



Bug Validator

by

Software Verify

Copyright © 2002-2026 Software Verify Limited

Bug Validator

Application flow tracing for Win32 applications

by Software Verify Limited

Welcome to the Bug Validator software tool. Bug Validator is an application flow tracer. A flow tracer is an application that monitors everything that another application does.

Bug Validator monitors each function call, function return, line visit and exception. At each step the function arguments, local variables and processor registers are stored allowing you to view this data at a later time, perhaps whilst investigating a bug or crash.

We hope you will find this document useful.

Bug Validator Help

Copyright © 2002-2026 Software Verify Limited

All rights reserved. No parts of this work may be reproduced in any form or by any means - graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems - without the written permission of the publisher.

Products that are referred to in this document may be either trademarks and/or registered trademarks of the respective owners. The publisher and the author make no claim to these trademarks.

While every precaution has been taken in the preparation of this document, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use of information contained in this document or from the use of programs and source code that may accompany it. In no event shall the publisher and the author be liable for any loss of profit or any other commercial damage caused or alleged to have been caused directly or indirectly by this document.

Printed: December 2025 in United Kingdom.

Table of Contents

Foreword	1
Part I Overview	2
1 Notation used in this help	3
2 Introducing Bug Validator	4
3 Why Bug Validator?	5
4 What do you need to run Bug Validator?	6
5 How to buy Bug Validator	7
6 How does Bug Validator work?	7
7 What does Bug Validator do?	7
8 Supported Compilers	8
9 User Permissions	8
Part II Getting Started	9
1 Microsoft Compilers	10
2 Other Compilers	10
3 Quick Start	10
Part III The User Interface	13
1 First run configuration	14
2 Menu Reference	22
File menu	23
Launch menu	24
Edit menu	25
Settings menu	26
Managers menu	26
Deploy menu	27
Tools menu	27
Data Views menu	27
Software Updates menu	28
Help menu	29
3 Toolbar Reference	29
4 The status bar	30
5 Keyboard shortcuts	32
6 The main display	32
Icons	33
Execution History	33
Diagnostic	40
Floating Licence	46
7 User Interface Mode	48
8 UX Theme	48

9 Settings	49
Settings Dialog	50
Data Collection	53
Data Collection	53
Custom Hooks	54
Optimisation	58
Applications to Monitor	59
Filters	64
Hooked DLLs	64
Hooked File Extensions	67
Source File Filters	68
Load Settings Pattern Match	71
Instrumentation	74
Hook Insertion	74
Line Hooking	76
Hook Control	77
Hook Safety	79
Instrumentation Logging	80
Symbol Handling	81
Symbol Misc	81
Symbol Lookup	82
Symbol Servers	85
Data Display	89
Display Filters	89
Colours	91
Source Code Brow sing	92
Editing	93
File Locations	95
Path Substitutions	100
Diagnostic	102
Inter-Process Communication	104
ColInitializeEx	104
Data Transfer	106
Warnings	110
Warning	110
Don't Show Me Again	112
Session Already Loaded	113
Third Party DLLs	115
UI Global Hook DLLs	115
Environment Variables	118
Loading and saving settings	119
User Permissions Warnings	119
Session Already Loaded Dialog	120
10 Managers	121
Session Manager	121
11 Deploy	122
12 Tools	126
Edit Source Code	127
Refresh	131
Loaded Modules	132
DLL Debug Information	133
Symbol Path Truncation	138
Instrumentation Logging Data	141

Instrumentation Failure Data	142
Finding functions	143
Address Sanitizer DLLs	145
13 Software Updates	146
14 Loading, Saving, Exporting, Closing	153
Loading and Saving Sessions	154
Exporting	155
XML Export Tags.....	157
15 Starting your target program	157
Launching the program	159
Re-Launching the program	168
Injecting into a running program	169
Waiting for a program	172
Monitor a service	177
IIS	179
Monitor IIS and ISAPI.....	179
Reseting and stopping IIS.....	181
16 Stopping your target program	181
17 Command Line Builder	181
18 Data Collection	185
19 Help	186
 Part IV Command Line Interface	 192
1 Example Command Lines	193
2 Environment variables	195
3 Development environment	196
4 Target Program & Start Modes	198
5 User interface visibility	204
6 Session Management	206
7 Session Export options	207
8 Filter and Hook options	208
9 File Locations	210
10 Command Files	212
11 Help, Errors & Return Codes	213
12 Command Line Reference	216
13 Troubleshooting	218
 Part V API	 220
1 Source Files	222
2 Native API Reference	222
3 Calling the API via GetProcAddress	225
4 Convenience functions	226
 Part VI Working with IIS and Services	 228

1 NT Service API	230
Changes to the NT Service API	233
NT Service API Reference	235
Troubleshooting	239
2 Working with IIS	240
3 Example Source Code	241
Example Service Source Code	242
Example ISAPI Source Code	246

Part VII Examples 248

1 Example Application	249
Building the example application	250
2 Example NT Service	250
Building the example service	250
Building the example client	251
Building the example service utility	251
Monitoring the service	252
3 Example Application Launched from a Service	254
Building the service and application	255
Monitoring the application launched from the service	256

Part VIII Debug Information, Symbols, Filenames, Line Numbers 259

1 Visual Studio	260
2 Visual Basic 6	268

Part IX Frequently Asked Questions 269

1 General Questions	270
2 What file extensions does Bug Validator use for itself?	271
3 Crashes and error reports	272
4 Bug Validator Unrecoverable Error	274
5 What is in bvExceptionLogUI.txt?	275
6 How do I create a Power User on Windows XP?	276
7 Why does Bug Validator fail to load my symbols?	277
8 How do I examine the DbgHelp symbol search path?	279
9 Why are some functions not hooked?	281
10 Why are some lines not hooked?	282
11 Debug symbols and DbgHelp	282
12 Extensions, services and tools	290
13 Working with Debuggers	296

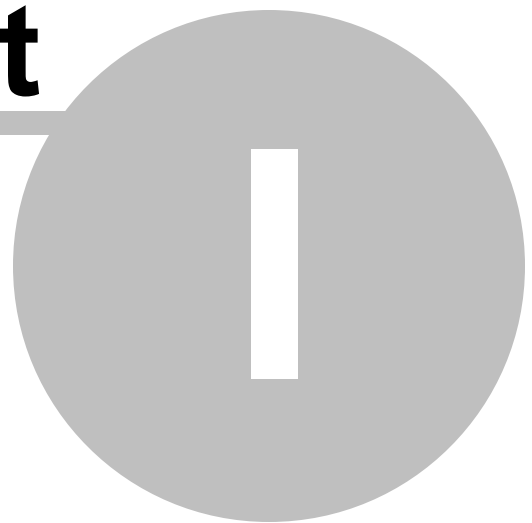
Part X Copyright notices 298

1 Udis86	299

Foreword

This is just another title page
placed between table of contents
and topics

Part



1 Overview

Welcome to the Bug Validator help manual. This section provides a brief overview of the capabilities of Bug Validator.

This help manual is available in Compiled HTML Help (Windows Help files), PDF, and online.

Windows Help	https://www.softwareverify.com/documentation/chm/bugValidator.chm
PDF	https://www.softwareverify.com/documentation/pdfs/bugValidator.pdf
Online	https://www.softwareverify.com/documentation/html/bugValidator/index.html

Tutorials for Bug Validator are available at <https://www.softwareverify.com/tutorial/bug-validator-tutorial/>.

1.1 Notation used in this help

> **Instruction** > **steps**

☰ **Menu action** > **steps**

Throughout the help you'll find instruction steps like this:

- **Filter...** > shows the session comparison private filters dialog

or

☰ **Settings Menu** > **Settings...** > **Data Collection** in the list > **Trace Hooks**

This is a shorthand notation for performing consecutive steps in the user interface.

The first example indicates that the action of clicking the **Filter...** will result in showing the dialog described.

The second example directs you to open the Settings menu (from the menu bar in this case), and then choose the Settings item, and in the dialog that appears, open the Data Collection option via the list and select the Trace Hooks child entry.

Right mouse button menu

Where you see this mouse menu the instruction is to use the right mouse button menu (a.k.a. popup menu or context menu) and select the menu option that follows this symbol.

For example: use  **Edit Source Code...**

Interactive images

Shown next to a picture, the hand symbol  indicates the image is interactive and can be clicked on in order to jump directly to the help section most relevant to the part of the image under the cursor.

External Links

You may see this symbol  after some links. Those links lead to an external website (shown in your default browser), as opposed to jumping to another section in the help. Naturally, if you have no internet access, these links will be unavailable.

For example: Software Verify Limited 

Notes

Warning notes

Notes pertaining to the current topic are indicated by the  symbol. Notes may include exceptions to a rule, items to watch out for, or other asides to the main topic.

Notes that act as warnings will use the similar  symbol, for example where there's a danger of crashing your application. Don't panic though - there aren't many of these!

→ See also

Where there are other pages in the help that have more detail on the topic at hand, or if there is additional reading that is not already linked within the content, you will find these sections linked after the → symbol.

1.2 Introducing Bug Validator

What is Bug Validator?

Bug Validator is an automatic execution history tracer for Windows.

Bug Validator works with versions of Windows from 10.0 through Windows XP, on x86 processors (and compatible).

What does Bug Validator do?

Bug Validator automatically detects which lines of your program have been executed, and records the execution history of the application. Bug Validator places a very low overhead on the performance of your application and does not require the target program to be recompiled or relinked. Bug Validator is intended to be used in software development labs to analyse software development bugs and crashes.

Additionally, the companion software tool Bug Validator Client can be used to analyse bugs that only happen at customer sites.

Bug Validator can decode the logs created by Bug Validator Client, allowing you to collect data at customer sites without compromising the intellectual property in your applications debugging symbols.

The main sections of Bug Validator

The user interface is split into two separate sections, each section dedicated to a different task. The two main sections are:

- **Execution History**

Display the execution history of the application, showing line by line execution history of the application. Selecting a specific line displays the appropriate source code in a syntax coloured window.

- **Diagnostic**

Diagnostic information collected by the stub. Information about lines that could not be hooked is displayed here.

1.3 Why Bug Validator?

Bug Validator allows you to record the execution history of an application and to decode the execution history logs recorded at customer sites using Bug Validator. When you are trying to analyse "customer only" bugs or bugs that crash without throwing exceptions or dropping into the debugger, Bug Validator is a valuable tool.

Sessions can be saved and reloaded in Bug Validator for later analysis. Sessions saved using Bug Validator can be loaded and decoded so that the cause of a crash at a customer site can be analysed. Sessions can be exported to HTML or XML for providing reports that are appropriate for the management team, quality assurance team, software engineering teams.

Reliable

Bug Validator has been created with the following criteria in mind.

1) Bug Validator must have no adverse effect on the program's behaviour.

Any hooks Bug Validator places into the target program's code must not affect the registers or the condition code flags of the program. The program must behave in the same way when being inspected by Bug Validator as when the program is running without being inspected by Bug Validator.

2) Bug Validator must be reliable and avoid causing the target program to crash.

Since we can't know exactly which DLLs and other components are present on every computer that Bug Validator is installed on, we have configured every hook, so that they can be enabled or disabled, and/or installed or not installed.

3) Bug Validator must be capable of having as little impact on the target programs performance as possible.

To do this we allow you to enable and disable as many or as few function hooks as you wish.

4) Bug Validator's user interface must be independent of the target program.

Bug Validator's user interface is independent of the target program.

- If the Bug Validator user interface crashes, your target program will not crash.
- If the target program crashes the Bug Validator user interface will not crash - you will still have data to work with.
- If the target program is stopped in the debugger, Bug Validator's user interface will continue to work.

5) Flexibility

Where there are multiple ways of presenting the data, the user should be given a choice over how that display works. Not all users like the same choices, so providing some choice over the display is always better than forcing all users to use the same settings.

1.4 What do you need to run Bug Validator?

- Microsoft® Visual Studio®

Bug Validator requires your application to be built using Microsoft® Visual Studio® 6.0 service pack 3 or later.

In practice you may find that applications built with Developer Studio 4.2 and later can be used with Bug Validator.

- User Privileges

Bug Validator uses the **CreateRemoteThread()** Win32 function. You must have access privileges that allow you to create threads in other programs. Typically **Administrator** and **Power User** user types have the appropriate privileges. Ordinary user accounts can be easily modified to have the required privileges. User Privileges are discussed in detail here.

- Registry Access Privileges

Bug Validator requires read access and write access to **HKEY_CURRENT_USER\Software\SoftwareVerification\CrashValidator**.

- Operating System

Because the **CreateRemoteThread()** Win32 function and named pipes are not available on Windows 95®, Windows 98® and Windows Me®, Bug Validator will not run on these platforms.

Bug Validator requires Windows NT® 4.0 or better.

We recommend that the minimum service pack levels are used.

- Windows NT 4.0, Service Pack 6
- Windows 2000, Service Pack 2
- Windows XP and later, no service packs issued at time of product release.

1.5 How to buy Bug Validator

Bug Validator is experimental software and cannot be purchased at this time.

Software Verify Limited
Suffolk Business Park
Eldo House
Kempson Way
Bury Saint Edmunds
IP32 7AR
United Kingdom

email sales@softwareverify.com
web <https://www.softwareverify.com>
blog <https://www.softwareverify.com/blog>
twitter <http://twitter.com/softwareverify>

1.6 How does Bug Validator work?

Bug Validator is a multi-part program. One part of the program (known as "the stub") is injected or linked into the target program and communicates with the Bug Validator user interface.

The stub is typically injected into the target program using the **CreateRemoteThread()** Win32 function. Communication between the stub and the user interface is done via named pipes. There is no human readable data format for the communication between the two parts of the program. Both the stub and the user interface are multithreaded. If required the stub can be linked into the program so that it does not need to be injected into the program.

The stub walks the entire program image detecting the start of each source code line using PDB and/or MAP files. Each line is checked to see if it can safely be hooked without corrupting the code for another line or function, or changing the function of the program. If the line can be hooked, it is hooked. If the line cannot be hooked the user interface is informed of the line hook failure.

As your program executes the hooks that were inserted track the program's execution and write it to a buffer that the user interface reads.

1.7 What does Bug Validator do?

Bug Validator provides functionality to track an application's execution history for Win32 programs running on Windows NT® 4.0, Windows 2000® and Windows XP. Bug Validator can also read and decode the encoded session logs created by Bug Validator, allowing bugs that only occur at customer sites to be analysed without compromising intellectual property contained in debugging symbols.

The execution history is displayed as a list, with the source code corresponding to any particular line shown in a window to the right.

There is no requirement to relink or recompile your application to use Bug Validator. All that is required is PDB files with debug information and/or MAP files with line number information (linker option /MAPINFO:LINEs).

1.8 Supported Compilers

Bug Validator will work with any PE format executable. Bug Validator supports C++ and Visual Basic.

C/C++ runtime functions are provided by individual compiler vendors. The following compilers are supported by Bug Validator.

- **Microsoft Developer Studio 4.0**
- **Microsoft Developer Studio 5.0**
- **Microsoft Developer Studio 6.0**
- **Microsoft Visual Studio 7.0 / .net 2002**
- **Microsoft Visual Studio 7.1 / .net 2003**
- **Microsoft Visual Studio 8.0 / .net 2005**
- **Microsoft Visual Studio 9.0 / .net 2008**
- **Microsoft Visual Studio 10.0 / .net 2010**
- **Microsoft Visual Studio 11.0 / .net 2012**
- **Microsoft Visual Studio 12.0 / .net 2013**
- **Microsoft Visual Studio 14.0 / .net 2015**
- **Microsoft Visual Studio 15.0 / .net 2017**
- **Microsoft Visual Studio 16.0 / .net 2019**
- **Microsoft Visual Studio 17.0 / .net 2022**
- **etc...**
- **Microsoft Visual Studio Express** - supported, but see also: building the examples for Visual Studio Express

<http://www.microsoft.com>

Microsoft Developer Studio and Microsoft Visual Studio products support both C++ and Visual Basic.

Microsoft use Microsoft's PDB proprietary symbol information format. If your compiler uses PDB symbol information it Bug Validator will be able to use that information.

- **Intel performance compiler**

The Intel compiler uses the Microsoft runtime already installed on your computer rather than supply its own.

<http://www.intel.com>

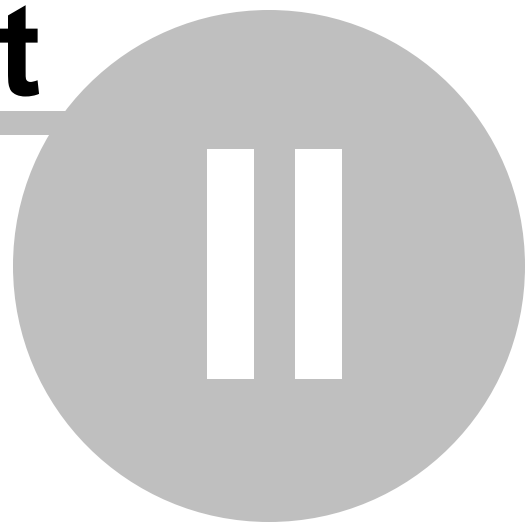
Intel use Microsoft's PDB proprietary symbol information format. If your compiler uses PDB symbol information it Bug Validator will be able to use that information.

If the compiler you are using is not listed here, please contact support@softwareverify.com for advice. We add compilers as we receive requests for them.

1.9 User Permissions

Bug Validator works on any standard user account and on administrator accounts.

Part



2 Getting Started

If you have never used Bug Validator before you have probably acquired Bug Validator because you wish to analyse the flow trace of your application. As such you may want to 'dive in' and identify application flow trace immediately. If you choose to read the manual first you will find out more about the product and how to use it to its full advantage.

For those that wish to 'dive in' this section is for you. For those that do not wish to 'dive in' please skip to the next chapter.

2.1 Microsoft Compilers

When working with applications built using compilers from Microsoft you must ensure that your application is built with debugging information.

Debugging information must be enabled for debug builds (enabled by default) and for release builds (disabled by default).

To enable debugging information for a DLL/EXE, open the project settings/solution for that DLL/EXE, select the build type (debug or release), then modify the settings for the compiler to enable debugging information **and** the settings for the linker to enable debugging information. You must enable **both** settings to generate debugging information.

2.2 Other Compilers

At the time of writing of this help file Bug Validator does not support compilers from any vendor apart from Microsoft (and Intel's Performance Compiler).

2.3 Quick Start

If you want to test the capabilities of Bug Validator you could choose to run the sample program supplied with Bug Validator - **crashValidatorExample.exe**. The program **crashValidatorExample.exe** demonstrates deadlocks, bad lock strategy and related errors..

- Please read these topics before starting:
What do you need to run Bug Validator?
User Permissions
- Your application needs to be compiled to produce debugging information and linked so that debugging information is available.
- If you have no debugging information but you do have a Microsoft format MAP file available the MAP file must contain line number information using the /MAPINFO:LINEs linker directive.

To start your program click on the launch icon on the session toolbar.



The launch program dialog will be displayed. If you have just installed the software you will be shown the launch wizard. If you have switched to Dialog user interface mode you will be shown the launch dialog (shown below).

Start an application and inject Validator into the process

☒ Collect data from application ☐ Collect Stdout

Application (*.exe or *.bat):

Application to monitor (*.exe):

Arguments: Launch count:

Startup Directory:

Environment Variables (override global environment variables):

File to supply to stdin (leave blank for none):

File to supply to stdout (leave blank for none):

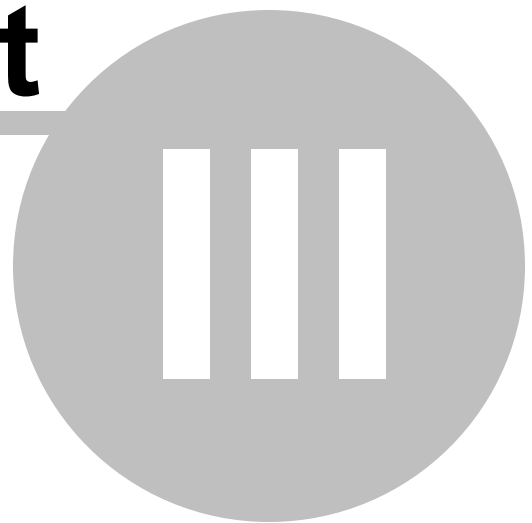
Previously started applications (double click to relaunch) ☐ Full Path ☒ Image Name

Admin	Application	Arguments	Directory	Environment
	bvExample.exe		E:\om\c\bugValidator\bvExample\...	
	dbgHelpBrowser.exe		E:\om\c\dbgHelpBrowser\Release\...	
	ColInitializeTest.exe		E:\om\c\testApps\ColInitializeTest\...	

The picture above shows the launch program dialog. If you wish to know how this dialog works click [here](#).

- 1) Click on the **Start** button to use a file browser to choose the program to launch. The program will be launched automatically.
- 2) Bug Validator will start the target program and inject the stub into the target program. A progress dialog will be displayed whilst the stub is being injected into the target program. The progress dialog lets you know what task it is performing during the injection sequence. When the stub is correctly installed in the target program the stub will establish communications with Bug Validator.
- 3) Data will be collected by Bug Validator until the target program exits.
- 4) When the target program exits, Bug Validator closes the session. The data collection icons on the session toolbar are disabled (the toolbar image will look like the image shown at the top of this section).

Part



3 The User Interface

Bug Validator is a two part program. The part that the user interacts with is the user interface. The other part of Bug Validator is known as the stub. The stub's function is to install and control the data hooks inserted into the target program. The user never interacts directly with the stub, so the stub will not be described in detail. The user influences how the stub behaves via the user interface. This section describes the various functions of the user interface so that you can get the most from using Bug Validator.

Typical usage of Bug Validator is:

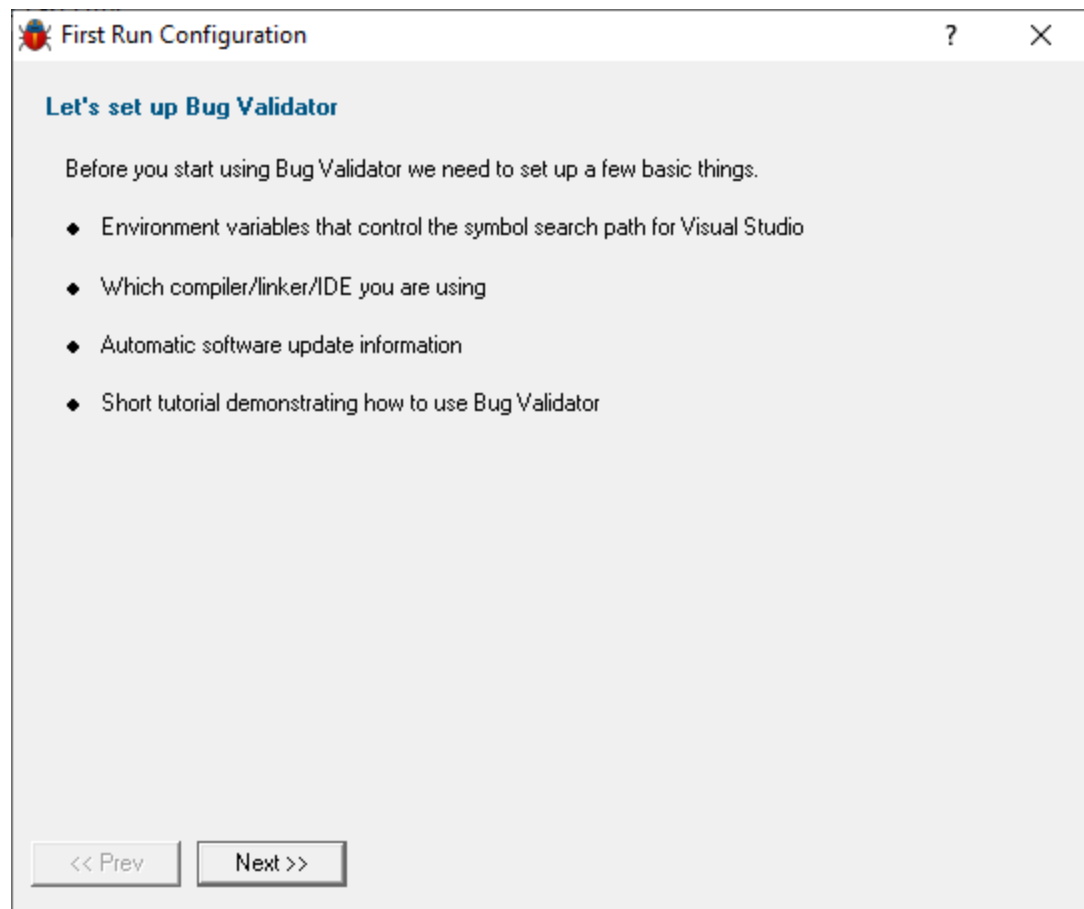
- Start the target program
- Collect the coverage data of the program.
- Close the program

3.1 First run configuration

First run configuration

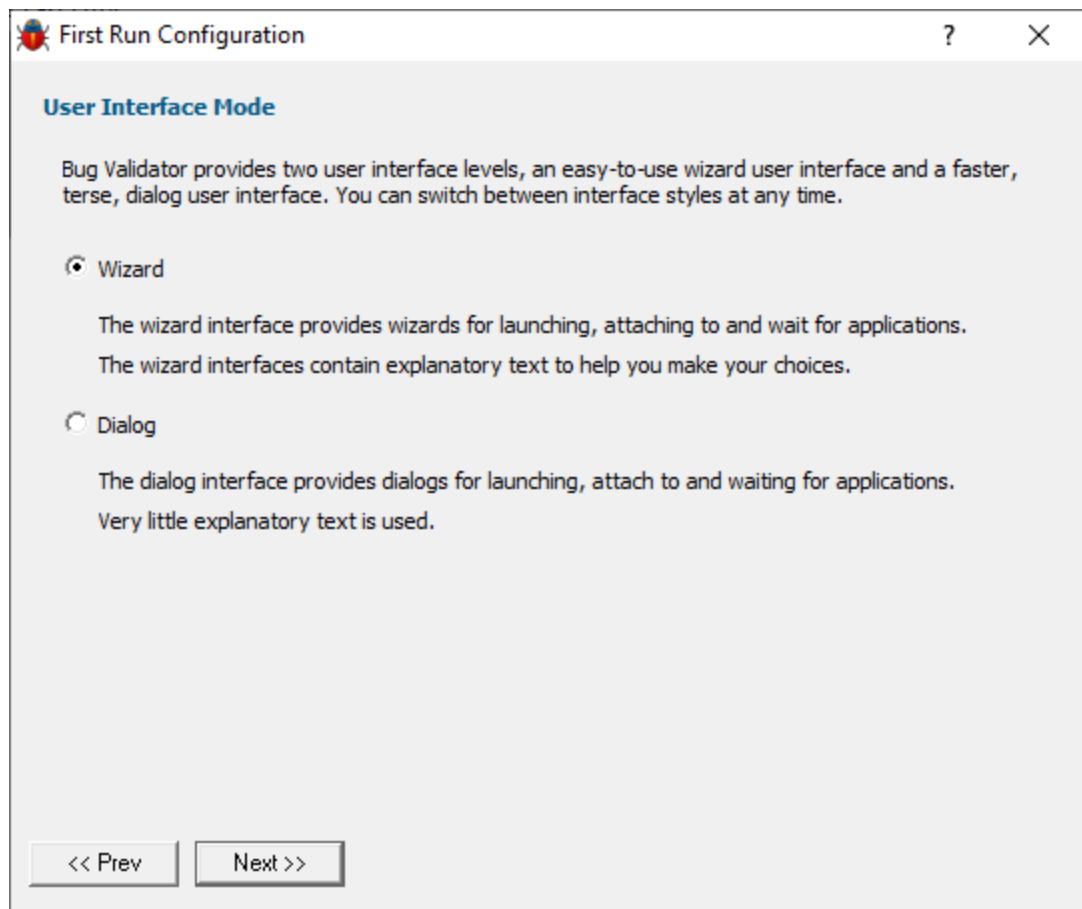
For new users of Bug Validator, a configuration wizard appears the first time you run the application.

The wizard collects a few details about environment, tools, update requirements (for non-evaluation users) and ends with a short video tutorial.



User interface mode

After the introductory page, the wizard presents options for configuring the how the launch, inject and wait dialogs present information to you.



- **Wizard mode** ➤ guides you through the tasks in a linear fashion
- **Dialog mode** ➤ all options are contained in a single dialog

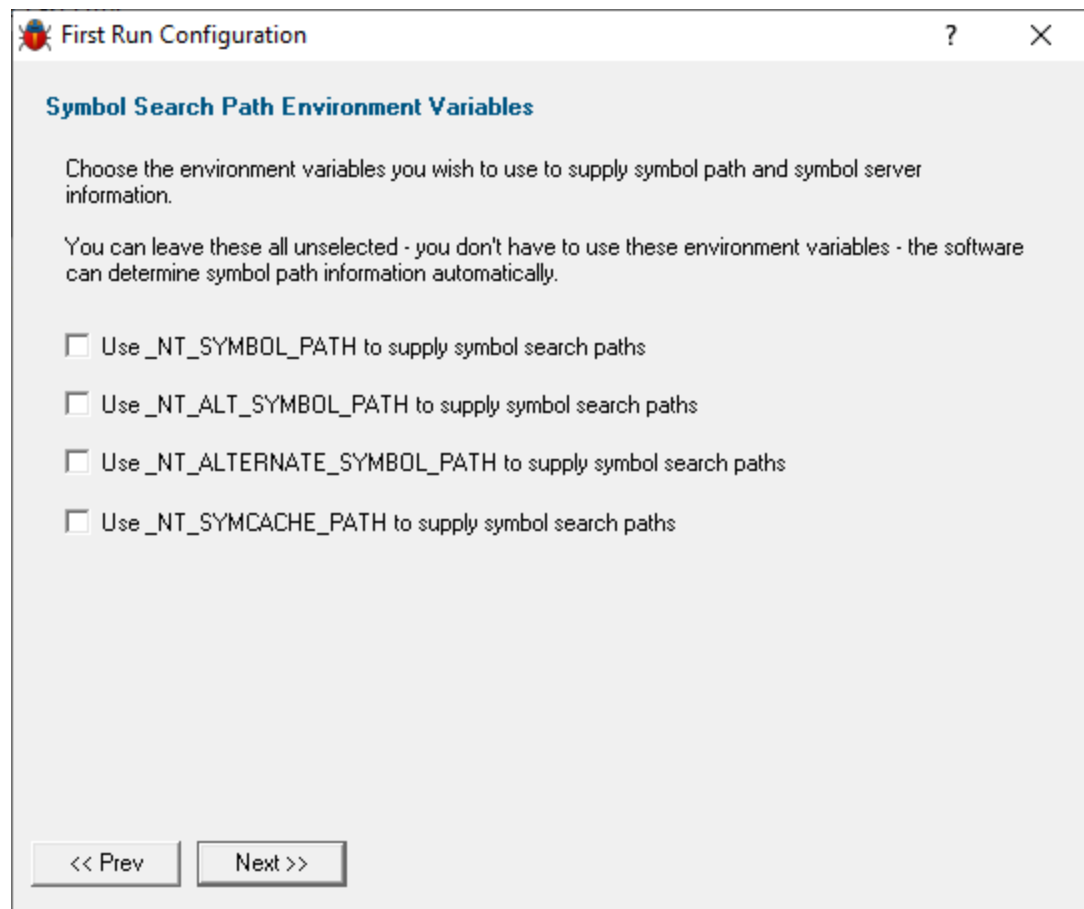
Experienced users will find this mode quicker to use

These settings can be changed at any time via the User Interface Mode option on the Settings menu.

Symbol search path environment variables

The next page of the wizard presents options for using environment variables for symbol search paths when finding PDB symbols.

You don't *have* to choose any of these options as Bug Validator will try to automatically determine symbol path information.



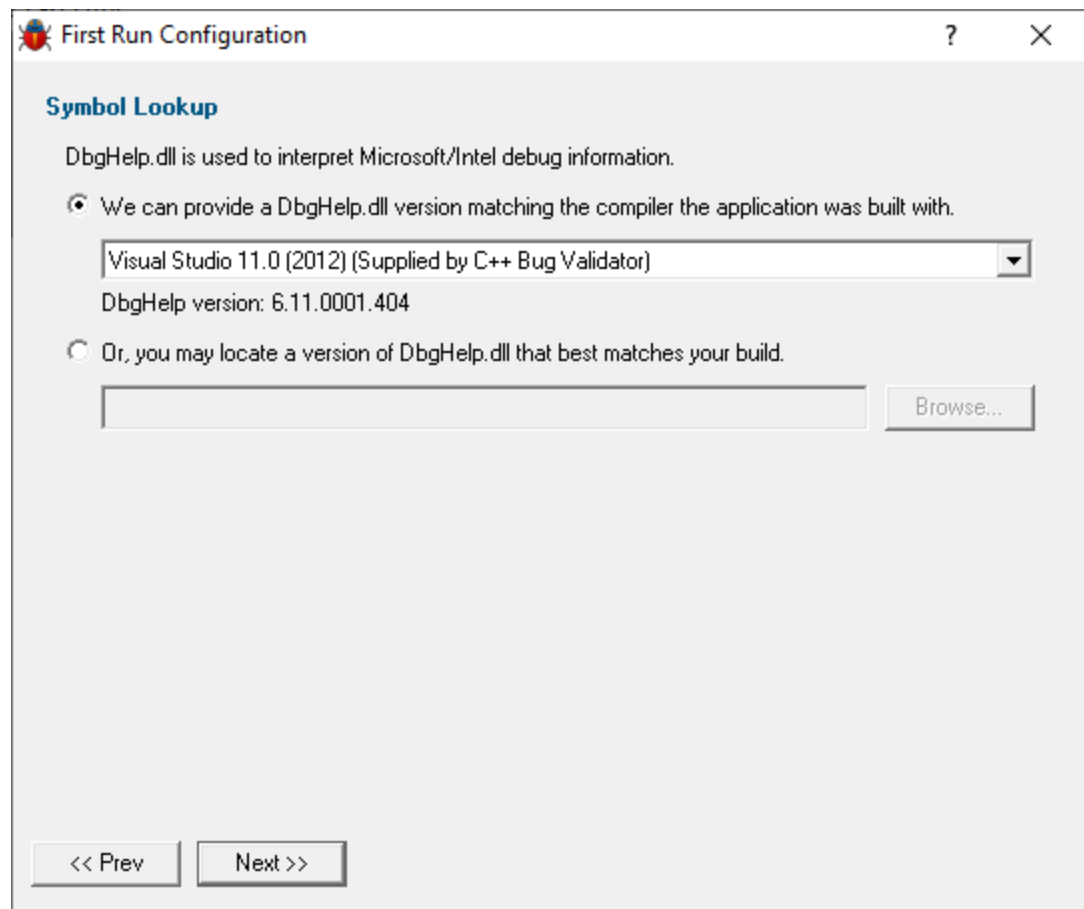
These environment settings can be changed at any time via the Configure Symbol Handling Environment Variables on the Symbol Server page of the Settings Dialog.

Symbol lookup

The next page of the wizard allows you to specify which IDE, Compiler or Linker you're using.

This is important as it affects how symbol lookup is performed. Visual Studio has various quirks in its history of symbol handling and we have to work around that.

The default settings are shown below, although the Visual Studio version may vary.

