running Analyses
About Viewing Results
Intel Inspector Filenames and Locations

Toolbar: Navigation

Use to navigate among windows where you can perform key Intel Inspector tasks.

Use This	Before/During/After Analysis Is Complete	To Do This
Analysis Type button	Before analysis	Open the **Analysis Type** window, where you can: • Choose and, if necessary, fine-tune a preset analysis type.

Use This	Before/During/After Analysis Is Complete	To Do This
		• Start the process of creating a custom analysis type if the combination of analysis type settings in the preset analysis types does not meet your needs. • Configure an analysis to investigate issues in an interactive debugging session.
	During and after analysis is complete	Open a read-only **Analysis Type** window, where you can review the analysis type settings for this analysis run.
Collection Log button	During analysis	Open the **Collection Log** window, where you can: • Confirm the application is still executing (thread activity columns are resizable). • Check analysis duration. • Check memory consumption of the target application plus the Intel Inspector. • View finalization progress. • Track analysis milestones and address analysis warnings. • Measure memory growth during a specific time period. • Search for new memory leaks during a specific time period. • Stop inspecting the application for issues, finalize the result collected thus far, open a new result tab, and display the result. • Stop inspecting the application for issues, finalize the result collected thus far, but close the new result tab. (You can open and display the result at a later date.)
	After analysis is complete	Open a read-only **Collection Log** window, where you can review analysis informational, warning, and error messages.
Summary button	During analysis	Open the **Summary** window, where you can view problems displayed in the order detected and choose problems of interest to display in the **Sources** window.
	After analysis is complete	Open the **Summary** window, where you can: • View prioritized problems and choose problems of interest to display in the **Sources** window. • Temporarily limit the problem list to only those items that meet specific criteria. • Graphically visualize the relationship between threads and code locations. • Change problem state and suppress problems to help you focus on only those problems that require your attention during this and future analysis runs. • Report result data in plain text format. • Merge state information from a result with an overlapping set of problems.

Use This	Before/During/After Analysis Is Complete	To Do This
		• Launch a new analysis in conjunction with a debugger to stop at problems of interest.
Target button	During and after analysis is complete	Open a read-only **Target** window, where you can review the project properties for this analysis run.

See Also

Workflows-Quick Paths to Maximizing Productivity

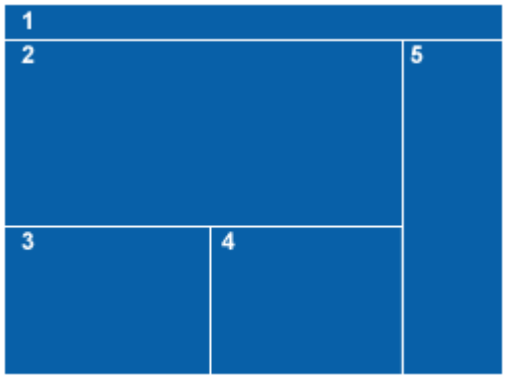
Window: Collection Log

During Analysis

The Intel Inspector**Collection Log** window shows analysis progress indicators and analysis milestones, such as when application execution ends, finalization begins and ends, and analysis is complete. It also shows analysis warnings and suggestions on how to address those warnings. Use this window to:

- Confirm the application is still executing (thread activity columns are resizable).
- Check analysis duration.
- Check memory consumption of the target application plus the Intel Inspector.
- View finalization progress.
- Track analysis milestones and address analysis warnings.
- View suppression summary statistics.
- Measure memory growth during a specific time period.
- Search for new memory leaks during a specific time period.
- Stop inspecting the application for issues, finalize the result collected thus far, open a new result tab, and display the result.
- Stop inspecting the application for issues, finalize the result collected thus far, but close the new result tab. (You can open and display the result at a later date.)

Window Layout



Window Panes and Toolbars

1. Toolbar: Navigation
2. Pane: Collection Log
3. Pane: Application Output
4. Pane: Collector Messages
5. Toolbar: Command

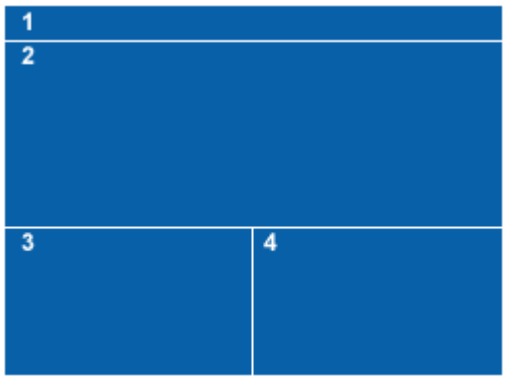
After Analysis is Complete

To access this window: Click the **Collection Log** button on the **Navigation** toolbar.

Use this window to:

- Review analysis informational, warning, and error messages.
- Re-inspect - run another analysis using the same analysis type as that in the current result.

Window Layout



Window Panes and Toolbars

1. Toolbar: Navigation
2. Pane: Collection Log
3. Pane: Application Output
4. Pane: Collector Messages

See Also

Examine Result Data During Analysis

Window: Compare Results

(One way) To access this Intel Inspector window:

- From the Visual Studio* menu, choose **Tools> Intel Inspector*version*> Compare...**.

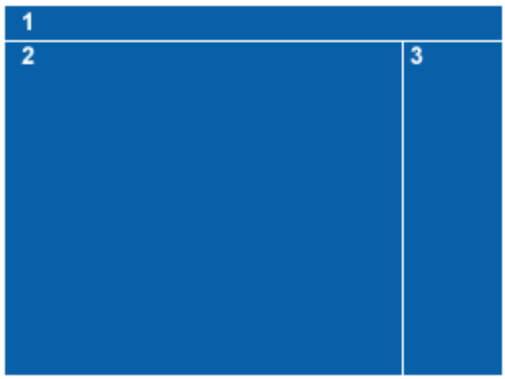
> **NOTE** In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- From the Standalone Intel Inspector GUI menu, choose **File** > **Open** > **Results Comparison...**.

Use this window to:

- Choose two results of the same analysis type.
- Compare them to identify issues that exist in one but not the other, or that exist in both.

Window Layout



Window Panes and Toolbars

1. Toolbar: Navigation
2. Pane: Compare Results
3. Toolbar: Command

See Also

Compare Results

Window: Import Result

To access this Intel Inspector window:

- Visual Studio* menu, choose **Tools > Intel Inspector*version* > Import...**.

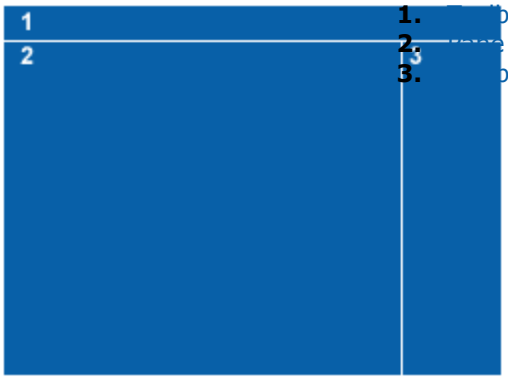
> **NOTE** In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

- Standalone Intel Inspector GUI menu, choose **File** > **Import Result...**.

Use this window to import the following into the current project:

- Intel Inspector result archive files
- Intel Inspector results not associated with a project
- Results from other Intel® error-detection products

> **Tip**
> Refinalize the imported result to apply current project symbol information. (Use the **Re-resolve** option on the **Solution Explorer** or **Project Navigator** pane context menu.)

Window Layout	Window Panes and Toolbars
	**1.** bar: Navigation **2.** : Import Result **3.** bar: Command

See Also

Export and Import Files

Window: Sources

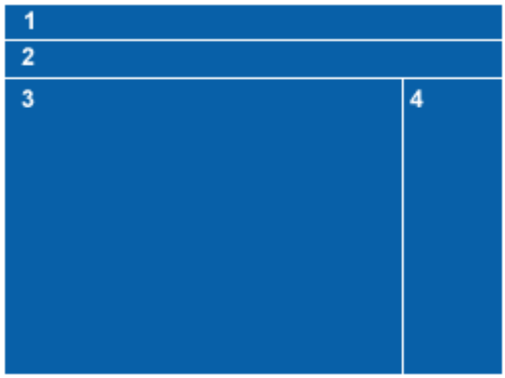
To access this Intel Inspector window: In the **Summary** window, double-click result data.

The **Sources** window shows a snapshot of source code for code locations and parent code locations associated with a detected problem. It also provides access to generic explanations of the errors associated with detected problems and corresponding source code in your default editor. Use this window to:

- Explore the relationship among code locations associated with a problem.
- Examine a snapshot of the relevant source code.
- Start resolving issues.

During Analysis

Window Layout	Window Panes and Toolbars
	1. Problem type **2.** Toolbar: Navigation **3.** Pane: Code and Stack containing one or more expanded or collapsed code location regions **4.** Toolbar: Command



After Analysis is Complete

Window Layout

Window Panes and Toolbars

1. Problem type
2. Toolbar: Navigation
3. Pane: Code and Stack containing one or more expanded or collapsed code location regions

See Also

Examine Result Data During Analysis
Interpret Result Data and Resolve Issues
View Results

Window: Summary After Analysis Is Complete

Intel Inspector displays this window after analysis is complete and when you open an existing result.

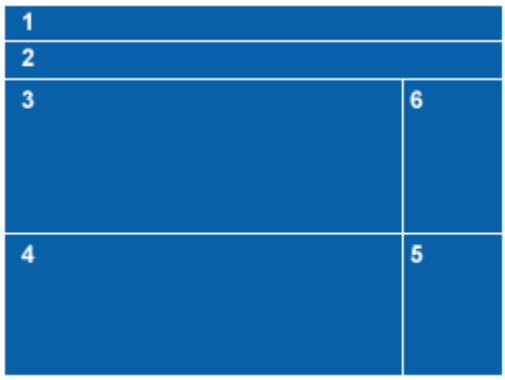
The **Summary** window shows application problems detected during analysis, problem severity, and associated code locations. It also provides access to generic error explanations and corresponding source code in your default editor. Use this window to:

- View prioritized problems and choose problems of interest to display in the **Sources** window.
- Temporarily limit the problem list to only those items that meet specific criteria.
- Graphically visualize the relationship between threads and code locations.
- Change problem state and suppress problems to help you focus on only those problems that require your attention during this and future analysis runs.
- Report result data in plain text format.
- Merge state information from a result with an overlapping set of problems.
- Launch a new analysis in conjunction with a debugger to stop at problems of interest.

Window Layout

Window Panes and Toolbars

1. Analysis Type



2. Toolbar: Navigation
3. Pane: Problems
4. Pane: Code Locations
5. Pane: Timeline
6. Pane: Filters

See Also

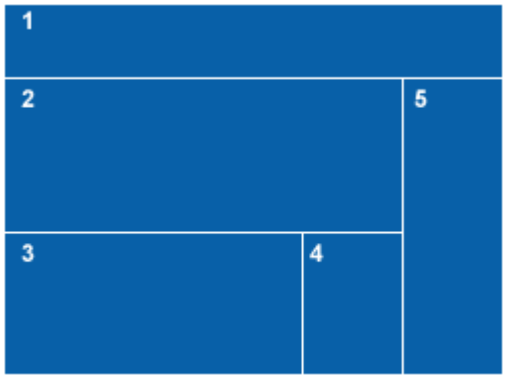
Choose Problems
Collaborate on Results
Interpret Result Data and Resolve Issues
Investigate Problems Using Interactive Debugging

Window: Summary During Analysis

To access this Intel Inspector window: Click the **Summary** button on the **Navigation** toolbar.

The **Summary** window shows application problems detected during analysis, problem severity, and associated code locations. It also provides access to generic error explanations and corresponding source code in your default editor. Use this window to view problems displayed in the order detected and choose problems of interest to display in the **Sources** window.

Window Layout



Window Panes and Toolbars

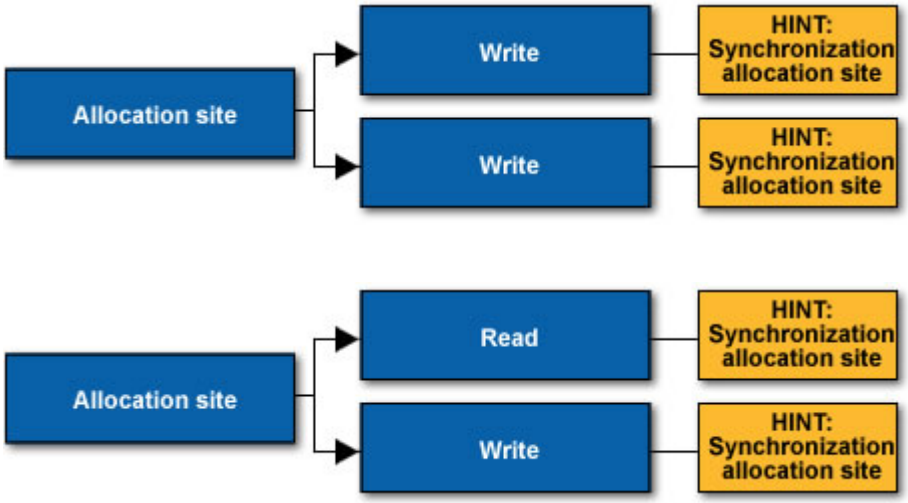
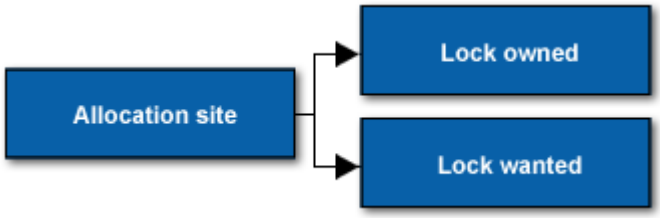
1. Toolbar: Navigation
2. Pane: Problems
3. Pane: Code Locations
4. Pane: Timeline
5. Toolbar: Command

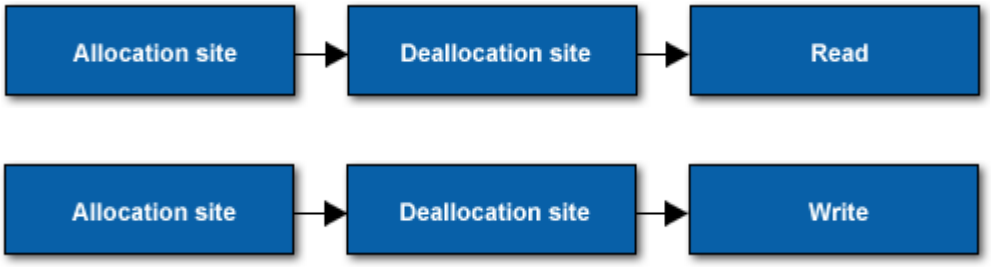
See Also

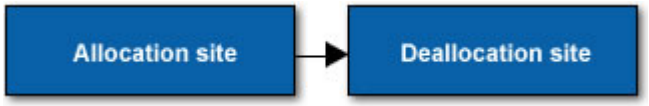
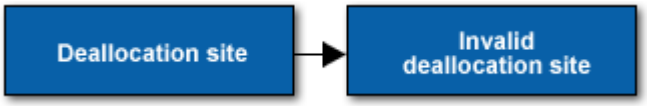
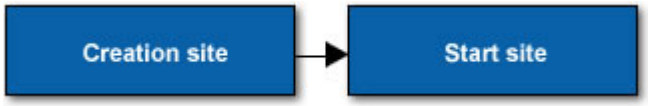
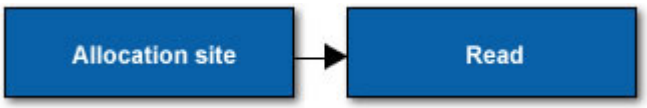
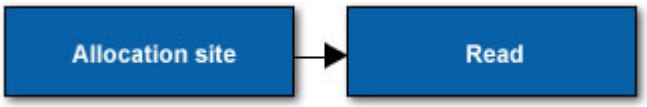
Choose Problems
Examine Result Data During Analysis

Problem Type Reference

Intel® Inspector currently classifies problems detected by analysis into the following types:

Problem Type	Code Location Relationships
Cross-thread stack access	
Data race	
Deadlock	
GDI resource leak	
Incorrect memcpy call	
Invalid deallocation	

Problem Type	Code Location Relationships
Invalid memory access	
Invalid partial memory access	
Kernel resource leak	
Lock hierarchy violation	
Memory growth	
Memory leak	
Memory not deallocated	

Problem Type	Code Location Relationships
Mismatched allocation/ deallocation	
Missing allocation	
Thread exit information	
Thread start information	
Unhandled application exception	
Uninitialized memory access	
Uninitialized partial memory access	

Parent topic: Intel® Inspector

See Also

Analysis Workflow

Cross-thread Stack Access

Occurs when a thread accesses a different thread's stack.

Syntax

ID	Code Location	Description
1	Allocation site	If present, represents the location and associated call stack where the accessed stack memory was allocated. For Microsoft Windows* threading, a thread's stack is allocated by the system when the thread is created, and the allocation site is implicit.
2	Stack owned	Represents the location and associated call stack where the thread owning the stack was created.
3	Stack cross access	Represents the location and associated call stack where the stack memory was accessed by a different thread.

Caution
It is unsafe if the two threads do not coordinate the access properly. This can happen even if the accesses do not have race conditions. Failure to coordinate safe accesses can potentially result in unexpected application behavior or even an application crash. However, the Intel Inspector does not distinguish between safe accesses and unsafe accesses. It is the user's responsibility to determine if a cross-thread stack access is safe or not.

Example

Preparation
```
int *p;
CreateThread(..., thread #1, ...);
CreateThread(..., thread #2, ...);
```

Thread #1
```
int q[1024]; // Allocated on Thread #1's stack
p = q;
q[0] = 1;
```

Thread #2
```
*p = 2; // Thread #1's stack accessed
```

If thread #1 and thread #2 are concurrent and there is no other synchronization between them, the Intel Inspector detects a **Cross-thread stack access** problem if `p = q;` in thread #1 executes before `*p = 2;` executes in thread #2.

The example shows the case where thread #1 publishes the address of a local stack variable `q` to the global pointer `p`, and later thread #2 accesses the stack of thread #1 by dereferencing the global pointer `p`, in effect writing to the variable `q` on thread #1.

Possible Correction Strategies

- Keep stack variables private to each thread.
- Avoid publishing addresses of stack variables and storing shared data on thread's stack. Instead, store shared data on the heap or in static variables.
- If you cannot avoid accessing a different thread's stack, establish a handshaking protocol between the owning thread and accessing thread to prevent accidental corruption or the reading of unexpected or garbage stack data.

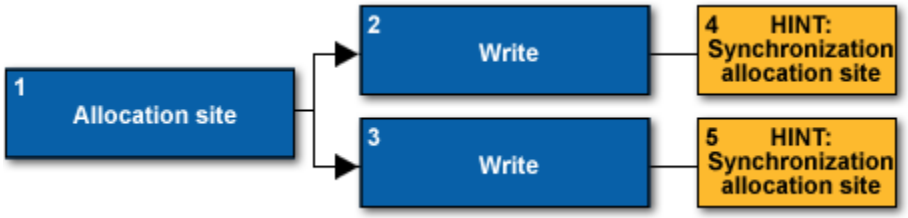
Caution
Sample Code Caveats

Data Race

Occurs when multiple threads access the same memory location without proper synchronization and at least one access is a write.

Write -> Write Data Race

Occurs when multiple threads write to the same memory location without proper synchronization.



ID	Code Location	Description
1	Allocation site	If present, and if the memory involved is heap memory, represents the location and associated call stack from which the memory block was allocated.
2	Write	Represents the instruction and associated call stack of the thread responsible for the first memory write.
3	Write	Represents the instruction and associated call stack of the thread responsible for the second memory write.
4	HINT: Synchronization allocation site	If present, represents the location and associated call stack of a synchronization object to protect the first memory write and resolve the **Data race** problem.
5	HINT: Synchronization allocation site	If present, represents the location and associated call stack of a synchronization object to protect the second memory write and resolve the **Data race** problem.

C Example

Preparation
```
CRITICAL_SECTION cs;
int *p = malloc(sizeof(int)); // Allocation Site
InitializeCriticalSection(&cs); // HINT for first Write
```

Thread #1
```
*p = 1; // First Write
```

Thread #2
```
EnterCriticalSection(&cs);
*p = 2; // Second Write
LeaveCriticalSection(&cs);
```

If thread #1 and thread #2 are concurrent and there is no other synchronization between thread #1 and thread #2, the Intel Inspector detects a **Data race** problem because the final value of `*p` depends on the write order and is therefore non-deterministic.

Fortran Example

Preparation	`include "omp_lib.h"`
	`integer(omp_lock_kind) lock`
	`integer, allocatable :: x`
	
	`allocate(x) ! Allocation site`
	`call omp_init_lock(lock) ! HINT for First Write`

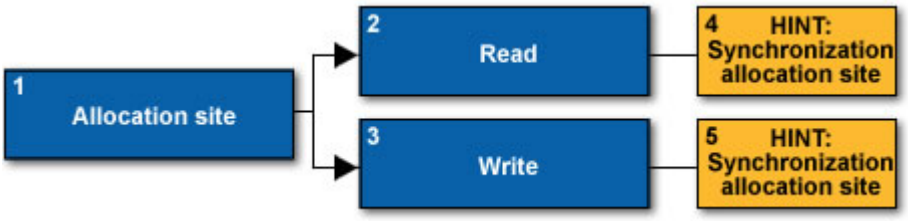
Thread #1	`x = 1 ! First Write`

Thread #2	`call omp_set_lock(lock)`
	`x = 2 ! Second Write`
	`call omp_set_lock(lock)`

If thread #1 and thread #2 are concurrent and there is no other synchronization between thread #1 and thread #2, the Intel Inspector detects a **Data race** problem because the final value of `x` depends on the write order and is therefore non-deterministic.

Read -> Write Data Race

Occurs when one thread reads while a different thread concurrently writes to the same memory location without proper synchronization.



ID	Code Location	Description
1	Allocation site	If present, and if the memory involved is heap memory, represents the location and associated call stack from which the memory block was allocated.
2	Read	Represents the instruction and associated call stack of the thread responsible for the memory read.
3	Write	Represents the instruction and associated call stack of the thread responsible for the memory write.
4	HINT: Synchronization allocation site	If present, represents the location and associated call stack of a synchronization object to protect the memory read and resolve the **Data race** problem.
5	HINT: Synchronization allocation site	If present, represents the location and associated call stack of a synchronization object to protect the memory write and resolve the **Data race** problem.

C Example

Preparation
```
CRITICAL_SECTION cs;
int *p = malloc(sizeof(int)); // Allocation Site
*p = 0;
InitializeCriticalSection(&cs); // HINT for Read
```

Thread #1
```
int x;
x = *p; // Read
```

Thread #2
```
EnterCriticalSection(&cs);
*p = 2; // Write
LeaveCriticalSection(&cs);
```

If thread #1 and thread #2 are concurrent and there is no other synchronization between thread #1 and thread #2, the Intel Inspector detects a **Data race** problem because the value of `*p` read by thread #1 depends on when the write by thread #2 occurs and is therefore non-deterministic.

The example shows the case if `x = *p` occurs in thread #1 before `*p = 2` occurs in thread #2.

Fortran Example

Preparation
```
include "omp_lib.h"
integer(omp_lock_kind) lock
integer, allocatable :: x
integer y

allocate(x) ! Allocation site
call omp_init_lock(lock) ! HINT for Read
```

Thread #1
```
y = x ! Read
```

Thread #2
```
call omp_set_lock(lock)
x = 2 ! Write
call omp_set_lock(lock)
```

If thread #1 and thread #2 are concurrent and there is no other synchronization between thread #1 and thread #2, the Intel Inspector detects a **Data race** problem because the value of `x` read by thread #1 depends on when the write by thread #2 occurs and is therefore non-deterministic.