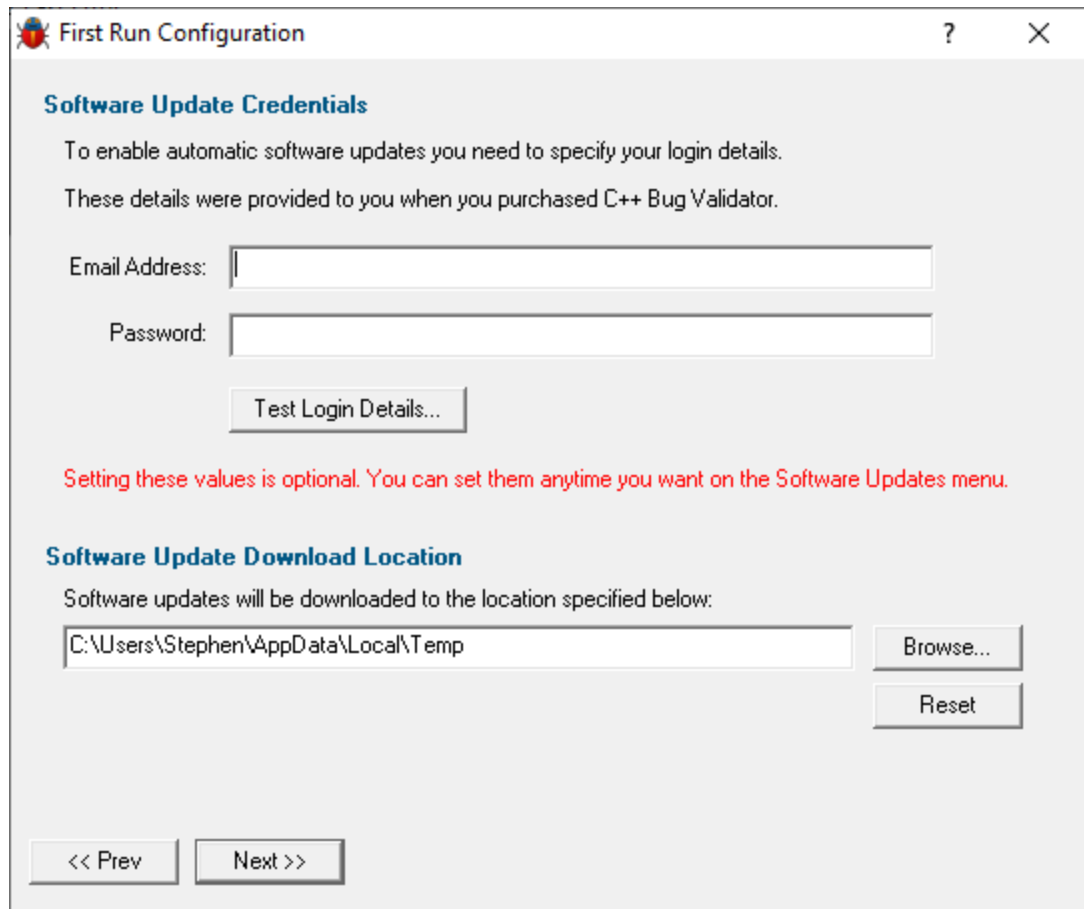


These lookup settings can be changed at any time on the Symbol Lookup page of the Settings Dialog.

Software update credentials

The software updates page of the wizard is only shown to non-evaluation users.

You can configure your software update credentials within the application and where updates are downloaded to.



The 'First Run Configuration' dialog box is titled 'First Run Configuration' and features a bug icon. It contains two main sections: 'Software Update Credentials' and 'Software Update Download Location'. The first section explains that login details are needed for automatic updates and provides input fields for 'Email Address' and 'Password', along with a 'Test Login Details...' button. A red note states that setting these values is optional. The second section explains that updates will be downloaded to a specified location, with a text box showing 'C:\Users\Stephen\AppData\Local\Temp' and buttons for 'Browse...' and 'Reset'. At the bottom are '<< Prev' and 'Next >>' buttons.

First Run Configuration

Software Update Credentials

To enable automatic software updates you need to specify your login details.
These details were provided to you when you purchased C++ Bug Validator.

Email Address:

Password:

Setting these values is optional. You can set them anytime you want on the Software Updates menu.

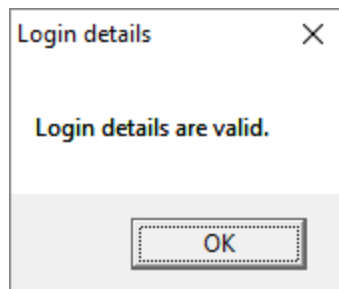
Software Update Download Location

Software updates will be downloaded to the location specified below:

You can test the login details to ensure they are valid:

- **Test login details** ➤ click to check your entered details are valid (requires an internet connection)

Valid details will be confirmed:

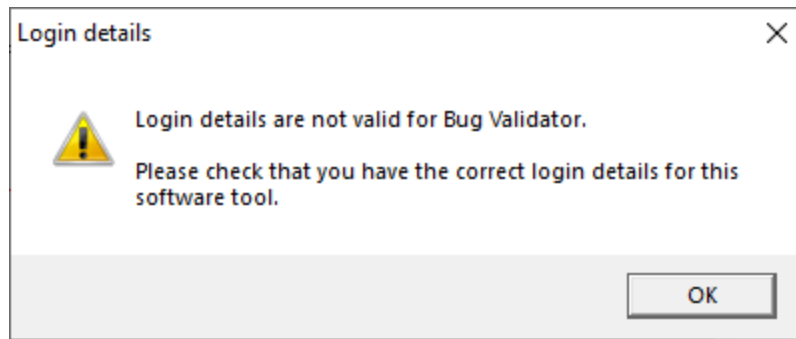


The 'Login details' dialog box is titled 'Login details' and has a close button. It displays the message 'Login details are valid.' in green text. At the bottom is an 'OK' button.

Login details

Login details are valid.

Invalid details may mean you entered credentials for another application in the Validator suite, or they could have been entered incorrectly.



You should have received the correct credentials when you purchased Bug Validator.

If you experience problems, check with your system administrator or contact Software Verify.

These update credentials can be changed at any time from the Software Updates menu.

Software update download location

It's important to be able to specify where software updates are downloaded to because of potential security risks that may arise from allowing the `TMP` directory to be executable.

We use the `TMP` directory as a default, but if you're not comfortable with that you can specify your preferred download directory. This allows you to set permissions for `TMP` to deny execute privileges if you wish.

An invalid directory, e.g. one that does not exist, will show text in red and will not be accepted until a valid folder is entered.

- **Reset** > reverts the download location to the user's `TMP` directory

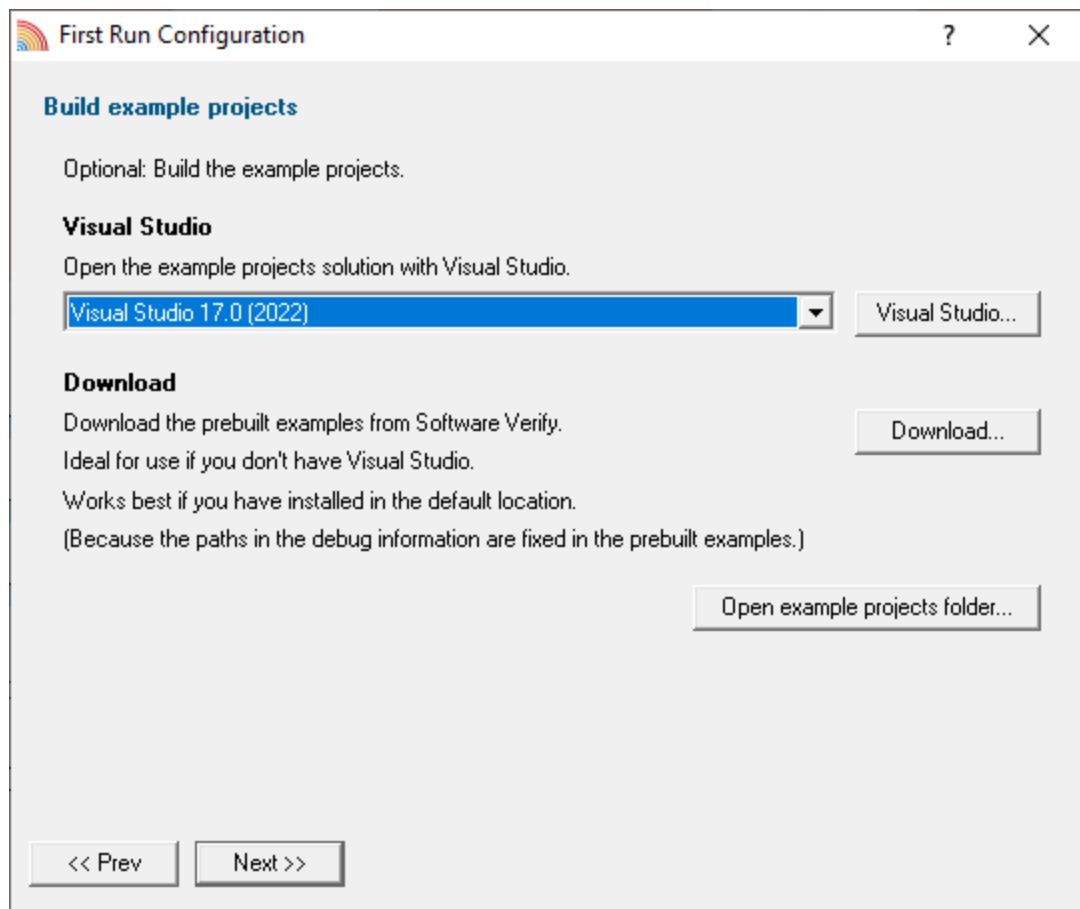
The default location is `c:\users\[username]\AppData\Local\Temp`

The download location can be changed at any time from the Software Updates menu.

Build example projects

The next page of the wizard allows you to build the example projects that ship with Bug Validator.

The example projects demonstrate various application types containing bug you may wish to investigate with Bug Validator.



- **Visual Studio...** ➤ opens the example projects solution with the version of Visual Studio selected
- **Download...** ➤ downloads a prebuilt version of the example projects, unzips them and installs them in the examples folder in the Bug Validator installation directory

If you choose this option and you have not installed Bug Validator in the default location (assuming a 64 bit OS) the source file paths in the debug information will be incorrect - you will need to use the File Locations settings to inform Bug Validator of the correct location(s).

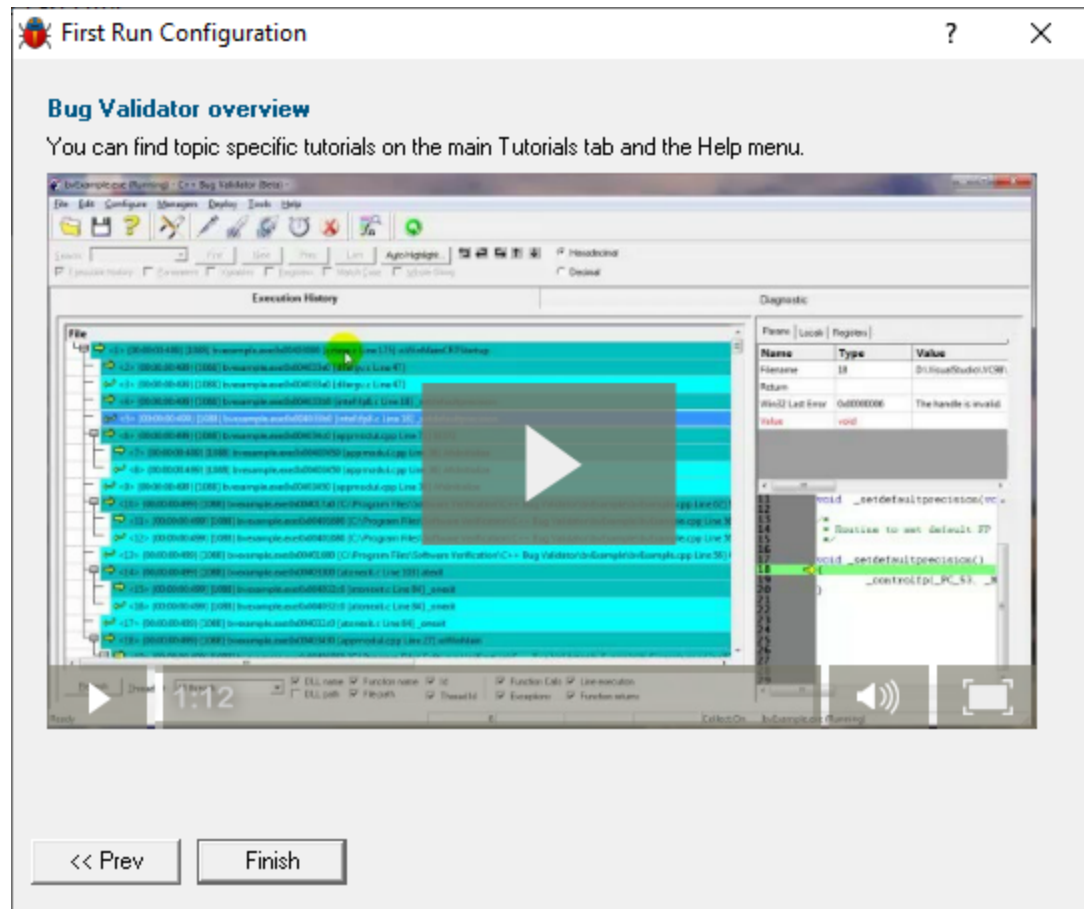
- **Open example projects folder...** ➤ opens the folder that contains all example projects

Video overview of application

The final page of the wizard presents a short video overview of Bug Validator.



The video has audio



More help is available via the Tutorials tab and the Help menu.

The video is also available on the product website. Visit <https://www.softwareverify.com/products.php> and find the product link for Bug Validator.

- **Finish** ➤ closes the First Run Configuration dialog leaving the application ready to use

3.2 Menu Reference

The menus provide access to all the major features in Bug Validator. The most common ones are also directly accessible via the toolbars.

The next few pages provide a convenient collection of links to the detailed help pages on each menu item.

Click in the picture below to jump to any menu's page:



[File](#) [Launch](#) [Edit](#) [Settings](#) [Managers](#) [Deploy](#) [Tools](#) [Data Views](#) [Software Updates](#) [Help](#)

3.2.1 File menu

The **File** menu allows you to:

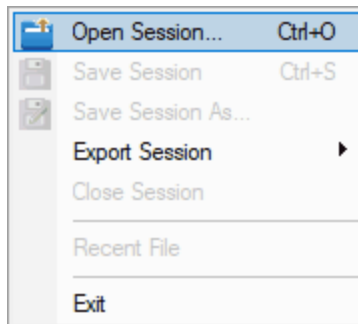
- open, close, export and save sessions
- exit the application

Most of these actions are also available via the standard or session toolbars.

Near the bottom of the menu, a list of recently used file names allows you to easily reload a previously saved session.



Click on an item in the picture below to find out more:



The last two items are not linked to topics. Exit is self explanatory and above that is a list of recently opened files.

3.2.2 Launch menu

The **Launch** menu allows you to:

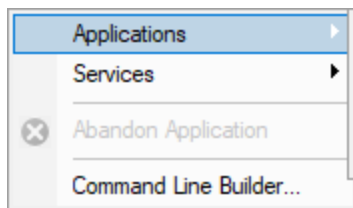
- start applications and restart applications
- inject into running applications
- wait for applications to start then attach to them
- monitor services and ISAPI extensions
- stop monitoring an application

Most of these actions are also available via the standard or session toolbars.

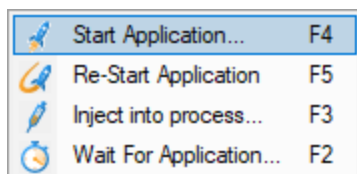
These actions are grouped into submenus according to whether they involve applications or services.



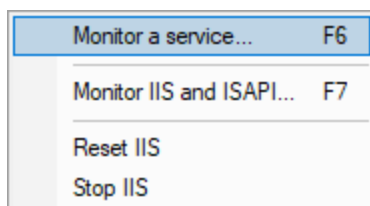
Click on an item in the pictures below to find out more:



Applications



Services



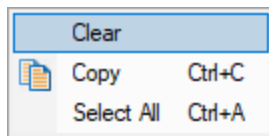
 In addition to the function key short cuts shown above, you can redisplay the previously chosen launch dialog by using **Ctrl + F4**

3.2.3 Edit menu

Selections and the clipboard

The **Edit** menu options can be used to clear all selected items in a table or tree, copy selected items (and relevant data where applicable) to the clipboard, or select all the items available.

Selected data is formatted into one line per row with a single space used to separate column data.



 **Select All** will include the header row as well as the data, and **Copy** will include the column titles.

For example, after running the example application, **Select All** on the Diagnostic Tab might show:

ID	Message
Information	Bug Validator x86 3.98
Information	Windows Version: 10.0 Windows 10
Information	Service Pack: 0.0
Information	Build: 19041
Information	64 bit Operating System
Information	Num Processors: 8
Information	Processor Type: 586
Information	VM Page size: 0x1000
Information	VM Paragraph size: 0x10000
Information	VM Minimum address: 0x00010000
Information	VM Maximum address: 0x7FFEFFFF
Information	16244: MB of physical memory
Process ID	664
User Account	S-1-5-21-3235464415-4188230944-672621225-1001
User Account	ZEUS\Stephen
Command line	nativeExample.exe"
Symbol Search Path	C:\WINDOWS\symbols\dl;
DbgHelp.dll version	C:\Program Files (x86)\Software Verify\Bug Validator x86\examples\nativeExample\ReleaseDynamic10_0\dbghelp.dll
DbgHelp.dll version	DbgHelp.dll version loaded into target: 10.0.16299.91
DbgHelp.dll version	DbgHelp.dll version expected: 10.0.16299.91

This would result in the following being copied to clipboard:

```
ID      Message
Information  Bug Validator x86 3.98
Information  Windows Version: 10.0 Windows 10
Information  Service Pack: 0.0
Information  Build: 19041
Information  64 bit Operating System
```

```

Information    Num Processors: 8
Information    Processor Type: 586
Information    VM Page size: 0x1000
Information    VM Paragraph size: 0x10000
Information    VM Minimum address: 0x00010000
Information    VM Maximum address: 0x7FFFFFFF
Information    16244: MB of physical memory
Process ID     664
User Account   S-1-5-21-3235464415-4188230944-672621225-1001
User Account   ZEUS\Stephen
Command line   nativeExample.exe"
Symbol Search Path C:\WINDOWS\symbols\dll;
DbgHelp.dll version C:\Program Files (x86)\Software Verify\Bug Validator x86\examples\nati
DbgHelp.dll version DbgHelp.dll version loaded into target: 10.0.16299.91
DbgHelp.dll version DbgHelp.dll version expected: 10.0.16299.91

```

3.2.4 Settings menu

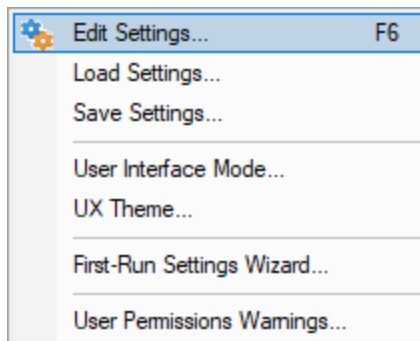
The **Settings** menu allows you to:

- choose the user interface mode (wizards or dialogs)
- change settings for global data and how it is displayed

Settings are also accessible via the session toolbar.



Click on an item in the menu below to find out more:

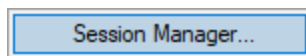


3.2.5 Managers menu

The **Managers** menu allows you to manage all sessions recorded using Bug Validator.



Click on an item in the menu below to find out more:

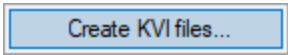


3.2.6 Deploy menu

The **Deploy** menu allows you to manage all tasks related to recording a flow trace on a customer machine without debugging information



Click on an item in the menu below to find out more:



3.2.7 Tools menu

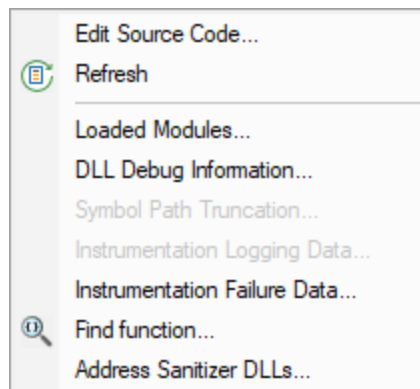
Tools

The Tools menu provides access to a few different tools including a couple not found on the Tools toolbar:

- A list of the modules loaded by your target application
- A list of the debug information status of modules loaded by your application
- A log of files, classes, functions, methods, or modules not instrumented, and reasons why not



Click on a menu item in the picture of the Tools Menu below to find out more:



3.2.8 Data Views menu

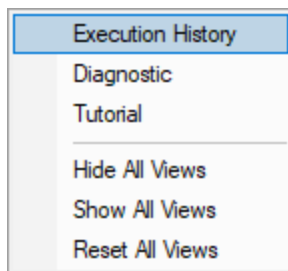
Data Views

The Data Views provides easy control of which tabs are displayed in the main view.

Selecting any of the items shows the relevant tab (if it's not visible already), and makes it the current selected tab.

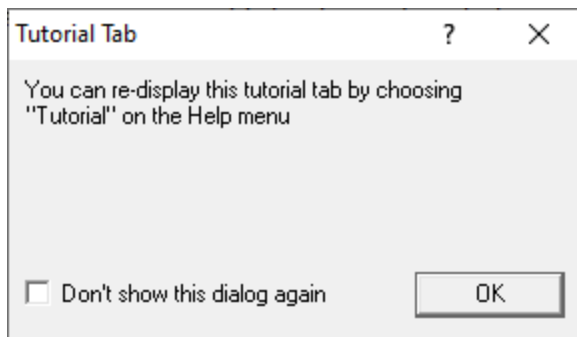
- **Hide All Views** ➤ hides all tabs *except* the one that's currently visible
- **Show All Views** ➤ shows *all* the listed tabs, and in their normal order
- **Reset All Views** ➤ shows only the most *popular* tabs, so excludes the Unit Tests, and the Files and Lines tabs

This is the default setting when you first use the software



When you hide a tab (by clicking the cross on the right of the tab header), you'll initially be reminded of where to go to show it again.

You can choose not to keep seeing this reminder.



The Execution History tab will always remain shown.

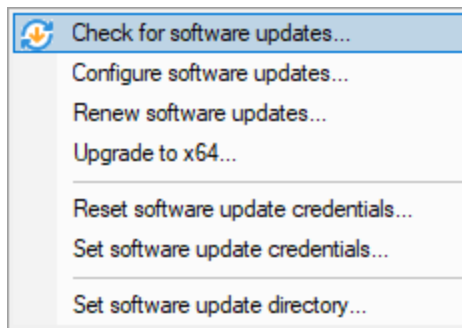
Hidden views are remembered between sessions.

3.2.9 Software Updates menu

Software Updates

All six items in this menu are covered in the Software Updates topic.



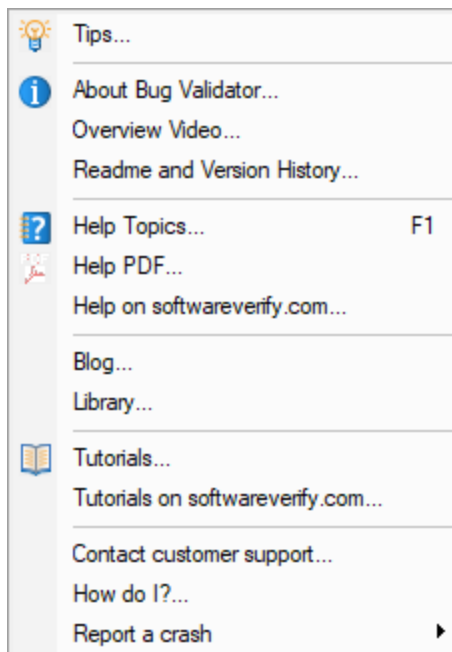


3.2.10 Help menu

Help Menu



Click on an item in the picture below to find out more about each item in the Help topic:



➔ Check out the Frequently Asked Questions too!

3.3 Toolbar Reference

This section lists the various toolbars in Bug Validator and provides links to the appropriate section of the help manual. The icons are described in sequence from left to right in the toolbar.

Standard



- Load session
- Save session
- Help
- Context sensitive help

Session



- Data collection settings
- Inject into application
- Launch application
- Re-Launch application
- Wait for application to start.
- Stop application

Data



- Start Collecting Data
- Stop Collecting Data
- Find function
- Refresh view

3.4 The status bar

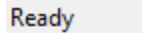
Bug Validator provides various items of status data on the status bar at the bottom of the main window. You can use the data item counts to give a very crude indicator of how data is being collected by the stub and sent to Bug Validator. The number of data items pending processing indicates how much has yet to be processed.

The status bar displays, from left to right, the following items

- Command description.
- Number of data items (symbols and related data) obtained from the target executable.
- Number of data items in the execution history buffer.
- Bytes processed.
- Data collection status on/off.
- Flow trace status.
- Target program name and time stamp.

Command description

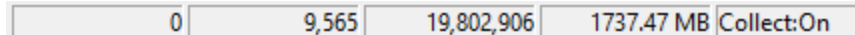
The command description displays **Ready** when awaiting a command. When the mouse moves over a menu item a description of the menu item is displayed in the command description area.


 A rectangular button with a light gray background and the word "Ready" in a dark gray font.

Data collection item count

The next five fields indicate if data collection is on or off and how much data has been collected. The data items displayed are:

- Number of symbols and other related data obtained from the executable.
- Number of data items in the execution history buffer
- Number of data items recorded (count includes those currently in the execution history buffer and those discarded)
- Size of the data buffer
- Data collection status


 A row of five rectangular fields with light gray backgrounds. The first field contains the number "0", the second "9,565", the third "19,802,906", the fourth "1737.47 MB", and the fifth "Collect:On".

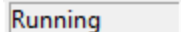
The boxes stay gray when the values are static, but will be coloured for a few seconds when the value changes:

-  The value increased
-  The value decreased

Flow trace status

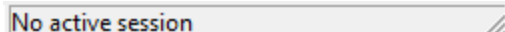
The status of the flow trace indicates what is currently happening.

- **Ready.** Waiting to start a run, or a run has finished and is waiting for you to analyze the data.
- **Starting.** Starting a run (hooks being installed etc).
- **Running.** Target executable is hooked and running.
- **Terminating.** Target executable has entered ExitProcess but has not yet finished executing.
- **Post Processing.** Target executable has finished executing. There is data that still needs to be processed.

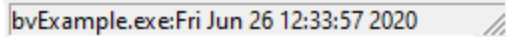

 A rectangular button with a light gray background and the word "Running" in a dark gray font.

Target program name

This displays **No active session** when there is no session running, terminating or loaded.


 A rectangular button with a light gray background and the text "No active session" in a dark gray font.

When a session is running, terminated or loaded, this displays the name of the target program followed by a timestamp.



3.5 Keyboard shortcuts

Keyboard shortcuts

The following shortcuts are available:

 +  Select All

 +  Copy

 +  Open session

 +  Save session

 Help (contextual for current view or dialog)

 Wait for application

 Inject into process

 Start application

 Restart application

 Monitor a Service

 Monitor IIS and ISAPI

3.6 The main display

The main display of Bug Validator consists of two tabbed windows. The tabbed windows are:

- Execution History
- Diagnostic

Each tabbed window can be closed by clicking the small [x] on the right hand side of the tab. The window can be redisplayed by going to the Data Views menu.

The Data Views menu has options for each tabbed window, plus **Hide All Windows**, **Show All Windows** and **Reset All Windows**.

Reset All Windows displays the tabbed windows that are displayed as the default configuration - this typically most of the tabbed windows with the lesser used tabbed windows hidden.

The tabbed window configuration is restored each time the software is closed and reopened, thus preserving any particular tabbed window configuration that is preferred compared to the defaults the software ships with.

3.6.1 Icons

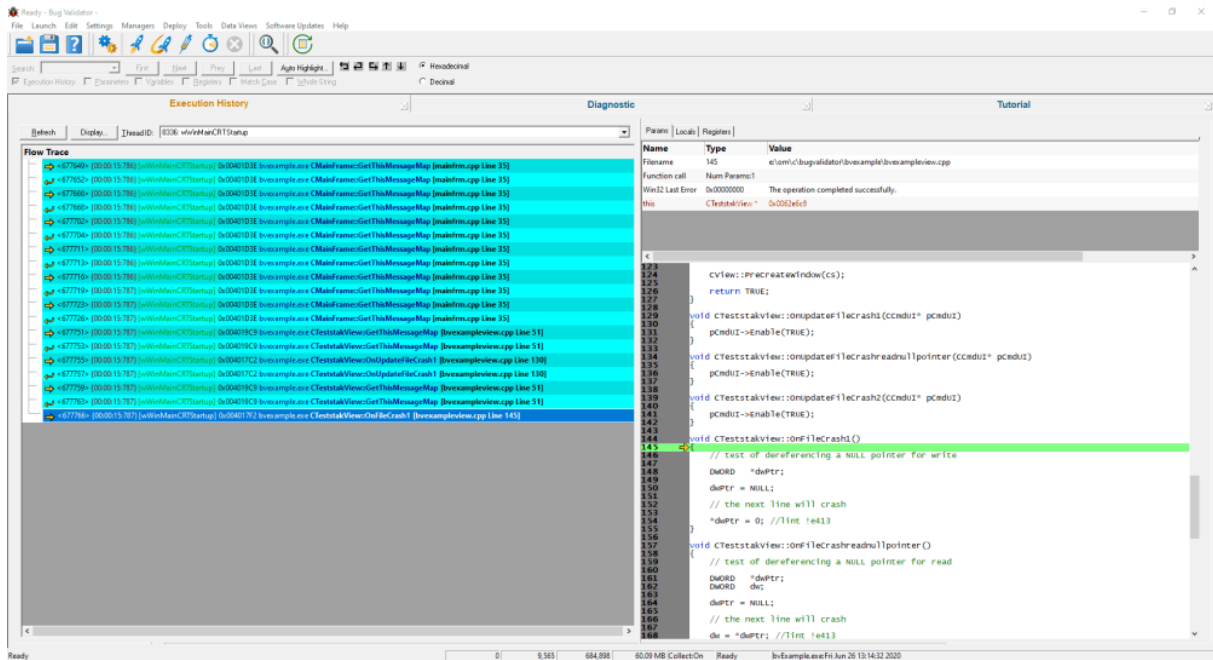
Some of the displays include an icon on the left border of the scrolled list/tree to indicate the type of data that is present on that line. The icons are shown below, with an explanation.

General

-  Option enabled.
-  Option Disabled.
-  Source code line indicator.
-  Source code.

3.6.2 Execution History

The **Execution History** tab on the tabbed window displays information about the program flow trace. This is known as the execution history view.



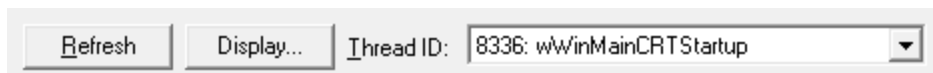
The execution history display is split into two panes. The left hand pane is a tree control showing the execution history lines, the right hand pane displays the parameters, local variables, registers and source code for a particular line. The tree control when expanded shows a small code fragment for the particular line.

Each line displays the thread ID, the DLL name, the program counter, the source file name and the line number for the source file. When multiple threads are displayed, each time the thread ID changes, the background colour for the lines is changed, so that changing execution context is easy to see on the display.

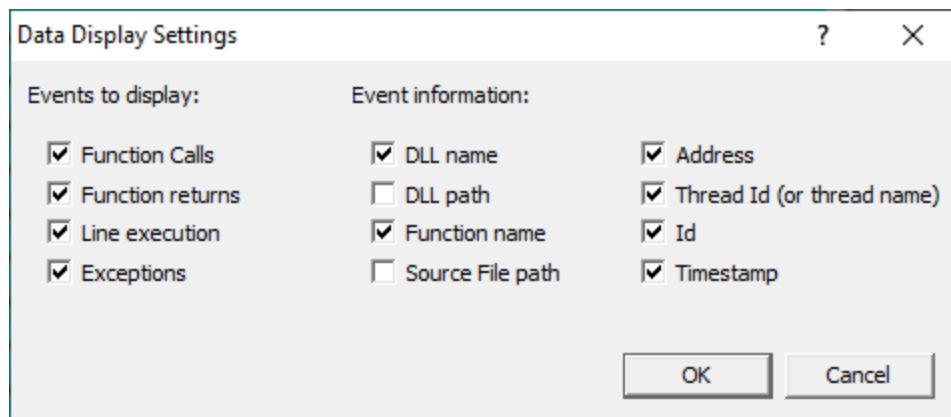
Parameters, local variables and registers are shown in a tabbed window above the source code.

Controls

The **file and line** controls are shown below.



- **Refresh** ➤ causes the display to update to show the most recently collected flow trace history for the specified thread(s).
- **Display...** ➤ shows the display settings dialog.



Each check box controls the display of data on the display.

The first column of check boxes control which events will be displayed.

The other two columns control how much data about each event is displayed.

- **Thread ID** ➤ select the thread flow trace you wish to view.

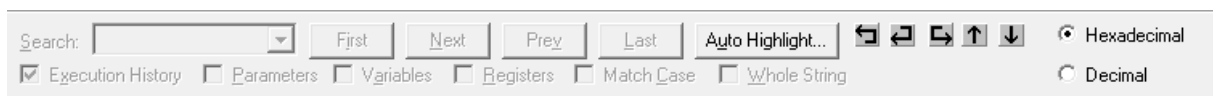
The Thread ID combo box lists the ID of each thread, and it's name (if available).

To show the execution history for all threads, choose **All threads**.

When displaying the execution history for all threads, the execution history shows the order in which each thread swaps in and out of the thread scheduling, this is highlighted by a change in the background colour for each thread.

Search

The search toolbar can be used to search for data in the execution history. Data can also be automatically highlighted by using the **Auto Highlight...** button.



To search for data in the execution history the **Search** field and the **First**, **Next**, **Prev**, **Last** buttons are used. Previous search texts are remembered in the combo box for easy reuse. The history for search texts is 30 items.

The **Match Case** and **Whole String** check boxes affect the pattern matching of the search. Selecting the **Match Case** check box makes text searches case sensitive. Deselecting the **Match Case** check box makes text searches case insensitive. Selecting the **Whole String** check box makes text searches match the entire string. Deselecting the **Whole String** check box makes text searches match the partial strings.

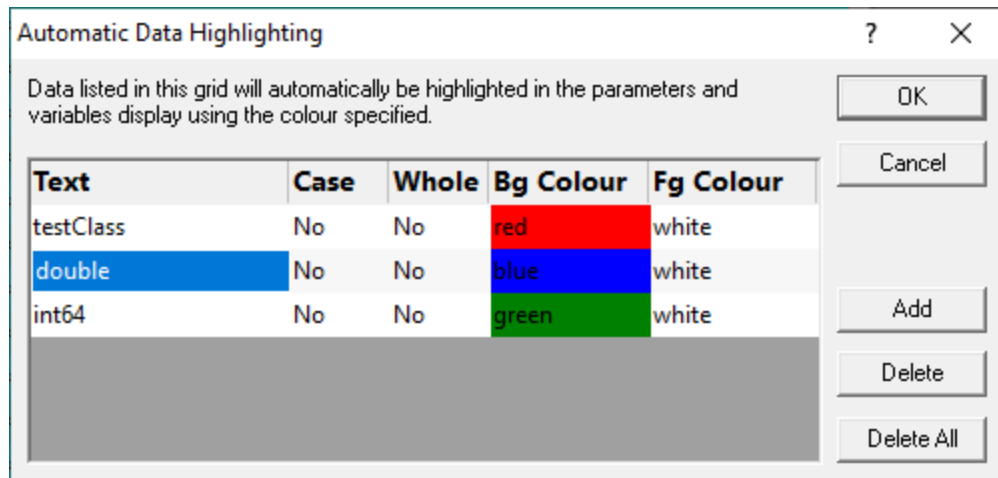
The **Execution History**, **Parameters**, **Variables**, **Registers** check boxes control which data areas are searched for a text match.

The **Hexadecimal** and **Decimal** radio boxes control the display of Parameters, Variables and Registers in hexadecimal and decimal format, respectively.

The navigation buttons provide the following functionality:

-  Go to caller.
-  Go to after caller.
-  Step into.
-  Previous line.
-  Next line.

- **Auto Highlight** > Opens the Auto Highlight dialog.



Data matching the search criteria for each item in the auto highlight dialog is highlighted in the data display above the source code window.

Params Locals Registers		
Name	Type	Value
Filename	338	e:\om\c\dbghelpbrowser\testapp\testapp
Line Execution	Num Vars:15	
Win32 Last Error	0x00000000	The operation completed successfully.
rl	int	0x00000000
tc	testClass	size:12, display:TODO
rS	short	0x0000
tcX	testClassX	size:40, display:TODO

The columns in the Automatic Data Highlighting dialog are as follows:

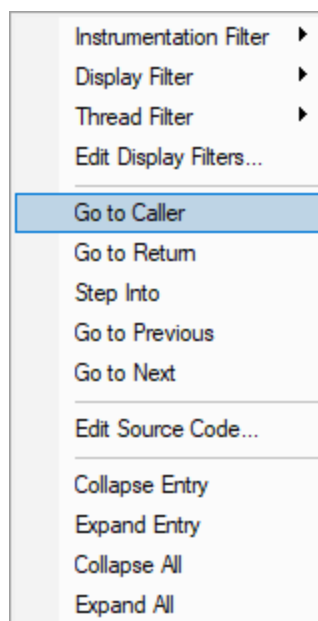
- **Text.** The text to search for.
- **Case.** Yes if the search is case sensitive. No if the search is case insensitive.
- **Whole.** Yes if the text is the entire string. No if the text is a partial match.
- **Bg Colour.** The background colour for any matching data display.
- **Fg Colour.** The foreground colour for any matching data display.

The controls are:

- **Add** ➤ Add an item. Click on each field to edit the contents. The **text** field is an edit box. The other fields are combo boxes.
- **Delete** ➤ Delete all selected items on the display.
- **Delete All** ➤ Delete all items on the display.

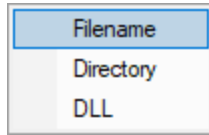
Context Menu

The execution history display provides a context menu.



Instrumentation Filter

The instrumentation filter allows you to prevent various files from being instrumented the next time the application is executed.



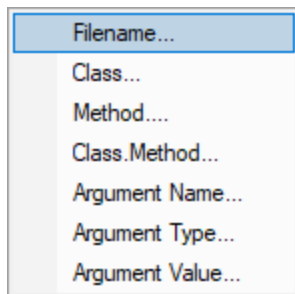
Filename. To exclude the selected item's filename from instrumentation.

Directory. To exclude the selected item's directory from instrumentation.

DLL. To exclude the selected item's DLL from instrumentation.

Display Filter

The display filter allows you to control the information displayed on the execution display.



Filename... To filter using the selected item's filename.

Class... To filter using the selected item's class name.

Method... To filter using the selected item's method name.

Class.Method... To filter using the selected item's class name and method name.

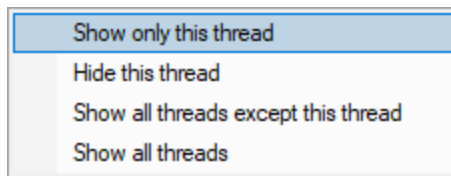
Argument Name... To filter using the selected item's argument name.

Argument Type... To filter using the selected item's argument type.

Argument Value... To filter using the selected item's argument value.

Thread Filter

The thread filter allows you to control which threads are displayed on the execution display



Show only this thread. Shows only this thread and hides all other threads.

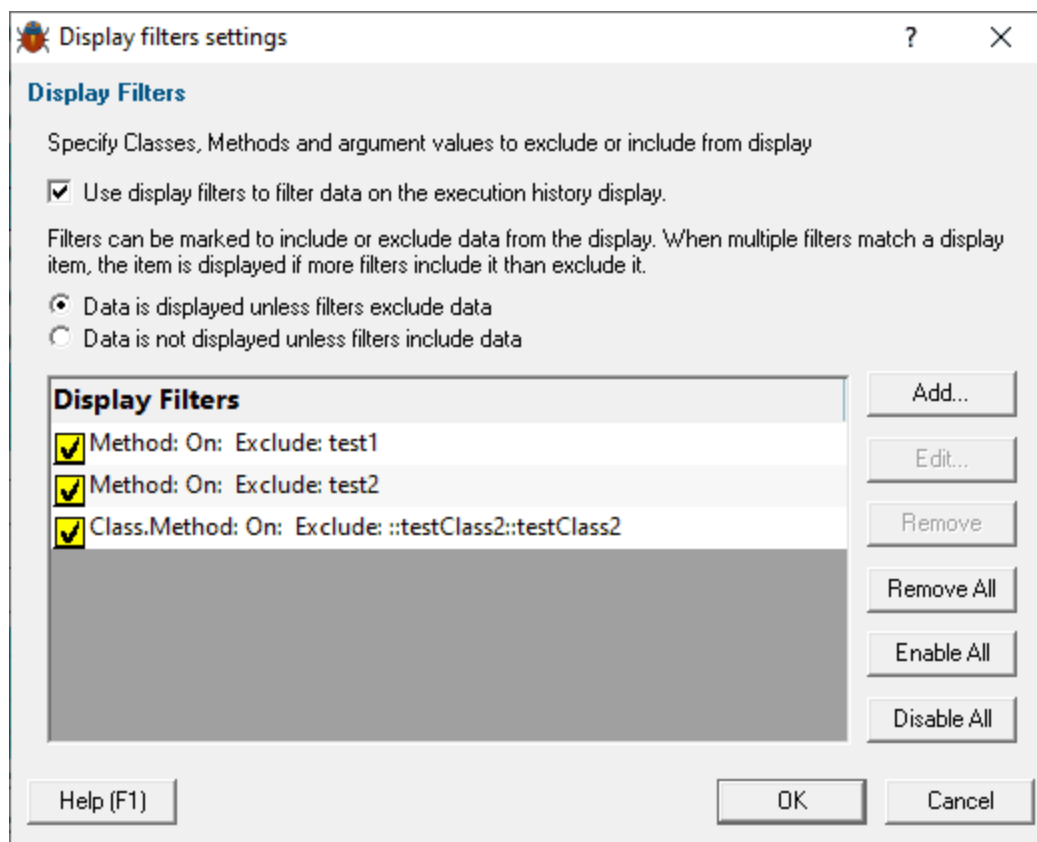
Hide this thread. Hides the selected thread. All other threads remain shown or hidden with no change to their display state.

Show all threads except this thread. Displays all threads except the selected thread.

Show all threads. Displays all threads.

Edit Display Filters...

Opens the display filters dialog.



Go to Caller

Move the current item to the caller of this item.

Go to Return

Move the current item to the return from this item.

Step Into

Move the current item to the first child item of this item.

Go to Previous

Move the current item to the previous item.

Go to Next

Move the current item to the next item.

Edit Source Code...

Edit the source code of this item.

Refresh

Refresh this display with the current data.

Refresh All

Refresh all displays.

Collapse Entry

Collapse this entry.

Expand Entry

Expand this entry.

Collapse All

Collapse all entries.

Expand All

Expand all entries.

3.6.3 Diagnostic

The **Diagnostic** tab displays information collected by Bug Validator about the target program.

There are four sub tabs:

- Diagnostic information
- Data captured from stdout and stderr
- Environment variables
- Child processes

Diagnostic

Diagnostic Stdout	
Show: All	Filter: Apply Filter
ID	Message
Information	Bug Validator 3.76
Information	Windows Version: 6.2 Windows 8
Information	Service Pack: 0.0
Information	Build: 9200
Information	64 bit Operating System
Information	Num Processors: 8
Information	Processor Type: 586
Information	VM Page size: 0x1000
Information	VM Paragraph size: 0x10000
Information	VM Minimum address: 0x00010000
Information	VM Maximum address: 0x7FFEFF
Information	16244: MB of physical memory
Command line	bvExample.exe*
Symbol Search Path	C:\Windows\symbols\dl;
DbgHelp.dll version	E:\om\c\bugValidator\bvExample\ReleaseDynamic10_0\dbghelp.dll
DbgHelp.dll version	DbgHelp.dll version loaded into target: 6.11.1.404
DbgHelp.dll version	DbgHelp.dll version expected: 6.11.1.404
DLL load address	Loaded at 0x00400000 to 0x0040efff: E:\om\c\bugValidator\bvExample\ReleaseDynamic10_0\bvExample.exe
Adding to PDB search path	PDB Search path, add directory:E:\om\c\bugValidator\bvExample\ReleaseDynamic10_0
DbgHelp Search Info	DBGHELP: Symbol Search Path: C:\Windows\symbols\dl;E:\om\c\bugValidator\bvExample\ReleaseDynamic10_0;
Symbol Search Path	C:\Windows\symbols\dl;E:\om\c\bugValidator\bvExample\ReleaseDynamic10_0;
DLL load address	Loaded at 0x77000000 to 0x77199fff: C:\Windows\SYSTEM32\ntdll.dll
DLL load address	Loaded at 0x75e00000 to 0x758bffff: C:\Windows\System32\KERNEL32.DLL
DLL load address	Loaded at 0x76d90000 to 0x76f8dfff: C:\Windows\System32\KERNELBASE.dll
DLL load address	Loaded at 0x746c0000 to 0x7475efff: C:\Windows\SYSTEM32\apphelp.dll
DLL load address	Loaded at 0x74430000 to 0x746b3fff: C:\Windows\SYSTEM32\AcLayers.DLL
DLL load address	Loaded at 0x75d00000 to 0x75e8ffff: C:\Windows\System32\msvcrt.dll
DLL load address	Loaded at 0x75640000 to 0x757d6fff: C:\Windows\System32\USER32.dll
DLL load address	Loaded at 0x74d00000 to 0x74d16fff: C:\Windows\System32\win32u.dll

Diagnostic information

When Bug Validator's stub is injected into the target program, it logs diagnostic information to the main window for inspection.

Examples of diagnostic data collected are below, and may be displayed with a message, although you may not see some of these if all is well:

Hooking information

- Ordinal hook found
- Function hook success or failure
- Delay loaded function hooked
- Possible hook found
- Function already hooked
- Hook at address

Other information

- DLL load address
- DbgHelp searching
- Image source line
- Unknown instruction found
- Disassembly of troublesome code
- Other text message

The locations of loaded DLLs are also displayed in the window for each `LoadLibrary()`, `LoadLibraryEx()` and `FreeLibrary()` in the target program.



If for whatever reason, you don't want to collect diagnostic information, you can switch it off in the Data Collection > Miscellaneous settings.

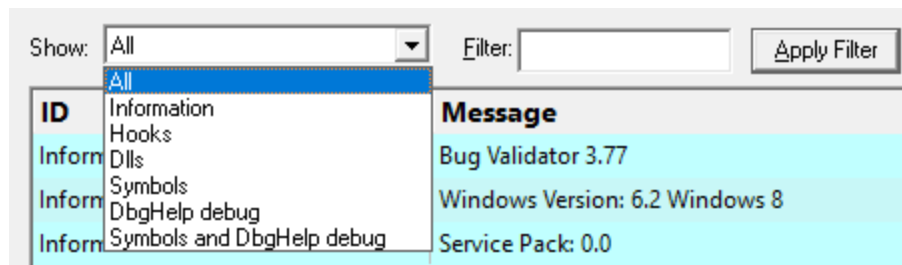
Unhooked lines

Messages indicating that particular source lines could not be hooked are common and are because there was not enough space to safely insert a hook for the particular line.

This is normal and is to be expected. The number of lines that cannot be hooked because of this will vary from program to program, and from debug mode to release mode based on what the program is doing and how the program is optimized.

Filtering diagnostic information

By default, all information is displayed, but you can filter the messages to show only one type:



- **All** > the default
- **Information** > operating system and environment information, etc
- **Hooks** > hooking success and failure messages
- **DLLs** > DLL related information
- **Symbols** > symbol loading status messages
- **DbgHelp debug** > messages from DbgHelp.dll about the DLL symbol search processes
- **Symbols and DbgHelp debug** > both the previous two

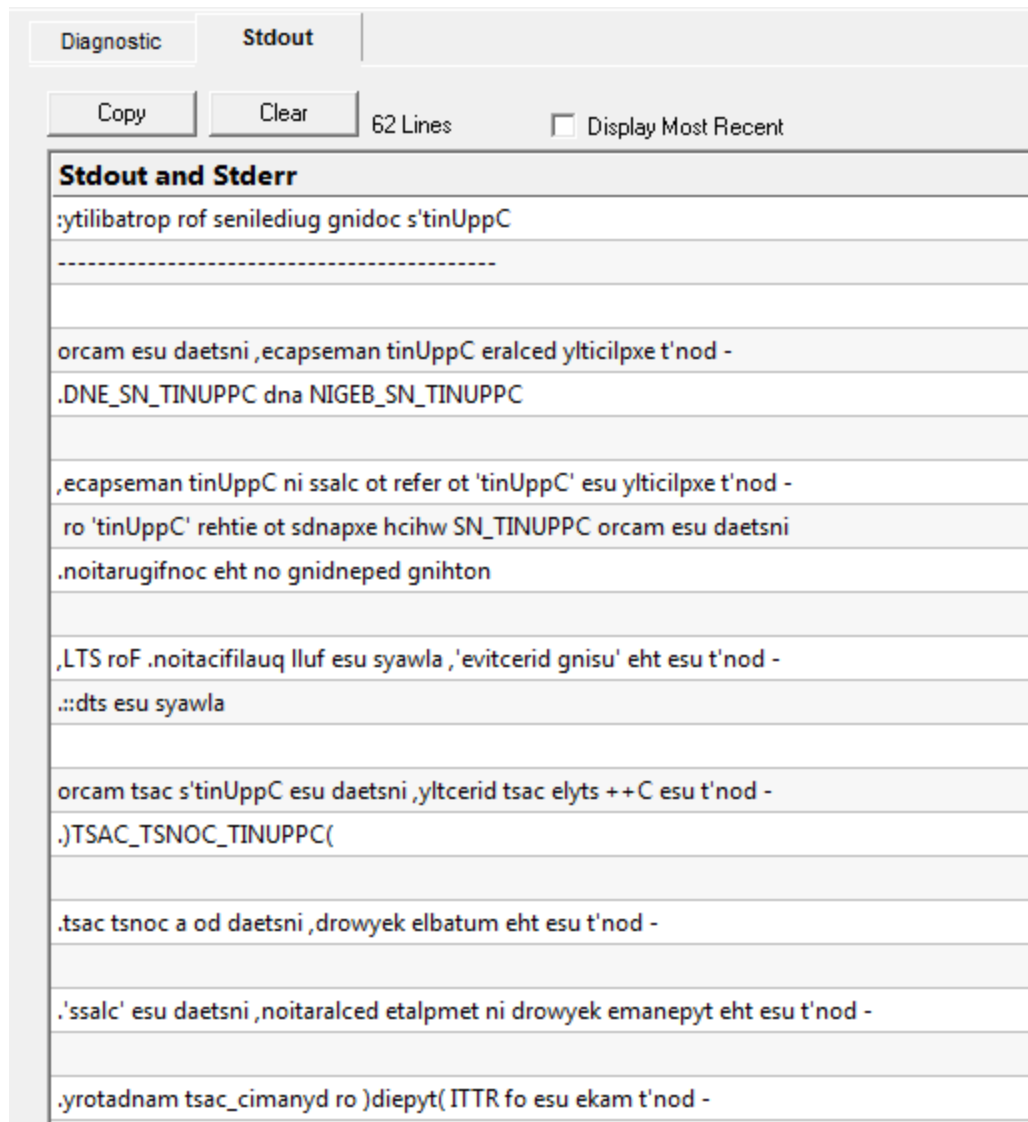
As well as filtering different *types* of lines, you can also search for specific terms:

- **Filter** > enter some text and **Apply Filter** to show lines with the term in the Message column



When identifying why symbols aren't loading, you'll find it's much easier when showing only the **DbgHelp debug** information.

Stdout and Stderr



The **Stdout** tab displays any data collected from stdout and stderr. ➡ The option to enable this data collection is specified on the launch dialog/wizard.

The above image shows some data collected from a program that reverses the characters in each line passed to it.

- **Copy** ➤ copy all data from the display on to the clipboard. For large amounts of data this can be time consuming.
- **Clear** ➤ clear the display of any captured data.
- **Display Most Recent** ➤ the display will be scrolled to ensure the most recently captured data is displayed.

Environment Variables

Environment variables tab displays environment variables from Bug Validator, environment variables from the program under test and environment variable substitution errors.

Choose which data you wish to view using the combo box at the top left of the tab.

Bug Validator environment variables

Diagnostic	Stdout	Env Vars	Child Processes
View: Bug Validator x86			
Bug Validator x86			
Name	Value		
==:	::\		
=D:	D:\		
=Z:	Z:\		
ALLUSERSPROFILE	C:\ProgramData		
APPDATA	C:\Users\stephen\AppData\Roaming		
CommonProgramFiles	C:\Program Files (x86)\Common Files		
CommonProgramFiles(x86)	C:\Program Files (x86)\Common Files		
CommonProgramW6432	C:\Program Files\Common Files		
COMPUTERNAME	DOBRO		
ComSpec	C:\WINDOWS\system32\cmd.exe		
DriverData	C:\Windows\System32\Drivers\DriverData		
FPS_BROWSER_APP_PROFILE_STRING	Internet Explorer		
FPS_BROWSER_USER_PROFILE_STRING	Default		
HOMEDRIVE	C:		
HOMEPATH	\Users\stephen		

Target application environment variables

If you launched the target application from Bug Validator the target application's environment variables will be similar to those in Bug Validator, but with some additional env vars to control .Net profilers and and some other SVL_ prefixed env vars to communicate various data to Software Verify components that are loaded.

If you launched the target application as a standalone application, or service and used one of our APIs to connect to Bug Validator, the environment variables shown will reflect those in force at the time the application/service was started, and the account that application/service is running on.

Diagnostic	Stdout	Env Vars	Child Processes
View: Application being monitored			
Process ID: 3088 e:\om\c\abcMusicTutor\release\abcMusicTutor.exe			
Name	Value		
==:	::\		
=C:	C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE		
=D:	D:\		
=E:	E:\om\c\coverageValidator		
=Z:	Z:\		
ALLUSERSPROFILE	C:\ProgramData		
APPDATA	C:\Users\stephen\AppData\Roaming		
ArmServerInfo	000B0B42		
CommonProgramFiles(x86)	C:\Program Files (x86)\Common Files		
CommonProgramFiles	C:\Program Files (x86)\Common Files		
CommonProgramW6432	C:\Program Files\Common Files		
COMPLUS_ProfAPI_ProfilerCompatibilitySetting	EnableV2Profiler		
COMPUTERNAME	DOBRO		
ComSpec	C:\WINDOWS\system32\cmd.exe		
CORECLR_ENABLE_PROFILING	1		

Environment variable errors

The environment variable errors display shows the name of the environment variable that could not be found, the string containing the environment variable, a comment indicating where the string came from (in this example, the command line), and a timestamp.

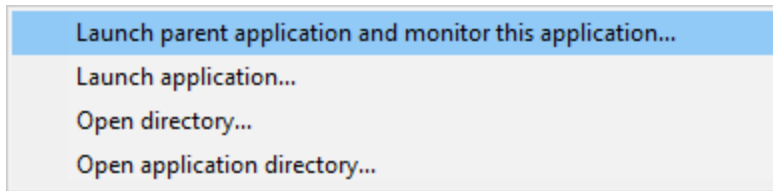
Diagnostic	Stdout	Env Vars	Child Processes
View: Environment variable errors			
Environment variable substitution failures			
Name	Value	Comment	Time
SRC_DIR	e:\SSRC_DIRS\model	Commandline: -fileLocations	2022-02-28:09:20:42
SRC_DIR	e:\SSRC_DIRS\view	Commandline: -fileLocations	2022-02-28:09:20:42
SRC_DIR	e:\SSRC_DIRS\controller	Commandline: -fileLocations	2022-02-28:09:20:42
SRC_DIR	e:\SSRC_DIRS\test	Commandline: -fileLocations	2022-02-28:09:20:42

Child Processes

Information about child processes, and the appropriate launch parameters passed to CreateProcess (and related functions) are displayed on this tab.

Diagnostic	Stdout	Env Vars	Child Processes		
Process ID: 13056 E:\om\c\testApp\testLaunch\other\release\testLaunch\test.exe					
#	Pid	Application	Command Line	Directory	Function
0	9138	testLaunchApp.exe			CreateProcessW
1	13932	E:\om\c\testApp\testLaunch\other\release\testLaunchApp.exe			CreateProcessW
2	12552	dotNetExample10_0.exe			CreateProcessW
3	9972	E:\om\c\testApp\testLaunch\other\release\dotNetExample10_0.exe			CreateProcessW
4	6088	E:\om\c\testApp\testLaunch\other\release\dotNetExample10_0.exe arg1			CreateProcessW
5	6188	E:\om\c\testApp\dotNetTestApp\dotNetCoreSelfContained\ConsoleApp\bin\Debug\netcoreapp3.1\dotNetCoreSelfContained\ConsoleApp.exe			CreateProcessW
6	15076	c:\program files\dotnet\dotnet.exe E:\om\c\testApp\dotNetTestApp\dotNetCoreConsoleApp\dotNetCoreConsoleApp\bin\Release\netcoreapp2.1\dotNetCoreConsoleApp.dll			CreateProcessW

A context menu is provided to allow you to perform some actions with the launched application data.



- **Launch parent application and monitor this application...** ➤ the launch application dialog is displayed configured to launch the parent application and monitor this application
- **Launch application...** ➤ the launch application dialog is displayed configured to launch and monitor this application
- **Open directory...** ➤ Windows Explorer is launched to view the contents of the launch directory (the directory field is empty nothing will be shown)
- **Open application directory...** ➤ Windows Explorer is launched to view the directory that contains the application (if the application specification has no path nothing will be shown)

3.6.4 Floating Licence

The **Floating Licence** view displays information about the computers using the floating licence.

This view is only displayed if a floating licence has been purchased. Evaluation users will not see this view.

Acquire Licence		Release Licence		2 user, LS Floating Licence		Num licensed: 2 (of 2). This computer is licenced.	
User # (max:...	Computer Name	Computer User	Identifier	ID	Software Tool	Computer Id	IP Address
1	DOBRO	Stephen	15838-358c4c8909d5522a8...	15838	Coverage Validator x64	821934:31D7F6f643d7aa	217.155.63.140
2	SURFACEPRO3	Stephen	15838-c95a3beef7ac9d95e...	15838	Coverage Validator x64	C2335E:098624:189ba9d2	217.155.63.140

The screenshot above show two computers using the same 2 user floating licence, that has maintenance id 15838. Both computer users are licenced and can use the software.

On startup the software automatically checks to see if a floating licence is available, and acquires the licence if possible. This takes a few seconds to process, after startup of the software.

An internet connection is required for floating licences to work. The licence server is managed and run by Software Verify.

Licence information

The information show in this display allows you to identify which of your colleagues are using the software and which versions of the software are in use.

- User
The user id (1 to number of licensed users).
- Computer Name
The name of the computer

- **Computer User**
The login name of the user of the computer.
- **Identifier**
The unique identifier for this licence, used on the licence server.
- **ID**
The maintenance id for the software.
- **Software Tool**
The software tool and version of the software that is running on that computer.
- **Computer ID**
The unique id for this computer.
- **IP Address**
This computer's IP address.

Unlicensed users

Acquire Licence		Release Licence		2 user, LS Floating Licence				This computer is not licensed.			
User # (max: 2)	Computer Name	Compute...	Identifier	ID	Software Tool	Computer Id	IP Address				
1	XPS13	Stephen	15838-6516e4b3edba411bc864c47ed365accf	15838	Coverage Validator x64	4E1D96:DA467F:bad1e473	217.155.63.140				
2	SURFACEPRO3	Stephen	15838-c95a3beef7ac9d95ee097fa9770cc512	15838	Coverage Validator x64	C2335E:098624:189ba9d2	217.155.63.140				
Licence not available - Maximum limit reached		DOBRO	Stephen	CVu202311010985ACFd-0000-3dde-0200	15838	Coverage Validator x64	821934:31D7F6:f643d7aa				

If any additional users are trying to get a licence for the software, but there are not enough licences, they will also be shown in the display, but with red text on a yellow background.

Please note that on the machine of an unlicensed user the status information will be different.

The software checks to see if a licence has been released on a periodic basis, so that if a licence is released by another user, it can be acquired by the next waiting user.

Releasing a licence

If you have finished using a licence and wish to let a team mate use the software, you have two choices.

You can close Bug Validator, releasing the licence as it closes.

Or you can keep Bug Validator running by manually releasing the licence. Do this by clicking the **Release Licence** button.

Acquiring a licence

If you have released a licence you will need to actively reclaim a licence when you wish to use Bug Validator again. You can start this procedure by clicking the **Acquire Licence** button.

3.7 User Interface Mode

The user interface provides two modes. These are Wizard and Dialog. The mode controls the way in which data is presented to you when you are setting options that control Bug Validator, and when you are attaching to an application, launching an application or waiting for an application to start.

Wizard

In Wizard mode, the following interfaces are provided as wizard interfaces.

- Attach to application.
- Launch application.
- Wait for application to start.
- Data collection and Data settings dialog. This dialog provides a simplified interface to control commonly used settings of Bug Validator. All other settings are set to the default value.

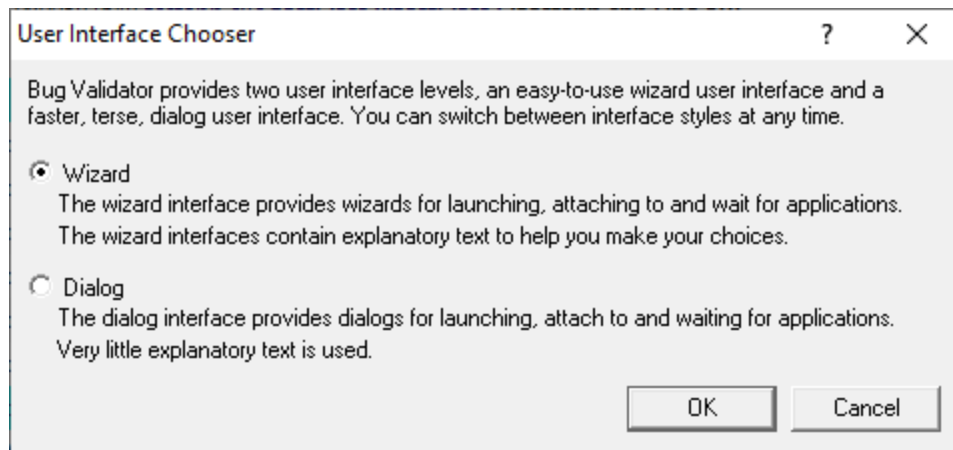
Dialog

In Dialog mode, there are no wizards. The emphasis with Dialog mode is making quick selection of options without lots of explanatory text.

Setting the user interface mode

To set the user interface mode

 **Settings** menu > **User Interface Mode...** > shows the user interface chooser dialog



Select the mode you want to use and click OK.

3.8 UX Theme

The user interface provides three UX themes.

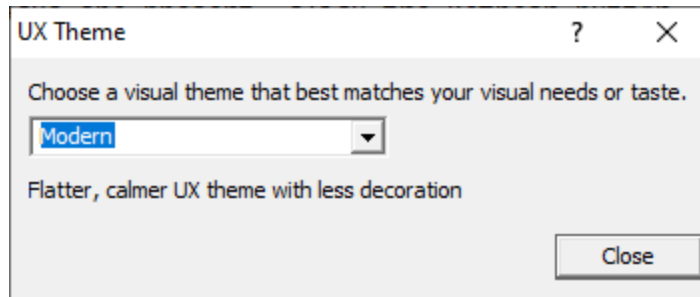
- Modern. The look and feel of current Software Verify tools.
- Classic. The look and feel of previous Software Verify tools.

- High Contrast. A higher contrast version of the Modern theme.

Setting the UX theme

To set the UX theme

 **Settings** menu > **UX Theme...** > shows the UX Theme chooser dialog



Changing the UX theme will update some of the colours that you can modify with the colour settings dialog.

3.9 Settings

Bug Validator allows a fine degree of control of which data is displayed. This control is provided by some global settings which affect all displays on Bug Validator. In addition to the global settings each of the main display windows has its own local controls. This section describes the global settings available from the display settings dialog.

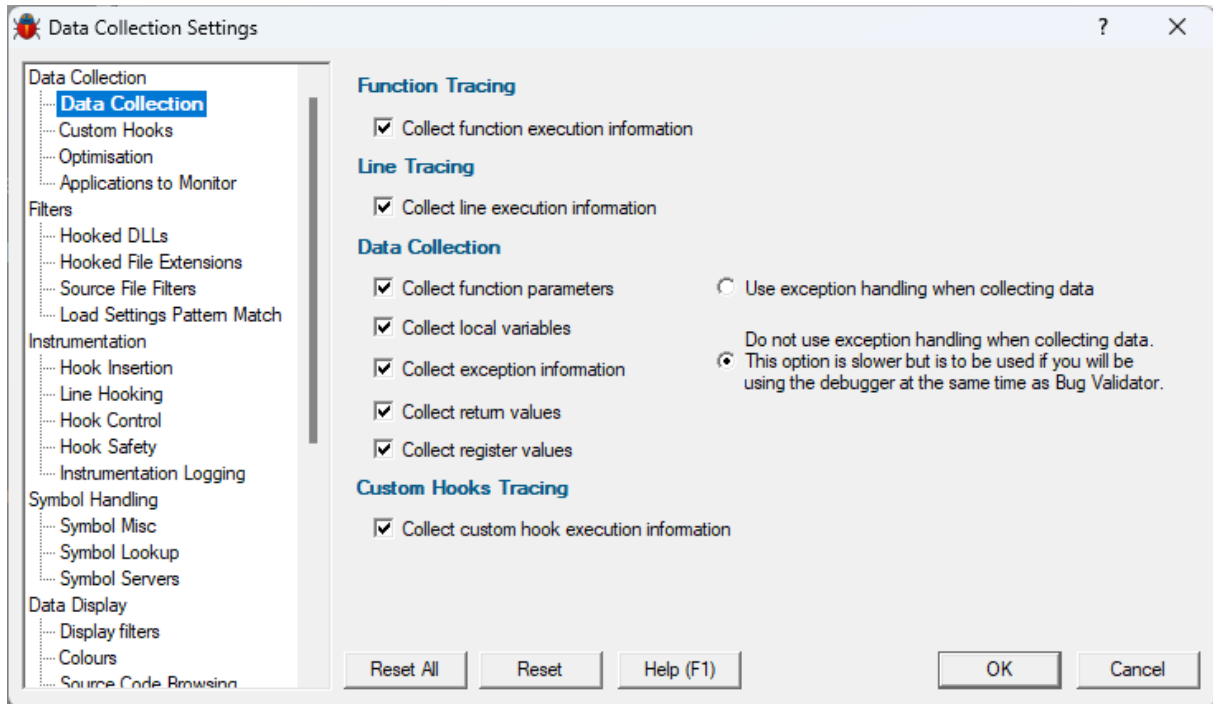
To view the display settings dialog

 **Settings** menu > **Edit Settings...** > shows the edit settings dialog

or click on the display settings icon on the session toolbar.



The settings dialog is displayed.



Reset

The display settings dialog has a button labeled **Reset** at the bottom left of the dialog. This button resets all display related settings in Bug Validator, not just the settings visible on the current tab of the dialog.

3.9.1 Settings Dialog

The Data Collection Settings dialog allows you to control all the global settings in Bug Validator that affect the way data is collected and displayed. There are also local display options on most of the main tabs.



This page has a warning about use of the **Reset** button.

Opening the settings dialog

To view the settings dialog, choose **Settings** menu ➤ **Edit Settings...**

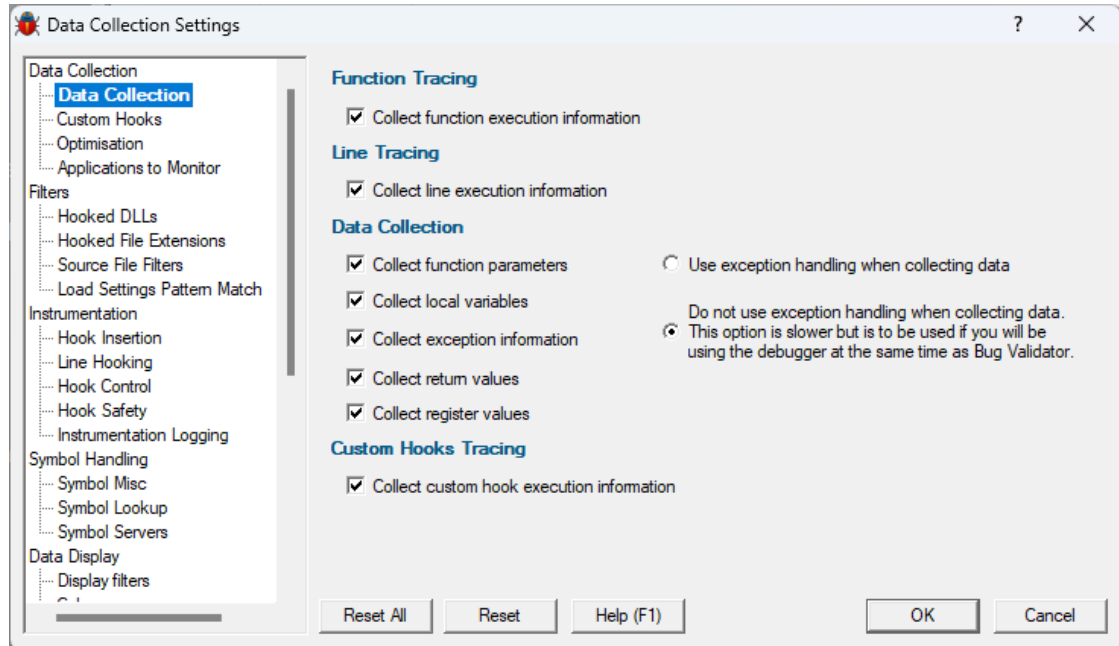
Or use the option on the Session Toolbar:



Using the settings dialog

The dialog has a scrolled list on the left hand side, grouping the topics. When a topic is clicked, its related controls are displayed on the right hand side.

The default display of the dialog is shown below with the first topic selected.



After selecting a topic, you can also use the cursor up and down arrow keys to change the selected item.

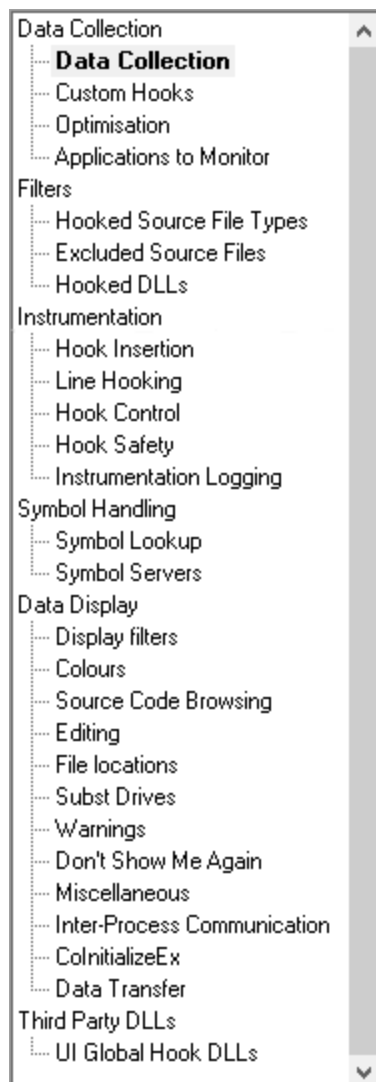
Entering a character on the keyboard cycles through topics starting with that letter.