The example shows the case if `y = x` occurs in thread #1 before `x = 2` occurs in thread #2.

About the HINTs

The hints are generated based on the memory and synchronization events in the application, not the logic of the application.

There can be multiple solutions to the problem, and the suggested solution may not be the best solution.

It is up to you to select the best solution.

Possible Correction Strategies

- First consider the algorithm. Not all sequential algorithms can run directly in parallel or be parallelized. If the algorithm is sequential in nature and cannot be parallelized, consider changing it to a concurrent algorithm or running the code sequentially.
- Privatize memory shared by multiple threads so each thread has its own copy.
  - For Microsoft Windows* threading, consider using `TlsAlloc()` and `TlsFree()` to allocate thread local storage.

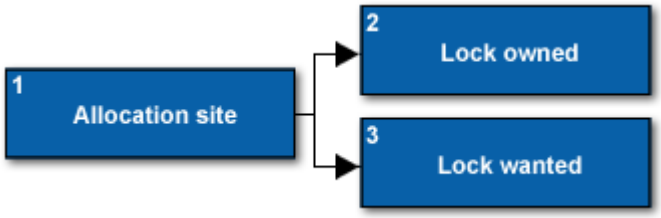
  - For OpenMP* threading, consider declaring the variable in a `private`, `firstprivate`, or `lastprivate` clause, or making it `threadprivate`.
  - Consider using thread stack memory.
- Synchronize access to the shared memory using synchronization objects.
  - For Microsoft Windows* threading, consider using mutexes or critical sections.
  - For OpenMP* threading, consider using atomic or critical sections or OpenMP* locks.

Caution
Sample Code Caveats

Deadlock

Occurs when two or more threads are waiting for each other to release resources (such as mutexes, critical sections, and thread handles) while holding resources the other threads are trying to acquire. If none of the threads release their resources, then none of the threads can proceed.

Syntax



ID	Code Location	Description
1	Allocation site	If present, represents the location and associated call stack where the resource was created.
2	Lock owned	Represents the location and associated call stack of the thread holding the object requested by another thread.
3	Lock wanted	Represents the location and associated call stack of the thread requesting the object held by another thread.

NOTE

- **Deadlock** problems are usually, but not always, caused by **Lock hierarchy violation** problems. If the Intel Inspector detects a **Deadlock** problem caused by a **Lock hierarchy violation** problem, it reports only the **Deadlock** problem.
- Intel Inspector cannot detect a **Deadlock** problem involving more than four threads.

C Example

Preparation

```
CRITICAL_SECTION cs1;
CRITICAL_SECTION cs2;
int x = 0;
int y = 0;
InitializeCriticalSection(&cs1); // Allocation Site (cs1)
InitializeCriticalSection(&cs2); // Allocation Site (cs2)
```

Thread #1

```
EnterCriticalSection(&cs1); // Lock Owned (cs1)
x++;
EnterCriticalSection(&cs2); // Lock Wanted (cs2)
y++;
LeaveCriticalSection(&cs2);
LeaveCriticalSection(&cs1);
```

Thread #2

```
EnterCriticalSection(&cs2); // Lock Owned (cs2)
y++;
EnterCriticalSection(&cs1); // Lock Wanted (cs1)
x++;
LeaveCriticalSection(&cs1);
LeaveCriticalSection(&cs2);
```

If thread #1 and thread #2 are concurrent and there is no other synchronization between them, the Intel Inspector detects a **Deadlock** problem if synchronization occurs in the following order:

1. `EnterCriticalSection(&cs1);` in thread #1
2. `EnterCriticalSection(&cs2);` in thread #2

Fortran Example

Preparation

```
include "omp_lib.h"
integer(omp_lock_kind) lock1
integer(omp_lock_kind) lock2
call omp_init_lock(lock1)
call omp_init_lock(lock2)
```

Thread #1

```
call omp_set_lock(lock1)
. . .
call omp_set_lock(lock2)
. . .
call omp_unset_lock(lock2)
. . .
call omp_unset_lock(lock1)
```

Thread #2

```
call omp_set_lock(lock2)
. . .
call omp_set_lock(lock1)
. . .
call omp_unset_lock(lock1)
. . .
call omp_unset_lock(lock2)
```

If thread #1 and thread #2 are concurrent and there is no other synchronization between them, the Intel Inspector detects a **Deadlock** problem if synchronization occurs in the following order:

1. `call omp_set_lock(lock1)` in thread #1

2. `call omp_set_lock(lock2)` in thread #2

Possible Correction Strategies

- Do not use multiple synchronization objects if one synchronization object is sufficient.
- Use recursive synchronization objects such as recursive mutexes if a thread must acquire the same object more than once.
- Avoid the case where two threads wait for each other to terminate. Instead, use a third thread to wait for both threads to terminate.
- Establish a global lock hierarchy and honor the same lock hierarchy in each thread. For example:
  - C language: If you have critical sections `cs1` and `cs2` and establish a global lock hierarchy (`cs1` , `cs2`), always acquire `cs1` before acquiring `cs2` and release `cs1` after releasing `cs2`.
  - Fortran language: If you have locks `lock1` and `lock2` and establish a global lock hierarchy (`lock1`, `lock2`), always acquire `lock1` before acquiring `lock2` and release `lock1` after releasing `lock2`.
- C language: Consider acquiring multiple synchronization objects at the same time using, for example, Microsoft Windows* system APIs such as `WaitForMultipleObjects()`.

NOTE
Sample Code Caveats

GDI Resource Leak

Occurs when a GDI object is created but never deleted.

Syntax

ID	Code Location	Description
1	Creation site	Represents the location and associated call stack from which the object was created.

Example

```
HPEN pen = CreatePen(0, 0, 0);
return;
```

Possible Correction Strategies

Use the appropriate function to delete the object after use.

Creation Function	Deletion Function
`CreateBitmap`, `CreateBitmapIndirect`, `CreateCompatibleBitmap`, `CreateDIBitmap`, `CreateDiscardableBitmap`, `LoadBitmap`,	`DeleteObject`

Creation Function	Deletion Function
CreateBrushIndirect, CreateDIBPatternBrush, CreateDIBPatternBrushPt, CreateHatchBrush, CreatePatternBrush, CreateSolidBrush	DeleteObject
CreateDC, CreateCompatibleDC	DeleteDC
CreateEllipticRgn, CreateEllipticRgnIndirect, CreatePolygonRgn, CreatePolyPolygonRgn, CreateRectRgn, CreateRectRgnIndirect, CreateRoundRectRgn, ExtCreateRegion	DeleteObject
CreateFont, CreateFontIndirect, CreateFontIndirectEx	DeleteObject
CreatePalette, CreateHalftonePalette	DeleteObject
CreatePen, CreatePenIndirect, ExtCreatePen	DeleteObject

Caution
Sample Code Caveats

Incorrect memcpy Call

Occurs when an application calls the `memcpy` *function with two pointers that overlap within the range to be copied. This condition is only checked on Linux* systems. On Windows* systems, this function is safe for overlapping memory.*

Syntax

ID	Code Location	Description
1	Call site	Represents the location from which the `memcpy` function was called.

Example

```
char *p = (char *)malloc(10);
memcpy(p+3, p, 5);
```

NOTE Linux* operating system/GNU gcc* compiler: The Intel Inspector cannot check for inlined calls to `memcpy`. Specify `-fno-builtin` to prevent inlining and allow the Intel Inspector to check for arguments.

Possible Correction Strategies

If the arguments represent the range you want to copy, call the `memmove` function instead of the `memcpy` function.

Caution
Sample Code Caveats

Invalid Deallocation

Occurs when an application calls a deallocation function with an address that does not correspond to dynamically allocated memory.

Syntax

ID	Code location	Description
1	Invalid deallocation site	Represents the location from which the invalid call to a deallocation function was made.

C Examples/Windows* OS

```
int x = 1;
int *p = &x;
VirtualFree(p, 1, MEM_DECOMMIT);
```

```
void *p = VirtualAlloc(NULL,1,MEM_COMMIT,PAGE_READWRITE);
VirtualFree(p,1,MEM_DECOMMIT);
VirtualFree(p,1,MEM_DECOMMIT);
```

C Example/Linux* OS

```
void *p = mmap(NULL, 8, PROT_READ, MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);
munmap(p, 8);
munmap(p, 8);
```

Possible Correction Strategies

If	Do This
This code location should free memory.	Change the code to pass in dynamically allocated memory.
This code location does not need to free memory.	Remove the call to the deallocation function.

Caution
Sample Code Caveats

Invalid Memory Access

Occurs when a read or write instruction references memory that is logically or physically invalid.

Syntax

ID	Code Location	Description
1	Allocation site	If present, represents the location and associated call stack from which the memory block containing the offending address was allocated.
2	Deallocation site	If present, represents the location and associated call stack from which the memory block containing the offending address was deallocated. The deallocation makes the access to the offending memory address logically invalid.
3	Read or Write	Represents the instruction and associated call stack responsible for the invalid access. If no allocation or deallocation is associated with this problem, the memory address might be one of the following: • Logically invalid stack space (below the current stack pointer value) • Memory that is physically not allocated to the process • Memory that has been deallocated **NOTE** The offset, if shown in the **Code Locations** pane, represents the byte offset into the allocated buffer where the **Invalid memory access** occurred.

C Examples

Heap example:

```
char *pStr = (char*) malloc(20);
free(pStr);
strcpy(pStr, "my string"); // Invalid write to deallocated memory
```

Stack example (set **Analyze stack accesses** to Yes when you configure the analysis):

```
void stackUnderrun()
{
   char array[10];
   strcpy(array, "my string");
   int len = strlen(array) - 1;
   while (array[len] != 'Z') // Will read memory outside of array
      len--;
}
```

Fortran Example

```
integer, allocatable :: i, b(:)

allocate( b(100))

! Body of Console1
do i = 1, 100
    b(i) = i
end do

deallocate(b)
! Invalid write to deallocated memory
b(1) = 1
```

Possible Correction Strategies

If	Do This
An invalid pointer dereference (corrupt or invalid value) occurs. This is usually a logic error in a sequential algorithm. For example, the pointer may increment past the end or decrement before the beginning of dynamically allocated memory. This pointer address value can be the address of memory that is marked invalid to access.	When dereferencing a pointer, ensure: • The pointer refers to a valid object. • Pointer arithmetic stays within the bounds of the dynamically allocated memory.
A stale reference to memory occurs. You can easily recognize this situation if allocation and deallocation information is present. For example, if a thread holds a pointer to the heap, and another thread releases the memory back to the heap, then all future accesses to this memory are invalid.	You need a clear concept of object ownership to ensure one thread does not mistakenly release an object back to the heap while another thread considers the pointer valid. Consider reference counting.
The Intel Inspector detects an invalid memory access when enhanced dangling pointer check is disabled.	Run another memory error analysis with enhanced dangling pointer check enabled to potentially report allocation and deallocation information.

Caution
Sample Code Caveats

Invalid Partial Memory Access

Occurs when a read or write instruction references a block (2-bytes or more) of memory where part of the block is logically invalid.

Syntax

ID	Code Location	Description
1	Allocation site	If present, represents the location and associated call stack from which the memory block adjacent to the offending address was allocated.
2	Deallocation site	If present, represents the location and associated call stack from which the memory block containing the offending address was deallocated. The deallocation makes the access to the offending memory address logically invalid.
3	Read or Write	Represents the instruction and associated call stack responsible for the partial invalid access. If no allocation or deallocation is associated with this problem, the memory address might be one of the following: • Logically invalid stack space (below the current stack pointer value) • Memory that is physically not allocated to the process **NOTE** The offset, if shown in the **Code Locations** pane, represents the byte offset into the allocated buffer where the **Invalid partial memory access** occurred.

Example

```
struct tally
{
   int num;
   char count;
};
```

```
struct tally *pCurrent = (struct tally *)malloc(5); // incorrect size allocated
struct tally *pRoot = (struct tally *)malloc(sizeof(struct tally));
pCurrent->num = 1;
pCurrent->count = 1;
*pRoot = *pCurrent; // will result in partial invalid read
```

NOTE
Different compilers and optimization levels can produce different assembly for block copies of memory. Depending on the generated assembly, this example might produce an invalid memory access problem.

Possible Correction Strategies

The typical cause of an invalid partial memory access problem is a miscalculation of the required size of an object. Determine the correct size for object creation.

Caution
Sample Code Caveats

Kernel Resource Leak

Occurs when a kernel object handle is created but never closed.

Syntax

ID	Code Location	Description
1	Creation site	Represents the location and associated call stack from which the handle was created.

C Example

```
HANDLE hThread = CreateThread(0, 8192, work0, NULL, 0, NULL);
return;
```

Fortran Example

```
ThreadHandle = CreateThread(security, stack_size, Thread_Proc,loc(ivalue),
CREATE_SUSPENDED, thread_id)
   end
```

Possible Correction Strategies

Use the appropriate function to close the handle after use.

Creation Function	Close Function
BeginUpdateResource	EndUpdateResource
CreateConsoleScreenBuffer	CloseHandle
CreateEvent, OpenEvent	CloseHandle
CreateFile, ReOpenFile	CloseHandle
CreateFileMapping, OpenFileMapping	CloseHandle
CreateIoCompletionPort	CloseHandle
CreateJobObject	CloseHandle
CreateMailslot	CloseHandle
CreateMemoryResourceNotification	CloseHandle
CreateMutex, OpenMutex	CloseHandle
CreatePipe, CreateNamedPipe	CloseHandle
CreateProcess, OpenProcess	CloseHandle
CreateProcessAsUser, CreateProcessWithLogon	CloseHandle
CreateSemaphore, OpenSemaphore	CloseHandle
CreateThread, CreateRemoteThread, OpenThread	CloseHandle
CreateToken, CreateRestrictedToken	CloseHandle
CreateToolhelp32Snapshot	CloseHandle
CreateWaitableTimer, OpenWaitableTimer	CloseHandle
DuplicateHandle	CloseHandle
DuplicateToken	CloseHandle
FindFirstChangeNotification	FindCloseChangeNotification
FindFirstFile, FindFirstFileEx, FindFirstFileTransacted	FindClose
FindFirstStreamW, FindFirstStreamTransactedW	FindClose
InitializeCriticalSection, InitializeCriticalSectionAndSpinCount	DeleteCriticalSection
LogonUser	CloseHandle
OpenEventLog, OpenBackupEventLog	CloseEventLog
OpenProcessToken, OpenThreadToken	CloseHandle

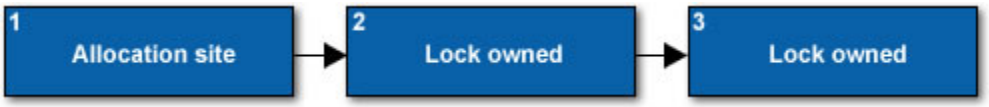
Creation Function	Close Function
`RegisterEventSource`	`DeregisterEventSource`
`WSASocket, socket`	`closesocket`

Caution
Sample Code Caveats

Lock Hierarchy Violation

Occurs when the acquisition order of multiple synchronization objects (such as mutexes, critical sections, and thread handles) in one thread differs from the acquisition order in another thread, and these synchronization objects are owned by the acquiring thread and must be released by the same thread.

The Intel Inspector reports a **Lock hierarchy violation** problem as multiple problems in a problem set. Each problem shows a portion of the **Lock hierarchy violation** from the perspective of a single thread.



ID	Code Location	Description
1	Allocation site	If present, represents the location and associated call stack where the synchronization object acquired by a thread (usually the object acquired first) was created.
2	Lock owned	Represents the location and associated call stack where a synchronization object was acquired by a thread.
3	Lock owned	Represents the location and associated call stack where another synchronization object was later acquired by the same thread.

NOTE
Deadlock problems are usually, but not always, caused by **Lock hierarchy violation** problems. If the Intel Inspector detects a **Deadlock** problem caused by a **Lock hierarchy violation** problem, it reports only the **Deadlock** problem.

C Example

Preparation

```
CRITICAL_SECTION cs1;
CRITICAL_SECTION cs2;
int x = 0;
int y = 0;
InitializeCriticalSection(&cs1); // Allocation Site (cs1)
InitializeCriticalSection(&cs2); // Allocation Site (cs2)
```

Thread #1
```
EnterCriticalSection(&cs1); // Lock Owned (cs1)
x++;
EnterCriticalSection(&cs2); // Lock Owned (cs2)
y++;
LeaveCriticalSection(&cs2);
LeaveCriticalSection(&cs1);
```

Thread #2
```
EnterCriticalSection(&cs2); // Lock Owned (cs2)
y++;
EnterCriticalSection(&cs1); // Lock Owned (cs1)
x++;
LeaveCriticalSection(&cs1);
LeaveCriticalSection(&cs2);
```

If thread #1 and thread #2 are concurrent and there is no other synchronization between them, the Intel Inspector detects a **Lock hierarchy violation** problem instead of a **Deadlock** problem if synchronization occurs in the following order:

1. `EnterCriticalSection(&cs1);` in thread #1
2. `EnterCriticalSection(&cs2);` in thread #1
3. `EnterCriticalSection(&cs2);` in thread #2
4. `EnterCriticalSection(&cs1);` in thread #2

Fortran Example

Preparation
```
include "omp_lib.h"
integer(omp_lock_kind) lock1
integer(omp_lock_kind) lock2
call omp_init_lock(lock1)
call omp_init_lock(lock2)
```

Thread #1
```
call omp_set_lock(lock1)
. . .
call omp_set_lock(lock2)
. . .
call omp_unset_lock(lock2)
. . .
call omp_unset_lock(lock1)
```

Thread #2
```
call omp_set_lock(lock2)
. . .
call omp_set_lock(lock1)
. . .
call omp_unset_lock(lock1)
. . .
call omp_unset_lock(lock2)
```

If thread #1 and thread #2 are concurrent and there is no other synchronization between them, the Intel Inspector detects a **Lock hierarchy violation** problem instead of a **Deadlock** problem if synchronization occurs in the following order:

1. `call omp_set_lock(lock1)` in thread #1
2. `call omp_set_lock(lock2)` in thread #1
3. `call omp_set_lock(lock2)` in thread #2
4. `call omp_set_lock(lock1)` in thread #2

Possible Correction Strategies

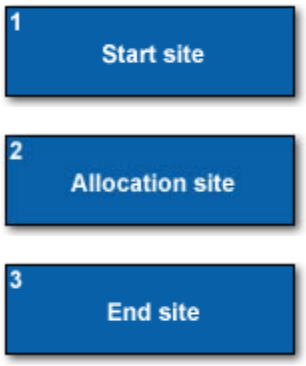
- Do not use multiple synchronization objects if one synchronization object is sufficient.
- Use recursive synchronization objects such as recursive mutexes if a thread must acquire the same object more than once.
- Avoid the case where two threads wait for each other to terminate. Instead, use a third thread to wait for both threads to terminate.
- Establish a global lock hierarchy and honor the same lock hierarchy in each thread. For example:
  - C language: If you have critical sections `cs1` and `cs2` and establish a global lock hierarchy (`cs1` , `cs2`), always acquire `cs1` before acquiring `cs2` and release `cs1` after releasing `cs2`.
  - Fortran language: If you have locks `lock1` and `lock2` and establish a global lock hierarchy (`lock1`, `lock2`), always acquire `lock1` before acquiring `lock2` and release `lock1` after releasing `lock2`.
- C language: Consider acquiring multiple synchronization objects at the same time using, for example, Microsoft Windows* system APIs such as `WaitForMultipleObjects()`.

Caution
Sample Code Caveats

Memory Growth

Occurs when a block of memory is allocated but not deallocated within a specific time segment during application execution.

Syntax



ID	Code Location	Description
1	Start site	Represents a point in time.
2	Allocation site	Represents the location and associated call stack of a memory block allocated but not deallocated between the start site and end site. There may be no, one, or many allocation site code locations in a **Memory growth** problem.
3	End site	Represents a point in time.

NOTE
Intel Inspector distinguishes among **Memory leak**, **Memory not deallocated**, and **Memory growth** problem types in the following manner:

- **Memory leak** problems occur when a block of memory is allocated, never deallocated, and not reachable (there is no pointer available to deallocate the block). Severity level =  (**Error**).
- **Memory not deallocated** problems occur when a block of memory is allocated, never deallocated, but still reachable at application exit (there is a pointer available to deallocate the block). Severity level =  (**Warning**).
- **Memory growth** problems occur when a block of memory is allocated, but not deallocated, within a specific time segment during application execution. Severity level =  (**Warning**).

Example

```
void ProcessTransaction(TransactionContext x)
{
   ...
   char* m = (char*) malloc(128);
   ...
   return;
}

void WaitForTransactions()
{
   ...
   for (;;)
   {
      TransactionContext x = WaitForTransaction(); // Report end site
      ProcessTransaction(x);
   }
}
```

In this example, if the user clicks the **Reset Growth Tracking** button, performs the transaction, and then clicks the **Measure Growth** button, the Intel Inspector reports a single allocation of 128 bytes within `ProcessTransaction`.

Possible Correction Strategies

Identify the leaked memory blocks that should have been freed and call the appropriate free routine during the lifetime of the transaction.

NOTE
Intel Inspector does not perform a *reachability* analysis. Some of the reported memory blocks may:

- Be already unreachable
- Become unreachable later
- Be deallocated later
- Be still allocated but reachable when analysis ends

Caution
Sample Code Caveats

Memory Leak

Occurs when a block of memory is allocated, never deallocated, and not reachable (there is no pointer available to deallocate the block).

Syntax

ID	Code Location	Description
1	Allocation site	Represents the location and associated call stack from which the memory block was allocated.

NOTE
Intel Inspector distinguishes among **Memory leak**, **Memory not deallocated**, and **Memory growth** problem types in the following manner:

- **Memory leak** problems occur when a block of memory is allocated, never deallocated, and not reachable (there is no pointer available to deallocate the block). Severity level =

  (**Error**).
- **Memory not deallocated** problems occur when a block of memory is allocated, never deallocated, but still reachable at application exit (there is a pointer available to deallocate the block). Severity level =

  (**Warning**).
- **Memory growth** problems occur when a block of memory is allocated, but not deallocated, within a specific time segment during application execution. Severity level =

  (**Warning**).

C Example

```
char *pStr = (char*) malloc(512);
return;
```

Fortran Example

```
function LEAK
integer, pointer, dimension(:) :: ptr
integer LEAK(100)
integer :: val

allocate(ptr(100))
allocate(ptr(100))
allocate(ptr(100))
```

```
ptr(2)=val
val=ptr(1)
LEAK = ptr

end function LEAK
```

Possible Correction Strategies

Use the appropriate deallocation function to return the memory block to the heap after its last use.

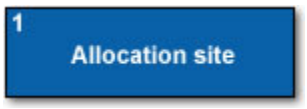
Platform	Memory Allocator	Memory Deallocator
C++ language	`new` operator	`delete` operator
	`new[]` operator	`delete[]` operator
C language	`malloc()`, `calloc()`, or `realloc()` functions	`free()` function
Fortran language	`allocate()` function	`deallocate()` function
Windows* API	Windows* dynamic memory functions such as `GlobalAlloc()` or `LocalAlloc()`	Appropriate functions, such as `GlobalFree()` or `LocalFree()`

Caution
Sample Code Caveats

Memory Not Deallocated

Occurs when a block of memory is allocated, never deallocated, but still reachable at application exit (there is a pointer available to deallocate the block).