Too many settings? It may seem that there is an overwhelming number of settings to worry about. Don't panic! The good news is that for new users, very few (if any) settings actually need to be changed to use the application in most cases, and even for experienced users, many groups of settings will not be needed. However, Bug Validator remains flexible for all our users in many different scenarios.



Click on any item in the picture below to find out more about the settings for that group.



Restoring the default settings

The settings dialog has **Reset All** and **Reset** buttons near the bottom left of the dialog which you can use to reset all global settings back to their default values.



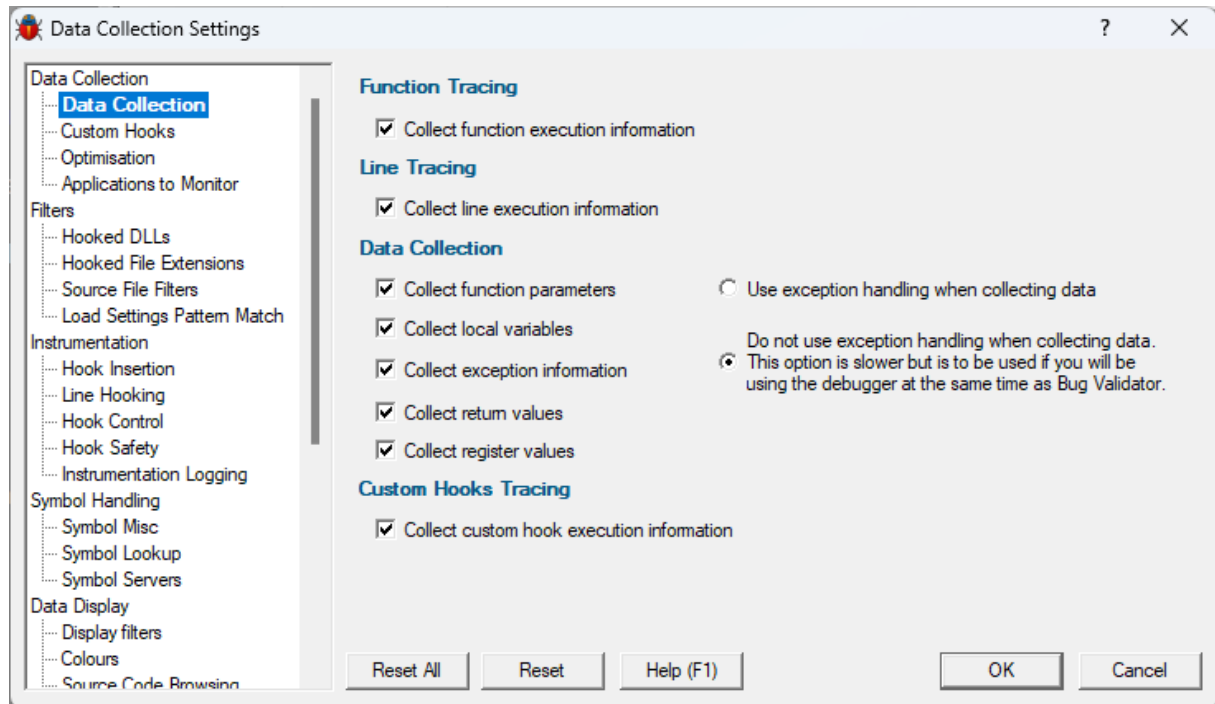
 The **Reset All** button resets **all global settings** in Bug Validator, not just the settings visible on the current tab of the dialog.

 The **Reset** button resets just the settings visible on the current tab of the dialog.

3.9.1.1 Data Collection

3.9.1.1.1 Data Collection

The **Data Collection** tab allows you to configure the type of data collected by Bug Validator.



Bug Validator collects execution information related to function calls, function returns and exceptions automatically. You can also record information about individual line execution by selecting the **Collect line execution information** check box.

Collecting information about function parameters, method variables, return values and exceptions can be useful. However collecting this information takes additional execution time and storing this information uses more space. For this reason Bug Validator allows you to choose which data to collect. Select the appropriate check boxes to enable collecting of function parameters, method variables, exception information, function return values and processor registers.

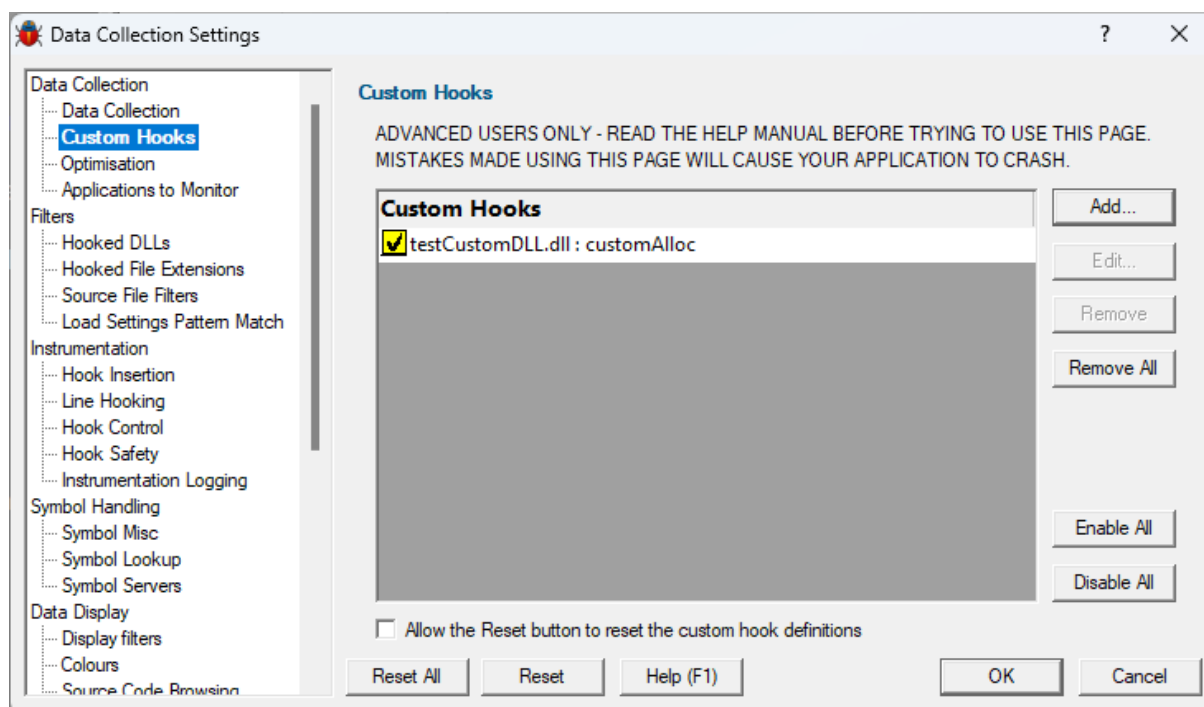
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.1.2 Custom Hooks

The **Custom Hooks** tab allows you to specify hooks for functions that Bug Validator does not know about. This allows you to monitor APIs in 3rd party products and in APIs that are released after Bug Validator was released.

This is a very advanced topic, requiring you to know the parameter list and return type of each function to be hooked and the calling convention for each function. **Failure to specify the information correctly may crash your application.** If in doubt do not attempt to use this feature of Bug Validator. This topic is also discussed in the Bug Validator tutorial, which can be found on the Help menu.



The picture above shows some custom hook definitions in the **testCustomDLL.dll** DLL. Some of the definitions are for functions with **__cdecl** calling conventions and some are for functions with the **__stdcall** calling convention. To edit an existing definition, double click the entry you want to edit, or click the **Edit...** button.

- **Add** ➤ add a new custom hook. The custom hook dialog is displayed.
- **Edit...** ➤ edit the selected custom hook. The custom hook dialog is displayed.
- **Remove** ➤ delete the custom hook that is selected on the display. Alternatively you can press the Delete key on the keyboard to delete the currently selected entry.
- **Remove All** ➤ delete all custom hook definitions.
- **Enable All** ➤ enable all custom hook definitions.
- **Disable All** ➤ disable all custom hook definitions.

Typically, after setting the custom hook definitions, you will not want these reset when you choose to reset the other settings to the defaults by pressing the **Reset** button. However if you do wish to reset the custom hooks at this point you can select the **Allow the Reset button to reset the custom hook definitions** check box.

Custom Hook Dialog

The custom hook dialog allows you to edit the definition of a custom hook.

A custom hook dialog for a function using the `__cdecl` calling convention is shown below. The function prototype for this function is:

```
extern "C" void *customAlloc1(int size);           // input param
```

The function has been specified using the `extern "C"` specifier so that the function name has no C++ name mangling decoration. The function is called **customAlloc1** and is in the DLL **testCustomDLL.dll**, it uses the `__cdecl` calling convention, takes one input parameter. The custom hook is marked as an allocator by the `Alloc` definition. For datatype purposes it has been specified as returning the `void *datatype`. The custom hook will monitor the return parameter, marked as a pointer and the single input parameter, marked as a size specifier.

Custom Hook ? X

DLL Name: testCustomDLL.dll OK

Function Name: customAlloc Cancel

Function Ordinal: -1 (-1 to ignore ordinal)

Calling convention: ☒ CDECL calling convention (caller cleans up stack ☺)
☐ STDCALL calling convention (callee cleans up stack (Win32))
☒ Enabled

Specify the Return type for the function

Name	Type
Return Value	void

Specify parameter types for the function

Name	Type
size	void *

Add
Remove
Remove All

DLL Name

Type the name of the DLL without a file path in the DLL Name field. The name is case insensitive.

Function Name

Type the name of the function in the Function Name field. The name should appear exactly as shown in the Export Address Table of the DLL (or the Import Address Table of any DLL that uses the function).

Function Ordinal

If the function is imported by ordinal (for example `MAPIAllocateBuffer@8` in `MAPI32.DLL`), type the ordinal (as a decimal integer) in the Function Ordinal field. If the function is not imported by ordinal, type -1.

For the example `MAPIAllocateBuffer@8` in `MAPI32.DLL`, the ordinal is 13. The ordinal can be found in the Export Address Table of the DLL.

Calling convention

Select the radio box that corresponds to the calling convention of the function. *If you don't know the calling convention, DO NOT GUESS. Using the wrong calling convention will crash your application.*

Number of Parameters

Type the number of parameters the function takes. This is the number of parameters the function has, not the number of parameters that you want to monitor. *If you don't know the number of function*

parameters, DO NOT GUESS. Using the wrong number of function parameters will crash your application when used with the stdcall calling convention.

Enabled

Select the check box to make the custom hook enabled. Deselect the check box to disable the custom hook. This setting only takes effect each time your application starts - changing this in the middle of a session has no effect.

Return type

The parameters section allows you to define the return value to be monitored.

Parameters

The parameters section allows you to define the parameters to be monitored.

Add

Click **Add** to add a parameter definition.

Remove

Click **Remove** to remove the selected parameter.

Remove All

Click **Remove All** to remove all parameters.

The return types and each parameter definition can be edited by clicking on the field in the grid. A combo box will be displayed with appropriate choices. The various columns are:

Name

This column is the name of the parameter.

Type

This column indicates what datatype the parameter is. The following values are valid.

void
char
wchar_t
TCHAR
int
short
long
WORD
DWORD
float;
double

void *
char *
wchar_t *
TCHAR *
int *
short *
long *
WORD *
DWORD *

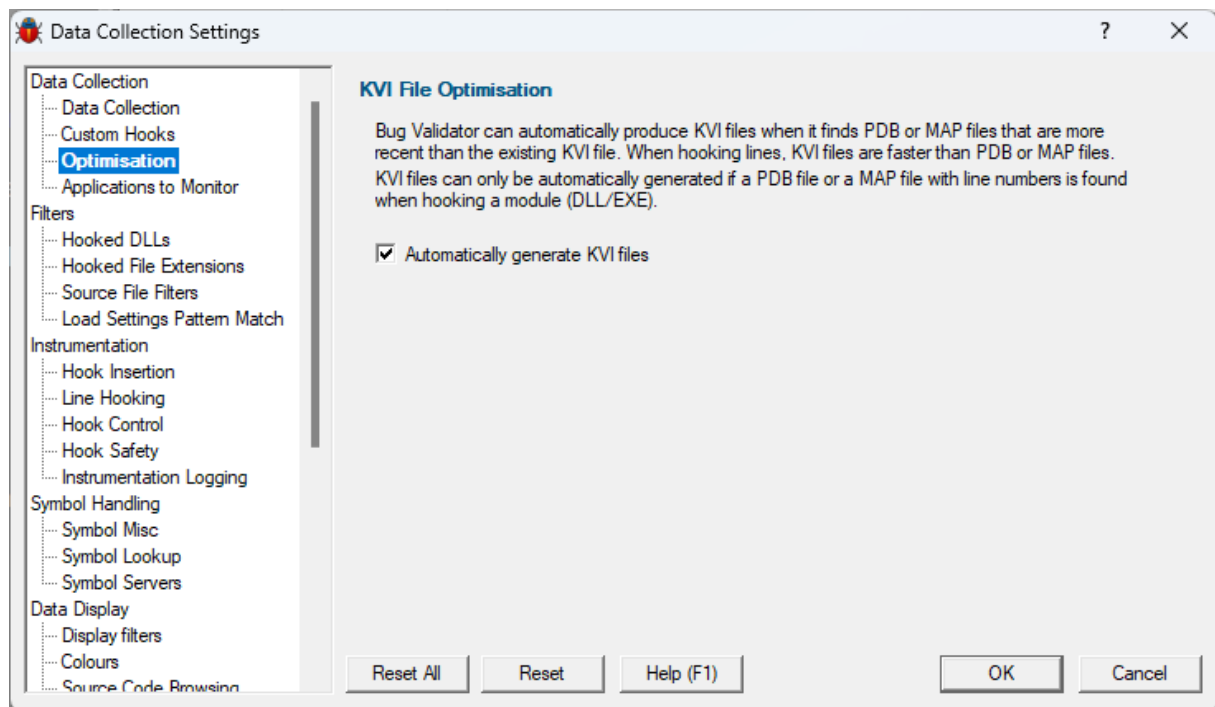
float *
double *

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.1.3 Optimisation

The **Optimisation and Editing** tab allows you to configure the editing options for Bug Validator.



KVI File Optimisation

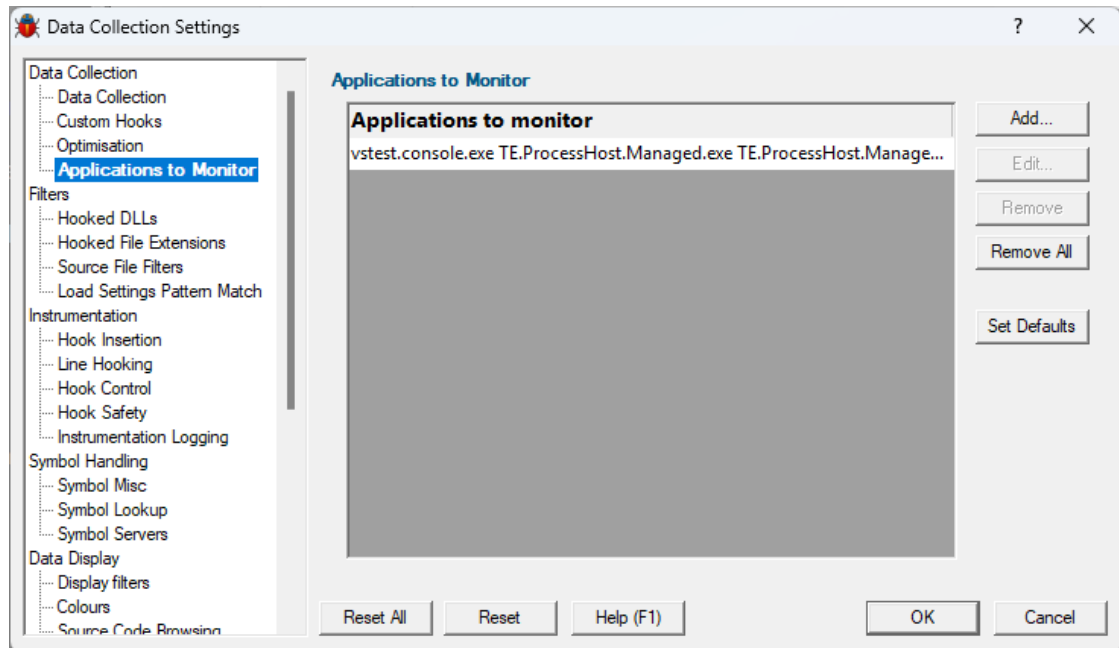
KVI Files are generated by Bug Validator and are used by Bug Validator at customer sites. In addition Bug Validator can also use KVI files in preference to MAP files or PDB files. Normally KVI files are created by choosing the Create KVI Files... option on the Deploy menu. If you want Bug Validator to automatically produce KVI files when it finds more recent PDB and/or MAP files, select the **Automatically generate KVI files** check box.

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.1.4 Applications to Monitor

If your target program launches other child applications then the **Applications to Monitor** page lets you choose which ones to monitor.



Monitoring child applications

You may have a case where the program you need to start is not the one you are interested in.

Your program may launch child applications and it may be one of *those* that you want to monitor with Bug Validator.

An example might be for unit testing where a test program spawns one or more child applications, or it might launch the same application multiple times.

The applications to monitor

The main list of **Applications to monitor** shows programs you may want to launch and the child applications they subsequently start - i.e. the you may be interested in monitoring.

Once a definition has been added, you can then use the Application to Monitor setting on the Launch Dialog or wizard to choose which of these child applications you actually want to monitor in a given session.

Managing the applications to monitor

The list contains a set of definitions - each one being for a different launch program.

For each launch program you can set the child applications you might want to monitor later.

An application is defined by its type (native and .Net, or .Net Core), the application executable name, and for .Net Core applications an additional application DLL that is used to identify the application.

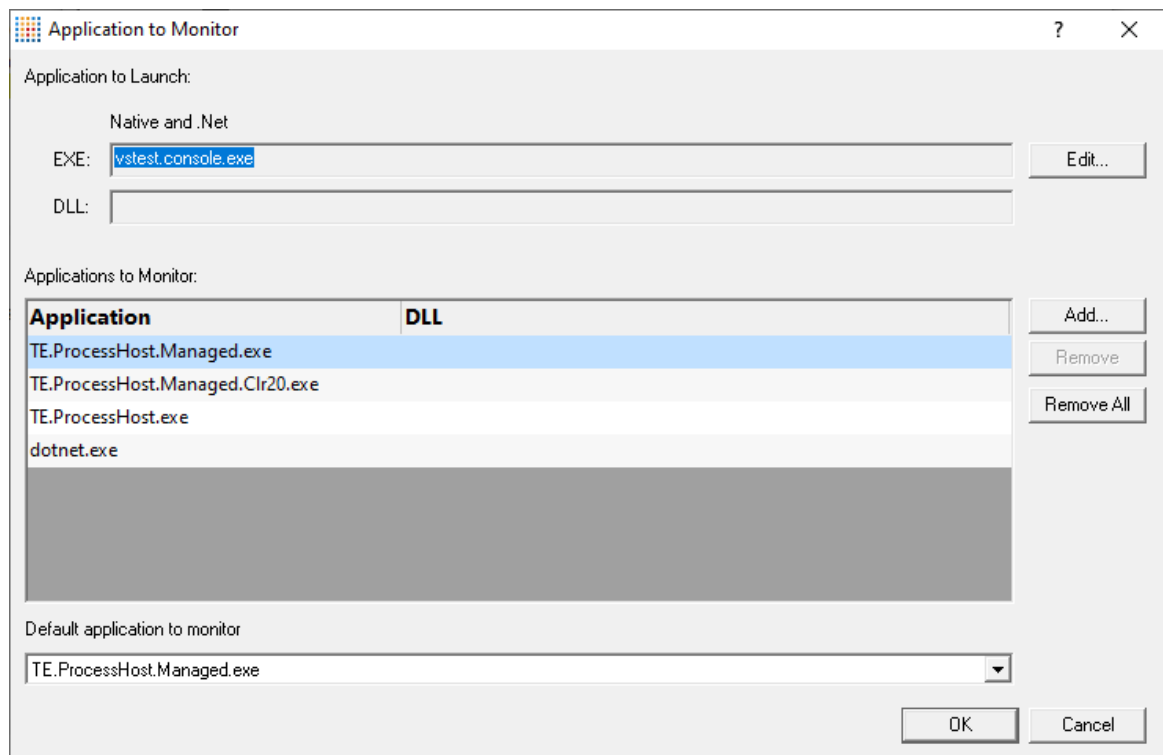
- **Add** > add a new module definition using the **Application to Monitor** dialog below
- **Edit** > modify a selected definition in the list, using the **Application to Monitor** dialog again
- **Remove** > removes any selected definitions in the list
- **Remove All** > clears the list
- **Set Defaults** > reset the list of known applications to those as configured with a new install of Bug Validator

The defaults are currently setup for Microsoft's Visual Test software `vstest.console.exe`.

The Application to Monitor dialog

The **Application to Monitor** dialog lets you define or edit a launch program and it's child applications.

The values you specify here are the ones used on the launch dialog and launch wizard to customize which application actually gets monitored.



- **Application to Launch** ➤ **Edit...** to select the initial starting application that will be *launching* the applications you want to monitor

Any executable names found in the selected program will automatically be displayed in the list of **Applications to Monitor**.

If you don't wish to use these automatic names you can **Remove** them.

- **Add** ➤ add an additional application that you know will be started by the launch program

Child applications that you add are used *without* the path.

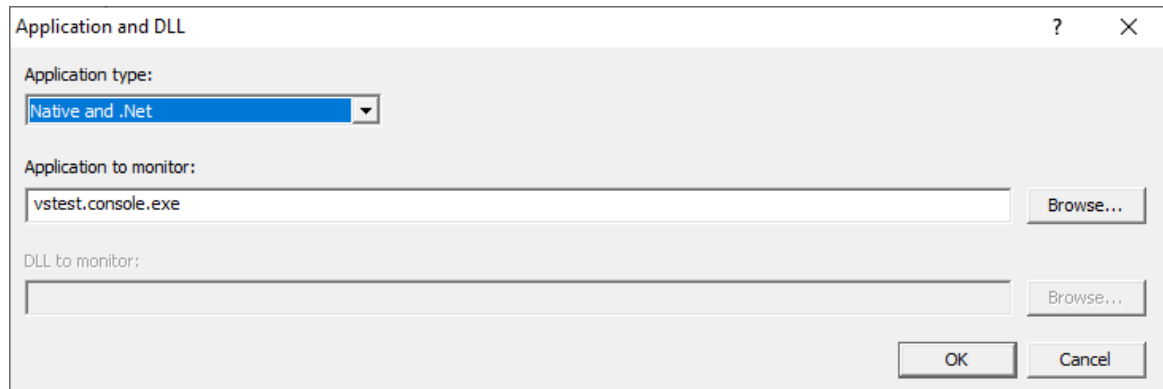
Excluding the path gives more scope for matching launched application names if they are launched with a different path.

- **Remove** ➤ removes any selected applications in the list
- **Remove All** ➤ clears the list
- **Default application to monitor** ➤ choose the appropriate item to be the default item

The default application will be selected on the launch dialog (or wizard) whenever the *start* program is specified as the one at the top of this dialog.

The Application and DLL dialog

The **Application and DLL** dialog lets you define or edit a launch program and a launch DLL.



- **Application type** ➤ choose the type of application
 - Native and .Net
 - .Net Core (Framework Dependent)
 - .Net Core (Self Contained)
- **Application to monitor** ➤ edit or **Browse...** the application EXE to monitor.

This can be an executable name or the full path to the executable. For example **test.exe** or **c:\unitTests\test.exe**.

For native applications this is the application executable.

For .Net Framework applications this is the application executable.

For .Net Core Framework-dependent applications this is most likely going to be **c:\program files\dotnet\dotnet.exe**.

For .Net Core Self-contained applications this is the application executable.

- **DLL to monitor** ➤ edit or **Browse...** the application DLL to monitor. This field is only needed for .Net Core applications.

This can be an executable name or the full path to the executable. For example **test.dll** or **c:\unitTests\test.dll**.

For native applications this is not used.

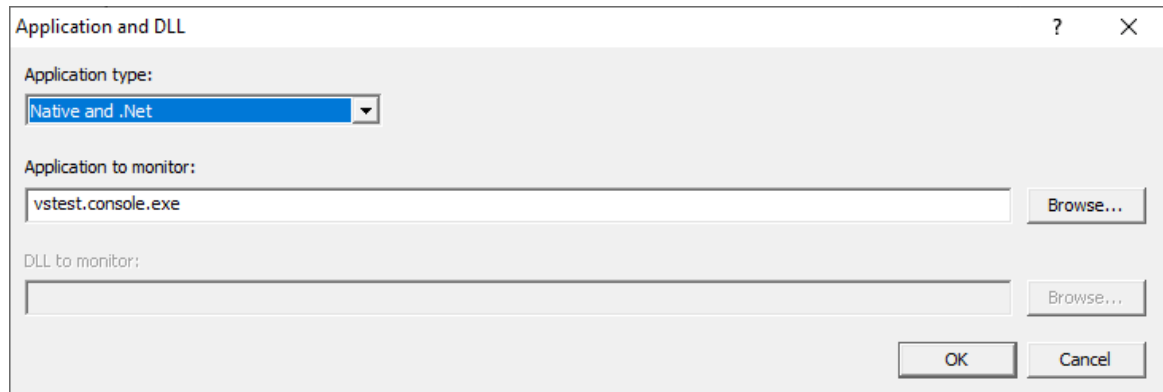
For .Net Framework applications this is not used.

For .Net Core Framework-dependent applications this is the application dll. (the name of the dll that you would pass to dotnet.exe on the command line).

For .Net Core Self-contained applications this is the dll that has the same name as the application executable. (for theApp.exe, the dll name is theApp.dll).

Example Dialogs

Native



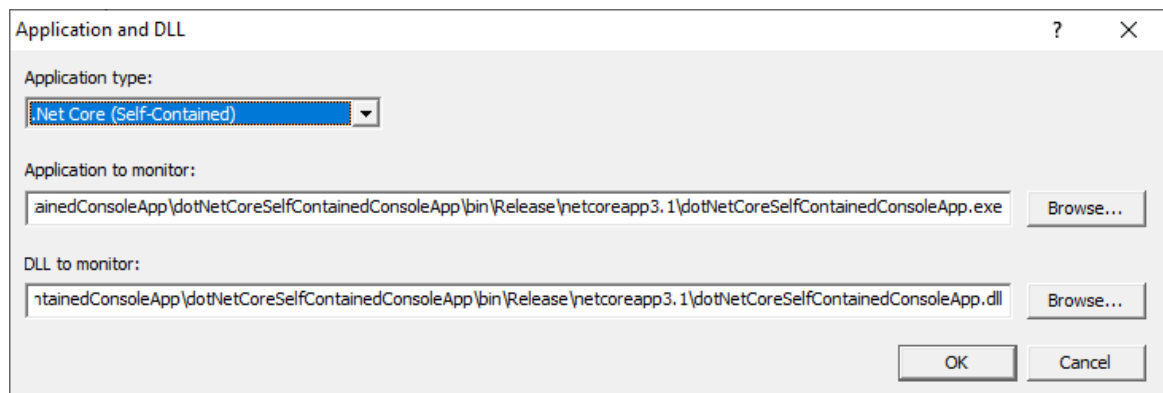
.Net



.Net Core (Framework-dependent)



.Net Core (Self-contained)



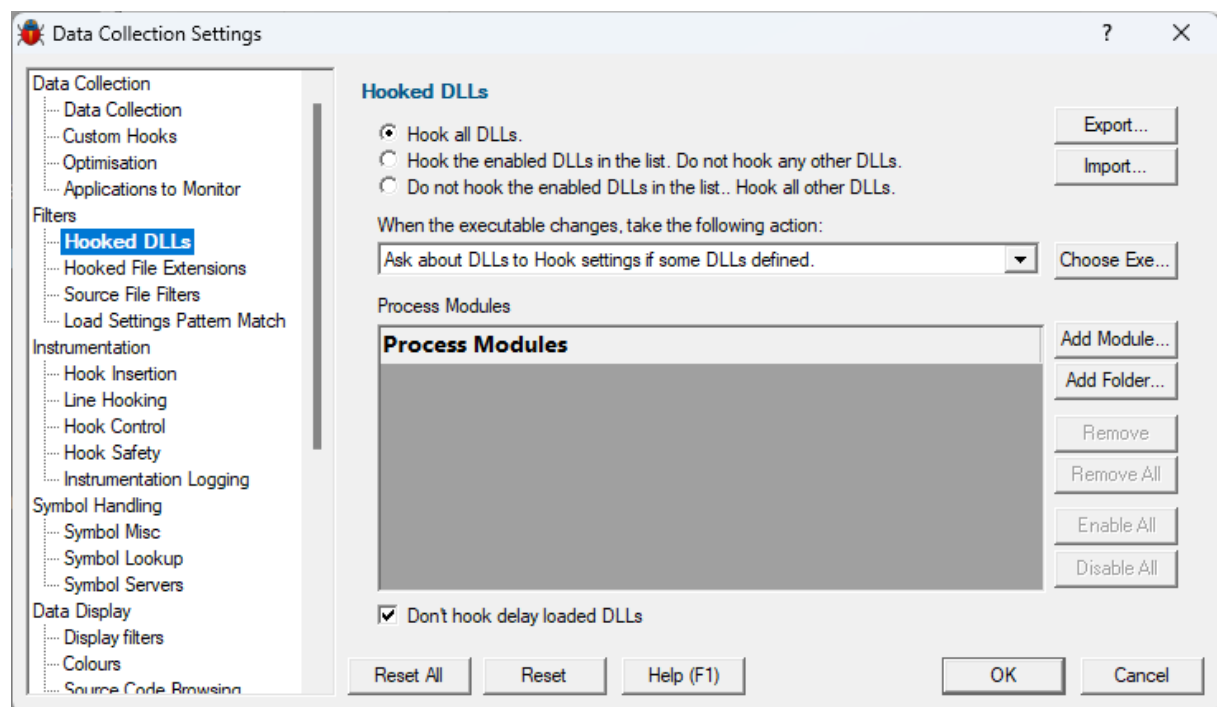
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.2 Filters

3.9.1.2.1 Hooked DLLs

The **Hooked DLLs** tab allows you to specify which DLLs should not be hooked.



Hooking Rules

DLLs are hooked according to a set of rules. There are three rules available.

- Hook all DLLs. All DLLs will be hooked regardless of the settings in the list.
- Hook the enabled DLLs in the list. Only the DLLs in the list that are enabled will be hooked. All other DLLs will not be hooked.
- Do not hook the enabled DLLs in the list. The DLLs in the list that are enabled will not be hooked. All other DLLs will be hooked.

Process Modules

The Process Modules list specifies all DLLs and .EXE in the target application. The default is that each DLL and .EXE is enabled. A DLL or .EXE that is disabled does not have any hooks put into that DLL. This list can also specify folders that should be hooked or that should not be hooked. All DLLs found inside a specified folder will be hooked or will not be hooked according to the same rules used for individual DLLs.

DLL names should be written as just the module name without the path. You can use the * wildcard to specify groups of DLLs. For example MFC*.dll will specify all DLLs starting with the MFC prefix. To edit a DLL name double click in the text field. Wildcards are not supported for folder names.

To enable a DLL for use with the hooking rules, click in the yellow box so that a tick mark is displayed.

To disable a DLL for use with the hooking rules, click in the yellow box so that a tick mark is not displayed.

Any DLLs in the list override the DLL Hook Insertion settings on the Hook Insertion tab.

- **Choose Exe...** ➤ to populate the list of DLLs with all the dependent modules of an application. A file selection dialog will be displayed to select the application.
- **Add Module...** ➤ add a dll to the list of DLLs. A file selection dialog will be displayed to select the DLL.
- **Add Folder...** ➤ add a folder to the list of DLLs that will not be hooked. A folder selection dialog will be displayed to select the folder.
- **Remove** ➤ remove the selected DLLs and folders from the list.
- **Remove All** ➤ remove all DLLs and folders from the list.
- **Enable All** ➤ enable all DLLs in the list.
- **Disable All** ➤ disable all DLLs in the list.
- **Export...** ➤ to export the hooking rule and the list of DLLs to hook. A file selection dialog is displayed to allow you to choose the file to export to.
- **Import...** ➤ to import a previously exported rule and list of DLLs to hook. A file selection dialog is displayed to allow you to choose the file to import from.

Delay loaded DLLs

If you do not want to hook DLLs that use delay loading techniques select the **Don't hook delay loaded DLLs** checkbox.

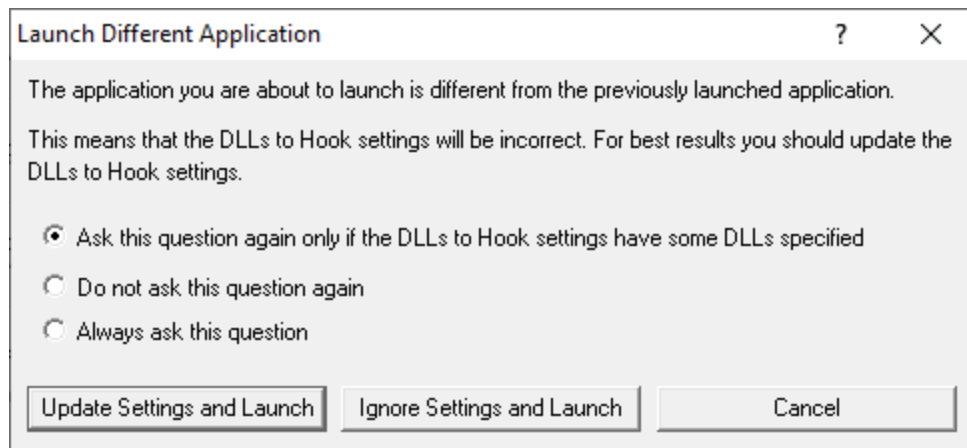
Launching new Applications

The process of performing Unit Tests often results in testing multiple applications. This will result in the list of DLLs that you are working with changing. It is therefore possible that you may forget to update the list of DLLs to hook. This could cause incorrect coverage results if you have the hooking rules set to hook a specific set of DLLs. To help prevent this, Bug Validator provides the option of warning you about these settings when you change the executable you are testing.

The options are:

- Ask about DLLs to Hook settings if some DLLs defined. You will only be asked about the Hooked DLLs settings if you have defined some DLLs (enabled or disabled) in the list.
- Always ask about DLLs to Hook settings. You will always be asked about the Hooked DLLs settings.
- Never ask about DLLs to Hook settings. You will never be asked about the Hooked DLLs settings.

When Bug Validator asks you about the Hooked DLLs settings, the following dialog is displayed.



If you wish to change when you are asked this question choose the appropriate radio box.