Syntax



ID	Code Location	Description
1	Allocation site	Represents the location and associated call stack from which the memory block was allocated.

NOTE
Intel Inspector distinguishes among **Memory leak**, **Memory not deallocated**, and **Memory growth** problem types in the following manner:

- **Memory leak** problems occur when a block of memory is allocated, never deallocated, and not reachable (there is no pointer available to deallocate the block). Severity level =  (**Error**).
- **Memory not deallocated** problems occur when a block of memory is allocated, never deallocated, but still reachable at application exit (there is a pointer available to deallocate the block). Severity level =  (**Warning**).
- **Memory growth** problems occur when a block of memory is allocated, but not deallocated, within a specific time segment during application execution. Severity level =  (**Warning**).

C Example

```
static char *pStr = malloc(512);
return;
```

Fortran Example

```
integer, allocatable, save, dimension(:)  :: notdeallocatedptr(:)
allocate(notdeallocatedptr(200))
```

Possible Correction Strategies

Use the appropriate deallocation function to return the memory block to the heap after its last use.

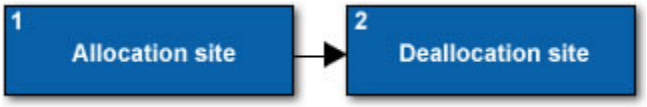
Platform	Memory Allocator	Memory Deallocator
C++ language	`new` operator	`delete` operator
	`new[]` operator	`delete[]` operator
C language	`malloc()`, `calloc()`, or `realloc()` functions	`free()` function
Fortran language	`allocate()` function	`deallocate()` function
Windows* API	Windows* dynamic memory functions such as `GlobalAlloc()` or `LocalAlloc()`	Appropriate functions, such as `GlobalFree()` or `LocalFree()`

Caution
Sample Code Caveats

Mismatched Allocation/Deallocation

Occurs when a deallocation is attempted with a function that is not the logical reflection of the allocator used.

Syntax



ID	Code Location	Description
1	Allocation site	Represents the location and associated call stack from which the memory block was allocated.
2	Deallocation site	Represents the location and associated call stack attempting the deallocation.

Example

```
char *s = (char*)malloc(5);
delete s;
```

Possible Correction Strategies

Use the appropriate deallocation function to return the memory block to the heap after its last use.

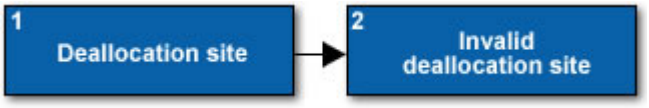
Platform	Memory Allocator	Memory Deallocator
C++ language	`new` operator	`delete` operator
	`new[]` operator	`delete[]` operator
C language	`malloc()`, `calloc()`, or `realloc()` functions	`free()` function
Fortran language	`allocate()` function	`deallocate()` function
Windows* API	Windows* dynamic memory functions such as `GlobalAlloc()` or `LocalAlloc()`	Appropriate functions, such as `GlobalFree()` or `LocalFree()`

Caution
Sample Code Caveats

Missing Allocation

Occurs when an invalid pointer is passed to a deallocation function. The invalid address may point to a previously released heap block.

Syntax



ID	Code Location	Description
1	Deallocation site	If present, represents the location and associated call stack from which the memory block referenced by the address was previously deallocated. The previous deallocation makes all subsequent deallocation attempts invalid.
2	Invalid deallocation site	Represents the location and associated call stack attempting the deallocation.

C Example

```
char* pStr = (char*) malloc(20);
free(pStr);
free(pStr);
```

Fortran Example

```
integer, target :: b(100)
integer, pointer :: pmiddle
pmiddle => b(4)
DEALLOCATE( pmiddle )
```

Possible Correction Strategies

If	Do This
A double deallocation occurs.	Remove the extraneous deallocation.
An invalid deallocation occurs.	Fix the pointer value passed to the deallocator.

Caution
Sample Code Caveats

Thread Exit Information

Occurs when the Intel Inspector detects thread termination. This problem is really informational feedback.

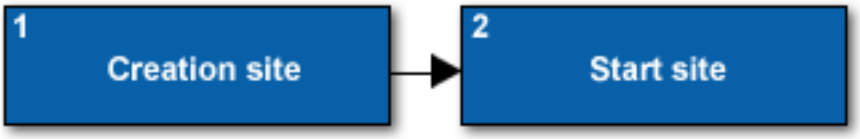
ID	Code Location	Description
1	Exit site	Represents the location where a thread terminated.

Thread Start Information

Occurs when the Intel Inspector detects the creation of a thread. This problem is really informational feedback useful for confirming the number and location of threads created during application execution and data collection.

You should expect at least two such messages when inspecting a multithreaded application. If the Intel Inspector reports only one such message for a multithreaded application, you may be executing the application on a single core machine where the application creates, by default, the same number of threads as cores. If this is the case, increase the number of threads generated by the application in order to use the full range of the Intel Inspector capabilities.

NOTE
Intel Inspector may be able to detect the creation of a thread but not the start or creation site.



ID	Code Location	Description
1	Creation site	Represents the location and call stack where a thread was created.
2	Start site	Represents the location where the thread started execution.

Unhandled Application Exception

Occurs when the application undergoing analysis crashes because of an unhandled exception thrown by the application.

Syntax

ID	Code Location	Description
1	Exception	Represents the instruction that threw the exception.

C Example

The following C example is overly simplified to provide possible scenarios where these types of errors may occur.

```
void problem1 (int *y)
{
   *y = 5;
}
void problem2()
{
   int *x = new int;
}
```

Two exceptions are possible in this example:

- `y` may not reference a valid memory address; therefore the write instruction may cause an exception. If that exception is not properly handled, the Intel Inspector shows an unhandled exception pointing to the write instruction at `y`.
- If the application is out of memory, the allocation throws an exception. If the exception is unhandled by the application, the Intel Inspector shows an unhandled exception associated with the allocation.

Fortran Example

The following Fortran example is also overly simplified to provide possible scenarios where this type of error may occur.

```
function problem1(a)
    integer, allocatable :: a(:)

    a(5) = 1
    problem1 = 1
end function problem1

function problem2(a)
    integer, allocatable :: a(:)

    allocate(a(1000))
    problem2 = 1000
end function problem2
```

Possible Correction Strategies

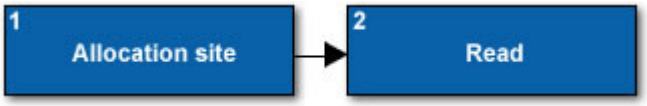
This problem usually indicates an existing bug in your application that manifests during analysis. In general, it is good programming practice to ensure allocations succeed and references are valid before using them.

Caution
Sample Code Caveats

Uninitialized Memory Access

Occurs when a read of an uninitialized memory location is reported.

Syntax



ID	Code Location	Description
1	Allocation site	If present, represents the location and associated call stack from which the memory block containing the offending address was allocated.
2	Read	Represents the instruction and associated call stack responsible for the uninitialized access. If no allocation is associated with this problem, the memory address might be in uninitialized stack space. **NOTE** The offset, if shown in the **Code Locations** pane, represents the byte offset into the allocated buffer where the **Uninitialized memory access** occurred.

C Examples

Heap example:

```
char* pStr = (char*) malloc(20);
char c = pStr[0]; // the contents of pStr were not initialized
```

Stack example (set **Analyze stack accesses** to Yes when you configure the analysis):

```
void func()
{
   int a;
   int b = a * 4; // read of uninitialized variable a
}
```

Fortran Examples

Heap example:

```
integer, allocatable :: a(:)
integer :: b

allocate( a(10) )
b = a(1) * 4
```

Stack example (set **Analyze stack accesses** to Yes when you configure the analysis):

```
integer :: a, b

b = a * 4
```

NOTE Buffers created by system calls linking processes to shared memory are flagged as allocated memory, which means the Intel Inspector does not report an **Uninitialized memory access** on these buffers.

Possible Correction Strategies

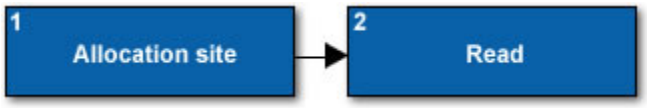
Initialize the offending memory prior to use.

Caution
Sample Code Caveats

Uninitialized Partial Memory Access

Occurs when a read instruction references a block (2-bytes or more) of memory where part of the block is uninitialized.

Syntax



ID	Code Location	Description
1	Allocation site	If present, represents the location and associated call stack from which the memory block containing the offending address was allocated.
2	Read	Represents the instruction and associated call stack responsible for the partial uninitialized access. If no allocation or deallocation is associated with this problem, the memory address might be in stack space. **NOTE** The offset, if shown in the **Code Locations** pane, represents the byte offset into the allocated buffer where the **Uninitialized partial memory access** occurred.

C Example

```
struct person
{
    unsigned char age;
    char firstInitial;
    char middleInitial;
    char lastInitial;
};
struct person *p1, *p2;
p1 = (struct person*) malloc(sizeof(struct person));
p2 = (struct person*) malloc(sizeof(struct person));
```

```
p1->firstInitial = 'c';
p1->lastInitial = 'o';
*p2 = *p1; // will result in partial uninitialized read
```

Fortran Example

```
type node
    character data1
    character data2
end type node

! Variables
type(node) :: a, b

a%data1 = "a"
b = a
```

NOTE Buffers created by system calls linking processes to shared memory are flagged as allocated memory, which means the Intel Inspector does not report an **Uninitialized partial memory access** on these buffers.

Possible Correction Strategies

Determine the correct initialization for the memory being accessed.

Caution
Sample Code Caveats

Command Line Interface Support

You can use the Intel® Inspector command line interface (`inspxe-cl` command tool) to:

- Perform analysis and collect data as part of an automated or background task, so you can view the result at your convenience.
- Perform regression testing to determine if any recent source code changes introduced new memory or threading problems.
- Generate several different types of reports using a variety of formatting options.

Tip

- To access the most current command line documentation: Enter `inspxe-cl -help`
- To duplicate an analysis performed in the Intel Inspector GUI: Use the **Command Line** button on the **Analysis Type** window panes to copy the exact command to the clipboard.

- Command Syntax
- Command Syntax Alternatives and Rules
- Command Line Output
- Collecting Result Data
- Working with Reports
- Working with Suppressions
- inspxe-cl Actions, Options and Arguments

Command Syntax

The Intel Inspector`inspxe-cl` command syntax is:

```
$ inspxe-cl <-action> [-action-options] [-global-options] [-- application [application options]]
```

inspxe-cl	The name of the Intel Inspector command line tool.
<-action>	The action to perform, such as `collect` or `report`. There must be only one action per command. So you cannot, for example, collect data and generate a report with the same command.
[-action-option]	Action-options modify behavior specific to the action. You can have multiple action-options per action. Using an action-option that does not apply to the action results in a usage error.
[-global-option]	Global-options modify behavior in the same manner for all actions. You can have multiple global-options per action.
[-- application]	The target application to analyze.
[application-options]	Options for the application.

Tip

- To access the most current command line documentation: Enter `inspxe-cl -help`
- To duplicate an analysis performed in the Intel Inspector GUI: Use the **Command Line** button on the **Analysis Type** window panes to copy the exact command to the clipboard.

Example #1

Display product version information.

```
$ inspxe-cl -version
```

Example #2

This example:

- Runs a **Detect Deadlocks and Data Races (ti2)** analysis on the application `myApp`.
- Stores the collected data in a result in the specified `myRes` result directory, which is created in the current working directory by default.

```
$ inspxe-cl -collect ti2 -result-dir .\myResult -- myApp.exe
```

where:

- `-collect` = action
- `ti2` = argument of action
- `-result-dir` = action-option
- `myResult` = argument of action-option
- `myApp.exe` = target application

NOTE
If you do not supply an absolute pathname for the application, system path environment settings are used to locate it.

Example #3

This example generates a `mySup.sup` suppression file for all problems in the specified `myRes` result directory.

```
$ inspxe-cl -create-suppression-file .\mySupFile -result-dir .\myRes
```

where:

- `-create-suppression-file` = action
- `mySup` = argument of action
- `-result-dir` = action-option
- `myRes` = argument of action-option

NOTE
The `inspxe-cl` tool appends a `.sup` extension if no extension is specified.

See Also

inspxe-cl Actions, Options and Arguments
Command Line Output

Command Syntax Alternatives and Rules

Syntax Alternatives

The Intel Inspector`inspxe-cl` command offers the following syntax alternatives for specifying options and arguments:

Syntax	Alternative	Valid Command Snippets
`-option`	`--option`	`inspxe-cl -version` `inspxe-cl--version`
`-option argument`	`-option=argument`	`inspxe-cl -c ti2` `inspxe-cl -c=ti2`
`-option` (long name)	`-o` (short name, as defined in the -help)	`inspxe-cl -verbose` `inspxe-cl -v`

`-option` (long name)	`-n-n` (implicit short name composed from initial letters of option name fragments)	`inspxe-cl-create-suppression-file`
		`inspxe-cl -c-s-f`
		NOTE
		If an implicit short name is ambiguous, the `inspxe-cl` command reports a syntax error.

`-option argument1,argument2`	`-option=argument1 -option=argument2`	`inspxe-cl -option-file file1,file2`

Syntax	Alternative	Valid Command Snippets
		`inspxe-cl -option-file file1 -option-file file2` **NOTE** Make sure there are no spaces after commas.

You may mix and match syntax alternatives. For example, the following commands are equivalent:

```
inspxe-cl -collect ti2 -result-dir myRes -suppression-file mySup -- myApp
```

```
inspxe-cl -collect=ti2 -r myRes --s-f mySup -- myApp
```

Other Syntax Rules

- Options that appear before `[-- application [application options]]` can appear in any order.
- If an argument contains spaces, enclose the argument in quotation marks.
- There are two ways to specify multiple options. One is to use multiple option=value pairs; the other is to use the option followed by a list of comma-separated values with no spaces.
- Two single-character short names cannot be combined after a single dash.
- Both short names and long names are case-sensitive. For example, `-R` is the short name of the `report` action, and `-r` is the short name of the `result-dir` action-option.

See Also

inspxe-cl Actions, Options and Arguments
Command Line Output

Command Line Output

Output from the Intel Inspector`inspxe-cl` command includes the following:

- Result directory
- Result file
- Summary
- Log file
- Exit code
- Suppression file(s)
- Report(s)

Result Directory

A `collect` or `collect-with` action generates a result directory to contain the analysis result.

By default, the result directory name is `r@@@{at}` where:

- `@@@` - next available number for similarly named result directories in the current working directory. The series begins with 000.
- `{at}` - analysis type code. This code represents the analysis type used when the result was created.

For example: `r001ti3`, where:

- `001` - next available number
- `ti3` - the selected analysis type (`ti` stands for "threading error analysis" and `3` indicates that this uses the level 3 setting, which is the highest setting. The `ti3` analysis type is known as the **Locate Deadlocks and Data Races** setting.)

The result directory name may be assigned according to the default naming conventions, or you may use the `result-dir` option to specify the result directory name during collection. Your specified result directory name can include up to five @ counters, so the result directory can be specified in scripts without having to update the result directory value each time.

By default, result directories are created in the current working directory. If you want to specify an alternative location, use the `user-data-dir` option to specify the parent directory for result directories. If you use this option during collection, remember to include it for any other actions you might perform on these results, such as generating reports.

Caution
You cannot put multiple results into the same result directory. If you specify the same result directory for multiple analysis runs, an error is returned. Use the auto-increment counter (@@@) to work around this restriction.

Result File

During a `collect` or `collect-with` action, the `inspxe-cl` command tool creates a result directory in which it stores the `*.inspxe` result file. In addition to analysis data, the result file stores data about the analysis environment, including start time, host machine name, product ID, CPU count, and analysis type used for collection.

The result filename is derived from the result directory name. For example: If the result directory name is `r001ti3`, the result filename is `r001ti3.inspxe`.

Summary Output, Log and Errors

At the end of an analysis run, the `inspxe-cl` command tool also writes the following files to the result directory:

- `inspxe-cl.txt`: Summary that includes the total number of problems found and types of problems found.
- `inspxe-cl.log`: Information useful to the Intel support team if the `inspxe-cl` command encounters internal errors
- The summary is also written to `stdout`.
- Internal errors are written to `stderr`.

Exit Codes

The `inspxe-cl` command returns the following exit codes:

0	Success and no new problems detected
1	Usage error
2	Internal error
4	Application returned a non-zero exit code
8	At least one new problem detected
12	Application returned a non-zero exit code and at least one new problem detected

NOTE
Use the `return-app-exitcode` action-option if you want to return the application exit code instead of the `inspxe-cl` exit code.

Suppression Files

Suppression files can be created and used in the GUI or on the command line.

- Suppression files that suppress all problems in the code (suppress-all files) can only be created from the command line.
- Use the command line to convert third-party suppression files (Purify, Valgrind) to a format that can be used by Intel Inspector.
- Once created, you can use suppression files when performing analysis or generating reports.

Reports

After collecting a result, you can use the `report` action to generate one or more types of reports from the result:

- **status**: Brief statement of the total number of new problems found, and a breakdown by state.
- **problems**: Detailed list of all new problem sets found, including their location in the source code.
- **observations**: Detailed list of all code locations in the source code used to form new problem sets.
- **summary**: This report is generated automatically in .txt form, but can also be generated on demand. Brief statement of the total number of new problems found, and a breakdown by problem type.

By default, the report is output in text format to `stdout`.

- To write a report to a file, use the `report-output` option when generating the report.
- To specify CSV or XML format, use the `format` option.
- To control how a report displays, you can use the `sort-asc`, `sort-desc` and `filter-include` options.
- To view a report from a result directory other than the default, use the `result-dir` option.

See Also

inspxe-cl Actions, Options and Arguments
Command Syntax
Intel Inspector Filenames and Locations
Working with Suppressions from the Command Line

Collecting Result Data from the Command Line

You can use the Intel Inspector`inspxe-cl` command to perform analysis and collect data as a background or automated task, which is particularly useful when the estimated collection time is lengthy, or when you want analysis and report generation to be part of your regularly scheduled processes.

This documentation uses the terms *analysis* and *data collection* as synonyms for the same process. When you initiate analysis, there are *collectors* that perform analysis based on your specified *analysis type* or *knob* values. The collected analysis data is packaged into a result.

The result is not directly viewable, but can be opened in the GUI or used to generate one or more kinds of reports. Some additional analysis occurs when reports are generated, so report generation may require source files in addition to the result.

Getting Started with Analysis on the Command Line

Here are the basic steps you would follow the first time you inspect your application for memory and threading issues:

1. Set up your command line environment. See the following Process Example section for details.
2. Invoke the `inspxe-cl` command with the `collect` action, specifying one of the lower-level memory error analysis types to inspect your application for memory issues.
3. Use the `collect` action with one of the threading error analysis types to inspect your application for threading issues.
4. After each analysis, you can read the Summary report, and then open your result for viewing in the GUI, or use the `report` action to generate a report from the analysis result.

Tip

- Collections that use lower-level analysis types take less time to complete than collections that use higher-level analysis types, and provide more of an overview, which are two reasons why you generally want to begin testing with the lowest-level analysis types.
- To duplicate an analysis performed in the Intel Inspector GUI: Use the **Command Line** button on the **Analysis Type** window panes to copy the exact command to the clipboard.

How the Collection Process Works

There are two actions that you can use to initiate analysis and data collection: `collect` and `collect-with`. These actions are very similar, as both perform the same basic collection process. The `collect` action provides a variety of predefined analysis types from which to select. If none of these predefined analysis types meet your specific needs, you can use the `collect-with` action and specify your own knob options.

By default, the collection process:

- Performs the specified type of analysis.
- Collects data and writes it to a result file in a result directory.
- Finalizes the result and generates a summary of all detected problems, writes the Summary to `stdout` and also saves it to an XML file in the result directory.
- Creates a log for troubleshooting purposes, and saves it in the result directory.
- Returns an exit code.

How to Perform Analysis on the Command Line

Here is a general example of how to perform analysis using the `inspxe-cl` command tool.

1. You can open a command prompt window and use the following command to run the script that establishes Intel Inspector environment settings: `C:\"Program Files (x86)"\Intel\oneAPI\setvars.bat`.
2. To inspect the `myApp` application for memory leak problems, use the `collect` action with the `mi1` (**Detect Leaks**) analysis type, and specify the search directory where all the source, binary and symbol files are stored.

   ```
   inspxe-cl -collect mi1 -search-dir all=C:\myProject -- myApp
   ```

   By default, the result is written to a result directory under the default current working directory. Assuming this is the first result directory, it will be named `r000mi1`. A Summary is sent to `stdout`, and saved in the result directory as `inspxe-cl.xml`.

NOTE
If you do not supply an absolute pathname for the application, system path environment settings are used to locate it.

3. Next, inspect the `myApp` application for deadlock and data races, using the `collect` action with the `ti2` (**Detect Deadlocks and Data Races**) analysis type, and specify the same search directories that you used for the memory analysis.

   ```
   inspxe-cl -collect ti2 -search-dir all=C:\myProject -- myApp
   ```

   By default, the result is written to a result directory under the current working directory. Assuming this is the first result directory, it will be named `r000ti2`. The summary is sent to `stdout`, and saved in the `r000ti2` result directory as `inspxe-cl.xml`.

Next Steps

1. Review the Summary of detected problems in each of the `inspxe-cl.xml` files.
2. If one of the summaries contains anything other than 0 new problems, you can enter `inspxe-gui` to launch the result in the standalone GUI so you can view details and have easy access to your code editor and other tools. You can also use the `report` action to generate a report.

See Also

inspxe-cl Actions, Options and Arguments
Command Line Output
Command Syntax
Investigate Results

Working with Reports from the Command Line

Reporting Result Data from the Command Line

After performing an Intel Inspector analysis, data is collected into a result that can be viewed in the GUI or used to generate a report from the command line. By default, a report is written to `stdout` in text format, but the `inspxe-cl` command provides a number of options you can use when generating a report.

Report Types

The `report` action can be used with the following report types:

summary	Brief statement of the total number of problems detected during analysis, with a breakdown by problem type. After each `collect` or `collect-with` action, a summary report is generated by default, sent to `stdout` and also saved as an XML file in the current working directory.
status	Brief statement of the total number of problems detected in the result, with a breakdown by problem state.
problems	Detailed report of detected problem sets, including their location in the source code, such as the problem type, severity, module and line number.
observations	Detailed report of all code locations (observations) included in defined problem sets, such as the allocation site, function, module and line number.

Report Process Summary

1. Set up the `inspxe-cl` command environment.
2. Invoke the `inspxe-cl` command with the `collect` or `collect-with` action to inspect the application.
3. Invoke the `inspxe-cl` command with the `report` action to generate a report of detected problems.

Report Process Example

1. Open the command prompt window.

   NOTE
   You can open a command prompt window and use the following command to run the script that establishes Intel Inspector environment settings: `C:\"Program Files (x86)"\Intel\oneAPI\setvars.bat`.

2. In a command window, inspect the `myApp` application for memory problems and store the result in the `myRes007mi2` directory in the current working directory:

   ```
   inspxe-cl-collect mi2 -result-dir myRes007mi2 -- myApp
   ```

   NOTE
   - `mi2` is the short name for the **Detect Memory Problems** analysis type.
   - If you do not supply an absolute pathname for the application, system path environment settings are used to locate it.

3. Print to `stdout` a list of new memory problems found in the `myRes007mi2` result:

   ```
   inspxe-cl-report problems -result-dir myRes007mi2
   ```

Status Report Example

Generate a status report for the specified result and display it to standard out. `-R` is the short name of the `-report` action, and `-r` is the short name of the `-result-dir` action.

```
$ inspxe-cl -R status -r myRes007mi2
```

The output status report:

```
181 problem(s) found
15 Investigated
166 Not investigated
Breakdown by state:
13 Confirmed
2 Fixed
166 New
```

Next Step

Interpret result data.

See Also

inspxe-cl Actions, Options and Arguments
Command Line Output
Command Syntax
Saving and Formatting Reports from the Command Line
Interpreting Result Data from the Command Line

Saving and Formatting Reports from the Command Line

By default, a report is written to `stdout` in text format, but the Intel Inspector`inspxe-cl` command provides a number of options you can use when generating a report.

Saving a Report to a File

To save a report to a file, you must use the report-output option when generating a report, and specify a pathname for the output file. By default, most reports are saved as a text file, but you can also choose CSV or XML. Whichever file type you choose, filtering and sorting options are available for formatting your report.

Example

Generate a status report from the most recent result and save it as `status-report.txt` file in the current working directory.

```
$ inspxe-cl -report status -report-output status-report.txt
```

Generating a Report in CSV File Format

Use the `format` option with the *CSV* format.

If you choose the CSV format and want to use a delimiter other than the default comma, you must use the csv-delimiter option to specify either a *comma* or *tab* delimiter.

CSV Report Example

Generate an observations report in CSV format using tab delimiters, and save it as `observations.csv`.

```
$ inspxe-cl -report observations -format csv -csv-delimiter tab -report-output ./out/
observations.csv
```

Sorting Report Data

There are a pair of options that you can use to sort report data: sort-asc and `sort-desc`. Use the `sort-asc` action-option to organize a report in ascending order of the specified field(s), or use `sort-desc` to sort it in descending order. You can specify up to three different fields.

Data Sorting Example

Generate a problems report from the most recently collected result, and display the problems by line number in ascending order.

```
$ inspxe-cl -report problem -sort-asc line
```

Filtering Report Data

You can filter a report by specifying columns and values that should be included or excluded, and these can be used singly or in combination.

Syntax

```
-filter | -f <column_name> = <value>
```

To display data that equals the specified values only, run the `amplxe-cl` command with the `report` action and `filter` option as follows:

```
$ inspxe-cl -report <report_type> -filter <column_name> = <value>
```

where

- `<column_name>` is the column name (*module*, *function*, and so on)
- `<value>` is the value to include

Specify multiple filter items by using multiple `-filter` option attributes.

NOTE

- Multiple values for the same column are combined with 'OR'.
- Values for different columns are combined with 'AND'.

Report Filtering Examples

Generate a Problems report that only includes problems in the source file combine.cpp that have not yet been investigated.

```
$ inspxe-cl -report problems -filter source=combine.cpp -filter
investigated=not_investigated
```

Generate a Problems report, filtering it so that only critical problems are reported.

```
$ inspxe-cl -report problems -user-data-dir "My Inspector Results - myProjectName" -f
severity=critical
```

See Also

inspxe-cl Actions, Options and Arguments
Command Line Output
Command Syntax
About Reporting Result Data

Interpreting Result Data from the Command Line

Result data is easiest to interpret when viewed in the Intel Inspector GUI, which provides multiple views of the result and easy access to tools that you can use to manage and resolve detected problems. However, you can use the `inspxe-cl` command to generate reports that can be viewed outside the GUI and saved for future reference.

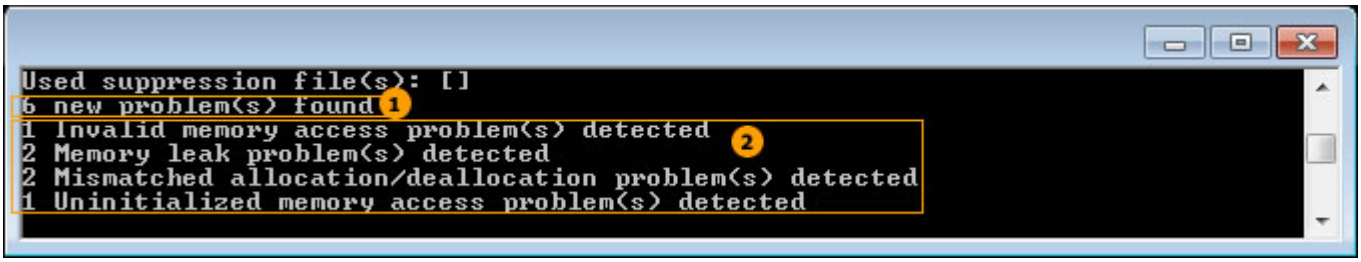
The quickest ways to launch a result are:

- With a command window open, enter `inspxe-gui`. This opens the most recently created result in the Intel Inspector GUI.
- Double click the result file. This opens the result in an IDE, if one is configured as a preference, or the Intel Inspector GUI.

Report Example: Summary

By default, a Summary report is automatically generated and saved after a result is created. Here is a Summary report displayed in `stdout`, with a breakdown by state of memory problems found in the `myRes007mi2` result:

```
2 problem(s) found
2 Not investigated
Breakdown by state:
2 New
```

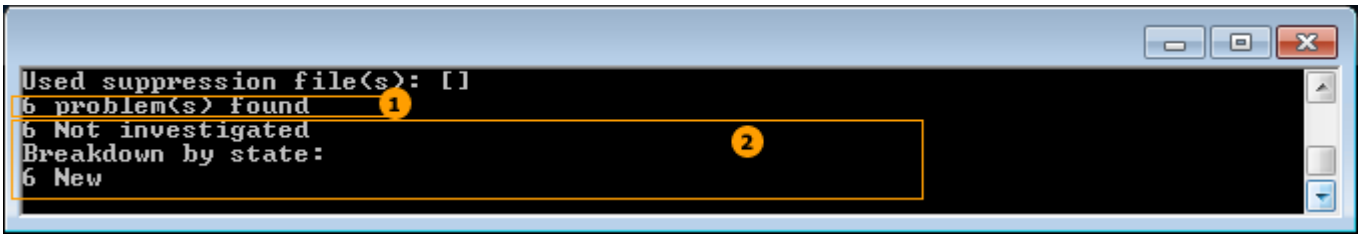


1. 6 new problem(s) found = Total new problem sets that do not match rule(s) in provided suppression file (no suppression file provided in this example)
2. New problem sets by problem type

Report Example: Status

Print to `stdout` a breakdown by state of new memory problems found in the `myRes007mi2` result:

```
inspxe-cl-report status -result-dir myRes007mi2
```

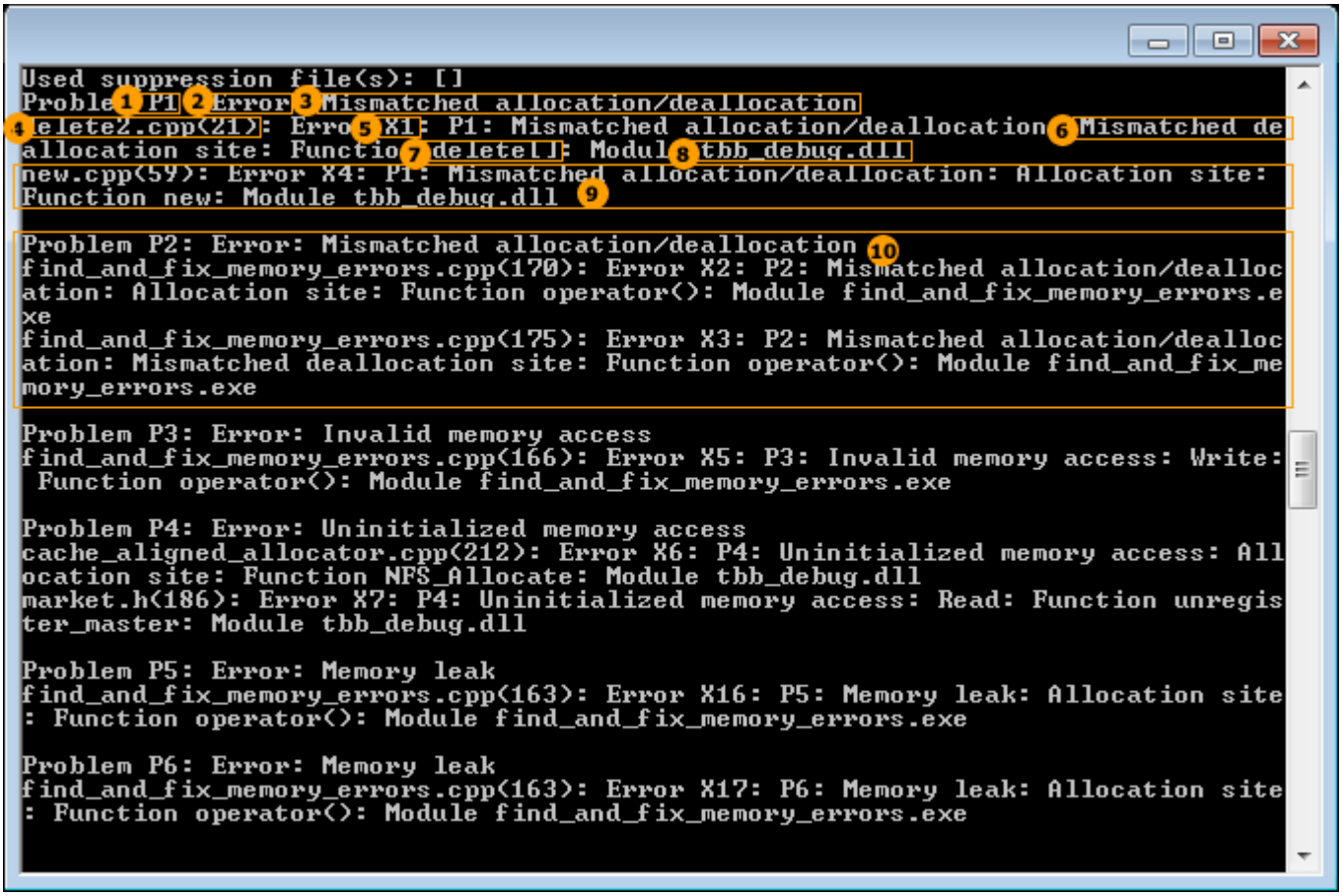


1. 6 new problem(s) found = Total new problem sets that do not match rule(s) in provided suppression file (no suppression file provided in this example)
2. New problem sets by state

Report Example: Problems

Print to `stdout` a list of new memory problems found in the `myRes007mi2` result:

```
inspxe-cl-report problems -result-dir myRes007mi2
```

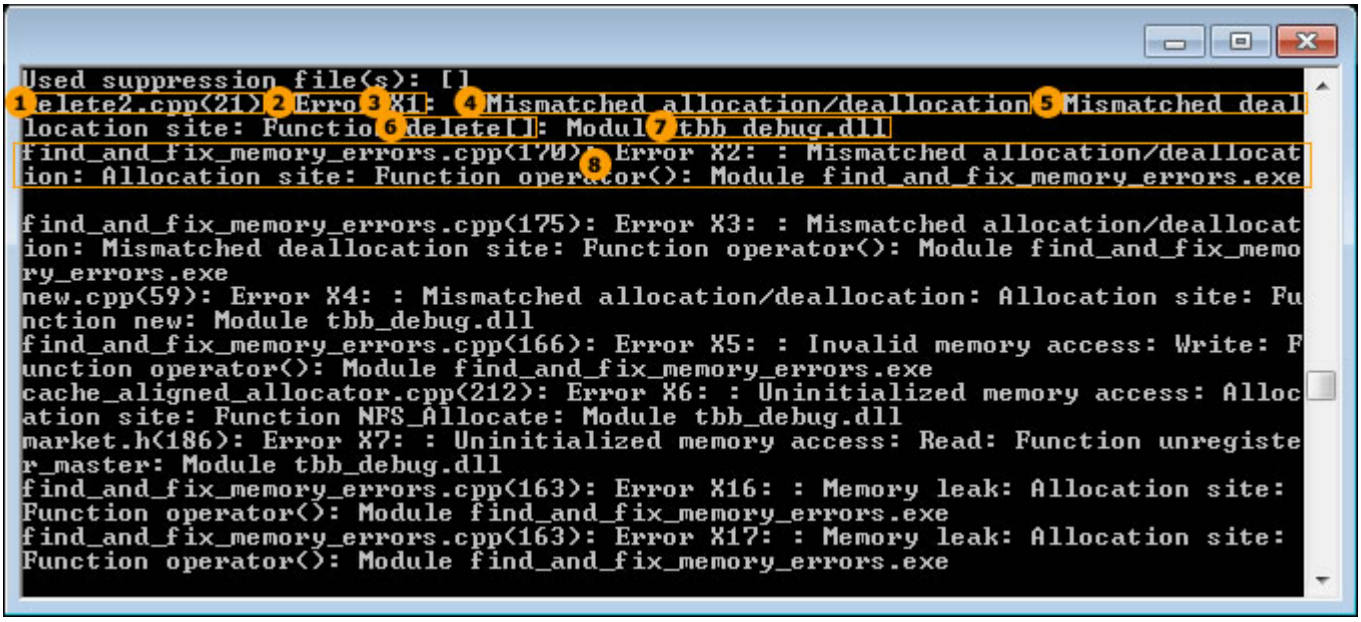


1. P1 = Unique identifier for a problem set
2. Error = Severity of the problems in the P1 problem set
3. Mismatched allocation/deallocation = Problem type of problems in the P1 problem set
4. delete2.cpp(21) = Source file location and line number of the X1 code location
5. X1 = Unique identifier for a code location in the P1 problem set
6. Mismatched deallocation site = Classification of the X1 code location
7. delete[] = Function name associated with the X1 code location
8. tbb_debug.dll = Executable or library name associated with the X1 code location
9. Information for the X4 code location in the P1 problem set
10. Information for the P2 problem set

Report Example: Code Locations

Print to `stdout` a list of code locations in new memory problems found in the `myRes007mi2` result:

```
$ inspxe-cl -report observations -result-dir myRes007mi2
```



1. delete2.cpp(21) = Source file and line number of the X1 code location
2. Error = Severity of the problem containing the X1 code location
3. X1 = Unique identifier for a code location
4. Mismatched allocation/deallocation = Problem type of the problem containing the X1 code location
5. Mismatched deallocation site = Classification of the X1 code location
6. delete[] = Function name associated with the X1 code location
7. tbb_debug.dll = Executable or library name associated with the X1 code location
8. Information for the X2 code location

Next Step

Use your development environment tools to resolve issues.

See Also

inspxe-cl Actions, Options and Arguments
Command Line Output
Command Syntax
Reporting Result Data from the Command Line

Working with Suppressions from the Command Line

Suppression files - files containing suppression rules - can be created and applied using the Intel Inspector`inspxe-cl` command tool, or using Intel Inspector GUI tools. When analysis is performed on the command line, the ability to apply suppressions is automatically enabled regardless of project or result settings.

A suppression file is a collection of suppression rules for problems identified during analysis that you want excluded from future analysis results. For example, these problems could be in code that is not your responsibility, in third-party libraries, or false positives. When a suppression file is used during a subsequent analysis run, suppressed problems are ignored so that you can focus on unsuppressed and newly detected problems.

From the command line, you can create a suppress-all file to suppress all detected problems that remain in a specified result. When the code has changed and there are new problems that you want to suppress, you can create additional suppression files through the command line for these new unfixed problems. To create or delete individual suppression rules from a suppression file, you can use the Intel Inspector GUI tools. To edit suppression rules, you can use a text editor.

NOTE
Suppression files can be used along with other problem-suppression tactics, such as assigning a problem the **Not a Problem** state, then filtering out all problems with this state when viewing problems in the GUI or generating a report on the command line. You can also propagate this state assignment forward into other results using one of the following techniques:

- To merge problem state assignments from one existing result to another, use the `merge-states` option. State assignments from the merge-from result are used to update state assignments in the merge-to result. Problem states in the merge-to result are edited according to a set of rules.
- To propagate problem state assignments forward when performing analysis, finalization or generating a report, use the `baseline-result` option.

Limitations of Suppression Rules

Intel Inspector suppression rules generally remain effective for edited source code, but when lines of code are changed within a function where a problem is marked, the problem may not be recognized the next time the suppression file is used.

This is because Intel Inspector identifies a problem based on source code lines relative to the start of a function, rather than absolute line number values. This makes it easier to track problems marked for suppression as lines of code are inserted and deleted in source files. However, adding or deleting lines within a function, prior to the location of a suppressed problem, can cause a problem to no longer match the suppression rule. In these cases, the problem is designated as a new problem in the subsequent analysis. If Inspector did not do this, suppressions would fail any time a line of code is added or removed.

Tips for Working with Suppression Files

- Suppression rules are based on results for a particular analysis type, so the same general category of analysis must be used to generate a suppression file and when using it in subsequent analysis runs. For example, you would use one set of suppression files for threading error analysis, and a different set for memory error analysis.
- Put all suppression files created for a specific type of analysis in a single, standard suppressions directory. This allows you to apply all the suppression files in this directory simply by specifying the directory path. To make it easy to switch between the GUI and the command line, store suppression files in the Microsoft Visual Studio* IDE standard suppressions directory: `C:\myProject\My Inspector Results-[project name]\suppressions`.
- If your suppression files are not in a single directory, option files provide an easy way to pass in a series of suppression file paths to a `-collect` or `-collect-with` action.

- You may use the same suppression file for multiple analysis runs, and may create and apply additional suppression files as needed.
- Each suppression rule has some overhead associated with it, so it can be worthwhile to edit the suppression file and combine multiple related rules.
- Conversion of third-party suppression files can only be performed through the command line.
- Suppressions can also be applied when generating a report.

Creating a Suppress-All File from the Command Line

When you want to suppress all problems remaining in an application, you can use the Intel Inspector`inspxe-cl` command to create a `suppress-all` file. Suppress-all files are expensive to produce and to apply, so be sure to consider other alternatives, such as problem states or manually assigning suppression rules, before using this method to filter problems.

In Intel® Inspector, you can create suppression files - files containing suppression rules - through the GUI or through the `inspxe-cl` command line interface, but the capabilities are not the same in both interfaces. GUI tools allow you manage suppression rules individually - for example, to add or delete a single suppression rule. But when you want to create suppression rules for all problems remaining in your application, known as a `suppress-all` file, you must use the command line.

Tip
Any number of suppression files can be used when performing analysis, but each suppression rule adds some overhead to the analysis process, so you will want to avoid redundantly suppressing the same code locations. Once your suppress-all file is created, you can manage it by editing suppression rules with a text editor, or adding or deleting suppression rules in the GUI. As an example, after creation, you might edit your suppress-all file in a text editor to merge multiple suppression rules into a smaller number of broader-scoped collection rules. As your code base changes, you might add suppression rules manually, and delete any that have become unnecessary. For more information on managing suppression files, see the links at the bottom of this topic.

Set Up

Before creating a suppression file, configure the **Project Properties** in the GUI to allow suppressions. This won't affect results generated from the command line, in which suppressions are always enabled, but it will affect the results of any analyses you might generate through the GUI.

Check your suppressions directory settings. To make it easy to switch between the GUI and the command line, store suppression files in the Microsoft Visual Studio* IDE standard suppressions directory: `C:\myProject\My Inspector Results-[project name]\suppressions`.

Choose a representative data set that exercises your code while minimizing unnecessary overhead.

A suppression file is created for a specific type of analysis, such as memory error or threading error analysis, so while you are working in the GUI, configure your analysis settings, and then use the Dialog Box Corresponding inspxe-cl Command Options to copy the command with the analysis type and other arguments to your clipboard, then save it to use on the command line.

Generate a Result that Only Contains Problems to Suppress

Your goal is to create a suppression file that contains only those problems that you want to suppress. To do this, you must run an analysis and fix all problems that should be fixed, and then run a second analysis so that the second result only contains those problems that you want to suppress. You will use this second result to generate your suppress-all file.

NOTE
Any problem state assignments are carried forward into the second result, but do not affect which problems are included in the suppress-all file. For example, even if the state is *Confirmed*, a suppression rule will be created for this problem in the suppress-all file.

1. Run an analysis on your application using your configured analysis settings.

Tip
If you initiate analysis through the GUI, you can view the analysis result immediately, use the **Copy the Command Line** tool to get the analysis command for use in subsequent analysis runs, and have easy access to your code editor.

2. Fix all detected problems that are unacceptable and should not remain in the application code. Be sure to fix detected errors only, so that no new problems are introduced before running the second analysis.
3. Using the same sample data set, if applicable, run your second analysis. Memory and threading analysis can be performed through the command line.

The result of this second analysis contains only those problems that you chose not to fix, so you are ready to create your suppression file.

Create the Suppress-All File

Now that your second result is ready, use the `inspxe-cl` command to create the suppress-all file.

The default name for suppression files is `default`.If you do not specify an extension, `.sup` is used. .

To make it easy to switch between the GUI and the command line, store suppression files in the Microsoft Visual Studio* IDE standard suppressions directory: `C:\myProject\My Inspector Results-[project name]\suppressions`.

The command syntax takes this general form:

```
inspxe-cl -create-suppression-file <PATHname> -result-dir <PATH>
```

For more information, see the option reference for create-suppression-file.

Next Steps

Once the suppress-all file exists for this type of analysis, you can use this suppression file in future analyses of this type, or when generating reports from results in which these problems are not already suppressed.

If you want to set up suppression files for a different kind of analysis, repeat this process. You may want to store the suppression file in a different directory.

See Also

inspxe-cl Actions, Options and Arguments
Command Syntax
Command Line Output
Applying Suppression Files from the Command Line
Collecting Result Data from the Command Line
Reporting Result Data from the Command Line
Collect Results
Investigate Results
Suppressions Support

Converting Third-Party Suppression Files from the Command Line

Convert suppression files from IBM Rational PurifyPlus*, Valgrind* or older versions of Intel® Inspector to a format compatible with Intel Inspector SP1 and above.*

You can use the `inspxe-cl` tool to convert the following types of suppression files to a format that can be used by the current version of Intel Inspector:

- IBM Rational* PurifyPlus* for Linux* OS
- Valgrind* for Linux* OS
- Previous versions of Intel Inspector for Linux* and Windows*. Old suppression files are in XML format, while new suppression files are in TXT format. Intel Inspector will load and use the older XML suppression files, so these do not have to be converted before use, but can convert these files if you wish to edit the suppression rules.

NOTE
Enter `inspxe-cl -help` to access the most current command line documentation.

Converted third-party suppression files define problems differently than Intel Inspector, so there will be differences in suppression rule structure and code locations. Incompatibilities may cause some suppression rules in the converted files to be rendered ineffective, while others may overlap suppression rules created in Intel Inspector, resulting in higher overhead.

Example: Converting a Purify Suppressions File

To convert a Purify or Valgrind suppression file, use the `convert-suppression-file` action, and specify values for `from` and `to`.

This example converts a Purify suppressions file, `my_old_suppressions.purify`, and saves it in Intel Inspector suppression file format as `my_new_suppressions.sup`.

```
$ inspxe-cl -convert-suppression-file -from .\my_old_suppression_file.purify -to
".\myResultDir\suppressions\mySup.sup"
```

Using a Converted Suppression File

A converted suppression file can be used like any other suppression file. You can check by opening the result in the GUI and applying the suppression file. Suppressed problems are crossed out when displayed in the GUI.