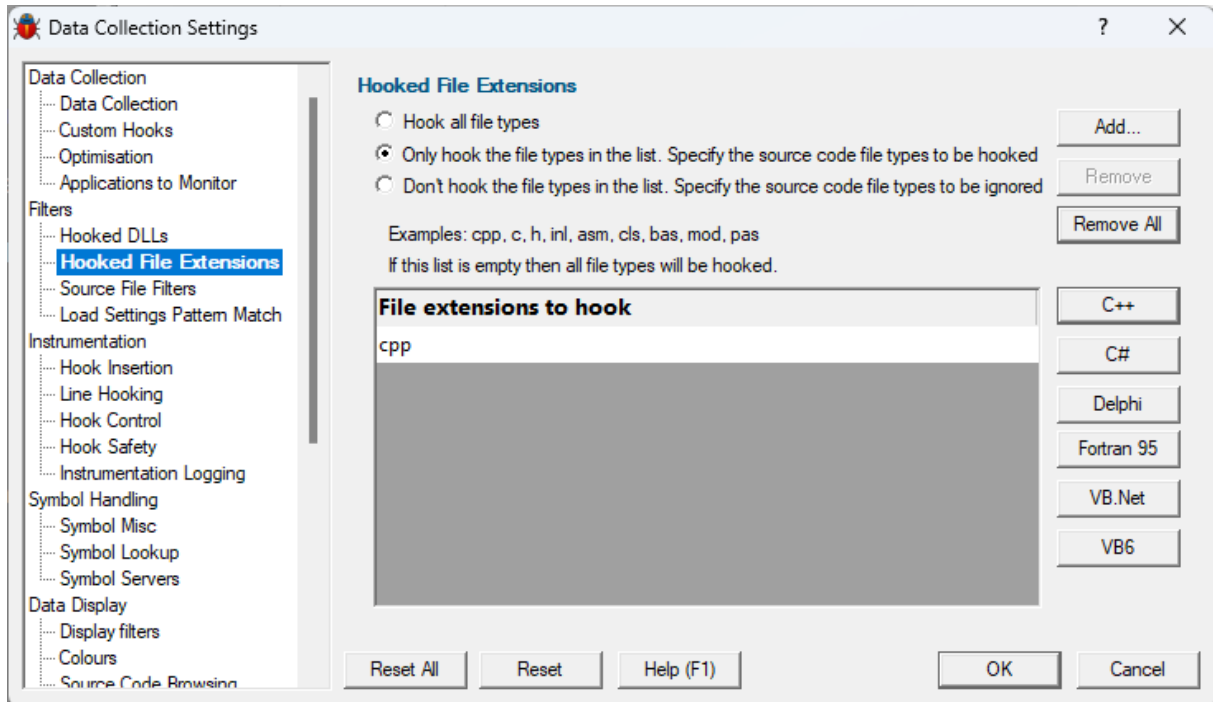
- **Update Settings and Launch** ➤ to edit the settings and then launch the application.
- **Ignore Settings and Launch** ➤ to launch the application without updating the settings.
- **Cancel** ➤ to cancel launching the application.

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.2.2 Hooked File Extensions

The **Hooked File Extensions** tab allows you specify which source code file types should be hooked and source code files should be excluded from hooking.



Which file types to hook - the hooking rule

By default, Bug Validator will try to hook all file types used by your application, but you can choose to list only those which should be included or excluded:

- **Hook all file types** ➤ hook everything - ignoring the settings in the list
- **Only hook the file types in the list** ➤ hook only the listed file types
- **Don't hook the file types in the list** ➤ ignore all the listed file types, and hook everything else

File extensions to hook

The file extensions to hook list allows you to specify a list of valid file extensions for source code files you want hooked. If the list is empty, then all source code files will be hooked.

For example, you may want to include C++ source and header files but exclude C source files. To do this you would add the following extensions to the list:

```
cpp
h
```

or if you use the less common forms of cxx, hxx and hpp:

cpp
cxx
hpp
hxx
h

The list of extensions omits 'c' which is the standard extension for C source files.

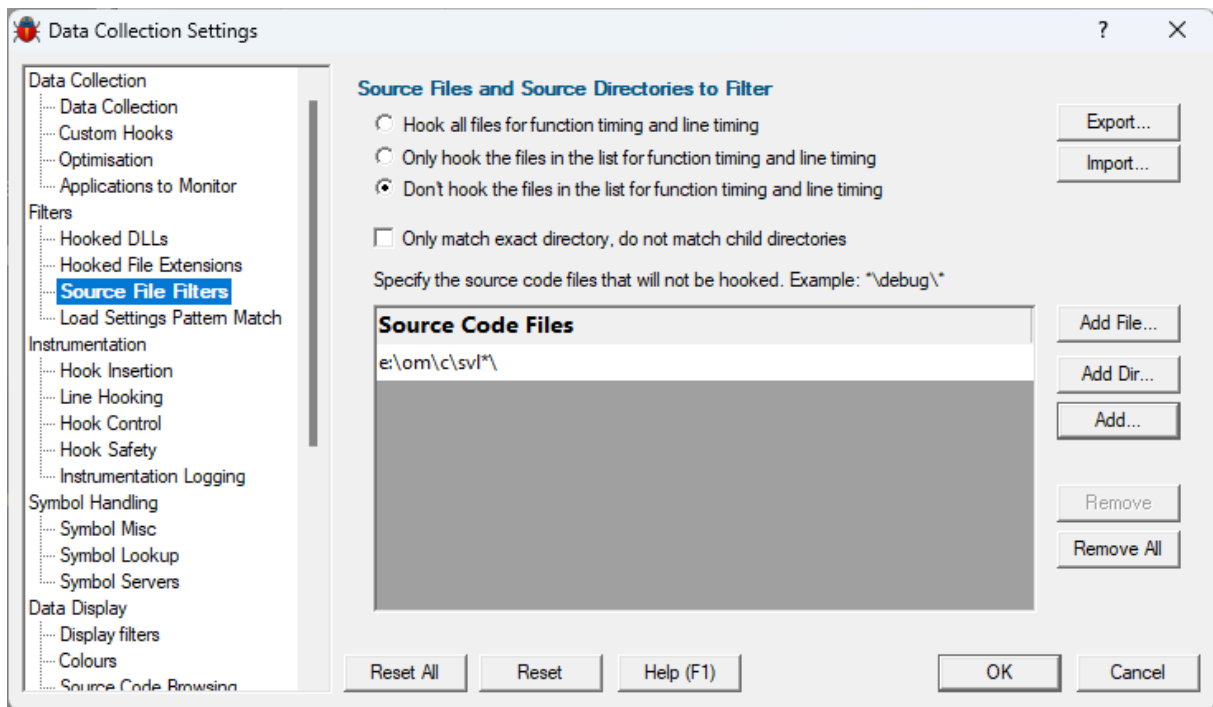
- **Add...** ➤ adds a new row to the list ➤ enter the extension you want to allow
- **Remove** ➤ removes selected items in the list
- **Remove All** ➤ removes all items, clearing the list
- **C++** ➤ adds the file extensions for C++
- **C#** ➤ adds the file extensions for C#
- **Delphi** ➤ adds the file extensions for Delphi
- **Fortran 95** ➤ adds the file extensions for Fortran 95
- **VB.Net** ➤ adds the file extensions for the VB.Net
- **VB6** ➤ adds the file extensions for the Visual Basic 6

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.2.3 Source File Filters

The **Source File Filters** tab allows you to specify files which should be hooked or which files should not be hooked.



Listing files to exclude or include

By default, Bug Validator will try to hook all source files used by your application, but you can choose to list only those which should be included or excluded

- **Hook all files...** ➤ hook all files
- **Don't hook the files in the list...** ➤ hook everything *except* the files in the list
- **Only hook the files in the list...** ➤ hook *only* the source files listed

In the list you can include files or directories. If using directories you may want only that specific directory, or everything underneath it (the default):

- **Only match exact directory...** ➤ check this so as *not* to recurse into child directories

Managing the list of files

Add files or directories on disk:

- **Add File...** ➤ navigate to the source files ➤ select one or more files ➤ click **Open** ➤ all the selected items are added
- **Add Dir...** ➤ navigate to the folder ➤ select it and click **OK** ➤ the folder is added to the list

Or manually enter a file:

- **Add...** > a new row is added to the list > Type the file path > press **Return** > the file is added to the list

Remove items as normal:

- **Remove** > removes selected items in the list
- **Remove All** > removes all items, clearing the list

Alternatively, press **Del** to delete selected items, and **Ctrl** + **A** to select all items in the list first.

Exporting and importing

Since the list of source files can be quite complicated to set up, you can export the settings to a file and import them again later. This is useful when switching between different target applications.

- **Export...** > **choose or enter** a filename > **Save** > outputs the filtered source files to the file
- **Import...** > **navigate** to an existing *.bxft file > **Open** > loads the filtered source files



The exported file can be used with the -sourceFileFilterHookFile command line option.

Wildcards

All file and directory specifications can contain the * wildcard.

Some examples will help:

Consider a project with three source directories:

```
e:\dev\srcMain\  
e:\dev\srcCommon\  
e:\dev\srcCustom\
```

Possible filters could be:

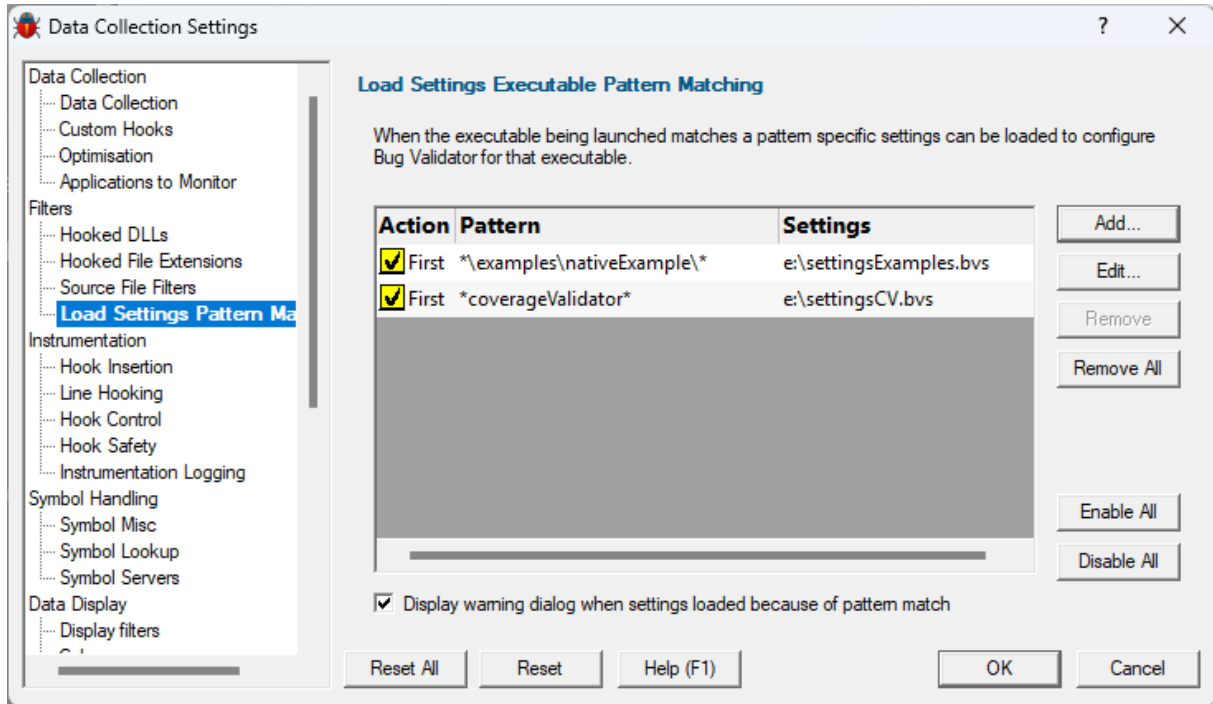
```
e:\dev\src*\*  
*\src*\*  
*\srcC*\*
```

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.2.4 Load Settings Pattern Match

The **Load Settings Pattern Match** tab allows you to configure loading of different settings depending on the executable being launched (or relaunched).



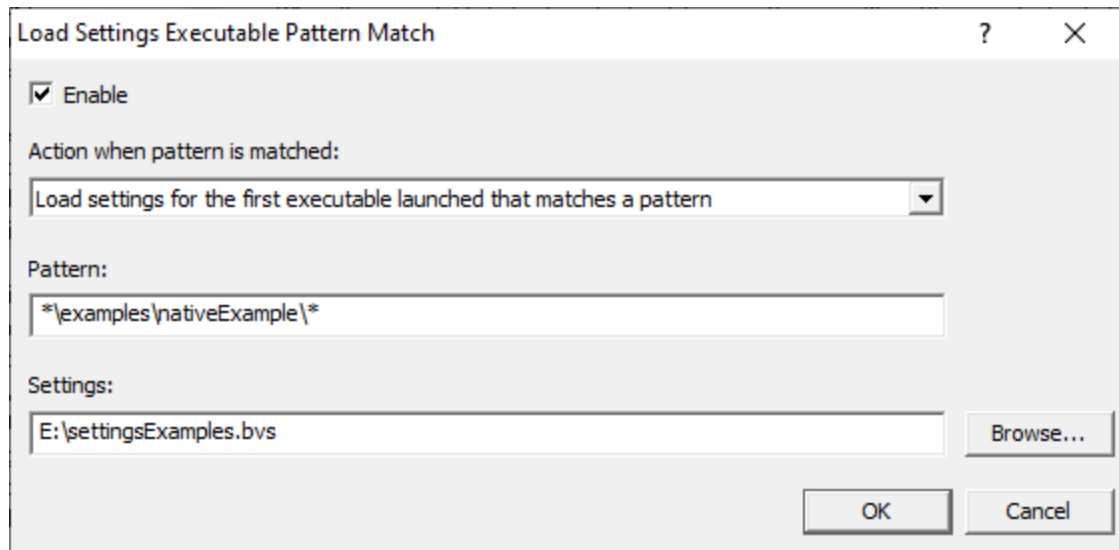
The grid shows one pattern match per line.

The buttons alongside allow you to Add, Edit and Remove patterns that you have created. You can also enable and disable them all.

- **Add...** ➤ display the pattern match dialog to create a pattern to match.
- **Edit...** ➤ display the pattern match dialog to edit the selected pattern.
- **Remove...** ➤ delete the selected pattern.
- **Remove All...** ➤ delete all selected patterns.
- **Enable All...** ➤ enable all patterns.
- **Disable All...** ➤ disable all patterns.

Pattern Match Dialog

The pattern match dialog allows you to create and edit pattern matches.



- **Enable** ➤ enable or disable this pattern.
- **Action** ➤ how to evaluate if this pattern is matched.
 - **Load settings for first executable...** ➤ the first executable that matches a pattern causes the settings to be loaded.
 - **Load settings for each executable...** ➤ each executable that matches a pattern causes the settings to be loaded provided it is not the same executable that previously loaded the settings.
 - **Load settings for every executable...** ➤ every executable that matches a pattern causes the settings to be loaded.

When a different pattern matches the first/each status is reset.

- **Pattern** ➤ a text pattern, including the * wildcard, to match an executable path.

Examples:

```
*\examples\nativeExample\*
c:\tests\*
e:\dev\myProject\release\*.exe
```

- **Settings** ➤ the full path to the settings you want to load if the action and pattern match an executable.

How does the pattern matching work?

It is probably easiest to demonstrate how pattern matching works with some examples.

Let's assume we have two patterns:

```
*\examples\nativeExample\* that will load e:\settingsExamples.bvs
*coverageValidator* that will load e:\settingsCV.bvs.
```

We'll cover each of the possible action criteria for a sequence of application launches, showing which settings are loaded and why.

- **Load settings for first executable...** ➤ the first executable that matches a pattern causes the settings to be loaded.

Launched application	Settings loaded	Reason
e:\examples\nativeExample\release\nativeExample.exe	e: \settingsExamples.b vs	1st application, new pattern
e:\examples\nativeExample\release\nativeExample.exe		repeat application, same pattern
e:\examples\nativeExample\debug\nativeExample.exe		2nd application, same pattern
c:\program files (x86)\software verify\coverage validator x86\coverageValidator.exe	e:\settingsCV.bvs	1st application, new pattern
e:\examples\nativeExample\release\nativeExample.exe	e: \settingsExamples.b vs	1st application, new pattern

- **Load settings for each executable...** ➤ each executable that matches a pattern causes the settings to be loaded provided it is not the same executable that previously loaded the settings.

Launched application	Settings loaded	Reason
e:\examples\nativeExample\release\nativeExample.exe	e: \settingsExamples.b vs	new application, new pattern
e:\examples\nativeExample\release\nativeExample.exe		repeat application, same pattern
e:\examples\nativeExample\debug\nativeExample.exe	e: \settingsExamples.b vs	new application, same pattern
c:\program files (x86)\software verify\coverage validator x86\coverageValidator.exe	e:\settingsCV.bvs	new application, new pattern
e:\examples\nativeExample\release\nativeExample.exe	e: \settingsExamples.b vs	new application, new pattern

- **Load settings for every executable...** ➤ every executable that matches a pattern causes the settings to be loaded.

Launched application	Settings loaded	Reason
e:\examples\nativeExample\release\nativeExample.exe	e: \settingsExamples.b vs	Every application
e:\examples\nativeExample\release\nativeExample.exe	e: \settingsExamples.b vs	Every application
e:\examples\nativeExample\debug\nativeExample.exe	e: \settingsExamples.b vs	Every application
c:\program files (x86)\software verify\coverage validator x86\coverageValidator.exe	e:\settingsCV.bvs	Every application
e:\examples\nativeExample\release\nativeExample.exe	e: \settingsExamples.b	Every application

vs

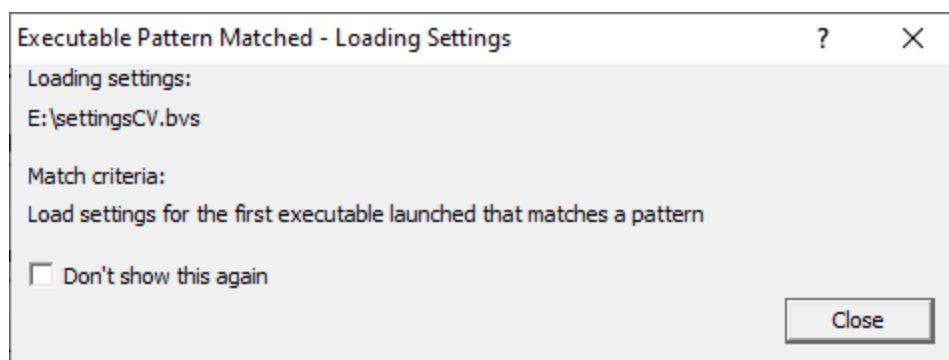
Warning

When a pattern is matched and the action criteria are satisfied the specified settings will be loaded.

A warning can be displayed at this point to remind you that the settings are being changed.

- **Display warning dialog...** ➤ the warning dialog will be displayed when the pattern match criteria are met.

The warning dialog looks like this:



Reset All - Resets **all** global settings, not just those on the current page.

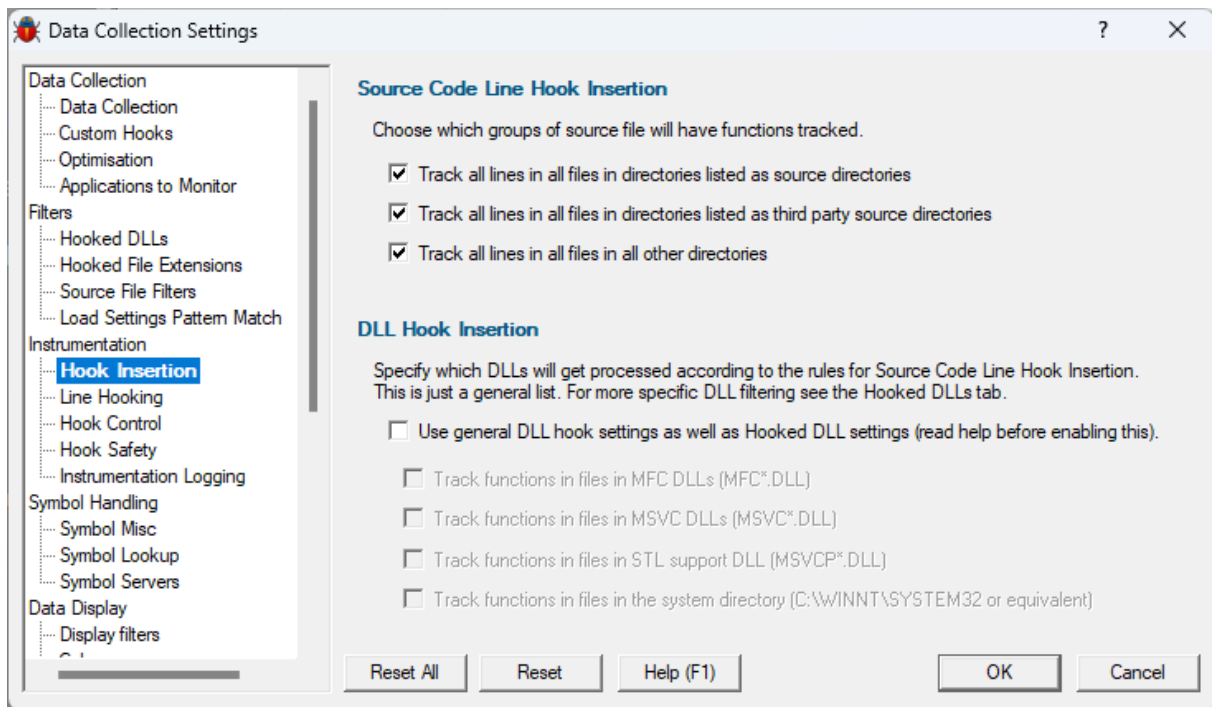
Reset - Resets the settings on the current page.

3.9.1.3 Instrumentation

3.9.1.3.1 Hook Insertion

The **Hook Insertion** tab allows you to specify which groups of files are hooked in the target program.

Once installed the group of installed hooks cannot be changed. If you wish to use a different set of hook groups you will need to start a new session with the target program.



Source Code Line Hook Insertion

This section allows you to specify which group of source code files will be hooked.

- **Track all lines in all files in directories listed as source directories.**
If this check box is selected any file referenced in a KVI file that is in one of the directories specified as a source directory will be hooked.
- **Track all lines in all files in directories listed as third party source directories.**
If this check box is selected any file referenced in a KVI file that is in one of the directories specified as a third party source directory will be hooked.
- **Track all lines in all files in all other directories.**
If this check box is selected any file that is not in one of the directories specified for source directories and/or third party source directories will be hooked.

Dll Hook Insertion

This section allows you to specify which DLLs will be processed according to the rules for Source Code Line Hook Insertion. The check boxes when selected enable the particular group of DLLs to be hooked. These DLLs are all system DLLs, and as such may not have much use if hooked for code coverage, as the source code may or may not be available, and even if available, your application will not be able to influence the code coverage for these files. To enable the use of these options select the **Use general DLL hook settings as well as Hooked DLL settings** checkbox.

- **Track functions in files in MFC DLLs.**
When selected all MFC dlls will be hooked.
- **Track functions in files in MSVC DLLs.**
When selected all MSVC dlls will be hooked.
- **Track functions in files in the STL support DLL.**
When selected the STL support dll (MSVCP(D).DLL) will be hooked.

- **Track functions in files in system directory.**

When selected all DLLs in the system directory will be hooked.

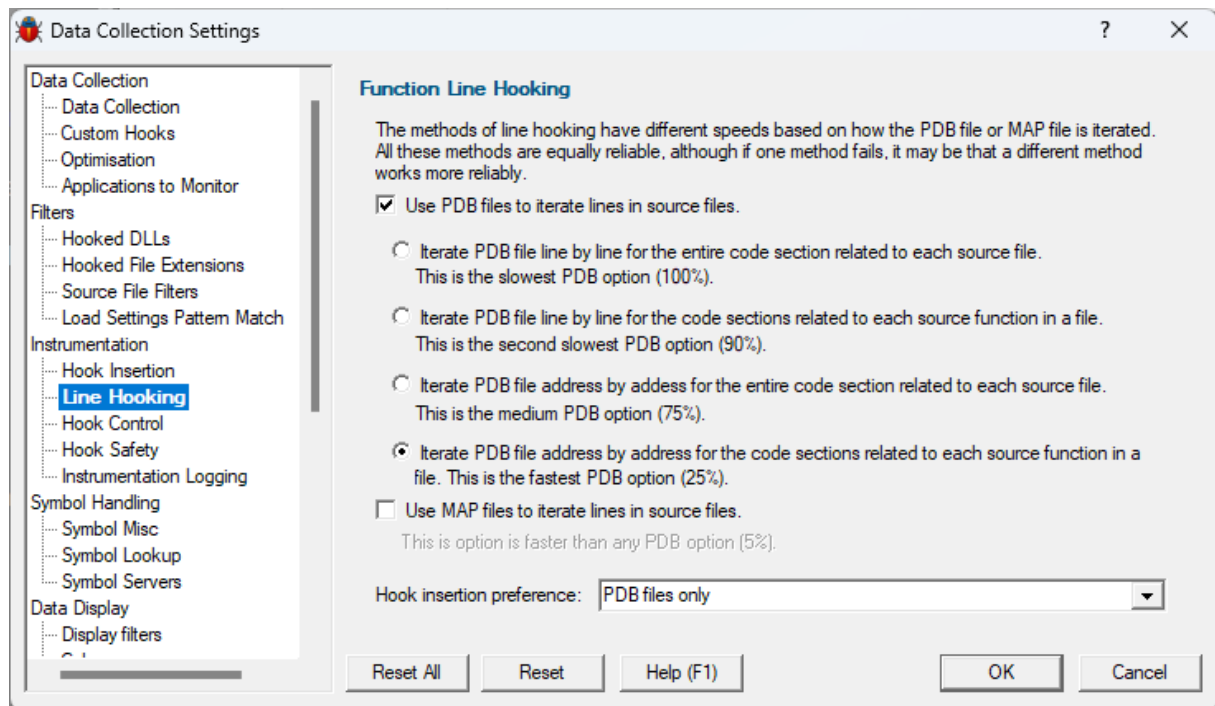
These settings are overridden if you specify DLLs in the list on the Hooked DLLs tab. As a general rule, if you are using the Hooked DLLs tab, it is best to disable the DLL Hook Insertion options by deselecting the **Use general DLL hook settings as well as Hooked DLL settings** checkbox.

Reset All - Resets all global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.3.2 Line Hooking

The **Line Hooking** tab allows you to specify which how the source code lines are discovered by Bug Validator.



To hook each source code line Bug Validator has to find the start address of each line and then check that hooking that line will not result in corruption of the code relating to other parts of the program. The data to determine the location of each source code line can be found in KVI files that contain line number information. For Microsoft DLLs, data in PDB files is used.

For KVI files, the line number information is explicit, so we provide one method of walking the source code lines for MAP files.

For PDB files we found that there were four related methods for obtaining source code line start addresses. Each method was subtly different, maybe more accurate, but slower, or faster and less reliable. As such we have included all of these methods, so that you can choose the method most appropriate for what you want to achieve.

To use PDB files select the Use **PDB files to iterate lines in source files** check box. To select the method of iterating the PDB file, choose the appropriate radio box.

To use KVI files select the **Use MAP files to iterate lines in source files** check box.

The hook insertion preference combo box allows you to choose how Bug Validator chooses between PDB files and KVI files. The options are:

- **PDB files only.**
Only PDB files are used for iterating files. KVI files are ignored.
- **KVI files only.**
Only KVI files are used for iterating files. PDB files are ignored.
- **Use KVI when no PDB.**
PDB files are used in preference to KVI files. KVI files are only used when the required PDB file cannot be found.
- **Use PDB when no KVI or no lines in KVI.**
KVI files are used in preference to PDB files. PDB files are only used when the required KVI file cannot be found, or the KVI file does not contain line number information.

Our tests have shown that KVI files with line numbers provide the fastest hooking method. KVI files with line numbers are not always available, so the **Use PDB when no KVI or no lines in KVI** option is the recommended option.

WARNING: Tests have shown that MAP files do not always contain the address of every line in the application. Some lines are omitted from the MAP, for no apparent reason. As a result, when using MAP files, sometimes lines are not hooked. If you use the **Use MAP when no PDB** option, PDB files will always be used in preference to MAP files, thus reducing the possibility of this happening.

IMPORTANT.

Due to daylight saving times it is possible for a MAP file to have an embedded timestamp that is different than the DLL timestamp by an hour.

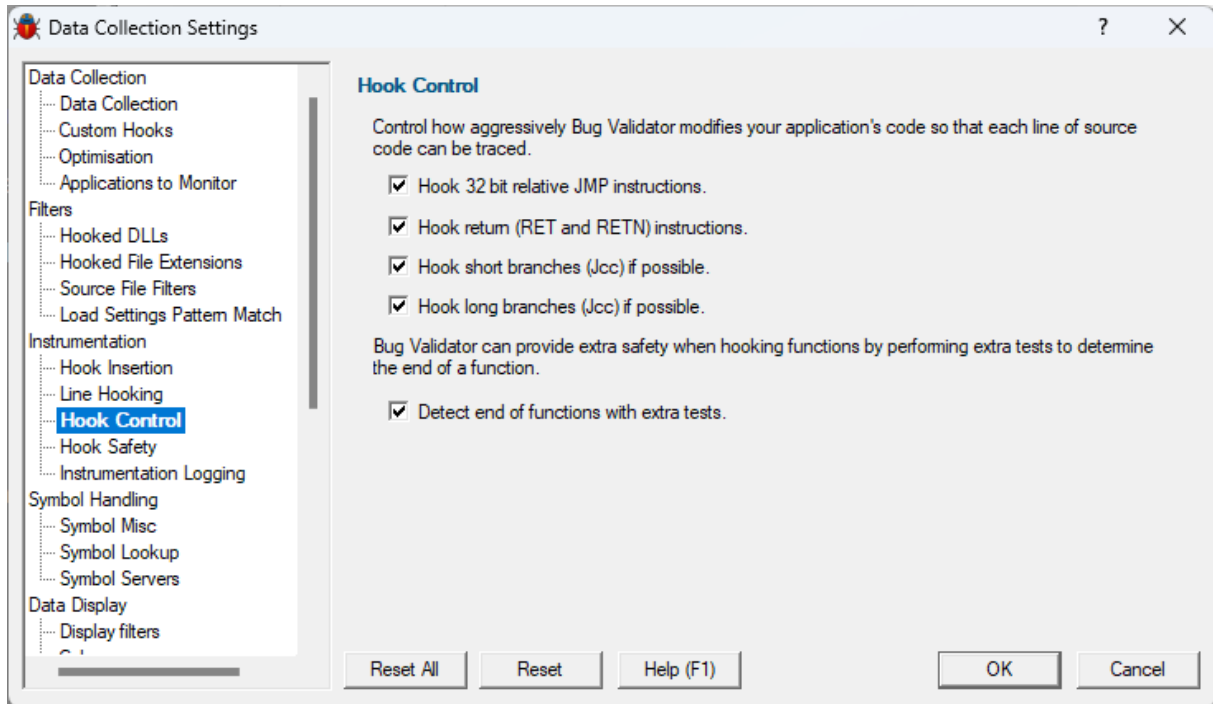
In these situations Bug Validator will not recognise the MAP as valid. The solution to this problem is to rebuild the application.

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.3.3 Hook Control

The **Hook Control** tab allows you to specify which how the source code lines are hooked and communicated to Bug Validator.



Hook Control

The next page allows you to specify how aggressive Bug Validator is when hooking lines that cannot be hooked as easily as other lines. Selecting these check boxes will make Bug Validator perform more safety checks when hooking lines and also enable lines to be hooked that would otherwise not be hooked.

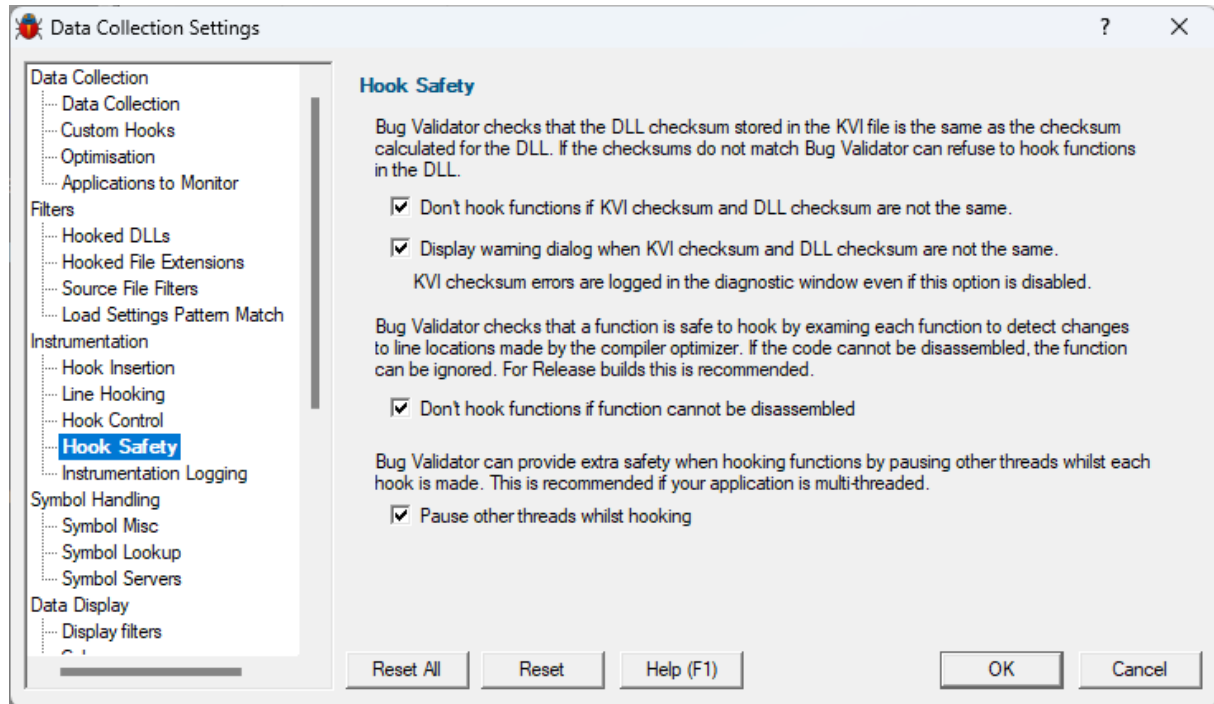
- **Hook 32 bit relative JMP instructions.**
When this check box is selected, Bug Validator will hook lines with JMP instructions in them.
- **Hook return (RET and RETN) instructions.**
When this check box is selected, Bug Validator will hook lines with RET or RETN instructions in them.
- **Hook short branches (Jcc) if possible.**
When this check box is selected, Bug Validator will hook lines with short branches in them.
- **Hook long branches (Jcc) if possible.**
When this check box is selected, Bug Validator will hook lines with long branches in them.
- **Detect end of functions with extra tests.**
When this check box is selected, Bug Validator will be more cautious when detecting the end of a function (to avoid overwriting the code for any function that after the function being hooked).

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.3.4 Hook Safety

The **Hook Safety** tab allows you to control how Bug Validator performs self-consistency checks.



KVI Checksum

Bug Validator checks that the DLL checksum and the checksum for the DLL in the KVI file are the same. This check is used to ensure that the KVI file is being used on the same version of the DLL as the DLL the KVI file was created for. If you want Bug Validator to perform this check, select the **Don't hook functions if KVI checksum and DLL checksum are not the same** check box..

If you want a warning dialog displayed when the checksums don't match, check the **Display warning dialog** check box.

Hook Safety

Bug Validator checks that lines and functions are safe to hook by disassembling each function to check that the optimizing compiler has not moved the locations of the start of each line. If the code cannot be disassembled due to finding unrecognised code sequences, the function is not hooked. If you want Bug Validator to ignore functions that cannot be disassembled, select the **Don't hook functions if function cannot be disassembled** check box.