See Also

inspxe-cl Actions, Options and Arguments
Command Line Output
Command Syntax
Collecting Result Data from the Command Line
Reporting Result Data from the Command Line
Collect Results
Investigate Results
Suppressions Support

Applying Suppression Files from the Command Line

After creating or converting suppression files, you can apply suppression files during analysis, finalization (symbol re-resolution) or when generating a report. There are several ways to pass suppression files to the Intel Inspector`inspxe-cl` command tool, including passing them in through an option file, which can be useful when you have many suppression files.

Ideally, a suppression file should be created for a specific analysis type, as described in Creating Suppression Files.

To make it easy to switch between the GUI and the command line, store suppression files in the Microsoft Visual Studio* IDE standard suppressions directory: `C:\myProject\My Inspector Results-[project name]\suppressions`.

NOTE
Suppression files can be used along with other problem-suppression tactics, such as assigning a problem the **Not a Problem** state, then filtering out all problems with this state when viewing problems in the GUI or generating a report on the command line. You can also propagate this state assignment forward into other results using one of the following techniques:

- To merge problem state assignments from one existing result to another, use the `merge-states` option. State assignments from the merge-from result are used to update state assignments in the merge-to result. Problem states in the merge-to result are edited according to a set of rules.
- To propagate problem state assignments forward when performing analysis, finalization or generating a report, use the `baseline-result` option.

Specifying Suppression Files

Depending on the number of suppression files you want to apply and their locations, you may want to specify a single suppression file, multiple individual suppression files, or the suppressions directory, as shown in these examples. The example commands all perform analysis using the `ti2`**Detect Deadlocks and Data Races** analysis type, and the project is named myTi2Analysis.

Single Suppression File

If you want to use a single suppression file only, for example, you may have created a suppress-all file on the command line, you can specify just this one file. In this command, a suppress-all file `myTi2SuppressAllFile.sup` is specified when running an analysis.

```
$ inspxe-cl -collect ti2 -s-f "C:\myTi2Analysis\My Inspector Results-myTi2Analysis
\suppressions\myTi2SuppressAllFile.sup"
```

Example: Multiple Individual Suppression Files, Possibly with Different Paths

You may want to specify multiple individual suppression files if you have suppression files stored in multiple directories, or if you only want to apply some of the suppression files stored in your project's suppressions directory. Use one of these two syntax variations when you want to pass in multiple individual suppression files.

Specify the `suppression-file` option followed by a comma-separated string of suppression file PATH/names (no spaces).

```
$ inspxe-cl -collect ti2 -s-f "C:\myTi2Analysis\Results\suppressions
\myTi2SuppressAllFile.sup","C:\myTi2Analysis\Results\suppressions
\myTi2SuppressAllFile5.sup" -- myApp
```

Or specify the suppression-file option multiple times, each one followed by a suppression file PATH/name.

```
$ inspxe-cl -collect ti2 -s-f "C:\myTi2Analysis\Results\suppressions
\myTi2SuppressAllFile.sup"-s-f "C:\myTi2Analysis\Results\suppressions
\myTi2SuppressAllFile5.sup" -- myApp
```

NOTE
It is easiest to use an option file to pass in multiple suppression files, as explained in the next section.

Example: Multiple Suppression Files in a Single Suppressions Directory

If you have multiple suppression files in the same directory and want to apply all the suppression files in this directory, you can simply specify the directory.

```
$ inspxe-cl -collect ti2 -s-f "C:\myTi2Analysis\Results\suppressions" -- myApp
```

Using an Option File to Pass in Multiple Suppression Files

If you have multiple suppression files but cannot specify the suppressions directory as described above, you can use the `option-file` option to pass in an option file with the `suppression-file` option and arguments. The option file is a plain text file with each option and its arguments followed by a newline. As you create new suppression files, you can add them to the option file as desired.

This option file is named `myOptionFile.txt`. It specifies suppression files from three different directories, including a suppress-all file from the baseline result directory, all suppressions from a suppressions directory that holds converted suppression files, and the latest suppression file from the current result directory.

```
-s-f "C:\myTi2Analysis\BaselineResult\suppressions\myTi2SuppressAllFile.sup"
-s-f "C:\myConvertedSuppressions\"
-s-f "C:\myNewTi2Analysis\Results\suppressions\myNewTi2Suppressions.sup"
```

Now `myOptionFile.txt` is passed into the analysis process, so all the suppression files specified in `myOptionFile.txt` are applied.

```
$ inspxe-cl -collect ti2 -option-file .\myOptionFile.txt -- myApp
```

Applying Suppression Files During Analysis

Use the `collect` or `collect-with` action with the `suppression-file` option to pass in one or more suppression files.

As previously described, it is easiest to configure analysis through the Intel Inspector GUI, then use the Dialog Box Corresponding inspxe-cl Command Options to copy the command with the analysis type and other arguments to your clipboard, and then save it for use on the command line or in scripts.

In this example, the `collect` action is used with the `ti2`**Detect Deadlocks and Data Races** analysis type. The command is simplified, so all other analysis settings are left to default. This command:

- Runs a **Detect Deadlocks and Data Races (ti2)** analysis on the application `myApp`.
- Uses the short name `-s-f` for the `suppression-file` option.
- Specifies the suppression directory, rather than individual suppression files. When the directory is specified, the `inspxe-cl` command tool applies all suppression files in this directory.
- To make it easy to switch between the GUI and the command line, the suppression files were stored in the Microsoft Visual Studio* IDE standard location for suppression files: `C:\myProject\My Inspector Results-[project name]\suppressions`.

```
$ inspxe-cl -collect ti2 -s-f "C:\myProject\My Inspector Results\suppressions" -- myApp
```

Applying Suppression Files When Generating a Problems Report

If a suppression file was not used during analysis and data collection, it can be applied when generating a Problems report. Use the `report` action with the `suppression-file` option to pass in one or more suppression files, or a suppressions directory. For most complete suppression of problems, the suppression file should be based on the same analysis type as the result.

This simple report-generation command uses all default settings except that it:

- Uses a result from a **Deadlocks and Data Races (ti2)** analysis.
- Uses the short name `-s-f` for the `suppression-file` option.

- Generates a Problems report of only new, unsuppressed problems.

```
$ inspxe-cl -report problems -s-f "C:\myProject\My Inspector Results\suppressions
\myTi2Suppress-AllFile"
```

See Also

inspxe-cl Actions, Options and Arguments
Command Line Output
Command Syntax
Collecting Result Data from the Command Line
Reporting Result Data from the Command Line
Suppressions Support
Tes for Regressions: Recommended Approach

Workaround for Disabled Suppressions from the Command Line

Rules for Enabling Suppressions

The rules for enabling suppressions vary depending on whether the analysis is performed in the Intel Inspector GUI or on the command line.

When analysis is performed in the GUI	The ability to apply suppressions to the result is determined by whether the **Allow Suppressions** setting was enabled in the **Project Properties** at the time that the result was created. Once the result is created, this setting is cannot be changed. Likewise, imported results retain their original settings.
When analysis is performed on the command line	The ability to apply suppressions is automatically enabled regardless of result or project settings. You can also use suppression files when generating a report.

What to Do When Suppressions Are Disabled in a Result

If you want to apply suppressions but they are disabled in your result, you have two main options: enable suppressions in the current project and perform the next analysis on the command line, or create a new project in which suppressions are allowed and run a new analysis.

Example: Workaround for Disabled Suppressions

By performing analysis on the command line, you can create a new result that has enabled suppressions, with all the problem state assignments and annotations as the old result. You can use the Copy Command tool to capture the basic analysis settings, but should also specify the old result as the baseline and apply suppressions to the new analysis.

Before running this command, you should enable suppressions in this project so that you will have the option of applying suppressions when initiating analysis in the GUI.

In this example, the `baseline-result` option provides the path to the baseline result directory. If this is not specified, the analysis uses the most recent result of the same analysis type in the current working directory by default.

The`suppression-file` option can pass in either the parent directory where suppression files are stored, or the PATHname of a specific suppression file. In this example, the suppressions parent directory is specified, so `inspxe-cl` applies whatever suppression files are in this directory during analysis. To make it easy to switch between the GUI and the command line, store suppression files in the Microsoft Visual Studio* IDE standard suppressions directory: `C:\myProject\My Inspector Results-[project name]\suppressions`.

```
$ inspxe-cl -collect ti2 -baseline-result ./old_result_dir -suppression-file ./
suppressions/ -- myApp.exe
```

The result of this analysis has suppressions enabled, contains all the problem settings and annotations as the previous result, and shows which problems are suppressed.

See Also

inspxe-cl Actions, Options and Arguments
Command Line Output
Command Syntax
Collecting Result Data from the Command Line
Collect Results
Suppressions Support

inspxe-cl Actions, Options and Arguments

This reference provides details on all options supported by the Intel Inspector`inspxe-cl` command tool.

Command options are categorized as follows:

- **action:** The action that the command performs, such as collect data or generate a report. You must specify exactly one action per command.
- **action-option:** Defines a behavior applicable to the specified action. Only use action-options that are supported by the action, or a usage error will result. You may have one or more supported action-options per command.
- **global-option:** Defines a behavior applicable to all actions. You may have one or more global-options per command.

Action-Option Usage Rules:

- If opposing action-options are used on the same command line, the last specified action-option applies.
- An action-option that is redundant or has no meaning in the context of the specified action is ignored.
- Attempted use of an inappropriate action-option which would lead to unexpected behavior returns a usage error.

Actions

Action	Purpose
collect	Run a memory or threading error analysis and data collection.
collect-with	Run an analysis and data collection with advanced knob settings.
command	Perform an action on a running analysis.
create-suppression-file	Generate a suppression file for all problems detected in a result.
export	Export a result file, optionally including code snippets and sources, to an archive file.
finalize	Redo symbol resolution for an existing result.
help	Display up-to-date information for all available command-line actions.
import	Create a new result from a legacy Intel Inspector result or raw data file.
knob-list	List knobs for the specified analysis type.

Action	Purpose
merge	Merge problem states from one result into another result.
report	Generate a summary, list of problems, or list of code locations report.
version	Display product version information.

app-working-dir

Specify the directory where the application will be run.

GUI Equivalent

Microsoft Visual Studio* IDE: **Project** > **Intel Inspector <version> Project Properties** > **Target tab** > **Application Launch Directory**

Intel Inspector standalone GUI menu: **File** > **Project Properties...** > **Target tab** > **Application Launch Directory**

Syntax

`-app-working-dir <PATH>`

Arguments

A string containing the PATH/name for the application.

Default

Defaults to the current working directory.

Actions Modified

`collect`, `collect-with`

Description

Use the `app-working-dir` action-option when performing a `collect` or `collect-with` action to run the application in a directory other than the current working directory during profiling. The process uses the specified directory to run the application, then returns to the current working directory. This option is especially useful when the application and data files are stored in different locations.

Example

Run a memory error analysis that answers the question **Does my target have memory access problems?** . The collection process changes to `J:\myAppDirectory` to run the `myApp` application, uses source files found in the directory specified by the `search-dir` option to finalize the result, writes the result in the default result directory, and then returns to the current working directory to create the `inspxe-cl.txt` Summary report file.

```
$ inspxe-cl -collect mi2 -app-working-dir J:\myAppDirectory -search-dir src:=.\mySources -- myApp
```

See Also

Setting Target Application Properties-Advanced
Command Line Output
Command Syntax

archive-name

Provide the filename when exporting results to an archived file.

GUI Equivalent

Export Archives

Syntax

`-archive-name <PATH>`

Arguments

PATH	A string containing the PATH/name for the archive file.

Actions Modified

`export`

Description

Use `archive-name` to specify the PATH/name for the `*.inspxez` archive file, when using the `export` action to archive a result.

Example

This command exports the result in the `myRes` result directory, including code sources as well as code snippets, and saves the archive as `my-archived-result.inspxez` in the current working directory.

```
$ inspxe-cl -export -archive-name my_archived_result.inspxez -include-sources -result-dir myRes
```

See Also

Command Line Output
Command Syntax

baseline-result

Propagate states and annotations from a specified baseline result when performing analysis or generating a report.

GUI Equivalent

Dialog Box: Options-State Management

Syntax

`-baseline-result <PATH>`

Arguments

A string containing the PATH/name for the baseline result. This may be an absolute path, or a path relative to the current working directory.

Default

When performing analysis, problem states and annotations from the most recently created result of the same type are carried forward to the new result.

Actions Modified

`collect`, `collect-with`, `finalize`, `report`

Description

Use the `baseline-result` action-option to specify a result other than the most recently created result of this type as the baseline for a new collection. You can also specify a baseline when finalizing a result or generating a report. Problem states and annotations from the specified baseline result are carried forward to the new result, newly finalized result, or report.

> **NOTE**
> Whatever type of analysis is used to create the baseline result, the results will align most closely if the same type is used when generating a subsequent result. A similar type can also be used, though an increased or reduced level of analysis will create some mismatches.

Example

In this example, the *mi3*, **Locate Memory Problems**, analysis type is used when analyzing `MyApp` and for subsequent analysis runs that use this baseline result.

To create the result that you want to use as a baseline, perform a memory analysis on `MyApp.exe` and save the result in the `mi3_baseline` result directory.

```
$ inspxe-cl -collect mi3 -result-dir ./mi3_baseline -- ./MyApp.exe
```

For a subsequent analysis run, the `baseline-result` option passes in the baseline result from the `mi3_baseline` result directory that was created in the previous step. Since the result directory for the new analysis result was not specified, it is written to a default-named result directory under the current working directory.

```
$ inspxe-cl -collect mi3 -baseline-result ./mi3_baseline -- ./MyApp.exe
```

See Also

States
Command Line Output
Command Syntax
Investigate Problems Using Interactive Debugging

collect

Run the specified type of analysis and collect data.

GUI Equivalent

Pane: Analysis Type-Memory Errors

Pane: Analysis Type-Threading Errors

Syntax

`-collect <analysis_type>`

`-c <analysis_type>`

Tip

- Both short names and long names are case-sensitive. For example, `-c` is the short name of the `collect` action, and `-C` is the short name of the `command` action.
- To duplicate an analysis performed in the Intel Inspector GUI: Use the **Command Line** button on the **Analysis Type** window panes to copy the exact command to the clipboard.

Arguments

Memory error analysis types

Analysis type code	Description
`mi1`	Memory error analysis that answers the question **Does my target leak memory?**
`mi2`	Memory error analysis that answers the question **Does my target have memory access problems?**
`mi3`	Memory error analysis that answers the question **Where are the memory access problems?**

Threading error analysis_types

Analysis type code	Description
`ti1`	Threading error analysis that answers the question **Does my target have deadlocks?**
`ti2`	Threading error analysis that answers the question **Does my target have deadlocks or data races?**
`ti3`	Threading error analysis that answers the question **Where are the deadlocks or data races?**

Modifiers

`app-working-dir`, `baseline-result`, `executable-of-interest`, `knob`, `module-filter`, `module-filter-mode`, `no-auto-finalize`, `no-summary`, `option-file`, `quiet`, `result-dir`, `return-app-exitcode`, `search-dir`, `suppression-file`, `user-data-dir`, `verbose`

Description

Use the `collect` action to run a memory or threading error analyis.

- The result is stored in the specified or default-named `r@@@{at}` result directory in the current working directory, where `@@@` represents the next available number and `{at}` represents the analysis type with preset configuration.
- Finalization occurs automatically unless overridden.
- By default, a Summary report is displayed to `stdout` and written to a .txt file in the current working directory.

Example

This command:

- Runs a **Detect Deadlocks and Data Races (ti2)** analysis on the application `myApp`.
- Stores the result in the default `r@@@ti2` result directory in the current working directory, where `@@@` represents the next available number.

- Generates a summary report of detected problems, and writes it to the `inspxe-cl.txt` file in the result directory.

```
$ inspxe-cl -collect ti2 -- myApp
```

This command performs the same type of analysis as that above, but exits without finalizing the result or generating a summary report.

```
$ inspxe-cl -c ti2 -no-auto-finalize -- myApp
```

See Also

Command Line Output
Command Syntax
Memory Error Analysis Types
Threading Error Analysis Types

collect-with

Run the collector with specified knob settings to perform analysis and data collection.

GUI Equivalent

Dialog Box: Custom Analysis

Syntax

`-collect-with <collector> -knob <knob-name>=<knob-value>`

Arguments

Collector values are:

runmc	Memory errors
runtc	Threading Errors

NOTE
Use the command `inspxe-cl -knob-list <collector>`, where `<collector>` is one of these two collector values, to see which knob values are available for this collector.

Modifiers

`app-working-dir`, `baseline-result`, `executable-of-interest`, `knob`, `module-filter`, `module-filter-mode`, `no-auto-finalize`, `no-summary`, `quiet`, `result-dir`, `return-app-exitcode`, `search-dir`, `suppression-file`, `user-data-dir`, `verbose`

Description

Use the `collect-with` action to run the specified type of analysis and data collection using the specified knob settings. Specify the collector, knob name and value, and target.

Example

This command:

- Performs memory error analysis on `myApp`.
- Uses the `detect-memory-leaks` knob setting.

```
$ inspxe-cl -collect-with runmc -knob detect-memory-leaks=true -- myApp
```

See Also

Memory Error Analysis Types
Threading Error Analysis Types
Command Line Output
Command Syntax

command

Perform the specified action on a running analysis.

GUI Equivalent

Toolbar: Command

Syntax

`-command <value>`

`-C <value>`

NOTE
Both short names and long names are case-sensitive. For example, `-c` is the short name of the `collect` action, and `-C` is the short name of the `command` action.

Arguments

Possible values are:

stop	Terminate target execution and collection for a threading or memory error analysis.
reset-leak-tracking	Reset the start point for finding memory leaks in the currently running memory error analysis. The default start point is the application start.
find-leaks	Search for new memory leaks since the currently running memory error analysis started (default) or since the last reset. View findings - in the form of zero, one, or more **Memory leak** problems - in the Problem report or GUI **Problems** pane. See *Description* section for prerequisites.
reset-growth-tracking	Reset the start point for measuring memory growth in the currently running memory error analysis. The default start point is the application start.
measure-growth	Measure memory growth since the currently running memory error analysis started (default) or since the last reset. View findings - in the form of one **Memory growth** problem - in the Problem report or GUI **Problems** pane. See *Description* section for prerequisites.

Default

The action is performed in the current working directory.

Modifiers

`option-file`, `quiet`, `result-dir`, `user-data-dir`, `verbose`

Description

NOTE
If you use the `result-dir` option to specify the result directory during analysis and data collection, you must use it when issuing these commands.

Use the `command` action to:

- Terminate the currently running threading or memory error analysis.
- Find memory leaks on demand in the currently running memory error analysis.
- Measure memory growth in the currently running memory error analysis.

On-demand Memory Leak and Memory Growth Detection

Prerequisites: On-demand memory leak detection and memory growth detection commands are valid only for the `mi2` or `mi3` analysis types. The knobs that support on-demand memory leak detection and memory growth detection are enabled in these analysis types by default: `enable-on-demand-leak-detection` and `enable-memory-growth-detection`.

Use `command` with `find-leaks` and `reset-leak-tracking` to gather memory leak information while an application is running. This is useful if:

- An application does not terminate (such as a server process).
- You want memory leak information, but you do not want to wait for an application to terminate.
- You want to determine if memory is leaked during a specific interval of application execution, or during a specific user action.
- You want to discard information about allocations performed during initialization as a way of filtering out allocations that are not currently of interest.

Use `command` with `measure-growth` and `reset-growth-tracking` to ensure an application uses no more memory than expected. This includes:

- Memory an application has allocated and still needs for future calculations
- Memory an application has allocated and no longer needs, but has not deallocated
- Memory an application has allocated and then leaked

For more precision, consider using these commands in tandem with APIs for custom memory allocation.

Examples

Terminate the analysis and data collection process that is currently running and storing the result in the `myRes` result directory.

```
$ inspxe-cl -command stop -result-dir myRes
```

After starting an analysis on the command line that is storing a result in the `myRes` result directory, measure memory growth from application start to this point. Note that the result directory used when starting the collection must be specified in all subsequent commands operating on that result. The command has no effect if `myRes` does not match a result directory for a currently running collection.

```
$ inspxe-cl -command measure-growth -result-dir myRes
```

See Also

Command Line Output
Command Syntax
Find Memory Leaks On Demand
Measure Memory Growth
Stop Analysis
APIs for Custom Memory Allocation

convert-suppression-file

Convert a third-party suppression file for use with Intel Inspector.

GUI Equivalent

None

Syntax

`-convert-suppression-file -from <PATH> -to <PATH>`

Arguments

from <PATH>	The PATH/name for the third-party suppression file (input) that is to be converted. This can be either a Purify or a Valgrind suppression file.
to <PATH>	The PATH/name for the new suppression file that is output after conversion to the Intel Inspector suppression file format. If no file extension is specified for the output suppression file, `.sup` is used.

This may be an absolute path, or a path relative to the current working directory. Both the `from` and `to` paths must be specified.

Modifiers

`option-file`, `quiet`, `result-dir`, `search-dir`, `user-data-dir`, `verbose`

Description

Use the `convert-suppression-file` action to convert a Rational Purify* or Valgrind* suppression file to the Intel Inspector suppression file format. Both the `from` and `to` pathnames must be specified. The default file extension is `.sup`.

Example

Convert the Purify suppressions file `my_old_suppressions.purify` and save it in Intel Inspector suppression file format as `"C:\myAppProj\My Inspector Results-[project name]\suppressions\"my_new_suppressions.sup`.

```
 $ inspxe-cl -convert-suppression-file -from my_old_suppressions.purify -to "C:\myAppProj\My Inspector Results-[project name]\suppressions\mySup"mySup
```

See Also

Command Line Output
Command Syntax
Working with Suppressions from the Command Line
Set Project Properties-Advanced

create-suppression-file

Generate a suppression file for all detected problems in the specified result.

GUI Equivalent

None

Syntax

`-create-suppression-file <PATH>`

Arguments

A string containing the PATH/name for the suppression file. This may be an absolute path, or a path relative to the current working directory. If no file extension is specified, `.sup` is used.

Modifiers

`option-file`, `quiet`, `result-dir`, `search-dir`, `suppression-file`, `user-data-dir`, `verbose`

Description

Use the `create-suppression-file` action to generate a suppress-all file for every problem detected in the specified result, regardless of problem state settings. The most recent result in the current working directory is used by default, or you can use the `result-dir` action-option to specify the result.

Once you have generated this suppression file, it can be applied when running analysis from the command line or GUI, and when generating a report from a result in which these problems were not suppressed.

Example

Generate a `mySup.sup` suppression file for all detected problems in the specified `r002ti2` result, and saves it to the standard Microsoft Visual Studio* IDE location for suppressions. This suppressions file can be applied to subsequent threading analysis, or when generating a report from an

```
 $ inspxe-cl -create-suppression-file "C:\myAppProj\My Inspector Results-[project name]
\suppressions\mySup" -result-dir r002ti2
```

See Also

Command Line Output
Command Syntax
Working with Suppressions from the Command Line
Creating a Suppress-All File from the Command Line
Suppressions Support
Defining Suppression Rules in the GUI
Set Project Properties-Advanced

csv-delimiter

Specify the delimiter when generating a report in tabular format.

Syntax

`-csv-delimiter <string>`

Arguments

comma | tab

Default

comma

Actions Modified

`report` with `format`

Description

Use the `csv-delimiter` action-option to specify a delimiter when generating a report in CSV file format.

Example

This example generates an observations report with tab delimiters and saves it as the file `observations.csv`.

```
$ inspxe-cl -report observations -format csv -csv-delimiter tab -report-output ./out/
observations.csv
```

See Also

Command Line Output
Command Syntax

executable-of-interest

Specify which executable to analyze.

GUI Equivalent

Dialog Box Project Properties-Target Child Application field

Syntax

`-executable-of-interest <executable-name>`

Arguments

`executable-name`	Name of executable to analyze for errors.