Multithreaded applications

If your application is multithreaded, there is the chance that Bug Validator may be modifying the code for a function whilst another thread is executing the function. To prevent errors in this case you can ensure

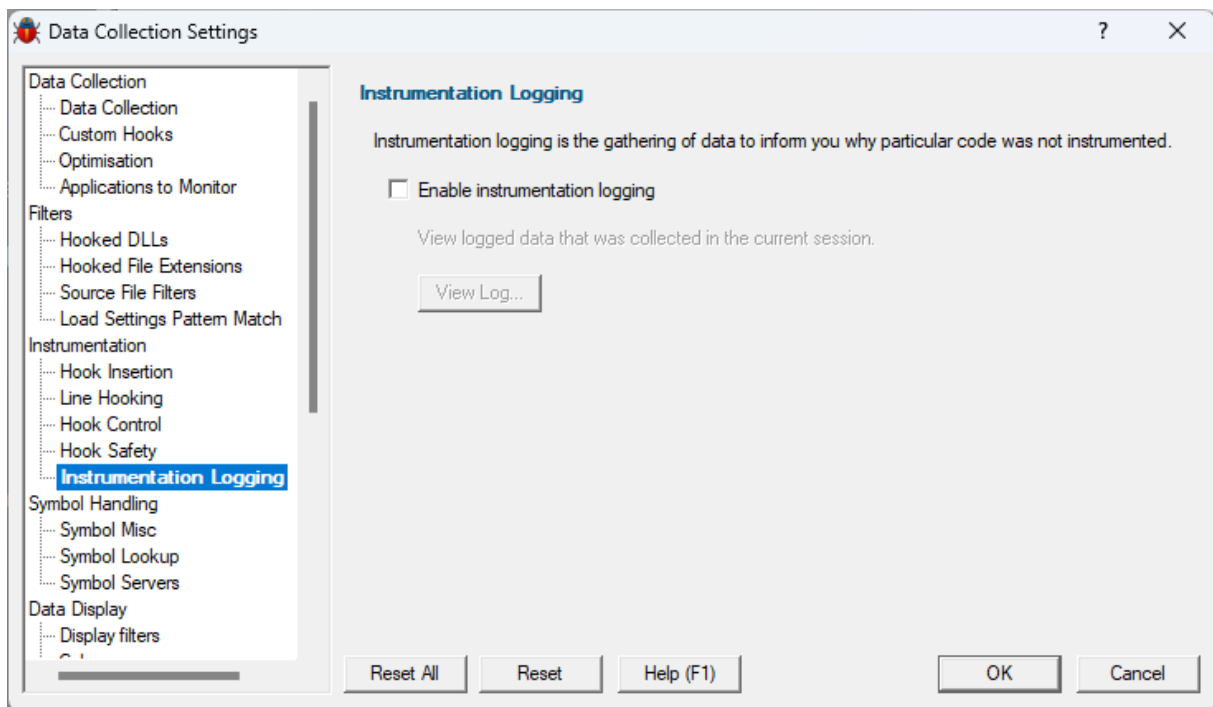
that Bug Validator pauses other threads whilst each line is hooked. To do this, select the **Pause other threads whilst hooking** check box.

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.3.5 Instrumentation Logging

The **Instrumentation Logging** tab allows you to control Bug Validator's internal logging.



If you enable instrumentation logging a log file will be created during instrumentation that indicates why each DLL, file and function was or was not instrumented according to the various settings and filters.

The instrumentation log can be useful to identify the reasons why a particular file or function or class is or is not be instrumented.

- **View Log...** ➤ to view the instrumentation log. You can also view the log from the Tools menu.

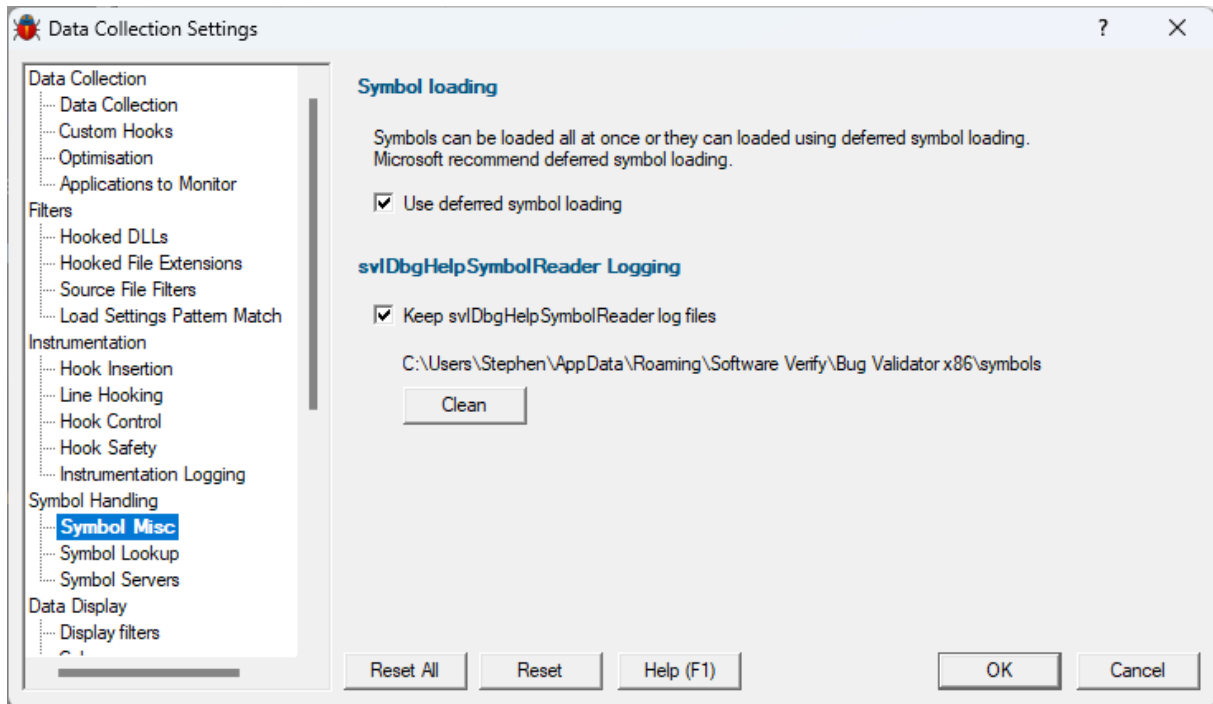
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.4 Symbol Handling

3.9.1.4.1 Symbol Misc

The **Symbol Misc** tab allows you to set miscellaneous symbol settings.



Immediate or deferred symbol loading

When converting program addresses to symbol names, you can choose immediate symbol loading, or defer loading until each symbol is needed.

- **Use deferred symbol loading** ➤ uses deferred symbol loading rather than 'all at once' (on by default)

Microsoft® recommend deferred symbol loading, claiming it is the fastest option. We give you the choice.

Symbol Reader Logging

Symbols are fetched from symbol servers using a helper process `svlDbgHelpSymbolReader.exe`. We log the command line and behaviour of this helper tool. This is displayed on the diagnostic tab.

If you wish the log files can be kept for later analysis. By default this option is turned off.

- **Keep svlDbgHelpSymbolReader log files** ➤ keep the log files after Bug Validator has finished processing them

The path to the directory containing the log files is shown.

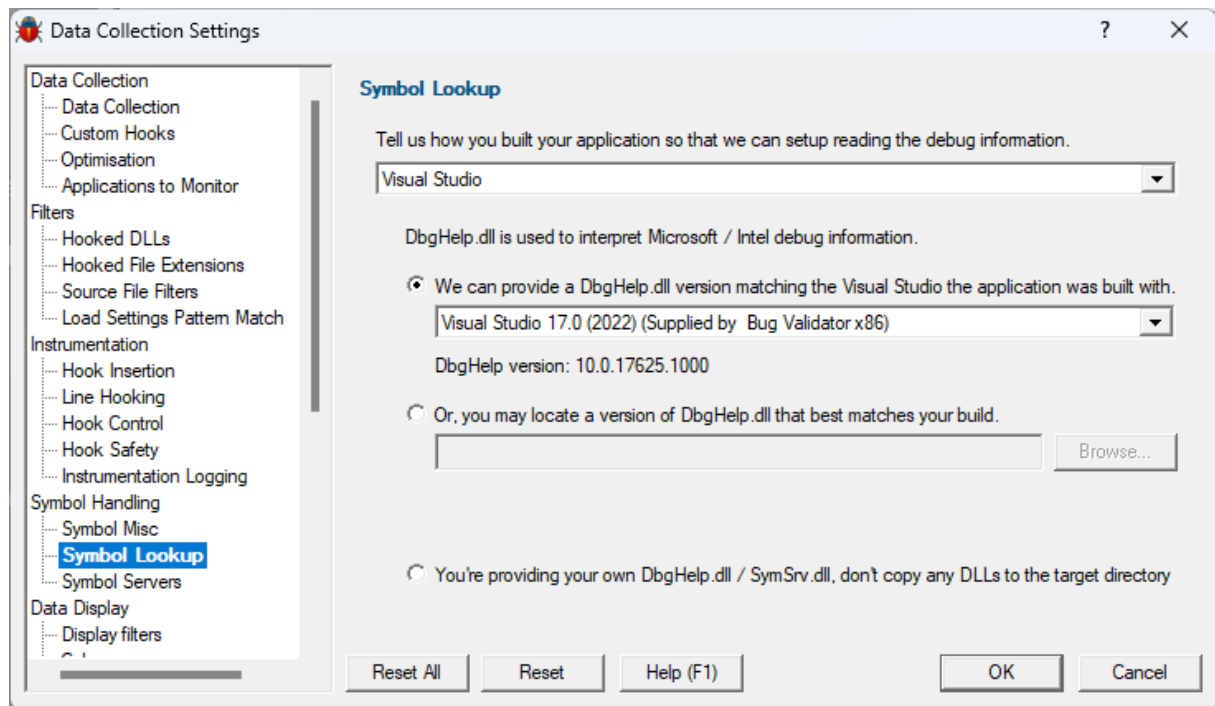
- **Clean** ➤ delete all svlDbgHelpSymbolReader log files

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.4.2 Symbol Lookup

The **Symbol Lookup** tab allows you to specify how Bug Validator acquires symbolic information about your application or service.



Compiler / IDE Choice

Use the first combo box to choose which compiler / IDE you used to build your software.

Bug Validator will use the appropriate methods to read your symbols.

The choices are:

- Visual Studio
- Visual Basic 6
- Rust
- Other

Symbol Lookup

- **We can provide a Dbghelp.dll** ➤ choose one of Bug Validator's known good DbgHelp.dll's based on the version of Visual Studio you are using

Bug Validator fetches symbols for your application using an appropriate symbol handler for the type of debugging information you have.

For Microsoft Visual Studio users each version of Visual Studio provides different debugging formats which are readable by the appropriate DbgHelp.dll supplied by Visual Studio. A given version of DbgHelp.dll is usually able to read earlier formats of Microsoft debugging information but is not able to read a future format. For example Visual Studio 2005 (version 8) can read Visual Studio 6 debug information but cannot read Visual Studio 2008 debug information.

Visual Studio 6.0 doesn't supply a DbgHelp.dll so we have provided one for use with Visual Studio 6.0.

Visual Studio 10 is unusual in that the DbgHelp.dll (6.12) supplied by Visual Studio cannot read the debug information created by Visual Studio. To solve this problem we have supplied DbgHelp.dll (6.11) as an alternative.

Bug Validator will choose the appropriate (most recent) version of Visual Studio automatically. You can override Bug Validator's choice by choosing the Visual Studio version from the **Visual Studio** combo box.

Specify your own DbgHelp.dll

- **Or, you may locate a version of DbgHelp.dll** ➤ specify your own DbgHelp.dll to use with Bug Validator

If you wish to explicitly specify which DbgHelp.dll to use choose the **Or, you may locate a version of DbgHelp.dll** option enter the path in the **DbgHelp.dll** edit field or use the **Browse...** button to select the dbgHelp.dll.

Note that the directory that contains DbgHelp.dll **should also contain symsrv.dll** if you wish to use symbol servers with Bug Validator.

Don't update DbgHelp.dll

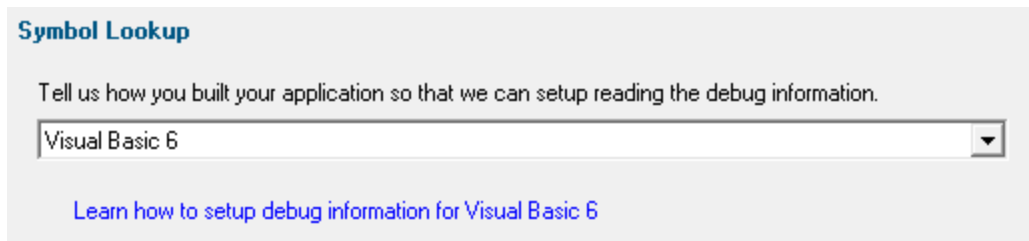
- **You're providing your own DbgHelp.dll** ➤ use the DbgHelp.dll that ships with your application

If your application needs to use a specific version of DbgHelp.dll that you're already providing with your application you should choose the **You're providing your own DbgHelp.dll** option to prevent Bug Validator from overwriting your DbgHelp.dll.

Note that the directory that contains DbgHelp.dll **should also contain symsrv.dll** if you wish to use symbol servers with Bug Validator.

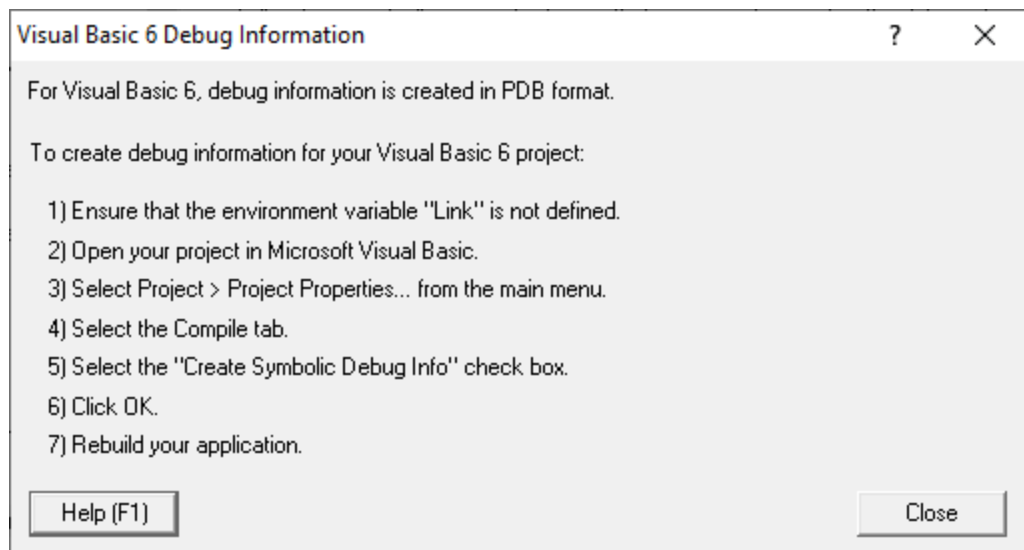
Symbol lookup for other compilers

If you are using another compiler click the link to see information about configuring debug information for that compiler.



After selecting the compiler, clicking the link will show a dialog box containing information relevant to the selected compiler.

For example:



Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.4.3 Symbol Servers

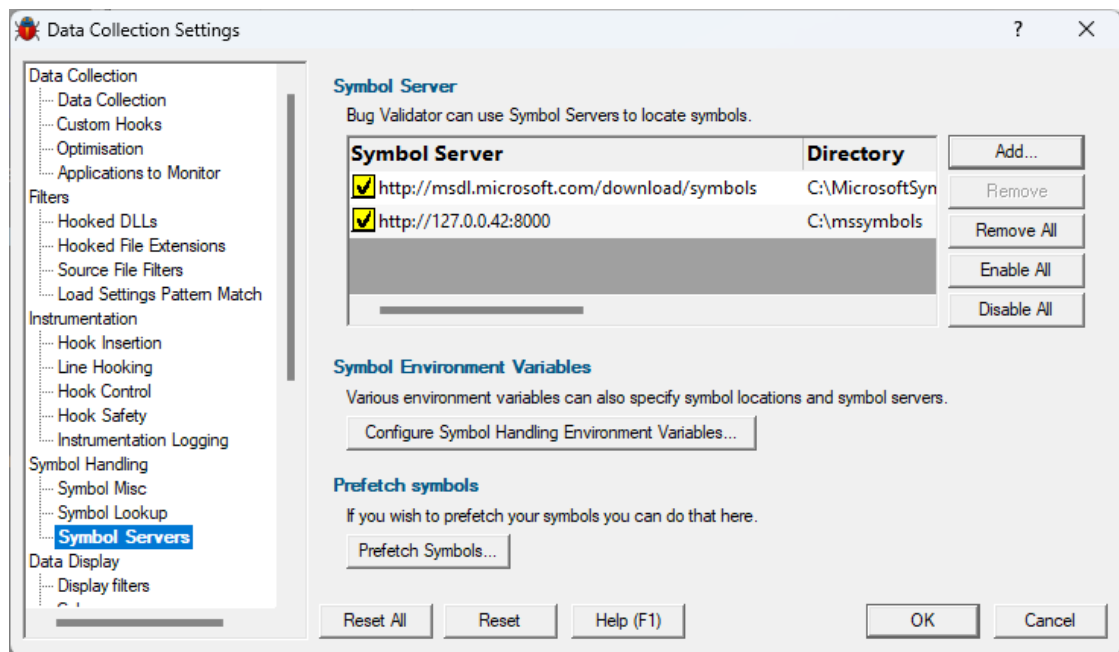
The **Symbol Servers** tab allows you to specify Symbol Servers to retrieve symbols used in your application.



You do not need to specify symbol servers if you do not wish to, and Bug Validator will work correctly without them.



Read on, or click a setting in the picture below to find out more.



Symbol servers

Symbol servers are entirely optional, but are useful for obtaining symbols from a centralized company resource or for obtaining operating symbols from Microsoft.

The default symbol server is the Microsoft symbol server used for acquiring symbols about Microsoft's operating system DLLs. You may also wish to add some symbol servers for any software builds in your organisation.

A symbol server is defined by at least the following:

- the symbol server dll to be used to handle the symbol server interaction
- a directory location where symbol definitions are saved
- the server location - a url

Each symbol server can be enabled or disabled allowing you to keep multiple symbol server configurations available without constantly editing their definitions.

You can define up to four symbol servers and more than one can be enabled at a time.

Symbol Server Errors

Any symbol server entry shown in red indicates there is a problem with parts of the definition of that symbol server.

In the image shown above the symbol server at <http://127.0.0.42:8000> cannot be reached. It is either offline or does not exist.

Managing symbol servers

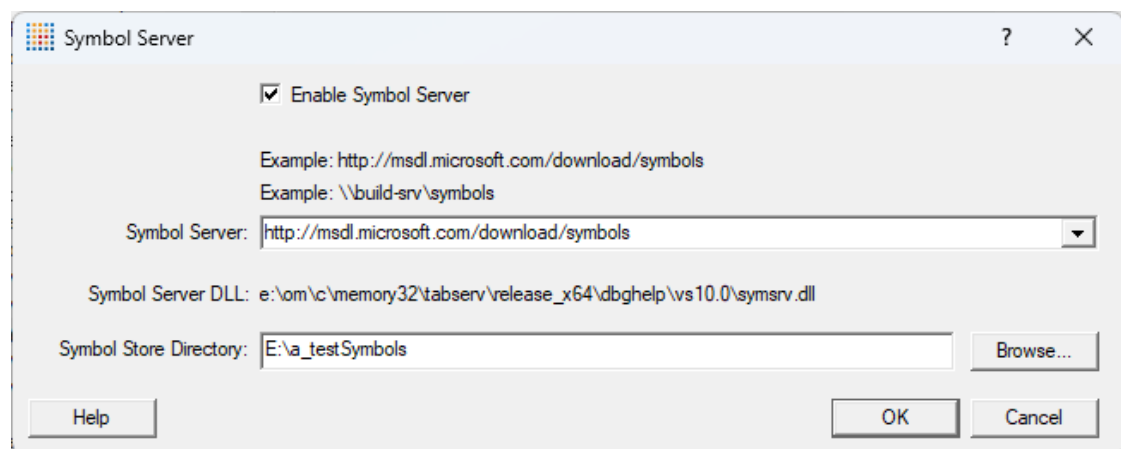
- **Add...** ➤ displays the symbol server dialog described below
- **Remove** ➤ remove selected symbol server(s) in the list
- **Remove All** ➤ remove all symbol servers
- **Enable All** ➤ enables all symbol servers in the list
- **Disable All** ➤ disables all symbol servers

You can also enable or disable an item in the list via the yellow check box at the left of each row.

To edit the details for a symbol server, just double click the entry in the list to show the symbol server dialog again.

Symbol server dialog

The dialog initially appears pre-populated with some default values and allows you to set up or edit the definition of a symbol server. Some of the default values can be changed.



- **Enable Symbol Server** ➤ enable or disable this server

The following three entries must be set to enable the **OK** button and define the symbol server.

 OK button not enabled? The OK button will only be enabled when the following entries have a valid value: - Symbol Server DLL names a dll present in the Memory Validator install directory. - Symbol Store Directory has been specified and exists. - Symbol Server URL has been specified (this value will not be checked for correctness).

- **Symbol Server** ➤ select a predefined public symbol server or enter the URL of the symbol server you wish to use - the Microsoft server is initially set as the default
- **Symbol Store Directory** ➤ enter or **Browse** to set the directory that will contain local copies of the downloaded symbols

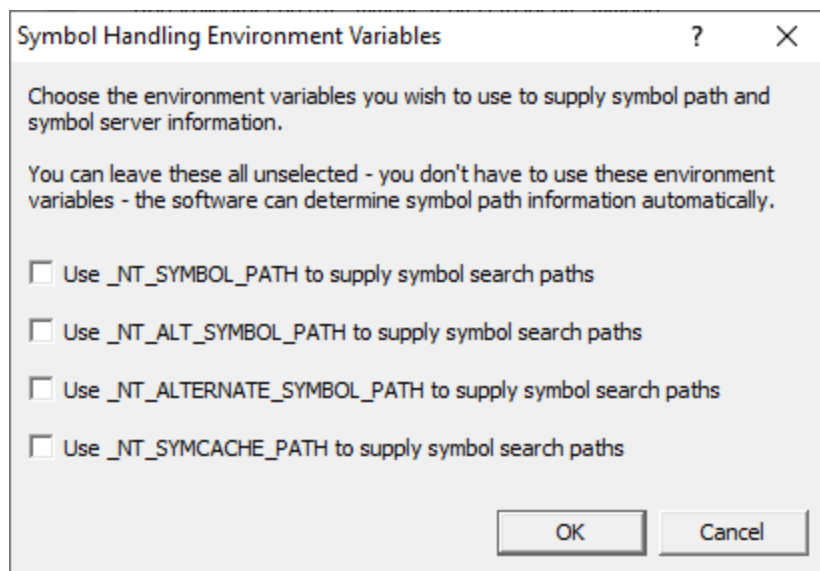
The **Symbol Server DLL** is set based on the Symbol Lookup settings you have chosen.

Environment variables related to symbols

If you wish, you can set some environment variables to supply symbol paths.

- **Configure Symbol Handling Environment Variables** ➤ opens the dialog below

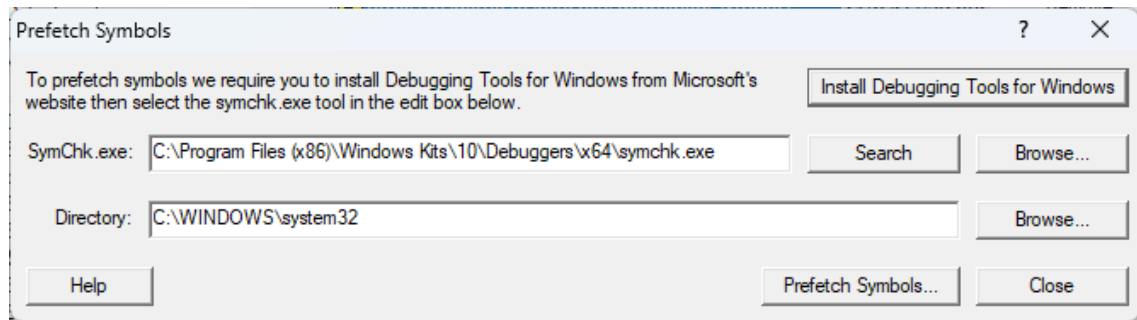
Check the desired options - if any.



Pre-fetching symbols

To avoid delays when using symbol servers, you can trigger the retrieval of symbols (by running SymChk.exe) to collect symbols for all executable files specified in the exe/dll which you associated with each symbol server.

- **Prefetch Symbols...** ➤ open the Prefetch Symbols dialog below to continue



Prerequisites for pre-fetching symbols

The pre-fetching of symbols requires the installation of Microsoft's Debugging Tools [\[1\]](#).

 You may already have Debugging Tools if you've previously installed the Windows Driver Kit (DDK) or the Windows SDK.

- **Install Debugging Tools for Windows** ➤ opens a web page (as above) to download and install the x86 or x64 Debugging Tools for Windows

After installing the Debugging Tools, you must specify the location of SymChk.exe from the installed area.

- **SymChk.exe** ➤ enter or **Search** to scan the computer for possible SymChk.exe executables. This search is usually fast, but depending on the computer could be time consuming. For that reason, we don't perform this search automatically.
- **SymChk.exe** ➤ enter or **Browse** to SymChk.exe location

A typical path might be `C:\WinDDK\7600.16385.1\Debuggers\symchk.exe`

- **Directory** ➤ enter or **Browse** to choose the directory you want to scan for dlls to fetch symbols

Getting the symbols

 Note that prefetching symbols may consume a large amount of disk space and download bandwidth.

You should ensure that you have at least 2GB or 3GB of disk free space, because of the total size of the download packages.

- **Prefetch Symbols...** ➤ runs SymChk.exe to get all the symbols

The symbols for each symbol server are stored in the associated symbol store directory.

 If no symbol servers are specified in the symbol server settings above, you'll see a warning dialog and no symbols will be fetched.

Command line pre-fetching of symbols with the SymChk utility

The section on Pre-fetching symbols above is a convenient alternative to manually using the SymChk.exe utility.

To avoid delays when using symbol servers, you can pre-fetch symbols using the SymChk.exe command line tool that is part of Microsoft's Debugging Tools [\[2\]](#).

You may want to add the folder of the Debugging Tools for Windows package to the PATH environment variable on your system so that you can access this tool easily from any command prompt.

Example:

To use SymChk.exe to download symbol files for all of the components in the `c:\windows\System32` folder, you might use the command:

```
symchk.exe /r c:\windows\system32 /s SRV*c:\symbols*http://msdl.microsoft.com/download/sym
```

where

`/r c:\windows\system32` finds all symbols for files in that folder *and any sub-folders*

`/s SRV*c:\symbols*http://msdl.microsoft.com/download/symbols` specifies the symbol path to use for symbol resolution.

In this case, `c:\symbols` is the local folder where the symbols will be copied from the symbol server.

➔ To obtain more information about the command-line options for SymChk.exe, type `symchk /?` at a command prompt.

Other options include the ability to specify the name or the process ID (PID) of an executable file that is running.

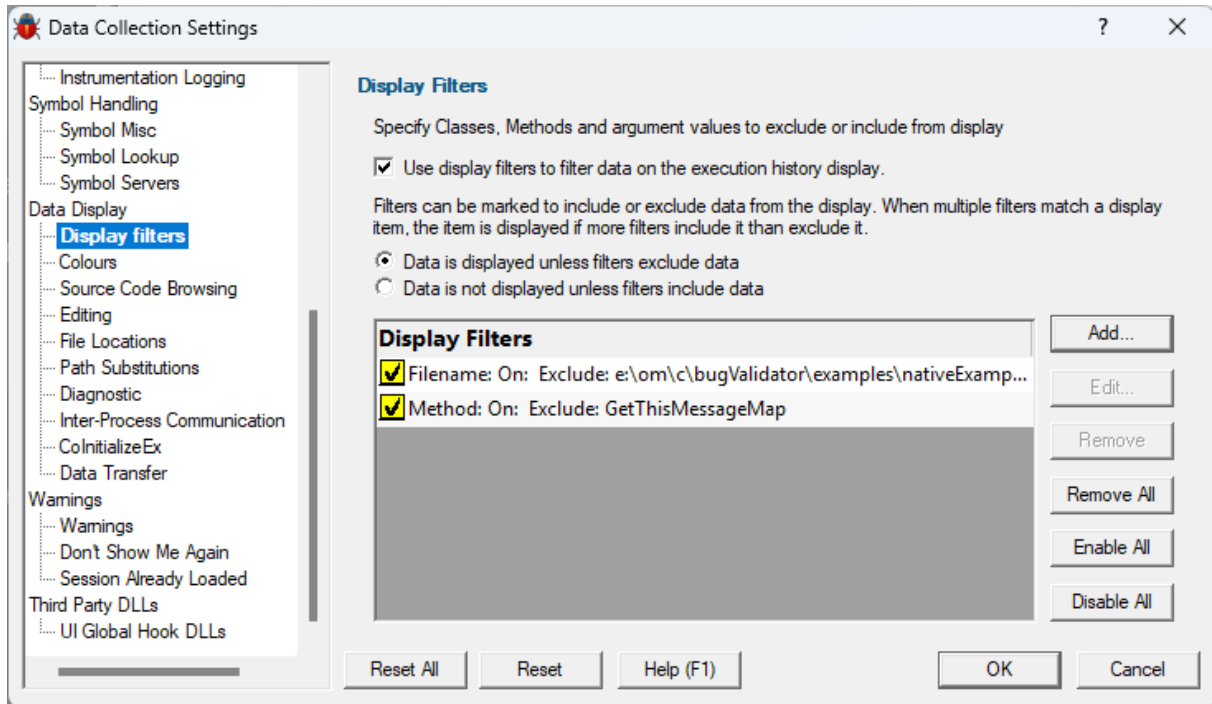
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.5 Data Display

3.9.1.5.1 Display Filters

The **Display Filters** tab allows you to configure filters to restrict the display of data on the execution history view.



The display filters work by specifying a data match which will include or exclude a data item from the display. When more than one filter matches an item of data, the data is displayed if the number of including filters is greater than or equal to the number of excluding filters. Data can be matched by filename, class, method, class and method, argument type, argument name, argument value (where argument means any parameter, variable or return value).

The inclusion and exclusion concept allows you to exclude data from one part of your application unless another criteria for display is met.

- **Choose Add...** ➤ to add a new filter. The Display Filter dialog is displayed.
- **Choose Edit...** ➤ to edit an existing filter. The Display Filter dialog is displayed.
- **Choose Remove** ➤ to remove selected filters from the list.
- **Choose Remove All** ➤ to remove all filters from the list.

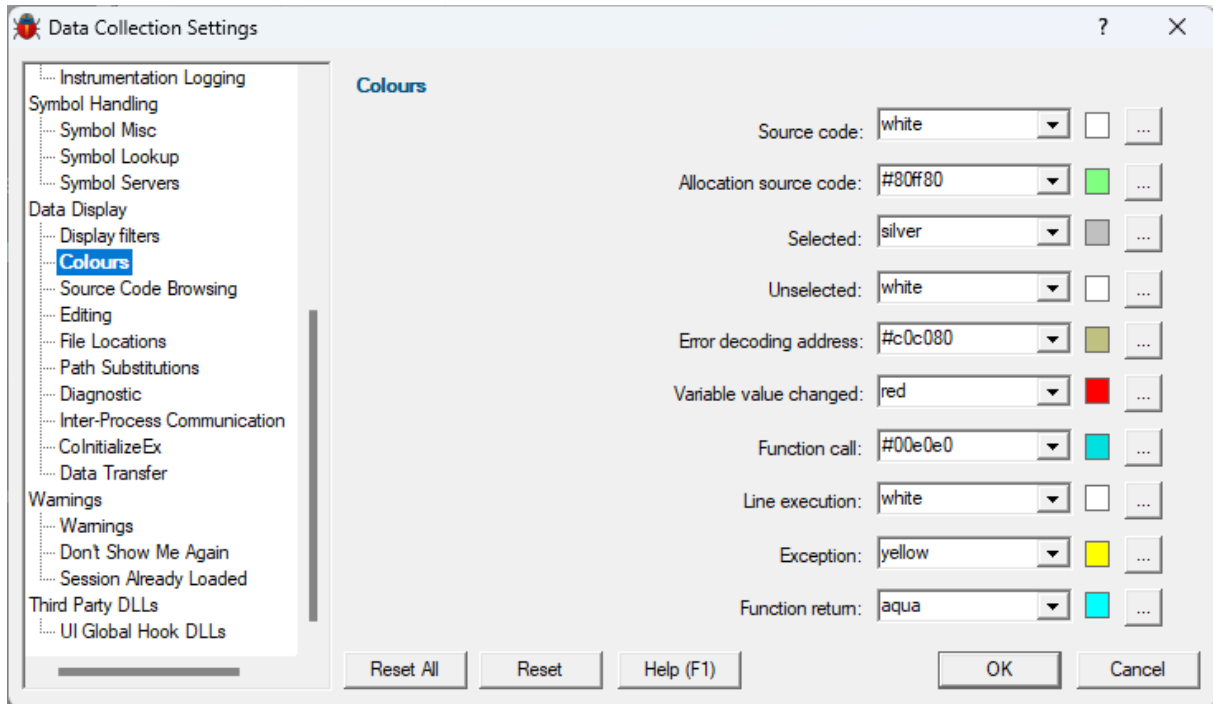
1. Select the filter type.
2. The appropriate edit fields are enabled. Enter the string(s) to filter.
3. Choose how the data matching is to be performed by selecting or deselecting the **Match case** check box and **Match whole string** check box.
4. Choose if the filter is going to include data or exclude data from the display.

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.5.2 Colours

The **Colours** tab allows you to specify the colours that will be used to represent each type of data item collected by Bug Validator.



For each colour there are two ways of specifying the colour to use.

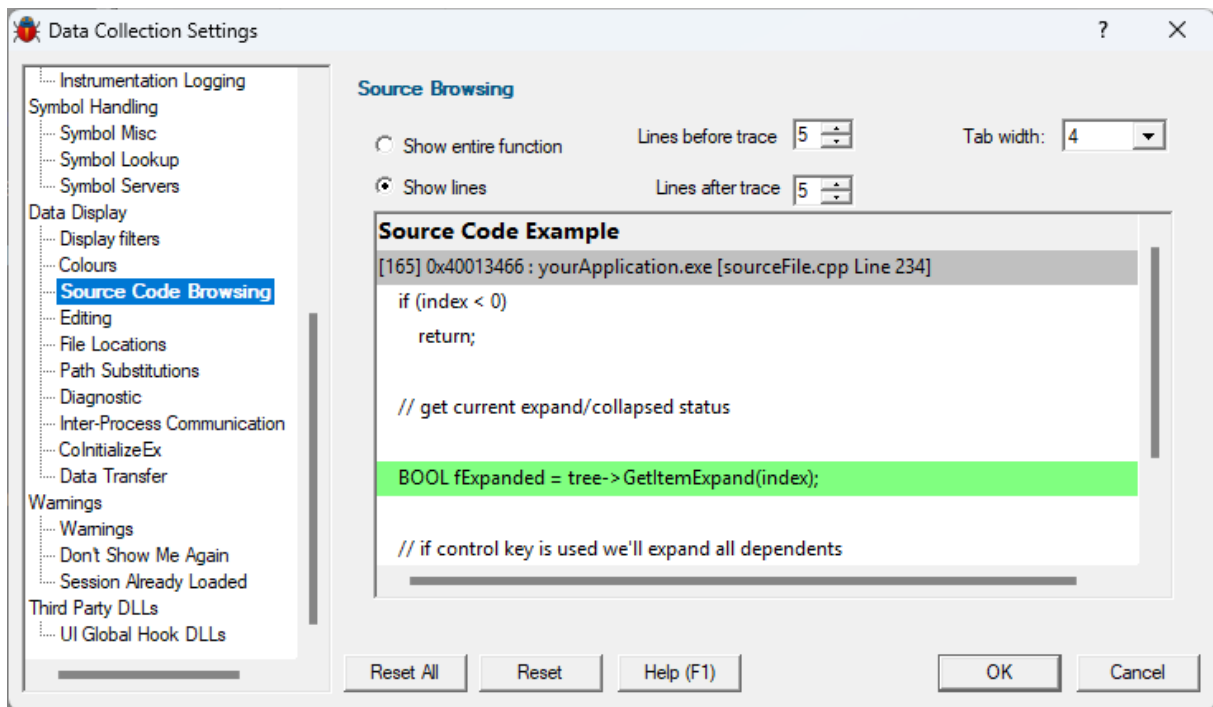
- Select a named colour from the combo box.
- Click on the button labeled ... to edit the colour using the standard Microsoft® colour choosing dialog.