Default

Data is collected on the specified target and all its child processes.

Actions Modified

`collect`

Description

Use the `executable-of-interest` action-option to specify an executable for the `collect` action. This is useful when the target, for example a script, launches additional executables and you only want to run the analysis on a particular executable (child application).

NOTE Multiple processes cannot be analyzed in a single analysis run.

Example

This command runs a leak detection analysis. The target `startApp` is run, but data collection does not begin until `startApp` launches the child application `myCounter`. Only leaks detected in the `myCounter` process are saved in the analysis result.

```
$ inspxe-cl -collect mi1 -executable-of-interest myCounter -- startApp
```

Command Line Output
Command Syntax

export

Export collected results and store as an archived file.

GUI Equivalent

Exporting Archives

Syntax

`-export`

Arguments

None

Modifiers

`archive-name`, `include-snippets`, `include-sources`, `result-dir`

Description

Use the `export` action to export a result for archiving as an *.inspxez file. Use with the `archive-name` action-option.

Example

This command exports the result in the `myRes` result directory, including code sources as well as code snippets, and saves the archive as `my-new-archived_result.inspxez` in the current working directory.

```
$ inspxe-cl -export -archive-name my-new-archived_result.inspxez -include-sources -result-dir
myRes
```

See Also

archive-name
action-option
Command Line Output
Command Syntax

filter

Specify filtering for a report.

GUI Equivalent

Toolbar Filter

Syntax

`-filter <filter_name>=value`

`-f <filter_name>=value`

Arguments

`function=value`	Include only problems with the specified function name(s).
`id=value`	Include only problems with the specified ID(s).
`investigated=value`	Set whether to include only problems that are investigated or not. Valid values are: `not_investigated | investigated`.

`line=value`	Include only problems with the specified line number(s).
`module=value`	Include only problems from the specified module(s).
`problem=value`	Include only problems with the specified problem type(s).
`severity=value`	Include only problems with the specified severity(s). Valid values are `remark | informational | caution | warning | error | critical`
`source=value`	Include only problems with the specified source file name(s).
`state=value`	Include only problems with the specified state(s).

Default

Unfiltered.

Actions Modified

`report`

Description

Use `filter` when you want to generate a report that only includes problems that meet your specified criteria.

Example

Generate a Problems report, filtering it so that only critical problems are reported.

```
$ inspxe-cl -report problems -user-data-dir "My Inspector Results - myProjectName" -f
severity=critical
```

Generate a Problems report that only includes problems in the source file `combine.cpp` that have not yet been investigated.

```
$ inspxe-cl -report problems -filter source=combine.cpp -filter investigated=not_investigated
```

See Also

sort-asc Sort report data by specified field(s) in ascending order.
sort-desc Sort report data by specified field in descending order.
About Reporting Result Data
Command Line Output
Command Syntax

finalize

Resolve symbols and form problems sets to finalize an existing result.

Syntax

`-finalize`

`-I`

Arguments

None

Modifiers

`baseline-result`, `option-file`, `quiet`, `result-dir`, `search-dir`, `user-data-dir`, `verbose`

Description

Finalization, or symbol resolution, uses debug information from binary files to resolve symbol information and form problem sets.

You can use the `finalize` action to:

- Finalize an un-finalized result.
- Re-finalize a result when the previous finalization did not include all necessary symbol information. Any previously resolved symbol information is discarded, and a new symbol resolution is performed, using symbol information from default locations. If the necessary symbol information is not found, the missing symbol information is retrieved from any specified search directories.

> **Important**
> To be sure you do not inadvertently use the wrong result, use the `result-dir` option to specify the result directory. If the `result-dir` option is not used, the most recently generated result under the current working directory is used by default.
>
> Intel Inspector does not check the validity of the module binaries, so improper finalization may result if the <option>finalize</option> action is used without specifying the same module binaries that were used to collect the result. If the necessary symbol information is not stored in a default location, use the `search-dir` option to specify the directory containing the binaries. Symbol information from the specified search directories is used only if the necessary information is not found in the default locations.
>
> The first time an unfinalized result is opened or loaded, the result is automatically finalized. So before opening an unfinalized result in the GUI, check settings in the module binaries containing symbol information are a result is opened through the GUI, by double-clicking, or if a report is generated without specifying the module binaries, improper finalization may result.

Finalization normally occurs automatically:

- When analysis is initiated through the GUI, finalization occurs immediately after the result is generated, as the Summary displays in the standalone GUI or Microsoft Visual Studio* IDE.
- When using the `collect` and `collect-with` actions to initiate analysis on the command line, finalization is performed automatically towards the end of the collection and result generation process, before the Summary is displayed. However, automatic finalization can be suppressed if the `no-auto-finalize` option is used with these actions.
- When the `report` action is used to generate a report from a non-finalized result. For example, when the `no-auto-finalize` option was used when collecting the result on the command line.
- When an unfinalized result is opened in the GUI.

Use the `finalize` action when:

- The `no-auto-finalize` option was used during collection and you want to specify the search directory with symbol information.
- The result was only partially resolved, and you want to redo symbol resolution using a more complete set of symbol information.
- There was an issue with symbol resolution, possibly because the `search-dir` option did not point to the appropriate directory.
- Unknown symbols and numbers appear when a report is generated from a result, or when a result is viewed in the standalone GUI or Microsoft Visual Studio* IDE. This can happen when incorrect symbol information was used during the original analysis or when finalization did not complete properly.

NOTE
For proper finalization, please make sure to use the <option>finalize</option> action
and specify the same module binaries you used to collect the result.

Alternate Options

Finalization normally occurs as part of the `collect` or `collect-with` actions, the first time that a result is opened in the Microsoft Visual Studio* IDE, or when a report is generated from a non-finalized result.

Example

This command uses the `collect` action with the `mi1` analysis type, so a **Detect Leaks** memory error analysis is performed to answer the question **Does my target leak memory?**. The `no-auto-finalize` option is used to suppress finalization, and the un-finalized result is written to the specified `myRes@@@mi1` result directory under the current working directory. The @@@ counter auto-increments, starting with 000, so if this were the first result directory created under the current working directory using this naming format, the result directory would be named `myRes000mi1`.

```
$ inspxe-cl -collect mi1 -no-auto-finalize -result-dir myRes@@@mi1
```

To finalize the previous un-finalized result, use the `result-dir` option to specify the `myRes000mi1` result directory. Since the result was never finalized, use the `search-dir` option to pass in non-default search directories containing the symbol information.

```
$ inspxe-cl -finalize -result-dir myRes000mi1 -search-dir bin:=nonstandardDirA,nonstandardDirB
```

See Also

collect
action
collect-with
action
Command Line Output
Command Syntax

format

Specify the output format for a report.

Syntax

`-format <string>`

Arguments

`text`	Output in plain text format.
`csv`	Output in tabular format, using a comma delimiter.
`xml`	Output in XML format.

Default

Plain text.

Actions Modified

`report`

Description

Use the `format` action-option to set the format of a report and write to a file.

By default, a report is written to standard output in text format, but the `inspxe-cl` tool provides a number of options you can use when generating a report.

- To save a report to a file, use the report-output option.
- Use the `format` option to choose a report format: Text, CSV, or XML. If you choose the CSV format and want to use a delimiter other than the default comma, use the csv-delimiter option to specify the delimiter.
- To sort and filter report content, use the `sort-asc`, `sort-desc` and `filter` options.

Example

Generates a Problems report in XML format for the most recently created result and saves it in the result directory as `myProblemReport.xml`.

```
$ inspxe-cl -report problems -report-output myProblemReport -format xml
```

This command generates a Problems report in XML format for the most recently created result and saves it in the result directory as `myProblemReport.xml`.

```
$ inspxe-cl -report problems -report-output myProblemReport -format xml
```

See Also

Command Line Output
Command Syntax

help

Display explanations of command line arguments.

Syntax

`-help` *`[action_name]`*

`-h` *`[action_name]`*

`-?` *`[action_name]`*

Arguments

`action_name`	Optional. The name of the action for which you want more detailed information.

Modifiers

None

Description

Use the `help` action to display a list of actions, or specify an action to display more detailed information, including the list of options appropriate for use with this action, and usage examples.

Example

This command displays all available help.

```
$ inspxe-cl -help
```

This command displays help for the `collect` action.

```
$ inspxe-cl -? collect
```

See Also
Command Line Output
Command Syntax

include-snippets
Include snippets when exporting results for archiving.

Syntax
```
-include-snippets
```

Arguments
None

Actions Modified
`export`

Description
Use `include-snippets` with the `export` action to include code snippets when exporting a result for archiving as an `*.inspxez` file.

Example
This command exports the result in the `myRes` result directory, including code snippets, and saves the archive as `my-archived-result.inspxez` in the current working directory.

```
$ inspxe-cl -export -archive-name my-archived-result.inspxez -include-snippets -result-dir myRes
```

See Also
include-sources
  action-option
Command Line Output
Command Syntax

include-sources
Include sources and snippets when exporting results for archiving.

Syntax
```
-include-sources
```

Arguments
None

Actions Modified
`export`

Description
Use `include-sources` with the `export` action to include code sources as well as snippets when exporting a result for archiving as an `*.inspxez` file.

Example

This command exports the result in the `myRes` result directory, including code sources as well as code snippets, and saves the archive as `my-archived-result.inspxez` in the current working directory.

```
$ inspxe-cl -export -archive-name my-archived-result.inspxez -include-sources -result-dir myRes
```

See Also

include-snippets
action-option
Command Line Output
Command Syntax

import

Import data collected using legacy Intel products.

GUI Equivalent

Window Import Result

Pane Import Result

Syntax

`-import <PATH>`

Arguments

`PATH`	PATH/name of result directory containing the result to import.

Modifiers

`option-file`, `quiet`, `result-dir`, `verbose`

Description

Use the `import` action to translate archived data files, or files generated with legacy Intel products. Supported file types are: `.insp`, `.pdr`, `.inspxez`.

> **NOTE** Legacy formats do not contain all the data fields that are included in the newer result files, so some data fields may be empty in reports generated from imported results.

Example

This command imports an archived result and generates a new result in the directory `myImportedResult`. The result in `myNewResult` can be opened in the GUI or used to generate reports.

```
$  inspxe-cl -import myArchivedResult.inspxez -result-dir myImportedResult
```

See Also

report action
Command Line Output
Command Syntax

knob

Specify additional settings for a collect or collect-with action.

GUI Equivalent

Pane: Analysis Type-Memory Errors

Pane: Analysis Type-Threading Errors

Dialog Box: Custom Analysis

Syntax

`-knob <knob-name>=<knob-value>`

`-k <knob-name>=<knob-value>`

Arguments

Available knobs and values are dependent on the specified analysis type. For a complete list, use the `knob-list` action.

<knob-name>	**Description**
`analyze-stack=true | false` Default: `false`	Analyze invalid and uninitialized accesses to thread stacks. Enabling this setting is useful when: • You want as thorough an analysis as possible. • An application calls `alloca()`. High cost. Recommendation: • Enable the first time you analyze an application and periodically thereafter. • Enable to analyze automatic variables. Supported analysis: `mi3`
`detect-invalid-accesses=true | false` Default: `true`	Detect problems where an instruction reads or writes an uninitialized or invalid memory location. Enabling this setting is useful when an application: • Exhibits unexpected behavior. • Shows evidence of uninitialized values in computations. High cost. Recommendation: Enable. Supported analysis: `mi2, mi3`
`detect-leaks-on-exit=true | false` Default: `true`	Enable this setting to report typical memory leaks in which the application allocates a memory block, never releases it, and doesn't keep a pointer to the block (e.g. **unreachable memory blocks**). Enabling is useful when an application: • Runs out of memory. • Appears to be using more memory than expected. Extremely low cost - especially if used only with the **Remove duplicates** checkbox selected.

<knob-name>	Description
	Recommendation: Enable. Supported analysis: `mi1, mi2, mi3`
`detect-resource-leaks=true | false` Default: `true`	Enable this setting to detect open kernel and GDI handles when the application ends. For example, the application may open a file, get its handle, but never close or release that handle until it stops running. On Windows, GDI resources are limited, and the application may experience some drawing issues if it uses more than ~10,000 of these types of handles at once (pen, bitmap, brush, etc.). Enabling is useful when analyzing Windows* GUI applications. Low cost. Recommendation: Enable the first time you analyze an application and periodically thereafter. Supported analysis: `mi1, mi2, mi3`
`enable-memory-growth-detection=true | false` Default: `true`	Set to true to enable buttons in the GUI that let you send commands during application execution. This will show you a list of **reachable and unreachable memory blocks** for a time segment. Enabling is useful for modeling memory usage patterns and ensuring a transactional application deallocates all memory allocations after a transaction completes. (Use in conjunction with the **Reset Leak/Growth Detection** button and **Show Leaks/Growth Now** button during analysis.) Low cost. Supported analysis: `mi1, mi2, mi3`
`offload-target=default | cpu` Default: `default`	Use this argument to set up the device on which your application executes during offloaded code analysis. • `default` value does not change device settings and application execution control is defined by profiling process • `cpu` value forces offloaded code regions to execute on a CPU during offloaded code analysis Supported analysis: `mi1, mi2, mi3, ti1, ti2, ti3`
`filter-guaranteed-atomics=true | false` Default: `false`	Enable this setting to skip reporting races on guaranteed atomic operations on Intel® P6 processor family or newer. For details, please refer to the *Guaranteed Atomic Operations* section of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. Use with caution, other architectures might have another policy on atomic operations. Supported analysis: `ti2, ti3`
`filter-guaranteed-atomics=true | false` Default: `false`	Enable this setting to skip reporting races on guaranteed atomic operations on Intel® P6 processor family or newer. For details, please refer to the *Guaranteed Atomic Operations* section of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. Use with caution, other architectures might have another policy on atomic operations.

<knob-name>	Description
	Supported analysis: `ti2, ti3`
`remove-duplicates=true | false` Default: `true`	Enable this setting to avoid showing duplicate occurrences of a detected problem in the **Code Locations** pane. Disabling is: • Useful when you need to fully visualize all threads and problem occurrences in relation to time • Low cost in terms of time; however, the number of duplicate errors could crowd out the number of unique errors. Recommendation: Enable. Supported analysis: `mi3, ti3`
`scope=normal | extreme` Default: `normal`	Normal/normal=Set memory access byte granularity to 4 bytes, do not detect data races on stack accesses, and defer memory check. High cost. Extremely thorough/extreme=Set memory access byte granularity to 1 byte, detect data races on stack accesses, and do not defer memory check. Extremely high cost. Supported analysis: `ti3`
`stack-depth=<value>` Available values are: `1 8 16 24 32` Default: `16`	Provide more or less call stack context for detected errors. A high setting is useful when analyzing highly object-oriented applications. A higher number does not significantly impact cost. Recommendation: Use only as large a value as an application requires to display complete call paths. Supported analysis: `mi1, mi2, mi3, ti1, ti2, ti3`
`still-allocated-memory=true | false` Default: `true`	Detect problems where a still-reachable block of memory is allocated but not released when the application stops executing. Cost is proportional to the number of memory blocks still allocated when the application stops executing. Recommendation: Enable this setting to investigate memory growth. Supported analysis: `mi1, mi2, mi3`
`terminate-on-deadlock=true | false` Default: `false`	Stop analysis and application execution if the Intel® Inspector detects a deadlock. Enabling this setting is useful when running your application as part of a kernel or unit testing suite. Low cost. Recommendation: Disable. Instead, use the corresponding knob in the command line interface to perform kernel or unit testing in a nightly scenario. If the Intel® Inspector identifies a deadlock, decide if it is appropriate to continue analysis. Supported analysis: `ti1, ti2, ti3`
`use-maximum-resources=true | false` Default: `false`	Potentially detect more data race and cross-thread stack access errors; however, do not optimize memory consumption and performance.

<knob-name>	Description
	High cost. Recommendation: Disable this setting to find most of your threading problems. After you find and fix the problems, enable the setting for more complete coverage. Supported analysis: `ti3`

Actions Modified

`collect`, `collect-with`

Description

Use the `knob` action-option when you want to specify an additional setting for a `collect` or `collect-with` action. For a full list of available knobs, use the `knob-list` action.

Example

This command uses the `collect-with` action to perform memory error analysis using the `detect-memory-leaks` knob on `myApp`.

```
inspxe-cl -collect-with runmc -knob detect-memory-leaks true -- myApp
```

See Also

knob-list
action
Managing Custom Analysis Types
Command Line Output
Command Syntax

knob-list

Display configurable settings for a collect or collect-with action.

GUI Equivalent

Pane: Analysis Type-Memory Errors

Pane: Analysis Type-Threading Errors

Dialog Box: Custom Analysis

Syntax

For the `collect` action:

`-knob-list <analysis_type>`

For the `collect-with` action

`-knob-list <collector>`

Actions Modified

`collect`, `collect-with`

Description

Use `knob-list` to display arguments to the `knob` option, based on the value of the analysis type or collector. Each knob represents a predefined setting that is available at this level of analysis.

For the `collect` action, the analysis type values are:

mi1 | mi2 | mi3	Use any one of these three different levels of memory error analysis, from narrowest to widest scope. Displays a list of knobs for this analysis type and level.
ti1 | ti2 | ti3	Use any one of these three different levels of threading error analysis, from narrowest to widest scope. Displays a list of knobs for this analysis type and level.

The `collect-with` action is more powerful, and has more settings available. For the `collect-with` action, the collector values are:

runmc	List knobs for memory error checking, with a brief description of each.
runtc	List knobs for threading error checking, with a brief description of each.

Example

Display a list of knobs for the `mi3` analysis type for the `collect` action.

```
$ inspxe-cl -knob-list mi3
```

Display a list of knobs for the `runtc` collector for the `collect-with` action.

```
$ inspxe-cl -knob-list runtc
```

See Also

collect
  action
collect-with
  action
knob action-option
Managing Custom Analysis Types
Command Line Output
Command Syntax

merge-states

Merge problem states from a result into problem states in another copy of the same result.

GUI Equivalent

Dialog Box Merge States

Syntax

`-merge-states <PATH>`

Arguments

The path for the merge-from result directory.

Default

Problem state values from the specified result directory are merged into state values in the most recently created result directory of the same type.

Modifiers

`option-file`, `result-dir`, `user-data-dir`

Description

When an analysis returns a large number of problems, you can create multiple copies of the same result to divide up the work of investigating and resolving problems. Then you can use the `merge-states` action to merge problem states from one copy of the result to another. When merging problem states on your system, state data from the specified result is merged into the most recent result of the same type in your current working directory, unless the *merge-to* result directory is specified by the `result-dir` option.

Problem states are merged according to these rules:

- `New` in the merge-from result is ignored; state value in the merge-to result is retained.
- `Not Fixed` in the merge-from result is ignored; state value in the merge-to result is retained.
- `Regression` in the merge-from result is propagated unless the state value in the merge-to result is `Not a Problem`, `Deferred`, `Confirmed` or `Fixed`.
- `Not a Problem` in the merge-from result is propagated unless the state value in the merge-to result is `Deferred`, `Confirmed` or `Fixed`.
- `Deferred` in the merge-from result is propagated unless the state value in the merge-to result is `Confirmed` or `Fixed`.
- `Confirmed` in the merge-from result is propagated unless the state value in the merge-to result is `Fixed`.

Example

Merge problem state values from the `merge-from-result` result directory into state values in the most recent result of the same type.

```
$ inspxe-cl -merge-states J:\myDirectory\merge-from-result
```

In this example, problem state data from the `merge-from-result` result directory is merged into state data in the result in the `merge-to-result` directory, as specified by the `result-dir` option.

```
$ inspxe-cl -merge-states J:\myDirectory\merge-from-result -result-dir ./merge-to-result
```

See Also

State Merge
Command Line Output
Command Syntax

module-filter

Set a filter to specify which application modules to analyze.

GUI Equivalent

Dialog Box Project Properties-Target

Syntax

`-module-filter <string>`

Arguments

A string specifying the name of the application(s), or child application(s), to analyze. Multiple names can be specified as a string of comma-separated values, with no spaces.

Default

Analyze and collect data on all modules.

Actions Modified

collect, collect-with

Description

Use the module-filter action-option to filter which application modules to include or exclude during collection.

- Specify which modules to include or exclude.
- Specified modules are included by default.
- To filter out the specified modules, use with the module-filter-mode option and specify exclude.

> **NOTE**
> See the module-filter-mode Option Reference for an example of how to exclude files.

Example

This command performs a memory analysis on myApp, but collects data only from the specified module childApp.

```
$ inspxe-cl -collect mi1 -module-filter childApp --myApp
```

See Also

Command Line Output
Command Syntax
module-filter-mode
action-option
suppression-file action-option
Defining Suppression Rules in the GUI
Set Project Properties-Advanced

module-filter-mode

Exclude filtered application modules from analysis.

GUI Equivalent

Dialog Box Project Properties-Target

Syntax

-module-filter-mode *exclude* | *include*

Arguments

A string whose value is either exclude or include.

Default

Application modules specified through the module-filter option are included in analysis and data collection.

Actions Modified

collect, collect-with

Description

When using the `module-filter` option to filter applications, the specified applications are included by default. When you want to exclude applications rather than including them, use `module-filter-mode` and set the value to *exclude*.

Example

Perform a memory analysis on `myApp`, but exclude data from `childApp`.

```
$ inspxe-cl -collect mi1 -module-filter childApp -module-filter-mode exclude -- myApp
```

See Also

module-filter
  action-option
Setting Project Properties-Advanced
Command Line Output
Command Syntax

no-auto-finalize

Perform collection without finalizing the result.

Syntax

`-no-auto-finalize`

`-auto-finalize`

Default

auto-finalize - Results are finalized during normal `collect` and `collect-with` actions.

Actions Modified

`collect`, `collect-with`

Description

Use the `no-auto-finalize` option to suppress finalization during a `collect` or `collect-with` action, so that symbol resolution and suppressions can be performed in a separate step. As the next step, you can be sure to use the `finalize` action to complete the process, specifying search directories containing the same module binaries used during collection.

> **Important**
> The first time an unfinalized result is opened or loaded, the result is automatically finalized. Inspector does not check the validity of the module binaries, so if a result is opened through the GUI, by double-clicking, or if a report is generated without specifying the module binaries, improper finalization may result. Improper finalization may also result if the `finalize` action is used without specifying the same module binaries that were used to collect the result. For proper finalization, please make sure to use the `finalize` action and specify the same module binaries that were used to collect the result.

Example

This command:

- Runs a **Detect Deadlocks and Data Races (ti2)** analysis on the application `myApp`.

- Stores the result in the default `r@@@ti2` result directory in the current working directory, where `@@@` represents the next available number.
- Does not complete the finalization process, and does not output a Summary report.

```
$ inspxe-cl -collect ti2 -no-auto-finalize -- myApp
```

See Also

Command Line Output
Command Syntax
create-suppression-file
action
finalize
action
Defining Suppression Rules in the GUI
Set Project Properties-Advanced

no-summary

Suppress the display of a summary after a collect action.

Syntax

`-no-summary`

`-summary`

Default

summary - A Summary of observed problems is displayed after a collect action, and written to a file as a Summary report.

Actions Modified

`collect`

Description

A summary of observed problems is generated after a collect action by default. Use the `--no-summary` action-option when you do not want a summary to be generated.

Example

This command performs the narrowest-scope threading analysis on `myApp` and generates no Summary report.

```
$ inspxe-cl -collect ti1 -no-summary -- myApp
```

See Also

report
action
About Reporting Result Data
Command Line Output
Command Syntax

option-file

Specify one or more files containing command line options.