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.5.3 Source Code Brow sing

The **Source Code Browsing** tab allows you control how source code is displayed.



Source Browsing

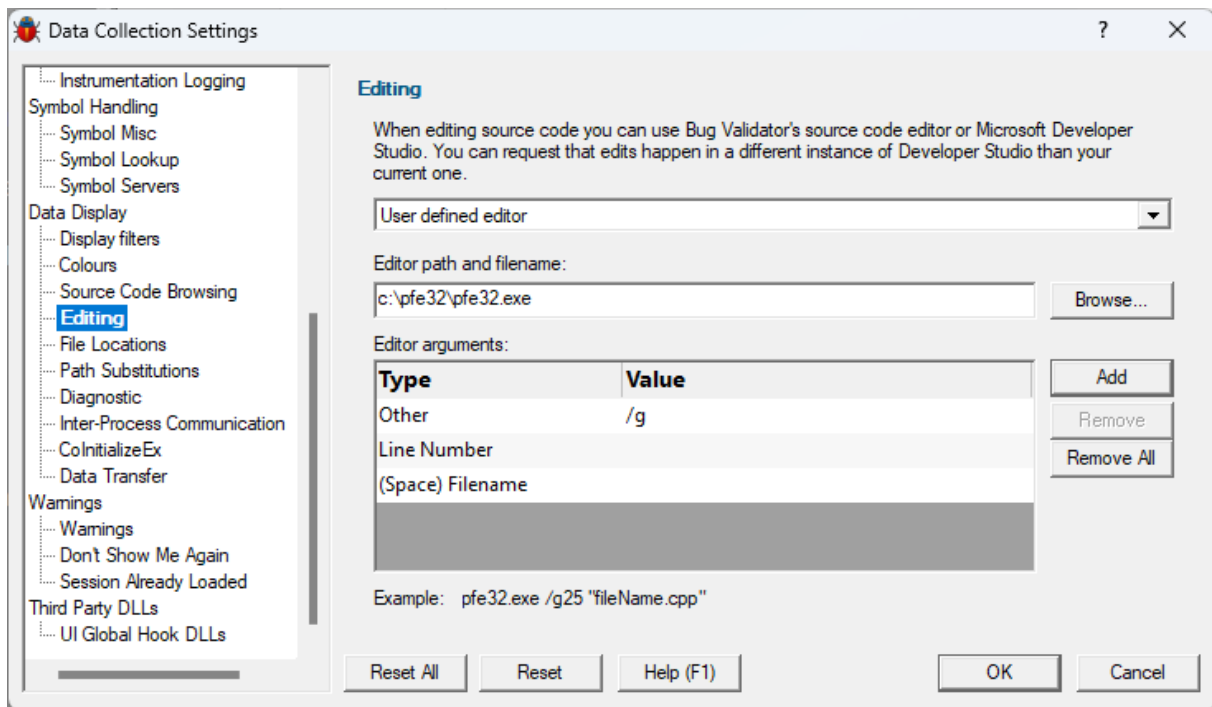
When expanding the callstack to view the source code in a view, Bug Validator can show the entire function containing the executed line or just a fragment of the surrounding code.

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.5.4 Editing

The **Editing** tab allows you to configure the editing options for Bug Validator.



Editing

When editing source code from within Bug Validator you can do the editing using Bug Validator's syntax coloured source code editor, using Microsoft® Developer Studio® 6.0, or using a custom editor definition. Select the appropriate entry in the combo box.

When using Microsoft® Developer Studio® to edit files, you can choose to edit source code using a currently open instance of Developer Studio® (probably the same one you are using to develop your application), or to open a new instance of Developer Studio®.

Custom Editor

To specify the editor to use, type the path to the editor in the edit field **Edit path and filename**, or use the **Browse...** button to use Microsoft's file dialog to specify the editor.

The picture above shows an example configuring the Programmers File Editor (Pfe32.exe) to edit a file and position the cursor at the appropriate line using the /g command line switch.

- **Choose Add...** ➤ to add a new argument. If you choose type "Other" you can specify a value in the value column.
- **Choose Remove** ➤ to remove the selected argument.
- **Choose Remove All** ➤ to remove all arguments.

If you specify no arguments, the editor will be passed the filename to edit. If the editor requires command line switches to specify the filename and/or the line number and/or any optional arguments, you must specify the arguments in the list of **Editor arguments**. Arguments are appended to the editor name in the order shown in the list. An example command line is shown below the list for the file fileName.py, line 25.

There are six types of argument:

- **(Space) Filename**. This appends a space followed by the filename.
- **Filename**. This appends the filename.
- **(Space) Line Number**. This appends a space followed by the line number.
- **Line Number**. This appends the line number.
- **Space**. This appends a space.
- **Other**. This appends the text typed in the **Value** column of the list.

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

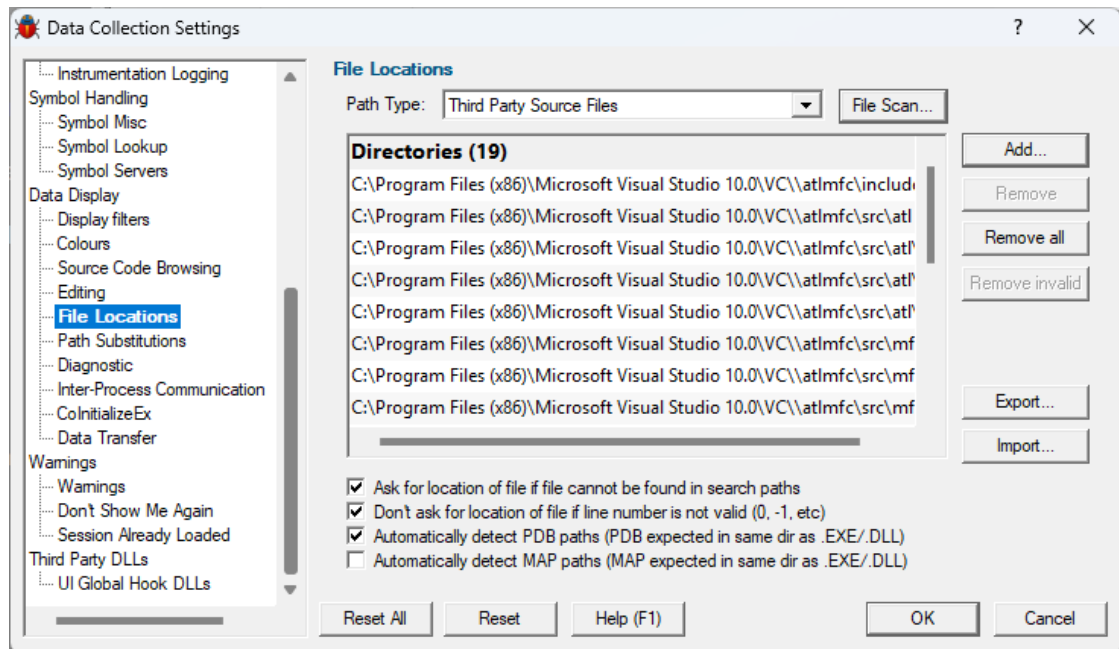
3.9.1.5.5 File Locations

The **File Locations** tab allows you to specify which directories Bug Validator should look in for source code files, whether that's your own or third party code.

The default settings are shown below:



Read on, or click on a setting in the picture below to find out more:



File locations

Sometimes the information Bug Validator has access to consists of the file name, but not the directory.

When this happens Bug Validator scans a set of directories that it knows about in order to find the file.

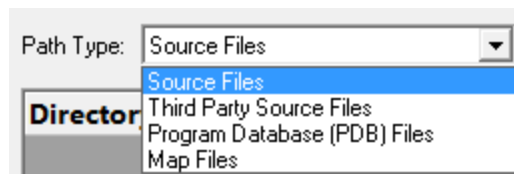
The options below allow you to specify those directories that should be searched for source files, PDB files and MAP files.

If a file can't be found, you'll get prompted for a location, but you can control this below as well.

Setting directories for a path type

There are four path types, and a separate list of directories to scan for each one.

- **Path Type** ➤ select the type of file with which you want to modify the list directory



 You don't *have* to specify any directories if you don't want to, or if you just don't have them. Nor do you have to give directories for *all* the path types.

Prompting for file locations

Whenever a file still cannot be found, then the default action is for a dialog to ask you where it is.

To avoid frequent user interruption, it is recommended that the directories for source code files (yours and third party) are specified, enabling Bug Validator to automatically load source code for browsing.

If however, you don't want to be prompted for locations, you can disable that too.

- **Ask for location of file...** ➤ untick to stop prompting for file locations

Even when prompting is switched on, it can still happen that the line in question is invalid anyway, e.g. line number 0 or -1.

The default is not to prompt for invalid lines, but if you want to know when that happens, just switch that behaviour off.

- **Don't ask for location of file if line number is not valid...** ➤ untick to be prompted for invalid lines anyway

PDB (program database) file paths

Normally PDB search paths are automatically generated, based on the same directories that .exe and .dll files are found in:

- **Automatically detect PDB paths** ➤ automatically detect PDB locations (the default)

However, it is recommended that you specify paths for PDB (program database) files, especially if your build environment dictates that PDB files are kept in different directories to their binaries.

If you don't automatically generate PDB paths and you don't specify any paths for PDBs, the search path will be defined as the current directory plus any paths found in the following environment variables:

- _NT_SYMBOL_PATH
- _NT_ALTERNATE_SYMBOL_PATH
- SYSTEMROOT

MAP file paths

It's recommended that you specify paths for Map files if your build environment means they are kept in different directories to their binaries.

If you don't specify any paths for Map files, then search paths are automatically generated, based on the same directories that .exe and .dll files are found in.

Manually adding path type directories

Once you have chosen your path type you can modify the list of files for each path type in the following ways:

- **Add** ➤ appends a row to the directory list ➤ enter the directory path

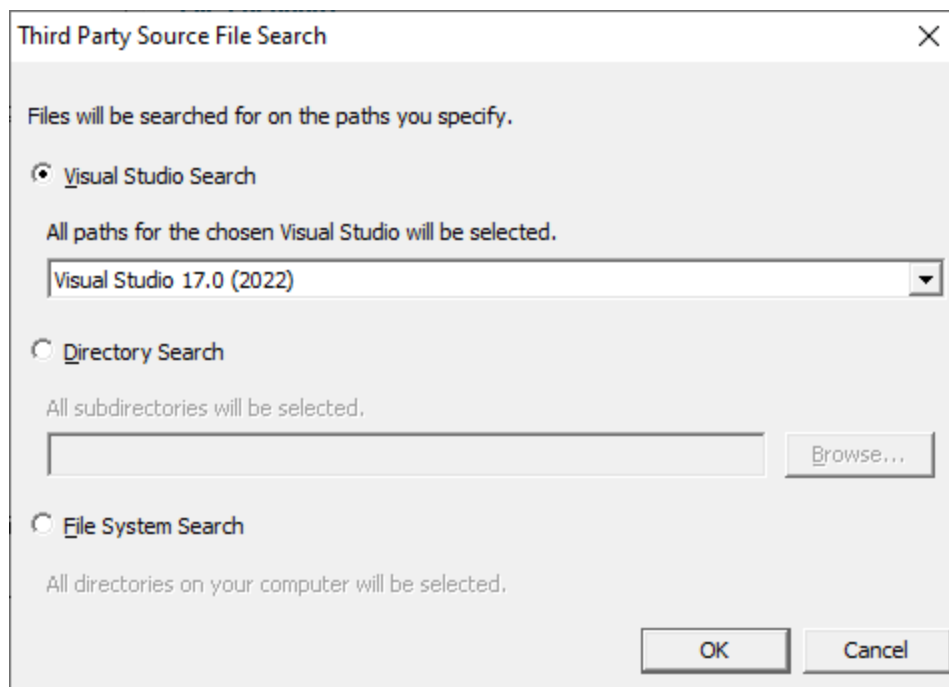
Edit a directory path by double clicking the entry. The usual controls apply for removing list items:

- **Remove** ➤ removes selected items from the list
- **Remove all** ➤ clears the list
- **Remove invalid** ➤ removes all items that are not valid directories from the list

Alternatively, press **Del** to delete selected items, and **Ctrl** + **A** to select all items in the list first.

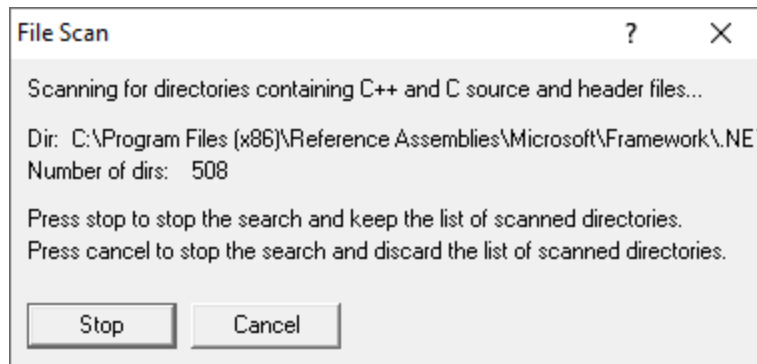
Scanning for directories to add

The **File Scan...** button displays the File Search dialog to provide three ways of specifying the files to scan.

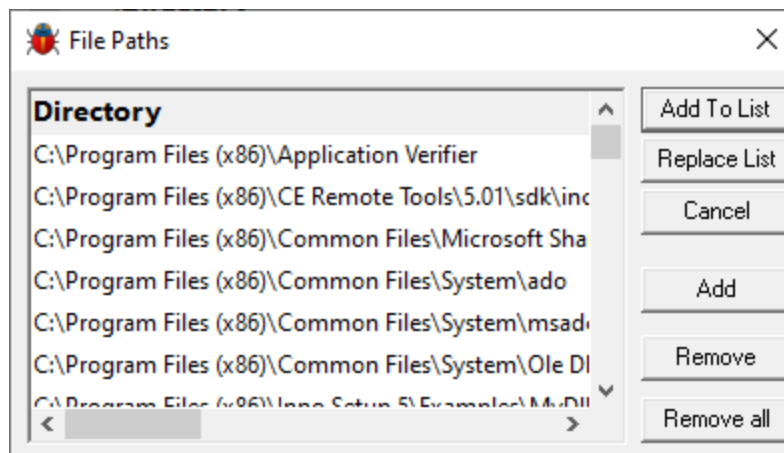


- **Visual Studio Search** ➤ choose the version of Visual Studio ➤ **OK** ➤ starts a scan for directories related to that version of Visual Studio
- **Directory Search** ➤ **Browse...** displays a directory browser ➤ **navigate** to a location you want to scan within ➤ **OK** ➤ starts a scan for directories
- **File System Search** ➤ **OK** ➤ starts a scan of all drives for directories containing files

All options will bring up a **File Scan** dialog indicating number of relevant directories found, and giving you a chance to **Stop** or **Cancel** the scan at any time:



Once the scan is complete you'll see the **File Paths** dialog showing you the scan results:



You can modify the list of resulting directories by adding, removing or editing, exactly as for the path type list above.

Once you're happy with the scan results, either append or replace the path type directories with the scan results.

- **Add To List** ➤ adds the scan results list to the path type directories and closes the File Paths dialog
- **Replace List** ➤ replaces the path type directories with the scan results
- **Cancel** ➤ discard the scan results and close the dialog

Exporting and Importing

Since the list of path types and their file locations can be quite complicated to set up and optimise, you can export the settings to a file and import them again later. This is useful when switching between different target applications.

- **Export...** ➤ **choose or enter** a filename ➤ **Save** ➤ outputs all the path types and their file locations to the file

- **Import...** > **navigate** to an existing *.bvxfl file > **Open** > loads the hooking rule and the list of modules

Export file format

The file format is plain text with one folder listed per line. Sections are denoted by a line containing [Files] (for source code files), [Third] (for third party source code files), [PDB] etc.

Example:

```
[Files]
c:\work\project1\
[Third]
d:\VisualStudio\VC98\Include
[PDB]
c:\work\project3\debug
c:\work\project3\release
[MAP]
c:\work\project3\debug
c:\work\project3\release
```

Checking directory scanning order

To see the order in which the *DbgHelp.dll* process checks directories to find symbols, see the diagnostic tab, showing *DbgHelp debug* in the drop-down.

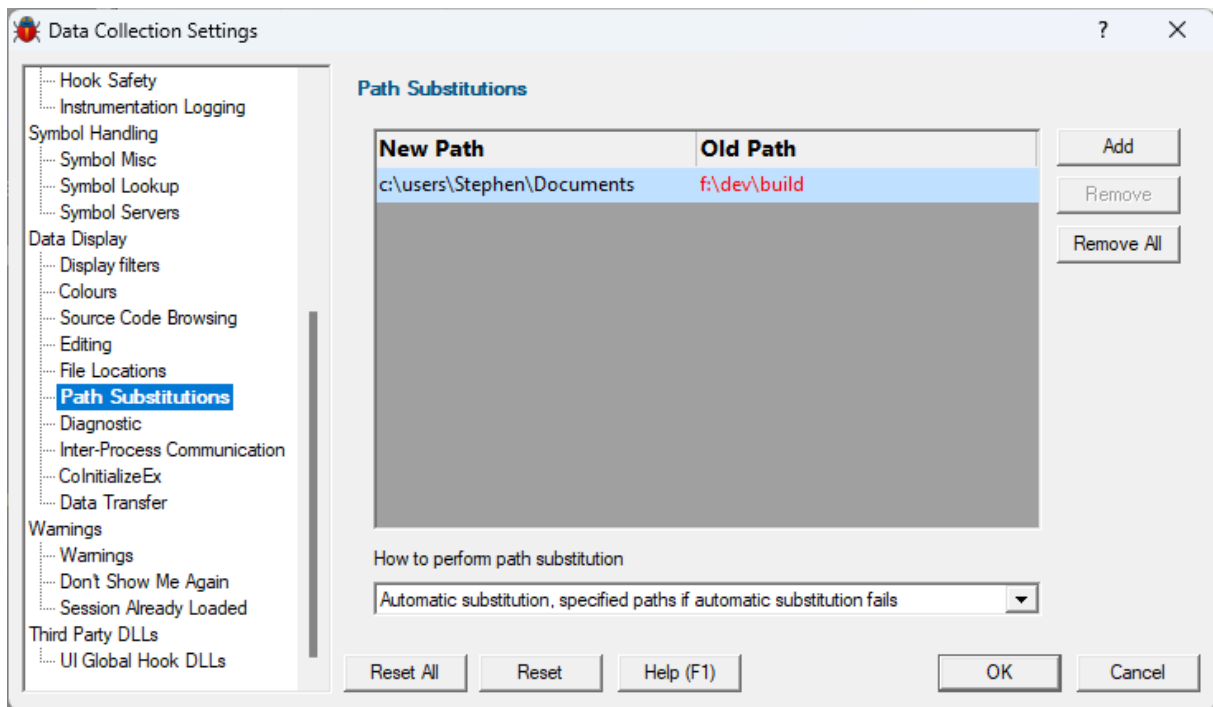
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.5.6 Path Substitutions

The **Path Substitutions** tab allows you to specify file path substitutions to handle copying builds from build machines to development or test machines .

The default settings are shown below:



Path Substitutions

Some software development schemes have multiple rolling builds of their software, often enabled by using substituted disk drive naming schemes.

When you download the build to your development machine for development and testing, debugging information may reference disk drives that don't exist on your machine, for example, drive X: while your machine only has C:, D:, and E: drives.

Or you may just be copying a build from a drive on a development machine to a subdirectory on a drive on your test machine.

These options let you remap the substitution so that the Bug Validator looks in the correct place for the source code.

- **Add** ➤ adds a row to the **File Paths Substitutions** table ➤ enter the new path that will replace the old path in the **New Path** column ➤ click in the **Old Path** column ➤ enter the path that is being replaced

For example, you might enter `c:\users\stephen\documents` for the new path and `f:\dev\build` for the old path.

You can double click to edit drives and paths in the table, or remove items:

- **Remove** ➤ removes selected substitutions from the list
- **Remove All** ➤ removes all substitutions from the list

Alternatively, press **Del** to delete selected items, and **Ctrl** + **A** to select all items in the list first.

Example: Changed disk drive

Project originally located at	m:\dev\build\testApp
Project copied to	e:\dev\build\testApp
New Path	e:\
Old Path	m:\

Example: Project copied to a new location

Project originally located at	f:\dev\build\testApp
Project copied to	C:\Users\Stephen\Documents\testApp
New Path	C:\Users\Stephen\Documents
Old Path	f:\dev\build

 The slashes do not have to match, a forward slash will match a backslash when comparing path fragments. This is deliberate - to improve ease of use with libraries built by different compilers (LLVM and compilers that use it use forward slashes, whereas Visual Studio etc use backslashes).

Path Substitution Method

Path substitution can be turned off, use only manually specified paths, perform automatic path substitution based on best guesses based on information in the executable, or a combination.

Use the combo box to choose the appropriate path substitution method. The default is automatic path substitution and if that fails to try path substitution using the manually specified paths.

- **No path substitution** > path substitution does not happen
- **Only substitute specified paths** > path substitution uses the manually specified paths
- **Automatic substitution only** > path substitution is performed automatically using information in the executable
- **Automatic substitution, specified paths if substitution fails** > an attempt at automatic path substitution is made, if this fails path substitution is performed using the manually specified paths

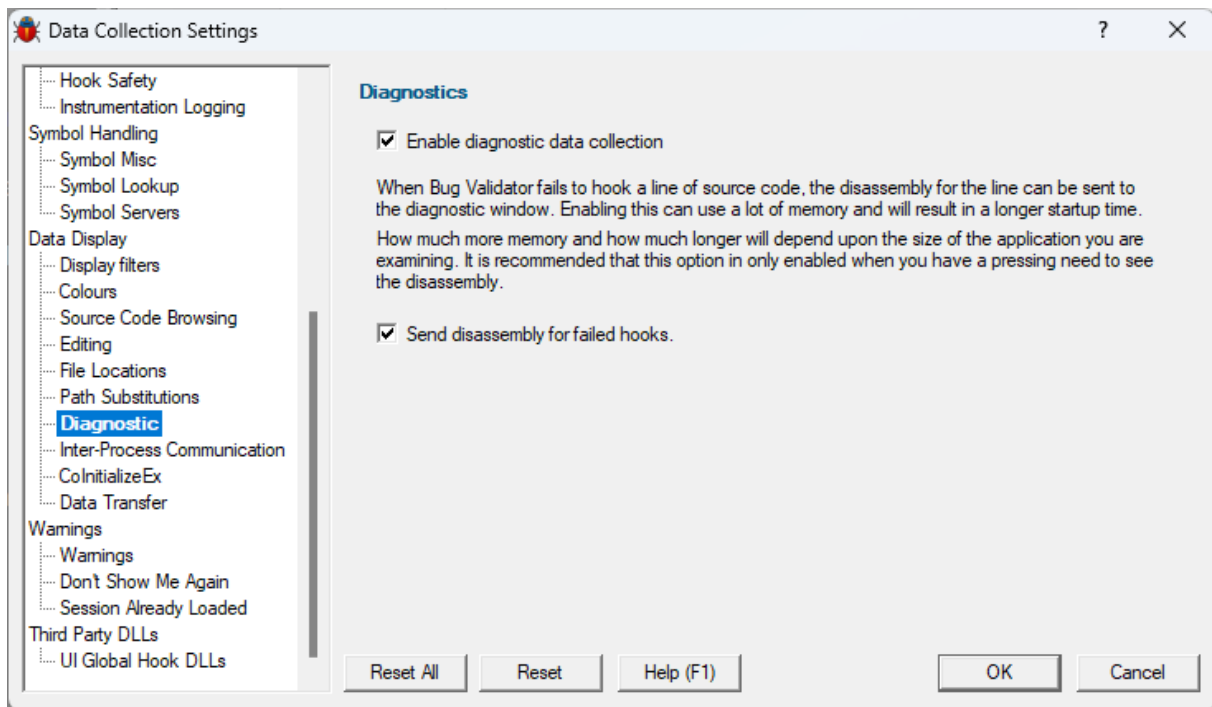
The default is **Automatic substitution, specified paths if substitution fails**.

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.5.7 Diagnostic

The **Miscellaneous** tab allows you to set diagnostic and symbol settings.



Diagnostics

Diagnostics

Collection: A lot of diagnostic information is collected and displayed on the diagnostic tab when attaching to a target program.

Some of this information is *always* sent to Bug Validator, but you may not want to see it all.

- **Enable diagnostic data collection** ➤ displays all diagnostic information in the diagnostic tab (on by default)

Disassembly: When hooking source code lines, some lines cannot be hooked due to the object code that corresponds to the source code location.

- **Send disassembly for failed hooks** ➤ shows the disassembly for lines that cannot be hooked (enabled by default)

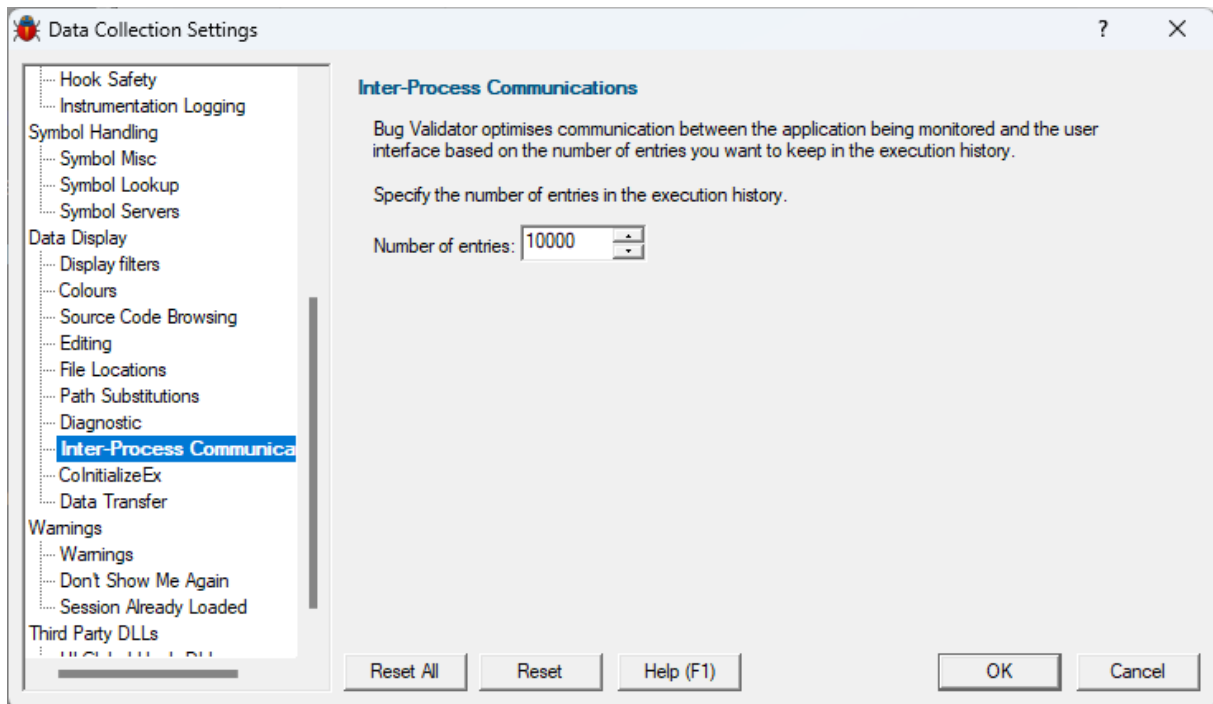
This *can* increase startup time and memory usage if used very frequently.

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.5.8 Inter-Process Communication

The **Inter-Process Communication** tab allows you to configure the size of the buffer used to communicate with the user interface.



Bug Validator communicates the execution history to the user interface via a buffer. Larger numbers for the number of entries mean that you will have more information in the execution history.

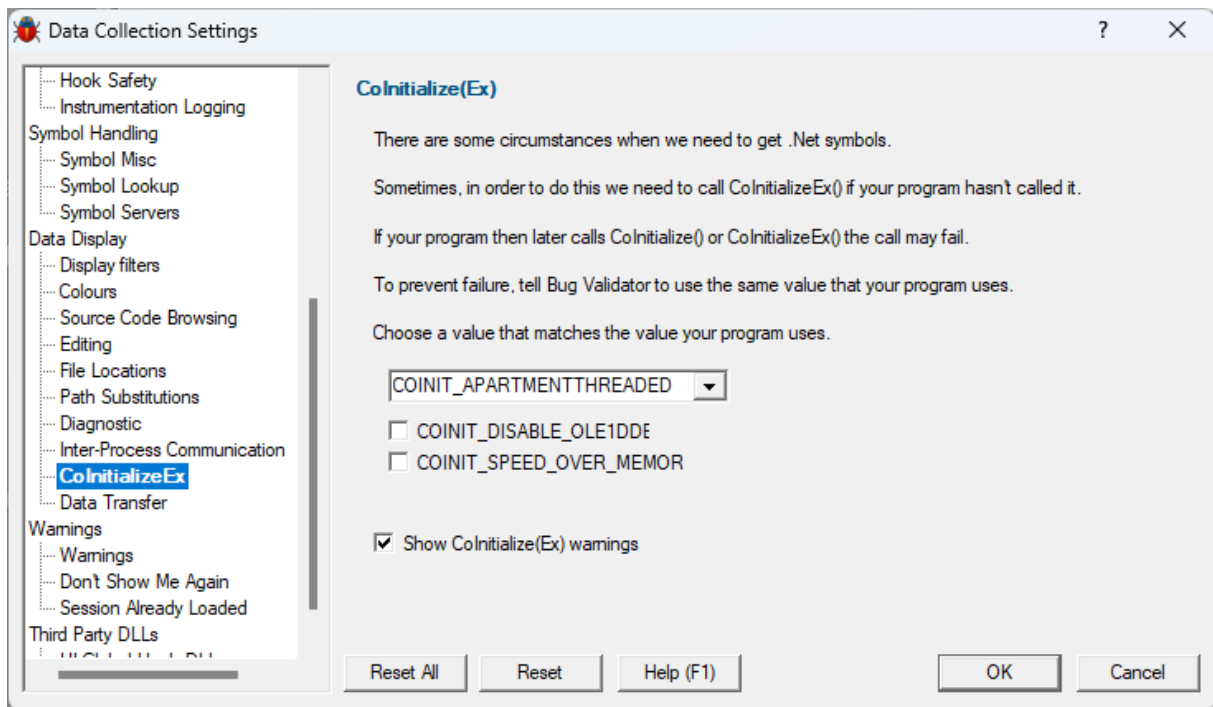
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.5.9 ColInitializeEx

The **ColInitializeEx** tab allows you to set the default behaviour used to initialize COM if Bug Validator needs to initialize COM to acquire symbols for .Net modules.

The default settings are shown below:



ColInitializeEx

In some situations the Validator needs to get .Net symbols and to do that COM needs to be initialized. This normally isn't a problem, but if your program also performs COM initialization and the sequence of events results in your COM initialization coming after the Validator's COM initialisation rather than getting the expected `ERROR_SUCCESS` return code you'll get either `ERROR_INVALID_FUNCTION` or `RPC_E_CHANGED_MODE`.

If you get `ERROR_INVALID_FUNCTION` this is OK, this just means you've called `ColInitialize()` or `ColInitializeEx()` multiple times with the same flags. **Your code needs to handle `ERROR_INVALID_FUNCTION` as not an error.**

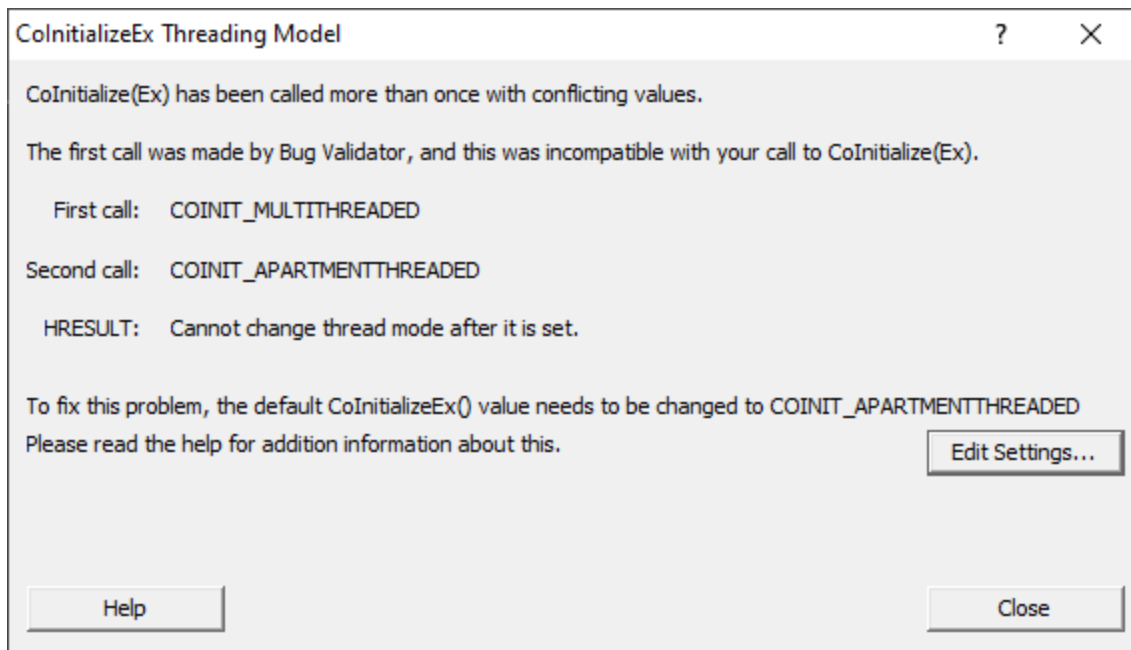
If you get `RPC_E_CHANGED_MODE` this means you need to change the Validator's default value to the same value your program is using. That's what this dialog allows you to do.

If you also wish to disable OLE DDE or favour speed rather than memory use we've provided appropriate options for you to select to add those flags to the threading mode.

See the Microsoft documentation for additional information on the behaviour of `ColInitialize()` and `ColInitializeEx()`.