Syntax

`-option-file <PATH>`

Arguments

A string containing the PATH/name for the one or more text files that contain additional command options. This may be absolute, or relative to the current working directory.

Actions Modified

This global option can be used with any action, including: `collect`, `collect-with`, `finalize`, `merge`, and `report`.

Description

Use the `option-file` global-option to specify one or more text files containing command line arguments, where each line include one or more arguments terminated by a newline character.

- Arguments specified in an option file are processed before any arguments that may be specified on the command line.
- If multiple option files are used, they are processed in the order in which they are specified on the command line.

Tip
Put commonly used options in a text file to shorten the command line and to create a reusable invocation syntax.

Caution
Options specified on the command line can override options in the option file(s), because options in the file(s) are processed prior to options passed in through the command line.

Example

In this example, an option file named `ti2Supp.txt` specifies the `suppression-file` option and path to the suppression file.

```
-suppression-file "C:\myAppProj\My Inspector Results-[project name]\suppressions\mySup"
```

This command performs the `collect` action, passing in the option file `ti2Supp.txt`.

```
$inspxe-cl -collect ti2 -option-file ti2Supp.txt -- myApp
```

When this option file is processed:

- It runs a **Detect Deadlocks and Data Races** analysis is performed on a target specified on the command line.
- It uses the `C:\myAppProj\My Inspector Results\suppressions\mySup.sup` suppression file, filtering out unwanted problems from the results according to rules in the suppressions file.
- Stores the result in the default `r@@@ti2` result directory in the current working directory, where `@@@` represents the next available number.
- Generates a summary report of new or unsuppressed problems, and saves the report as `inspxe-cl.txt` in the result directory.

See Also

Command Line Output
Command Syntax

quiet

Suppress non-essential information.

Syntax

`-quiet`

`-q`

Default

Off

Actions Modified

All

Description

Use the `quiet` global-option when you want only errors and warnings to be written to `stderr`.

Example

This command:

- Runs a **Detect Leaks (mi1)** analysis on the application `myApp`.
- Stores the result in the `r@@@mi1` result directory in the current working directory, where `@@@` represents the next available number.
- Generates a summary report of detected problems, and writes it to the `inspxe-cl.txt` file in the result directory.
- Generates an `insp-cl.loginspxe-cl.log` file that contains error, warning, and minimal status information.

```
$ inspxe-cl -collect mi1 -quiet -- myApp
```

See Also

verbose
global-option
Command Line Output
Command Syntax

report

Generate a report from result data collected during a previous analysis.

Syntax

`-report <value>`

`-R <value>`

NOTE
Both short names and long names are case-sensitive. For example, `-R` is the short name of the `report` action, and `-r` is the short name of the `result-dir` action-option.

Arguments

`summary`	Brief statement of the total number of new problems found and a breakdown by problem type. After each `collect` or `collect-with` action, a Summary report is generated by default, written to a text file in the current working directory, and sent to `stdout`.
`problems`	Detailed report of detected problem sets in the result, including their location in the source code.
`observations`	Detailed report of all code locations used to form new problem sets.
`status`	Brief statement of the total number of detected problems, the number that are *Not investigated*, and the breakdown by category.

Default

By default, a Summary report is automatically generated after running a `collect` or `collect-with` action, and then sent to `stdout` and to a file named `inspxe.xml`.

Modifiers

`baseline-result`, `csv-delimiter`, `filter`, `format`, `option-file`, `quiet`, `report-all`, `report-output`, `result-dir`, `search-dir`, `sort-asc`, `sort-desc`, `suppression-file`, `user-data-dir`, `verbose`

Description

Use the `report` action to generate the specified type of report from an analysis result.

By default, a report is written to standard output in text format, but the `inspxe-cl` tool provides a number of options you can use when generating a report.

- To save a report to a file, use the report-output option.
- Use the `format` option to choose a report format: Text, CSV, or XML. If you choose the CSV format and want to use a delimiter other than the default comma, use the csv-delimiter option to specify the delimiter.
- To sort and filter report content, use the `sort-asc`, `sort-desc` and `filter` options.

Example

Generate a problems report of all detected problems in the specified result `r001ti` and display it to `stdout`.

```
$ inspxe-cl -report problems -result-dir r001ti
```

Generate a status report for the most recent result and save it as `myThreadingStatus.txt` text format in the current working directory.

```
$ inspxe-cl -R status -report-output myThreadingStatus.txt
```

Status report output:

```
181 problem(s) found
15 Investigated
166 Not investigated
Breakdown by state:
13 Confirmed
2 Fixed
166 New
```

Compare two `t1` Detect Deadlocks results and generate a summary report that shows the differences.

```
$ inspxe-cl -R summary -r myRes000ti -r myRes001t1
```

See Also

About Reporting Result Data
Saving and Formatting Reports
About Interpreting Result Data and Resolving Issues
Command Line Output
Command Syntax

report-all

Create a detailed report particularly useful for regression testing purposes.

Syntax

`-report-all`

`-r-a`

Default

Off

Actions Modified

`report`

Description

Use the `report-all` action-option to create a detailed report that helps you thoroughly investigate a result. If you output the report in TXT or CSV format, it contains code snippets and call stacks. If you output the report in XML format, it also contains additional information normally available only when using the GUI.

> **Tip**
> Parse XML output to programmatically analyze it for regression testing purposes.

Example

Generate a detailed report of all detected problems in the specified result `r001ti` and display it to `stdout`.

```
$ inspxe-cl -report problems -report-all -result-dir r001ti
```

Generate a detailed problems report for the most recent result and save it as `myThreadingproblems.xml` in the current working directory.

```
$ inspxe-cl -R problems -report-all -report-output myThreadingproblems -format xml
```

See Also

Command Line Output
Command Syntax
Saving and Formatting Reports from the Command Line

report-output

Output a report file with the specified pathname.

Syntax

`-report-output <PATH>`

Arguments

`PATH`	A string containing the PATH/name for the report file.

Default

By default, a report is written to `stdout`.

Actions Modified

`report`

Description

Use `report-output` to save a report to a file rather than writing it to `stdout`. The file is created relative to the current working directory unless an absolute pathname is specified.

Example

Generate a status report from the most recent result and save it as `status-report.txt` file in the current working directory.

```
$ inspxe-cl -report status -report-output status-report.txt
```

See Also

About Reporting Result Data
Saving and Formatting Reports
Command Line Output
Command Syntax

result-dir

Specify the directory where the result is stored.

GUI Equivalent

Dialog Box: Options-Result Location

Syntax

`-result-dir <PATH>`

`-r <PATH>`

> **NOTE**
> Both short names and long names are case-sensitive. For example, `-R` is the short name of the `report` action, and `-r` is the short name of the `result-dir` action-option.

Arguments

`pathname`	A string containing the PATH/name for the directory where a result is stored. This may be an absolute pathname or a path relative to the current working directory. If the lowest directory in the pathname does not exist, it is created.

By default, the name of the result directory takes this general form: r@@@{at}.

- The prefix `r` indicates that this is a result directory. You may specify a different prefix if desired.
- The variables `@@@` represent an automatic counter that starts from 000 and is incremented by one as each subsequent result is created. There can be only one occurrence of this string in the pathname. When you perform a `collect` or `collect-with` action, this value is automatically incremented to the next available number. If you use this counter, your first result is numbered 000, the second is 001, and so on. If you perform an action on an existing result without specifying the pathname, the most recent result (which is assumed to be the result with the highest number) is used by default.
- The suffix `{at}` is a code representing the type of `collect`or `collect-with` action used to generate the result. This code indicates which collector, analysis type and preset configuration level (i.e., knob) were selected.

Default

The pathname uses the format r@@@{at} and stores the result in the current working directory.

Actions Modified

`collect`, `collect-with`, `command`, `create-suppression-file`, `finalize`, `report`

Description

Use the `result-dir` action-option to specify the pathname of a result.

- For the `collect` or `collect-with` action, use this to specify the directory where you want the result to be created. The result will have the same name as the directory name you specify.
- For other actions, use this to specify the result you want to use as input.

Using the `@@@` counter pattern in the directory name argument allows you to collect a result, or perform an action on a the most recent result, without specifying the exact result name.

If you specified a different prefix, such as myRes- (`myRes-@@@{at}`), and then perform a memory error analysis, followed by a threading error analysis, the result directories would be assigned the following names: `myRes-000mi1` and `myRes-001ti2`.

Caution
You cannot put multiple results into the same result directory. If you specify the same result directory for multiple analysis runs, an error is returned. Use the auto-increment counter (@@@) to work around this restriction.

Example

This command:

- Runs a **Detect Deadlocks and Data Races (ti2)** analysis on the application `myApp`.
- Stores the result in the `myRes000` result directory in the current working directory the first time `myRes@@@` is used. (The next invocation would generate a result directory named `myRes001`.)

- Generates a summary report of detected problems, and writes it to the `inspxe-cl.txt` file in the result directory.
- Generates a suppression file for all detected problems in the `myRes000` result (or the highest existing number of similarly named results).

```
$ inspxe-cl -collect ti2 -result-dir myRes@@@ -- myApp
```

This command compares the results of two t1 collections and generates a Summary report to get an overview of regression status.

```
$ inspxe-cl -R summary -r myRes000 -r myRes001
```

See Also

Command Line Output
Command Syntax

return-app-exitcode

Return the exit code from the target application.

Syntax

`-return-app-exitcode`

Default

By default, an `inspxe-cl` exit code is returned:

0	Success and no new problems detected
1	Usage error
2	Internal error
4	Application returned a non-zero exit code
8	At least one new problem detected
12	Application returned a non-zero exit code and at least one new problem detected

Actions Modified

`collect`, `collect-with`,

Description

Use the `return-app-exitcode` action-option to return the exit code returned by the target application, instead of Intel Inspector exit codes.

Example

This command:

- Runs a **Detect Deadlocks and Data Races (ti2)** analysis on the application `myApp`.
- Stores the result in the default `r@@@ti2` result directory in the current working directory, where `@@@` represents the next available number.
- Generates a summary report of detected problems, and writes it to the `inspxe-cl.txt` file in the result directory.
- Returns the exit code returned by `myApp`.

```
$ inspxe-cl -collect ti2 -return-app-exitcode -- myApp
```

See Also

Command Line Output
Command Syntax

search-dir

Specify an alternative location(s) for finding application support files.

GUI Equivalent

Dialog Box Project Properties-Source Search

Dialog Box Project Properties-Binary/Symbol Search

Syntax

`-search-dir all|bin|src|sym[:p|:r|:rp]=<PATH>`

Arguments

The pathname of a search directory, the type of file and search instructions.

`pathname`	The PATH/name of the search directory, and can include environment paths and absolute paths.
`File types`	• all: any of these file types • bin: binary files • src: source files • sym: symbol files
`:r`	Perform a recursive search of all subdirectories
`:p`	Specify the highest priority search directories, which will be searched prior to others. Use this to avoid picking up the wrong module by absolute path when moving a result between machines.
`:rp`	Combine these two arguments to perform a recursive search of all subdirectories and indicate highest priority search directories.

Example:`-search-dir all:r=C:\myProject`

For multiple arguments: You may specify this option multiple times in a single command, or specify multiple arguments in a comma-separated list (no spaces). For example, the following are equivalent:

```
-search-dir all:p=dir1 -search-dir sym=dir2 -search-dir bin:r=dir3
```

```
-search-dir all:p=dir1,sym=dir2,bin:r=dir3
```

Caution
Make sure there are no spaces after commas.

Default

For binary and symbol information files: System environment PATH settings and directory in which target resides.

For source files: Directories specified in debug information files.

Actions Modified

collect, collect-with, finalize,

Description

Use search-dir when you want to search non-standard directories during target execution, analysis, and finalization. This can be useful when:

- Source files reside somewhere other than where debug information indicates.
- Debug information files are not located with binary files.

Example

This example:

- Runs a **Detect Deadlocks and Data Races (ti2)** analysis on the application myApp.
- Searches the nonstandard subdirectory within the current working directory for support files needed to execute the myApp target during the collect action.
- Stores the result in the default r@@@ti2 result directory in the current working directory, where @@@ represents the next available number.
- Generates a summary report of detected problems, and writes it to the inspxe-cl.txt file in the result directory.

```
$ inspxe-cl -collect ti2 -search-dir all=nonstandard -- myApp
```

See Also

Command Line Output
Command Syntax
Searching Non-standard Directories in the Microsoft Visual Studio* IDE

sort-asc

Sort report data by specified field(s) in ascending order.

Syntax

-sort-asc *<value>*

-s-asc *<value>*

Arguments

function	Sort by function name alphabetical order.
id	Sort by problem id.
investigated	Sort by whether the problem has been investigated.
line	Sort by line number.
module	Sort by module.
problem	Sort by problem type.
severity	Sort by problem severity level.
source	Sort by source file name.
state	Sort by state.

Default

Varies by report type.

Actions Modified

`report`

Description

Use the `sort-asc` action-option to organize a report in ascending order of the specified field(s). You can specify up to three different fields.

> **NOTE** Legacy formats do not contain all these data fields, so some information may be missing in reports that display imported data.

Alternate Options

To sort in reverse order, use the `sort-desc` action-option.

Example

This example generates a Problems report from the most recently collected result data, and displays the problems in ascending order by line number.

```
$ inspxe-cl -report problem -sort-asc line
```

See Also

sort-desc
action-option
Command Line Output
Command Syntax

sort-desc

Sort report data by specified field in descending order.

Syntax

`-sort-desc <value>`

`-s-desc <value>`

Arguments

`function`	Sort by function name alphabetical order.
`id`	Sort by problem id.
`investigated`	Sort by whether the problem has been investigated.
`line`	Sort by line number.
`module`	Sort by module.
`problem`	Sort by problem type.
`severity`	Sort by problem severity level.
`source`	Sort by source file file.

`state`	Sort by state.

Default

Varies by report type.

Actions Modified

`report`

Description

Use the `sort-desc` action-option to organize a report in descending order of the specified field(s). You can specify up to three different fields.

> **NOTE** Legacy formats do not contain all the data fields, so some information may be missing in the reports shown for imported data.

Alternate Options

To sort in reverse order, use the `sort-asc` action-option.

Example

This command generates a Problems report from the most recently collected result data, and displays the problems in descending order of severity.

```
$ inspxe-cl -report problem -sort-desc severity
```

See Also

sort-asc
action-option
Command Line Output
Command Syntax

suppression-file

Use rules defined in a suppression file to exclude known problems during collection.

GUI Equivalent

Dialog Box Create Suppression

Syntax

`-suppression-file <PATH>`

Arguments

A string containing the PATH/name for the suppression file. This may be an absolute path, or a path relative to the current working directory.

Pathname can be either:

directory	The name of a directory containing suppression files.

filename	The name of a suppression file. May be an absolute pathname or name relative to the current working directory. The `inspxe-cl` tool appends a `.sup` extension if no extension is specified.

For multiple arguments: You may specify this option multiple times in a single command, or specify multiple arguments in a comma-separated list (no spaces). For example, the following are equivalent:

```
-suppression-file mySup1 -suppression-file mySup2.txt
```

```
-suppression-file mySup1,mySup2
```

Actions Modified

`collect`, `collect-with`, `report`

Description

Use the `suppression-file` action-option to apply one or more suppression files when performing the `collect` or `collect-with` action. You can use the `create-suppression-file` action to generate a suppression file that contains rules that define which problems should be excluded from the Summary of detected problems.

When `suppression-file` is used with the `collect` action, the process uses all files with a .sup extension in the directory.

Example

This command:

- Runs a **Detect Deadlocks and Data Races (ti2)** analysis on the application `myApp`.
- Uses the short name `-s-f` for the `suppression-file` option.
- Uses suppression files for `myProject`, found in the Microsoft Visual Studio* IDE standard location for private suppression files: `C:\myProject\My Inspector Results-[project name]\suppressions`.
- Stores the result in a default-named result directory under the current working directory.
- Generates a summary report of new or unsuppressed problems, and saves the report as `inspxe-cl.txt` in the result directory.

```
$ inspxe-cl -collect ti2 -s-f "C:\myProject\My Inspector Results\suppressions" -- myApp
```

See Also

Command Line Output
Command Syntax
Working with Suppressions from the Command Line
create-suppression-file
action
Set Project Properties-Advanced

user-data-dir

Specify the base directory for result paths.

Syntax

```
-user-data-dir <PATH>
```

Arguments

A string containing the PATH/name for the base directory. This may be an absolute path, or a path relative to the current working directory.

Default

The current working directory unless the `INSP_USER_DATA_DIR` environment variable is defined.

Actions Modified

`collect`, `collect-with`, `command`, `create-suppression-file`, `finalize`

Description

Use the `user-data-dir` action-option when you want to specify the base directory of result directories, but still use the default names for result directories and files.

- If used with the `collect` or `collect-with` actions, a default-named result directory is added below the specified base directory.
- To generate a report or perform another action on results under this base directory, use the `result-dir` option to specify the path to the result directory.
- If the `INSP_USER_DATA_DIR` environment variable is defined, this option overrides it.

Alternate Options

The `result-dir` action-option specifies the PATH/name of the directory in which the result is stored. If you want to use the default result directory name, but set a different path, use the `user-data-dir` option.

Example

This command:

- Runs a **Detect Deadlocks and Data Races (ti2)** analysis on the application `myApp`.
- Creates the default-named `r@@@ti2` result directory in the location specified by the `user-data-dir` action-option: `G:\inspResults`.
- Generates a summary report of detected problems, and writes it to the `inspxe-cl.txt` file in the result directory.

```
$ inspxe-cl -collect ti2 -user-data-dir G:\inspResults -- myApp
```

See Also

result-dir action-option
Command Line Output
Command Syntax

verbose

Print additional information.

Syntax

`-verbose`

`-v`

Default

Off

Actions Modified

All

Description

Use the `verbose` global-option when you want to write all informational messages (in addition to error and warning messages) to `stderr`.

Example

This command:

- Runs a **Detect Leaks (mi1)** analysis on the application `myApp`.
- Stores the result in the `r@@@mi1` result directory in the current working directory, where `@@@` represents the next available number.
- Generates a summary report of detected problems, and writes it to the `inspxe-cl.txt` file in the result directory.
- Generates an `inspxe-cl.log` file that contains error, warning, and all informational messages.

```
$ inspxe-cl -collect mi1 -verbose -- myApp
```

See Also

quiet global-option
Command Line Output
Command Syntax

version

Display product version information.

Syntax

`-version`

`-V`

Description

Use the `version` action to write Intel Inspector version information to `stdout`.

Example

Each of these commands displays product version information.

```
$ inspxe-cl -version
```

```
$ inspxe-cl -V
```

See Also

Command Line Output
Command Syntax

MPI Applications Support

MPI Analysis Workflow

To analyze the performance and correctness of an MPI application at the inter-process level, use the Intel® Trace Analyzer and Collector tool (located at `<installdir>/itac` directory after installation). The Intel Trace Analyzer and Collector attaches to the application through linkage (statically, dynamically, also through LD_PRELOAD or via the Intel Compiler `-tcollect` and `-tcollect-filter` options), or by using the itcpin tool. The tools collect information about events at the MPI level between processes and allow analyzing the performance and correctness of the MPI calls, deadlock detection, data layout errors, as well as risky or incorrect MPI constructs. The Intel Trace Analyzer and Collector data is correlated and aggregated across all processes and all nodes that participated in the execution run.

Beyond the inter-process level of MPI parallelism, the processes that make up the applications on a modern cluster often also use fork-join threading through OpenMP* and Intel® oneAPI Threading Building Blocks (oneTBB) . This is where the Intel® VTune™ Profiler and the Intel Inspector should respectively be used to analyze the performance and correctness of an MPI application.

At the high level the analysis workflow consists of three steps:

1. Use the `amplxe-cl` and `inspxe-cl` command-line tools to collect data about an application. By default, all processes are analyzed, but it is possible (and sometimes required for Intel VTune Profiler - there are certain collection technology limitations) to filter the data collection to limit it to a subset of processes. An individual result directory is created for each spawned MPI application process that was analyzed with MPI process rank value captured.
2. Post-process the result, which is also called *finalization* or *symbol resolution*. This is done automatically for each result directory once the collection has finished.
3. Open the content of each result directory through the GUI standalone viewer to analyze the data for the specific process. The GUI viewers are independent: Intel VTune Profiler and Intel Inspector have their own user-interfaces.

NOTE

- The file system contents should be the same on all nodes to make sure that the modules referenced in the collected data are available automatically on the host where the collection was initiated. This limitation can be overcome by manual copying of the modules for analysis from the nodes and adjusting the Intel VTune Profiler / Intel Inspector project search directories to make the modules found.
- For Intel VTune Profiler the CPU model and stepping should be the same on all nodes so that the hardware Event-based sampling operates with the same Performance Monitoring Unit (PMU) type on all nodes.

MPI Analysis Limitations

There are certain limitations in the current MPI profiling support provided by the Intel VTune Profiler / Intel Inspector:

- MPI dynamic processes are not supported by the Intel VTune Profiler / Intel Inspector. An example of dynamic process API is MPI_Comm_spawn
- The data collections that use the hardware event-based sampling collector are limited to only one such collection allowed at a time on a system. When the Intel VTune Profiler is used to profile an MPI application, it is the responsibility of the user to make sure that only one SEP data collection session is launched on a given host. Common ways to achieve this is using the host syntax and distribute the ranks running under the tool over different hosts.

See Also

Collect MPI Performance/Correctness Data
Finalize MPI Collected Data
View MPI Collected Data

Configure Installation Options

The default Intel® Inspector installation path is below `C:\Program Files (x86)\Intel\oneAPI\`.

NOTE
The per-user Hardware Event-based Sampling mode is not available on the Windows operating system at the moment.

Once installed, you can use the `setvars.bat` files to set up the appropriate environment (PATH, MANPATH) in the current terminal session. The tools capabilities are the same for Windows* and Linux* OS unless explicitly noted and described.

Collect MPI Performance/Correctness Data

To collect performance or correctness data for an MPI application with the Intel® VTune™ Profiler / Intel Inspector on a Windows* or Linux* OS, the following command should be used:

```
$ mpirun-n <N> <abbr>-cl -r my_result -collect <analysis type> my_app [my_app_ options]
```

where `<abbr>` is `amplxe` or `inspxe` respectively. The list of analysis types available can be viewed using `amplxe-cl-help collect` command.

As a result of using the collection commands, a number of result directories are created in the current directory, named as `my_result.0` - `my_result.3`. The numeric suffix is the corresponding MPI process rank that is detected and captured by the collector automatically. The usage of the suffix makes sure that multiple `amplxe-cl` / `inspxe-cl` instances launched in the same directory on different nodes do not overwrite the data of each other and can work in parallel. So, a separate result directory is created for each analyzed process in the job.

Sometimes it is necessary to collect data for a subset of the MPI processes in the workload. In this case the per-host syntax of `mpirun`/`mpiexec*` should be used to specify different command lines to execute for different processes.

When launching the collection on Windows OS, we recommend passing the `-genvall` option to the `mpiexec` tool to make sure that the user environment variables are passed to all instances of the profiled process. Otherwise, by default the processes are launched in the context of a system account and some environment variables (USERPROFILE, APPDATA) do not point where the tools expect them to point to.

There are also some specialties about stdout / stdin behavior in MPI jobs profiled with the tools:

- It is recommended to pass the `-quiet` / `-q` option to `amplxe-cl` / `inspxe-cl` to avoid diagnostic output like progress messages being spilled to the console by every tool process in the job.
- The user may want to use the `-l` option for `mpiexec`/`mpirun` to get stdout lines marked with MPI rank.

Example

The most reasonable analysis type to start with for the Intel VTune Profiler is hotspots, so an example of full command line for collection would be:

```
$ mpirun-n 4 amplxe-cl -r my_result -collect hotspots -- my_app [my_app_ options]
```

A similar command line for the Intel Inspector and its ti1/mi1 analysis types (the lowest overhead threading and memory correctness analysis types respectively) would look like:

```
$ mpirun-n 4 inspxe-cl -r my_result -collect mi1 -- my_app [my_app_ options]
```

```
$ mpirun-n 4 inspxe-cl -r my_result -collect ti1 -- my_app [my_app_ options]
```

Here is an example where there are 16 processes in the job distributed across the hosts and hotspots data should be collected for only two of them:

```
$
mpirun
-host myhost -n 14 ./a.out : -host myhost -n 2 amplxe-cl -r foo -c hotspots ./a.out
```

As a result, two directories will be created in the current directory: `foo.14` and `foo.15` (given that process ranks 14 and 15 were assigned to the last 2 processes in the job). As an alternative to specifying the command line above, it is possible to create a configuration file with the following content:

```
# config.txt configuration file
-host myhost -n 14 ./a.out
-host myhost -n 2 amplxe-cl -quiet -collect hotspots -r foo ./a.out
```

and run the data collection as:

```
$ mpirun-configfile ./config.txt
```

to achieve the same result as above (`foo.14` and `foo.15` result directories will be created). Similarly, you can use specific host names to control where the analyzed processes are executed:

```
# config.txt configuration file
-host myhost1 -n 14 ./a.out
-host myhost2 -n 2 amplxe-cl -quiet -collect hotspots -r foo ./a.out
```

When the host names are mentioned, consecutive MPI ranks are allocated to the specified hosts. In the case above, ranks 0 to 13, inclusive, will be assigned to myhost1, the remaining ranks 14 and 15 will be assigned to myhost2. On Linux, it is possible to omit specifying the exact hosts, in which case the distribution of the processes between the hosts will be done in round-robin fashion. That is, myhost1 will get MPI ranks 0, 2, and 4 thru 15, while myhost2 will get MPI ranks 1 and 3. The latter behavior may change in the future.

NOTE
In the examples this reference uses the `mpirun` command as opposed to `mpiexec` and `mpiexec.hydra` while real-world jobs might use the `mpiexec*` ones. `mpirun` is a higher-level command that dispatches to `mpiexec` or `mpiexec.hydra` depending on the current default and options passed. All the examples listed in the paper work for the `mpiexec*` commands as well as the `mpirun` command.

See Also
Support of Non-Intel MPI Implementations

Finalize MPI Collected Data

The finalization of the data (symbol resolution, conversion to the database) happens automatically after the collection has finished. It happens on the same compute node where the command line collection was executing, and so the binaries and symbol files will be located automatically by the `amplxe-cl` / `inspxe-cl` tools. In cases where the search process needs to be adjusted (common reason: need to point to symbol files stored elsewhere), the `-search-dir` option should be used with `amplxe-cl` / `inspxe-cl` as follows:

```
$ mpirun-np 128 amplxe-cl -q -collect hotspots -search-dir sym=/home/foo/syms ./a.out
```

View MPI Collected Data

Once the results are collected, the user can open any of them in the standalone GUI or generate a command line report. Use `inspxe-cl-help report` or `vtune-help report` to see the options available for generating reports.

To view the results through GUI, launch the `{vtune | inspxe}-gui <result path>` command or launch the `*-gui` tool and use the **File** > **Open** > **Result...** menu item to point to the result. Sometimes it is also convenient to copy the result to another system and view it there (for example, to open a result collected on a Linux cluster on a Windows workstation).

MPI functions are classified by the Intel® VTune™ Profiler as system ones making its level of support in this regard similar to Intel® oneAPI Threading Building Blocks (oneTBB) and OpenMP*. This helps the user to focus on his/her code rather than MPI internals.Intel VTune Profiler GUI *Call Stack Mode* and CLI `-stack-mode` switches can be used to turn on the mode where the system functions are displayed and thus the internals of the MPI implementation can be viewed and analyzed. The call stack mode *User functions+1* is especially useful to find the MPI functions that consume most of CPU Time (Hotspots analysis) or waited the most (Locks and Waits analysis). For example, assume there is a call chain `main() -> foo() -> MPI_Bar() -> MPI_Bar_Impl() -> ...` where `MPI_Bar()` is the actual MPI API function you use and the deeper functions are MPI implementation details. The call stack modes behave as follows:

- The default *Only user functions* call stack mode will attribute time spent in the MPI calls to the user function `foo()` so that you can see which of your functions you can change to actually improve the performance.
- The *User functions+1* mode will attribute the time spent in the MPI implementation to the top-level system function - `MPI_Bar()` so that you can easily see outstandingly heave MPI calls.
- The *User/system functions* mode will show the call tree without any reattribution so that you can see where exactly in the Intel MPI library the time was spent.

Intel VTune Profiler / Intel Inspector provide oneTBB and OpenMP support. It is recommended to use these thread-level parallel solutions in addition to MPI-style parallelism to maximize the CPU resource usage across the cluster, and to use the Intel VTune Profiler / Intel Inspector to analyze the performance / correctness of that level of parallelism. The MPI, OpenMP, and oneTBB features in the tools are functionally independent, so all usual features of OpenMP and oneTBB support are applicable when looking into a result collected for an MPI process.

Example

Here is an example of viewing the text report for functions and modules after a Intel VTune Profiler analysis (note that we open individual results each of which was collected for a specific rank of MPI process - `foo.14` and `foo.15` in the example above):

```
$ vtune -R hotspots -q -format text -r foo.14
Function Module CPU Time
-------- ------ --------
f        a.out  6.070
main     a.out  2.990

$ vtune -R hotspots -q -format text -group-by module -r foo.14
Module CPU Time
------ --------
a.out  9.060
```

MPI Analysis Limitations

There are certain limitations in the current MPI profiling support provided by the Intel® Inspector:

- MPI dynamic processes are not supported by the Intel Inspector. An example of dynamic process API is `MPI_Comm_spawn`
- Intel Inspector analyzes data races or deadlocks inside a single process and cannot detect inter-process data races or deadlocks.

Support of Non-Intel MPI Implementations

The examples in this section assume the usage of the Intel® MPI library implementation but the workflow will work with other MPI implementations, if the following is kept in mind:

- Intel® VTune™ Profiler and Intel Inspector tools extract the MPI process rank from the environment variables PMI_RANK or PMI_ID (whichever is set) to detect that the process belongs to an MPI job and to capture the rank in the result directory name. If the alternative MPI implementation does not set those

environment variables, the tools do not capture the rank in the result directory name and a usual automatic naming of result directories should be used. Default value for the `-result-dir` option is `r@@@{at}`, which results in sequence of result directories like `r000hs`, `r001hs`, and so on.
- The function/module patterns used for classification of time spent inside of the Intel MPI Library as system one may not cover all of modules and functions in the used MPI implementation. This may result in displaying some internal MPI functions and modules by default.
- The command-line examples in this section may need to be adjusted to work - especially when it comes to specifying different command lines to execute for different process ranks to limit the amount of processes in the job being analyzed.
- The MPI implementation needs to operate in cases when there is a tool process between the launcher process (`mpirun`/`mpiexec`) and the application process. This essentially implies that the communication information should be passed using environment variables, as most MPI implementations do. The tools would not work on an MPI implementation that tried to pass communication information from its immediate parent process. Intel is unaware of any implementations that have this limitation.

Additional MPI Resources

See the Intel® VTune™ Profiler, Intel® Inspector, online MPI documentation for more details at https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library-documentation.html.

There are also other resources available online related to the usage of the Intel VTune Profiler and Intel Inspector with the Intel MPI Library, such as *Hybrid applications: Intel MPI Library and OpenMP* at https://www.intel.com/content/www/us/en/developer/articles/technical/hybrid-applications-mpi-openmp.html.

Appendix

Product Design

Intel does not guarantee the Intel Inspector will detect or report every memory and threading error in an application.

- Not all logic errors are detectable.
- Heuristics used to eliminate false positives may hide real issues.
- Where reasonable, the Intel Inspector groups highly correlated problems into a single problem.

Incompatible or proprietary instructions in non-Intel processors may cause the analysis capabilities of this tool to function incorrectly. Any attempt to analyze code not supported by Intel® processors may lead to failures in this tool.

Sample Code Caveats

All code examples are designed only to illustrate product features. They do not represent best practices for creating multithreaded code. Results may vary depending on the nature of the analysis.

Intel Inspector Filenames and Locations

User-controlled Filenames and Locations

File Type	Default Name	Default Location
Project (`.inspxeproj`)	`config.inspxeproj` • `\My Inspector Results-[project name]\` directory in the solution directory (Visual Studio* IDE) • `\Inspector\Projects\[project directory]\` directory in the `%HOME%\My Documents` or `%HOME%\Documents\` directory (Standalone GUI)	**NOTE** The filename is system controlled; however, the file location is user controlled. The default location is • `\My Inspector Results-[project name]\` directory in the solution directory (Visual Studio* IDE) • `\Inspector\Projects\[project directory]\` directory in the `%HOME%\My Documents` or `%HOME%\Documents\` directory (Standalone GUI)
Result (`*.inspxe`) produced with preset analysis type	`r@@@{at}`, where: • `r` = constant • `@@@` = next available number • `{at}` = preset analysis type identifier	• `\My Inspector Results-[project name]\r@@@{at}\` directory in the solution directory (Visual Studio* IDE) • `\Inspector\Projects\[project directory]\r@@@{at}\` directory in the `%HOME%\My Documents` or `%HOME%\Documents\` directory (Standalone GUI)
Result (`*.inspxe`) produced with custom analysis type	`r@@@{at}`, where: • `r` = constant • `@@@` = next available number • `{at}` = user-controlled analysis type identifier	• `\My Inspector Results-[project name]\r@@@\` directory in the solution directory (Visual Studio* IDE) • `\Inspector\Projects\[project directory]\r@@@\` directory in the `%HOME%\My Documents` or `%HOME%\Documents\` directory (Standalone GUI)
Suppression (`*.sup`)	`default.sup`	• `\My Inspector Results-[project name]\suppressions\` directory in the solution directory (Visual Studio* IDE) • `\Inspector\Projects\[project directory]\suppressions\` directory in the `%HOME%\My Documents` or `%HOME%\Documents\` directory (Standalone GUI)
Result archive (`*.inspxez`)	`[result_name].inspxez`	• `\My Inspector Results-[project name]\` directory in the solution directory (Visual Studio* IDE) • `\Inspector\Projects\[project directory]\` directory in the `%HOME%\My Documents` or `%HOME%\Documents\` directory (Standalone GUI)

Examples

If you use the default result name template and location, run a threading error analysis using a medium-scope analysis type, and then run a memory error analysis using the narrowest-scope analysis type, the Intel® Inspector names and saves the results in the following manner:

- Visual Studio* IDE:
  - `\My Inspector Results-[project name]\r000ti2\r000ti2.inspxe`
  - `\My Inspector Results-[project name]\r001mi1\r001mi1.inspxe`
- Standalone Intel® Inspector GUI:
  - `\Inspector\Projects\[project directory]\r000ti2\r000ti2.inspxe`
  - `\Inspector\Projects\[project directory]\r001mi1\r001mi1.inspxe`

where

- `ti2` = threading analysis type with medium scope
- `mi1` = memory analysis type with narrowest scope

Other Filenames and Locations

File Type	Filename	File Location
Preset analysis type (`.cfg`)	• `mi1.cfg` • `mi2.cfg` • `mi3.cfg` • `ti1.cfg` • `ti2.cfg` • `ti3.cfg`	`\config\analysis_type` in the product installation directory **NOTE** The default installation path is below `C:\Program Files (x86)\Intel\oneAPI\inspector` (on certain systems, the directory is `Program Files`).
Custom analysis type (`.cfg`)	`[Command-line name].cfg`	`%HOME%\AppData\Intel\Inspector\analysis_type\` or `%HOME%\Application Data\Intel\Inspector\analysis_type\` directory

NOTE In Visual Studio* 2022, Intel Inspector provides lightweight integration. You can configure and compile your application and open the standalone Intel Inspector interface from the Visual Studio for further analysis. All your settings will be inherited by the standalone Intel Inspector project.

Notational Conventions

The following conventions may be used in this document.

Convention	Explanation	Example
Italic	Used for introducing new terms, denotation of terms, placeholders, or titles of manuals.	The filename consists of the *basename* and the *extension*. For more information, refer to the *Intel® Advisor User Guide*.
Bold	Denotes GUI elements	Click **Cancel**.

Convention	Explanation	Example
>	Indicates a menu item inside a menu.	**File > Close** indicates to select **Close** from the **File** menu.

`Monospace`	Indicates directory paths and filenames, or text that can be part of source code.	`ippsapi.h`
		`\alt\include`
		Use the `okCreateObjs()` function to...
		`printf("hello, world\n");`

*	An asterisk at the end of a word or name indicates it is a third-party product trademark.	Microsoft Windows XP*
Windows* OS, Windows operating system	These terms refer to all supported Microsoft* Windows* operating systems.	This table contains a summary of Windows* OS Linking Behavior.

Related Information

A variety of resources provide additional information on a number of topics. Some of the most useful are listed here.

Parallel Programming

You are strongly encouraged to read the following books for in-depth understanding of threading. Each book discusses general concepts of parallel programming by explaining a particular programming technology.

Technology	Resource
oneAPI Threading Building Blocks (oneTBB)	Reinders, James. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, July 2007 (http://oreilly.com/catalog/9780596514808/)
OpenMP* technology	Chapman, Barbara, Gabriele Jost, Ruud van der Pas, and David J. Kuck (foreword). *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, October 2007 (http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11387)
Microsoft Win32* Threading	Akhter, Shameem, and Jason Roberts. *Multi-Core Programming: Increasing Performance through Software Multithreading*, Intel Press, April 2006 (http://www.intel.com/intelpress/sum_mcp.htm).

Additional Reference Material

For additional technical product information, including white papers about Intel products, see https://www.intel.com/content/www/us/en/resources-documentation/developer.html

System APIs Supported During Memory Error Analysis

The following table lists all the 32-bit and 64-bit Windows* OS memory management functions currently supported by the Intel Inspector during memory error analysis. Check the Release Notes to see if support for new APIs has been recently added.

If an API is not supported, the Intel Inspector may report false positive (non-existent) memory problems in functions using the API.

C Library Memory Management Functions	
_aligned_free	_aligned_recalloc
_aligned_free_dbg	_aligned_recalloc_dbg
_aligned_malloc	_calloc_dbg
_aligned_malloc_dbg	_free_dbg
_aligned_offset_malloc	_malloc_dbg
_aligned_offset_malloc_dbg	_realloc_dbg
_aligned_offset_realloc	_recalloc_dbg
_aligned_offset_realloc_dbg	calloc
_aligned_offset_recalloc	free
_aligned_offset_recalloc_dbg	malloc
_aligned_realloc	realloc
_aligned_realloc_dbg	recalloc

C++ Language Features	
operator delete	operator new
operator delete[]	operator new[]

COM-related Functions	
SafeArrayCreate	SysFreeString
SafeArrayCreateVector	SysReAllocString
SafeArrayDestroy	SysReAllocStringLen
SysAllocStringByteLen	VariantClear
SysAllocStringLen	VariantCopy
SysAllocString	

Fortran Memory Management Functions	
allocate	deallocate
allocatable	

Global and Local Functions	
GlobalAlloc	LocalAlloc
GlobalFree	LocalFree
GlobalLock	LocalLock
GlobalRealloc	LocalReAlloc
GlobalUnlock	LocalUnlock

Heap Functions	
HeapAlloc	HeapFree
HeapCreate	HeapReAlloc
HeapDestroy	

oneAPI Threading Building Blocks (oneTBB) Memory Management Functions	
scalable_aligned_free	scalable_free
scalable_aligned_malloc	scalable_malloc
scalable_aligned_realloc	scalable_posix_memalign
scalable_calloc	scalable_realloc

Resource Management Functions	
_close	CreateRectRgnIndirect
_creat	CreateRemoteThread
_open	CreateRestrictedToken
_sopen	CreateRoundRectRgn
_sopen_s	CreateSemaphoreA
_wcreat	CreateSemaphoreW
_wopen	CreateSolidBrush
_wsopen	CreateThread
_wsopen_s	CreateToken
BeginUpdateResourceA	CreateToolhelp32Snapshot
BeginUpdateResourceW	CreateWaitableTimerA
CloseEventLog	CreateWaitableTimerW
CloseHandle	DeleteCriticalSection
CreateBitmap	DeleteDC
CreateBitmapIndirect	DeleteObject
CreateBrushIndirect	DeregisterEventSource
CreateCompatibleBitmap	DuplicateHandle
CreateCompatibleDC	DuplicateToken
CreateConsoleScreenBuffer	DuplicateTokenEx
CreateDCA	EndUpdateResourceA
CreateDCW	EndUpdateResourceW
CreateDIBPatternBrush	ExtCreatePen
CreateDIBPatternBrushPt	ExtCreateRegion
CreateDIBitmap	FindClose
CreateDiscardableBitmap	FindCloseChangeNotification
CreateEllipticRgn	FindFirstChangeNotificationA

Resource Management Functions	
CreateEllipticRgnIndirect	FindFirstChangeNotificationW
CreateEventA	FindFirstFileA
CreateEventW	FindFirstFileExA
CreateFileA	FindFirstFileExW
CreateFileMappingA	FindFirstFileTransactedA
CreateFileMappingW	FindFirstFileTransactedW
CreateFileW	FindFirstFileW
CreateFontA	FindFirstStreamTransactedW
CreateFontIndirectA	FindFirstStreamW
CreateFontIndirectExA	InitializeCriticalSection
CreateFontIndirectExW	InitializeCriticalSectionAndSpinCount
CreateFontIndirectW	LoadBitmapA
CreateFontW	LoadBitmapW
CreateHalftonePalette	LogonUserA
CreateHatchBrush	LogonUserW
CreateIoCompletionPort	OpenBackupEventLogA
CreateJobObjectA	OpenBackupEventLogW
CreateJobObjectW	OpenEventA
CreateMailslotA	OpenEventLogA
CreateMailslotW	OpenEventLogW
CreateMemoryResourceNotification	OpenEventW
CreateMutexA	OpenFileMappingA
CreateMutexW	OpenFileMappingW
CreateNamedPipeA	OpenMutexA
CreateNamedPipeW	OpenMutexW
CreatePalette	OpenProcess
CreatePatternBrush	OpenProcessToken
CreatePen	OpenSemaphoreA
CreatePenIndirect	OpenSemaphoreW
CreatePipe	OpenThread
CreatePolyPolygonRgn	OpenThreadToken
CreatePolygonRgn	OpenWaitableTimerA
CreateProcessA	OpenWaitableTimerW
CreateProcessAsUserA	ReOpenFile
CreateProcessAsUserW	RegisterEventSourceA
CreateProcessW	RegisterEventSourceW

Resource Management Functions	
CreateProcessWithLogonW	RtlDeleteCriticalSection
CreateRectRgn	RtlInitializeCriticalSection
	SetWindowRgn

Virtual Memory Functions	
VirtualAlloc	VirtualFree
VirtualAllocEx	VirtualFreeEx
VirtualAllocExNuma	

See Also

System APIs Supported During Threading Error Analysis

System APIs Supported During Threading Error Analysis

The following table lists the 32-bit and 64-bit Windows* OS threading and synchronization APIs currently supported by the Intel Inspector during threading error analysis. Check the *Release Notes* to see if support for new APIs has been recently added.

If an API is not supported, the Intel Inspector may report false positive (non-existent) data races problems in functions using the API.

C Library Memory Management Functions	
_calloc_dbg	_realloc_dbg
_expand	calloc
_expand_dbg	free
_free_dbg	malloc
_malloc_dbg	realloc

C++ Synchronization Classes	
std::mutex	std::recursive_timed_mutex
std::timed_mutex	std::condition_variable
std::recursive_mutex	std::condition_variable_any

Condition Variable And Slim Reader/Writer Lock Functions	
AcquireSRWLockExclusive	ReleaseSRWLockShared
AcquireSRWLockShared	SleepConditionVariableCS
InitializeConditionVariable	SleepConditionVariableSRW
InitializeSRWLock	WakeAllConditionVariable
ReleaseSRWLockExclusive	WakeConditionVariable

Critical Section Functions	
DeleteCriticalSection	InitializeCriticalSectionEx
EnterCriticalSection	LeaveCriticalSection
InitializeCriticalSection	TryEnterCriticalSection
InitializeCriticalSectionAndSpinCount	

Event Functions	
CreateEventA	PulseEvent
CreateEventW	ResetEvent
OpenEventA	SetEvent
OpenEventW	

Fortran Memory Management Functions	
allocate	deallocate
allocatable	

Handle And Object Functions	
CloseHandle	DuplicateHandle

Interlocked Operation Functions	
InterlockedCompareExchange	InterlockedExchange
InterlockedCompareExchange64	InterlockedExchangeAdd
InterlockedDecrement	InterlockedIncrement

Messaging Functions	
GetMessageA	PostMessageW
GetMessageW	PostThreadMessageA
PeekMessageA	PostThreadMessageW
PeekMessageW	SendNotifyMessageA
PostMessageA	SendNotifyMessageW

Mutex Functions	
CreateMutexA	OpenMutexW
CreateMutexW	ReleaseMutex
OpenMutexA	

Qt* Classes	
QMutex	QSemaphore
QReadWriteLock	QWaitCondition

Semaphore Functions	
CreateSemaphoreA	OpenSemaphoreW
CreateSemaphoreW	ReleaseSemaphore
OpenSemaphoreA	

Thread Functions	
CreateThread	ResumeThread
ExitThread	SuspendThread
OpenThread	TerminateThread

Threadpool Functions (supported only on versions 2.0 - 3.5 of the Microsoft .NET* runtime environment) **Microsoft .NET* 3.5 software support is deprecated in the Intel Inspector and will be removed after August, 2020. This deprecation does not apply to the Intel® VTune™ Profiler. See *Release Notes* for details.**	
CloseThreadpoolWork	SubmitThreadpoolWork
CreateThreadpoolWork	TrySubmitThreadpoolCallback
QueueUserWorkItem	WaitForThreadpoolWorkCallbacks

Virtual Memory Functions	
VirtualAlloc	VirtualFree
VirtualAllocEx	VirtualFreeEx

Wait Functions	
MsgWaitForMultipleObjects	WaitForMultipleObjectsEx
MsgWaitForMultipleObjectsEx	WaitForSingleObject
SignalObjectAndWait	WaitForSingleObjectEx
WaitForMultipleObjects	

See Also

System APIs Supported During Memory Error Analysis

Evaluation Features

Evaluation features are potential new Intel Inspector features:

- You can activate/deactivate by environment variable.
- For which Intel Corporation is actively seeking customer feedback.

Please send your feedback to our Online Service Center at https://supporttickets.intel.com/servicecenter.

Activating and Deactivating Evaluation Features

To activate an evaluation feature:

1. Close the Intel Inspector.

2. Set the environment variable `INSPXE_EXPERIMENTAL` to the appropriate value.
3. Re-open the Intel Inspector.

To deactivate an evaluation feature, close Intel Inspector and remove the environment variable.

Current Evaluation Features

There are no evaluation features at this time.

NOTE
Any evaluation feature descriptions in this section:

- Presume a basic understanding of currently released Intel Inspector features.
- Focus on the differences between evaluation and currently released features, or on how evaluation features enhance currently released features, or both.

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.