Runtime detection of ColInitializeEx conflict

When the above scenario happens, that the Validator has initialized COM before your code initializes COM and your call returns `RPC_E_CHANGED_MODE`, we display a dialog to warn you about this failure and provide you with the option of editing the default value for subsequent runs of your application.



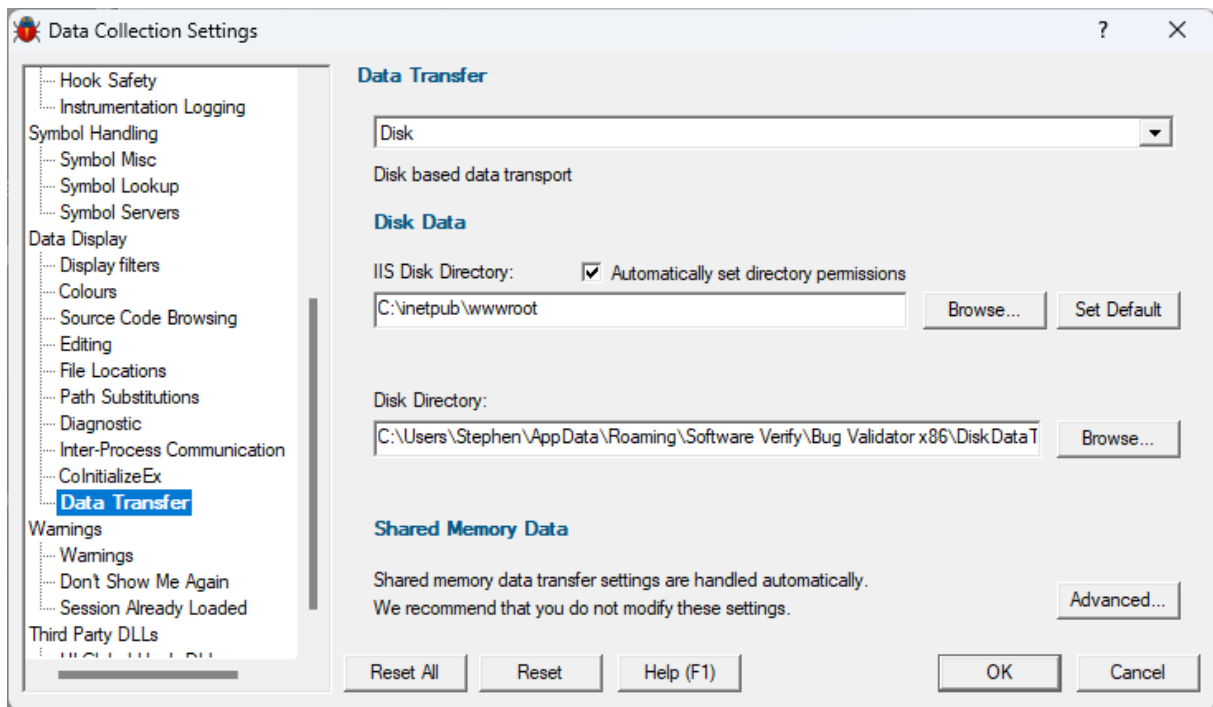
- **Edit Settings...** ➤ opens the ColInitializeEx dialog shown above.

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.5.10 Data Transfer

The **Data Transfer** tab allows you to specify the overall behaviour of data transfer between your application and Bug Validator.



Data Transport

Choose the type of data transport you wish to use.

- **Automatic.** Applications and services use shared memory to transfer data. IIS uses disk based data transfer.
- **Disk.** Applications, services and IIS use disk based data transfer.
- **High Volume.** Data transfer has no data throttling applied to it. This mode is for use with applications that generate very high volumes of data rapidly. They typically exceed the buffering capabilities of Bug Validator when working with shared memory. The High Volume setting uses a data transport that doesn't have a data-throttling requirement allowing the high volume application to continue without waiting.

Automatic

Under most circumstances data transfer between Bug Validator and the target program (desktop, service, etc) is via shared memory. This is handled automatically.

Disk Data

Some applications and services don't allow shared memory access. For these occasions we use a file based data transfer, where the files are stored in a directory of your choice.

We provide two options for disk-based data transfer, one for most applications and services, and one for Internet Information Server, as this operates in a very restricted environment.

Both options are configured automatically, but you can override either by typing the path to a suitable directory or using the Microsoft directory browser.

The ISS path you enter will be determined by the settings you have configured for IIS using the Internet Information Services Manager tool. We won't discuss that here because if you're using IIS we assume you already know how to configure IIS correctly.

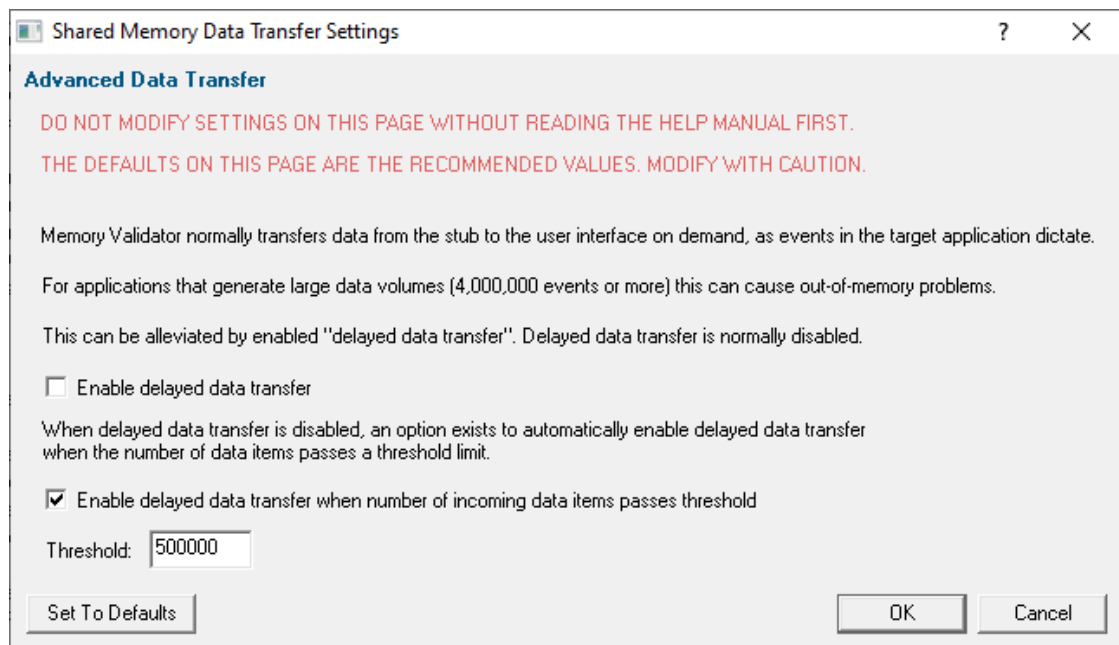
Advanced Shared Memory

Shared memory data transfer can also be configured **but we strongly recommend that you leave these settings alone.**



The **Data Transfer Helper** is a separate application supplied in the installation directory.

- **Advanced...** > opens the data transfer settings dialog.



Here be dragons!



Caution: Modifying the settings on this page and using the data transfer helper application can prevent Bug Validator from working correctly.

- **Set To Defaults** > if you have modified the settings, this resets them
 - ➔ See also the Reset to default buttons on the data transfer helper application below

If in doubt, don't modify these settings. If you promise to be careful, read on!

Delayed data transfer

Delayed data transfer is the process of throttling data rates in the stub so that the slower user interface can keep up with processing the data received.

In the stub, as an event occurs, data is queued and then sent to the user interface.

In the user interface, data from the stub is received and queued again for processing.

Any delay is usually in the slower user interface, but still not a problem for most applications.

However, some data intensive applications can generate so much data that the user interface gets swamped and can't process it all before running out of memory.

Temporarily limiting the data rate in the *stub* allows the user interface to stabilize the data processing.

Managing data rates

We recommend the default settings as shown above:

- disable delay data transfer for most applications
- enable *automatic* delay data transfer at a threshold of between 100,000 and 1,000,000 data items

If delayed data transfer is enabled all the time, the automatic options don't apply.

If you have more than 1GB RAM, you can raise these thresholds.

Data transfer helper application

A separate data transfer helper application is supplied in the installation directory.

The helper application can be used to modify low level settings that apply when delay data transfer is activated as above.

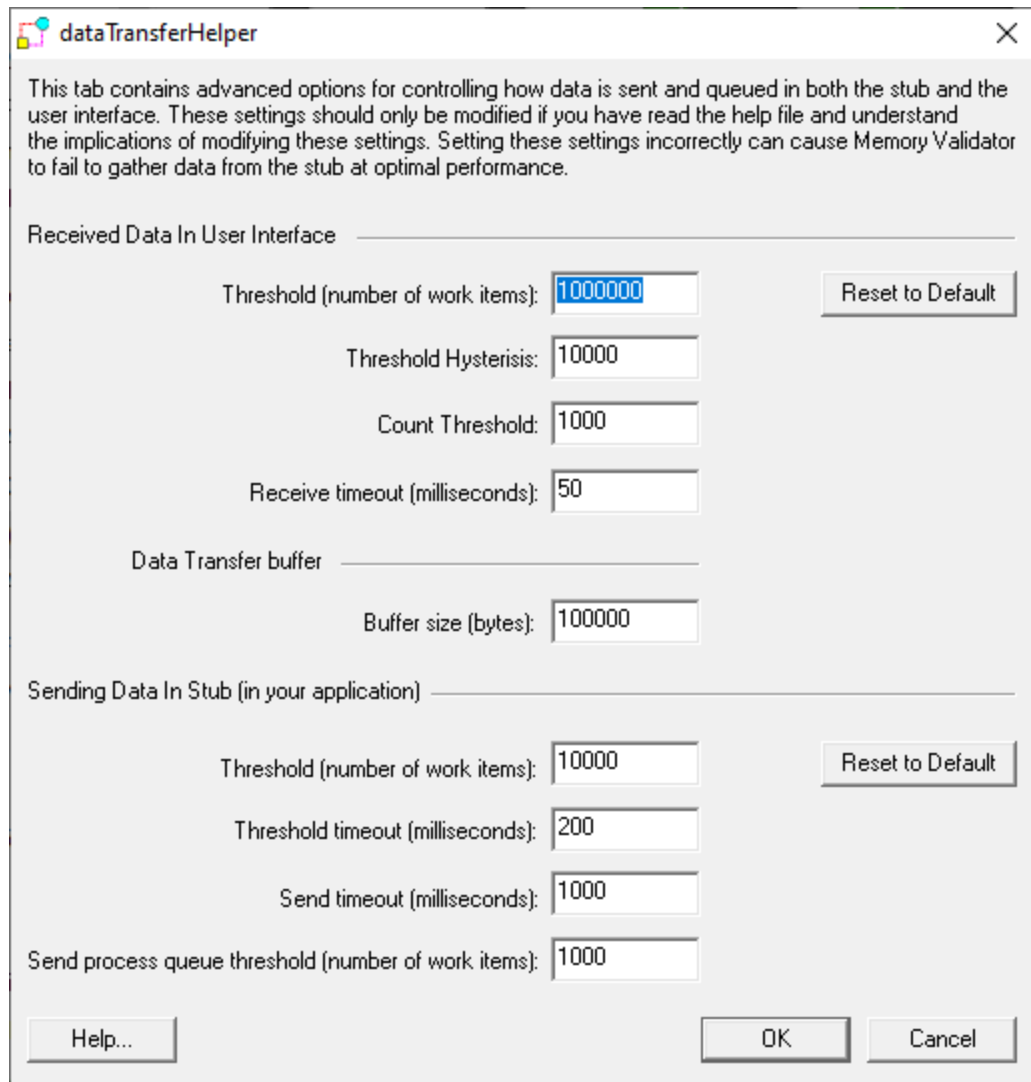


The helper should be used with care. We already warned of dragons above, but here we are, warning you again!

An HTML help page for this application is available by clicking the **Help** button on the helper application.

You can also find the help page directly as `dataTransferHelp.html`.

Please do take a moment to read the help before use.



Reset All - Resets **all** global settings, not just those on the current page.

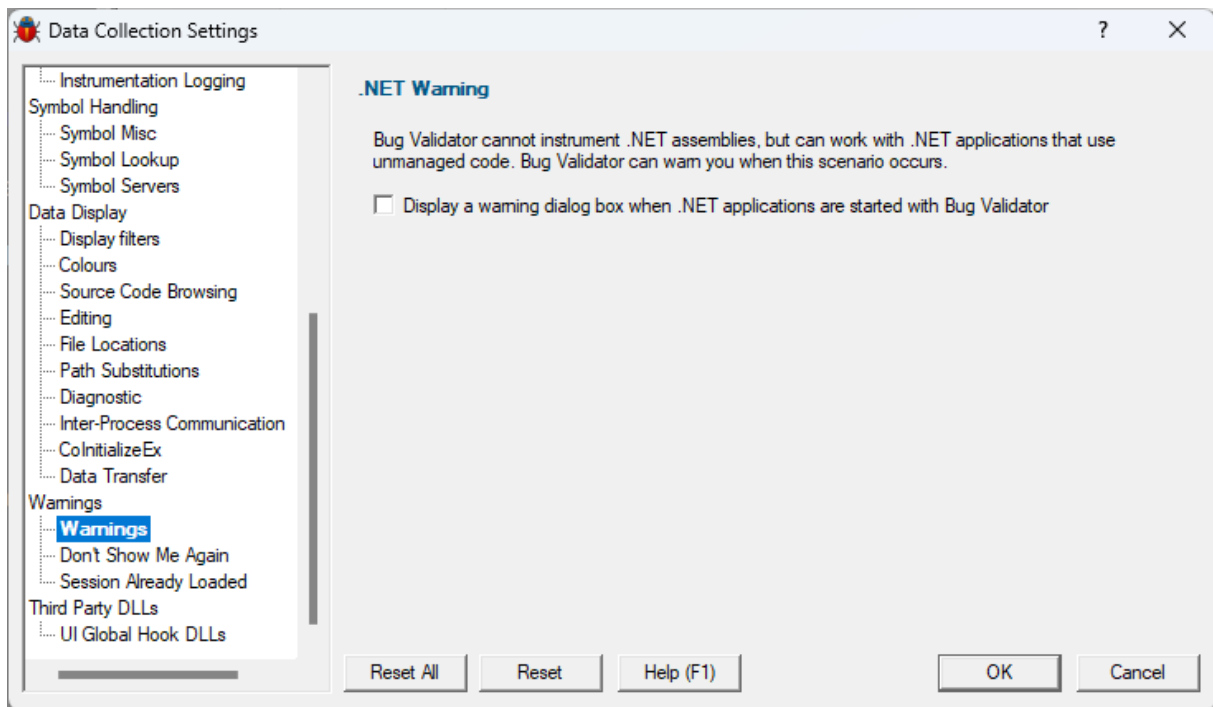
Reset - Resets the settings on the current page.

3.9.1.6 Warnings

Enter topic text here.

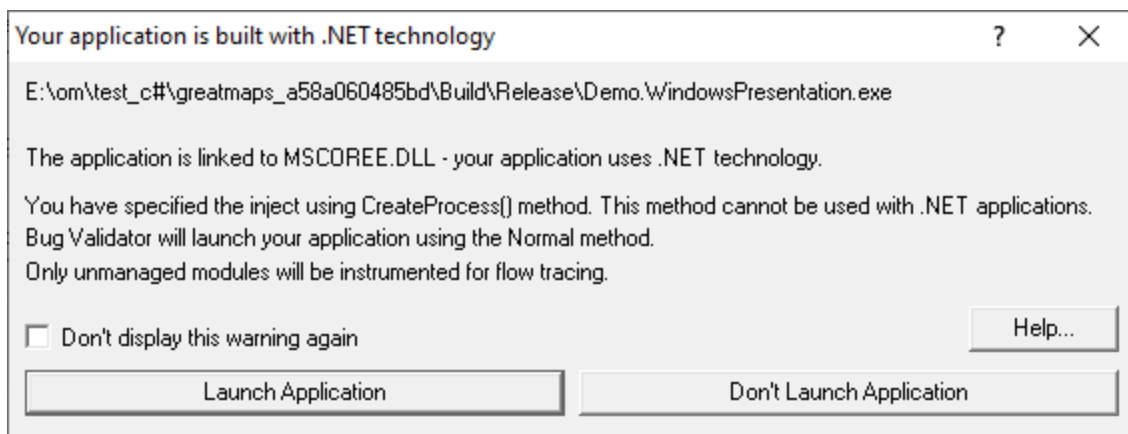
3.9.1.6.1 Warning

The **Warning** page allows you to control warnings that Bug Validator displays.



.NET Warning

Bug Validator cannot instrument .NET assemblies. This means that Bug Validator cannot monitor locks and thread synchronization primitives called from assemblies (also known as "managed code"). Bug Validator can monitor locks and thread synchronization primitives in any non-.NET DLLs in your application, even if your application is a .NET application. To help you identify when you are about to test a .NET application Bug Validator can display a warning dialog. The warning dialog is controlled by the check box in the .NET section of the dialog. Enable the check box to display the warning dialog. The warning dialog is shown below:

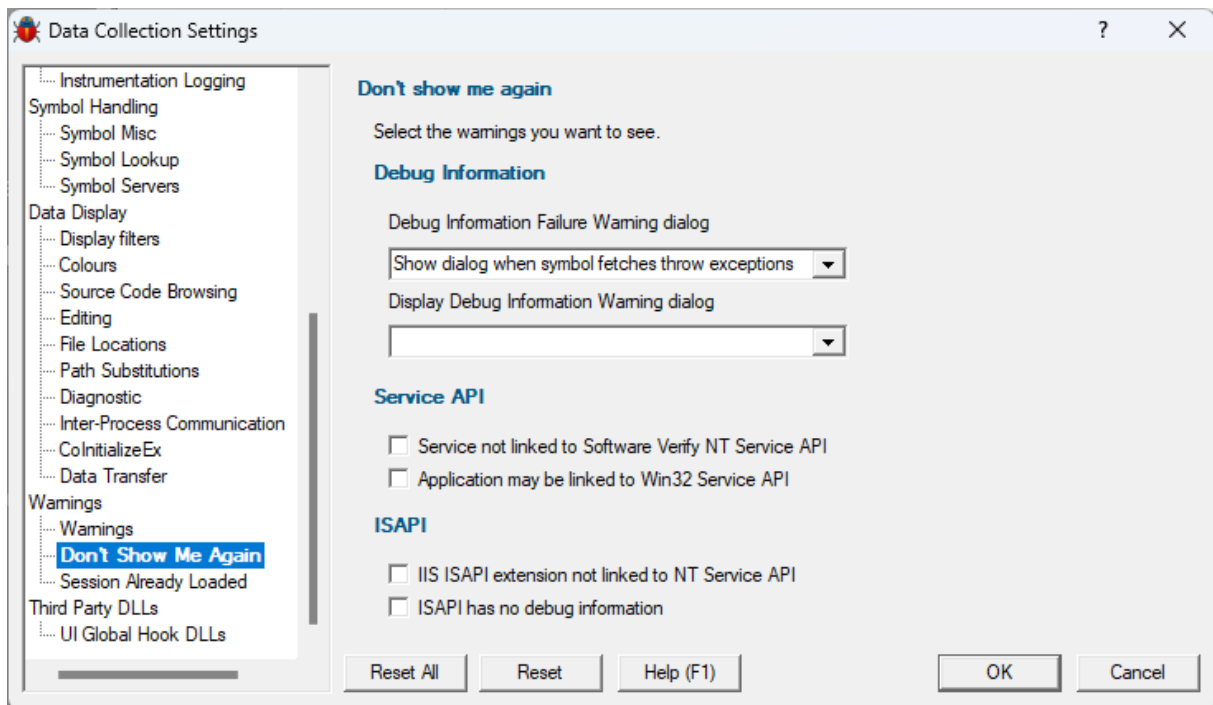


Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.6.2 Don't Show Me Again

The **Don't Show Me Again** page allows you to control warnings that Bug Validator displays.



Debug Information

Debug Information Failure Warning

When there is a failure collecting debug symbols a warning can be displayed. The options are:

- Always show dialog
- Never show dialog
- Show dialog when symbol fetches throw exceptions

Display Debug Information Warning

When no debug information is available for at least one module a warning can be displayed. The options are:

- Always show dialog
- Never show dialog
- Show dialog when debug information is missing

Services API

- **Service not linked to Software Verify NT Service API** ➤ warning will be shown if you try to monitor a service not linked to the Software Verify NT Service API. (on by default)

When trying to monitor a service Bug Validator can detect if the service is not linked to the NT Service API and display a warning.

It is possible to use the service API without linking to it (use GetProcAddress() to lookup the functions and call them) - in this case you would want to turn this warning off.

- **Application may be linked to Win32 Service API** ➤ warning will be shown if you try to start an application that appears to be a service - it uses Win32 Service APIs. (on by default)

ISAPI

NT Service API

When trying to monitor ISAPI extensions Bug Validator can detect if the ISAPI is not linked to the NT Service API and display a warning.

It is possible to use the service API without linking to it (use GetProcAddress() to lookup the functions and call them) - in this case you would want to turn this warning off.

Debug Information

Bug Validator can warn if the ISAPI has no debug information. There may be cases where you don't want to see this warning.

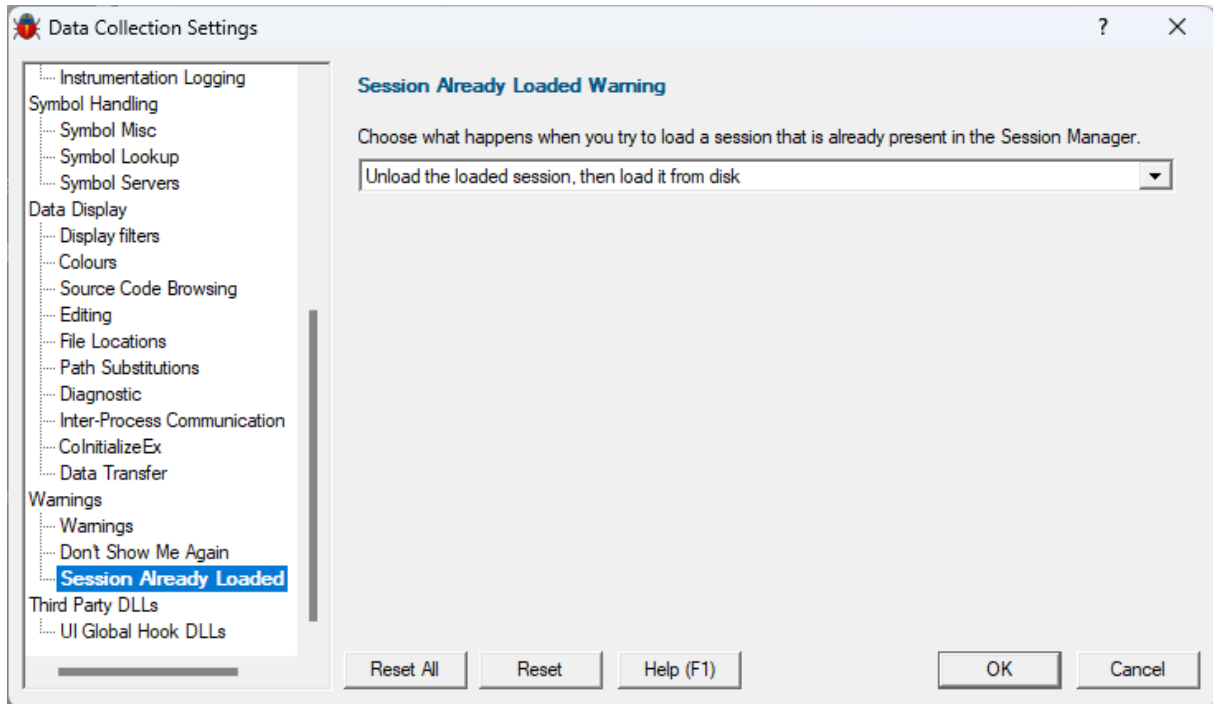
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.6.3 Session Already Loaded

The **Session Already Loaded** page allows you to control how Bug Validator behaves when you try to load a session that is already present in the Session Manager.

The default settings are shown below:



Sessions loaded into Bug Validator can be closed by many different actions:

- **File Close** ➤ user manually closing the session
- **Load Session** ➤ user manually loading a session
- **Launch** ➤ launching any application (native, .Net, etc) from the Bug Validator user interface
- **Inject** ➤ injecting into any application from the Bug Validator user interface
- **Wait for Application** ➤ waiting for any application and injecting into it when it launches, from the Bug Validator user interface
- **Command line** ➤ Bug Validator launches an application from the command line
- **Bug Validator started from the API** ➤ any use of the API that starts Bug Validator

When a session is closed in this manner, the session remains loaded into the Session Manager, unless it is purged based on the Session Manager settings.

This allows sessions to be kept in memory so that they can be inspected later.

Because of this arrangement, it is possible that you may try to load a session that is already loaded into the session manager.

When this happens, you have a choice of actions, allowing you to customize Bug Validator's behaviour.

- **Display the Session Already Loaded dialog** ➤ display the Session Already Loaded dialog. This is equivalent to an improved version of the experience prior to October 2025.
- **Display the Session Manager** ➤ display the Session Manager.
- **Display the loaded session** ➤ display the session that is already in memory without loading it from disk.
- **Unload the loaded session, then load it from disk** ➤ this option unloads the session found in memory, then loads the session from disk.

This option is provided to allow you to be sure you have the session you want loaded - in case you have the same data in two differently named files.

Default behaviour

Prior to October 2025, the default behaviour was **Display the Session Already Loaded** dialog.

From October 2025, the default behaviour is **Unload the loaded session, then load it from disk**.

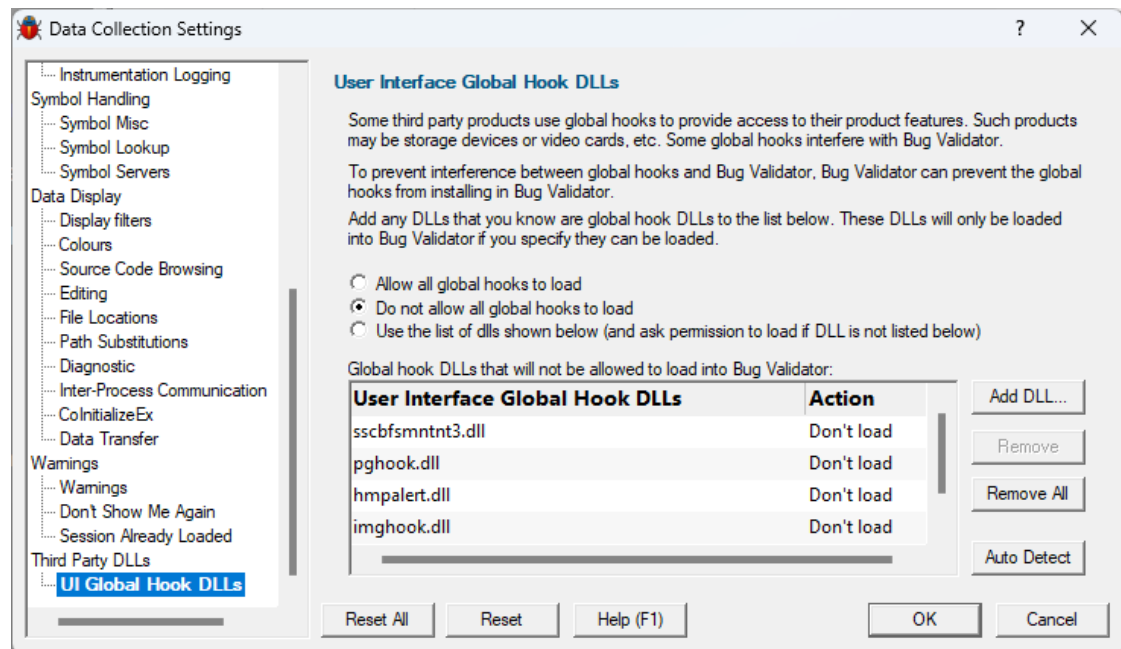
Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.1.7 Third Party DLLs

3.9.1.7.1 UI Global Hook DLLs

The **User Interface Global Hook DLLs** tab allows you to detect and specify global hook DLLs that may not be wanted in the Bug Validator user interface process



About global hook DLLS

Some third party products such as storage devices and video cards are supplied with software to help integrate the hardware device into the computer desktop environment.

An example is the Iomega® Zip® drive. This uses a global hook  via the IMGHOOK.DLL which allows the *browse for files* and *browse for folders* interfaces to correctly display all the storage devices on the computer, including the zip drive and any special options for the drive.

Some global (or *system*) hook DLLs can interfere with the correct operation of Bug Validator when it inserts hooks into the target program, (although the IMGHOOK.DLL mentioned above doesn't).

The settings below allow you to specify and/or detect DLLs that should be treated as global hook  DLLs.

Any DLL listed will fail to load into the target program when loaded via `LoadLibrary()` or `LoadLibraryEx()`.

For situations where the hook DLL is already present in the target program, it can optionally be forcibly unloaded. This may happen if it was loaded before Bug Validator attached to the process.

The user interface hook DLL loading rule

The default behaviour is not to allow the global hooks to load, but you can change this if necessary

- **Allow all global hooks to load** > allows all global hook DLLs to load into Bug Validator
- **Do not allow any global hooks to load** > prevent any global hook DLLs from loading (the default)
- **Use the list of DLLs shown** > provide per-DLL control over which DLLs load or don't load via the **User Interface Global Hook DLLs** list

Any global hook DLLs not listed will result in the user being asked for permission to load a DLL via the Global Hook Warning Dialog below

Managing user interface global hook DLLs

- **Add DLL...** > browse and select one or more DLLs > **Open** > adds the chosen DLLs to the **Global Hook DLLs** list

Having added a DLL to the list, you can change whether the DLL is allowed to load or not, by double clicking in the second column and changing the value: **Load** or **Don't load**

- **Remove** > removes any selected DLL from the list
- **Remove All** > removes all DLLs from the list

Auto detecting global hook DLLs

Bug Validator can detect any DLLs in its own process that are not ones it uses itself. Such DLLs are likely to be global hook DLLs:

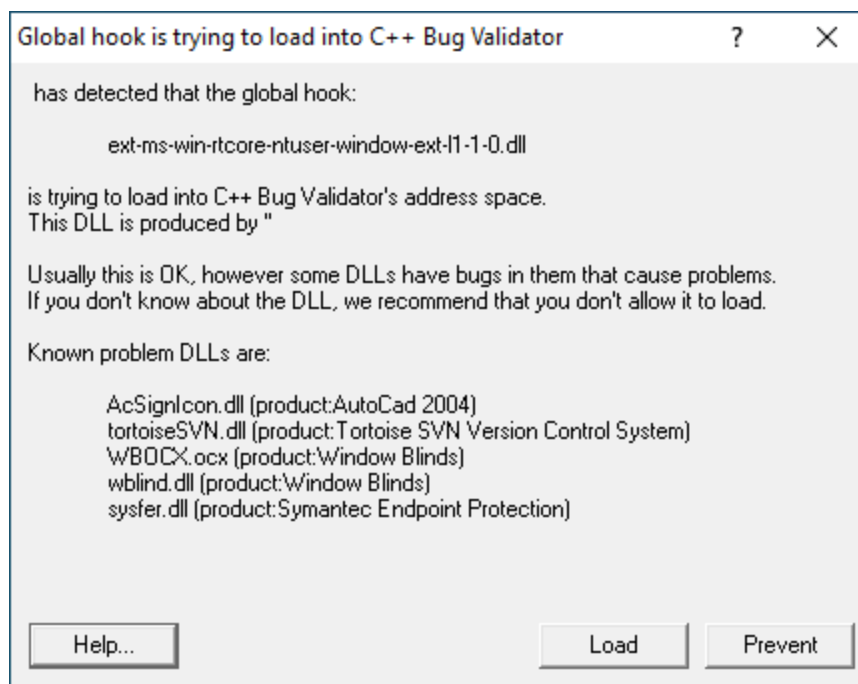
- **Auto Detect** > automatically detect DLLs which may be global hook DLLs, *adding* them to the **Global Hook DLLs** list

Global Hook Warning Dialog

When the global hook loading rule above is set to **Use the list of DLLs shown**, the **Allow load** column controls whether the hook DLL is loaded.

When a global hook is loaded that is *not* on the list of known global hooks, the user is presented with a warning dialog like that shown below.

The user can then accept or block the global hook from loading. The dialog lists a couple of known problematic DLLs.



- **Help** > displays this help page
- **Yes** > lets the DLL load
- **No** > blocks the DLL

Your response is automatically recorded in the **Global Hook DLLs** list, so that you won't be asked again.

Reset All - Resets **all** global settings, not just those on the current page.

Reset - Resets the settings on the current page.

3.9.2 Environment Variables

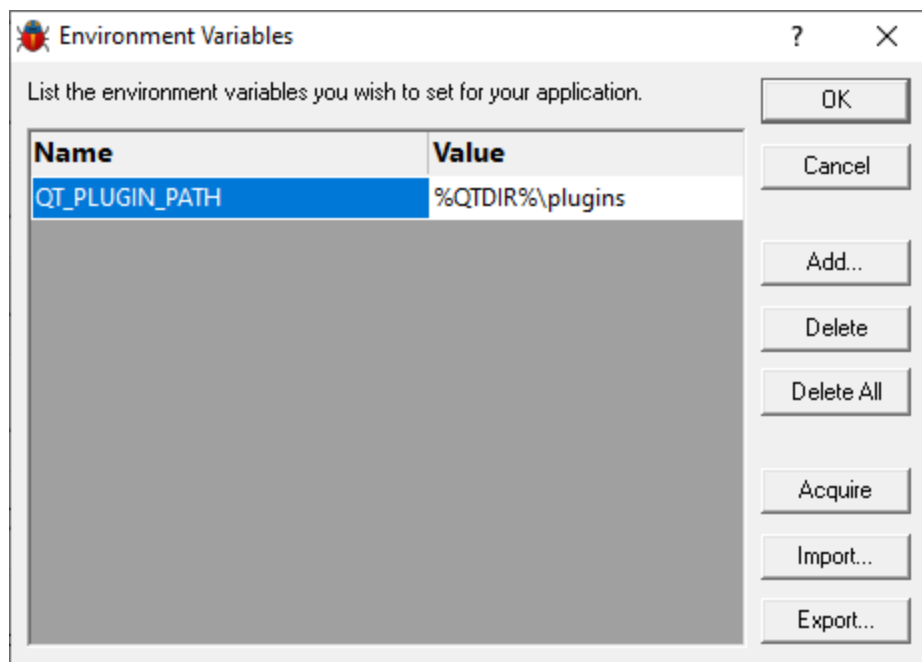
When launching an application, you might want to pass in some environment variables to your program.

The Environment Variables dialog lets you manage name/value pairs, including importing and exporting for use between programs or sessions.

The Environment Variables dialog

The dialog initially has no entries.

The example below shows the equivalent of `set QT_PLUGIN_PATH=%QTDIR%\plugins`



- **Add...** ➤ adds a new item to the list ➤ enter name in the first column, value in the second
- **Delete** ➤ deletes a selected item in the list
- **Delete All** ➤ clears the list
- **Acquire** ➤ fetches all system environment variables, *adding* them to the list
- **Import...** ➤ loads variables from a previously exported file, *adding* them to the list
- **Export...** ➤ saves all entries in the list to a file of your choice

The exported file is a simple ascii file with one entry per line of the form `name=value`

- **OK** ➤ accepts all changes

- **Cancel** ➤ ignores changes

3.9.3 Loading and saving settings

Saving and loading settings files

Bug Validator settings can be saved to a file and restored at any time.

 **Settings menu** ➤ **Save Settings...** ➤ save settings to a file

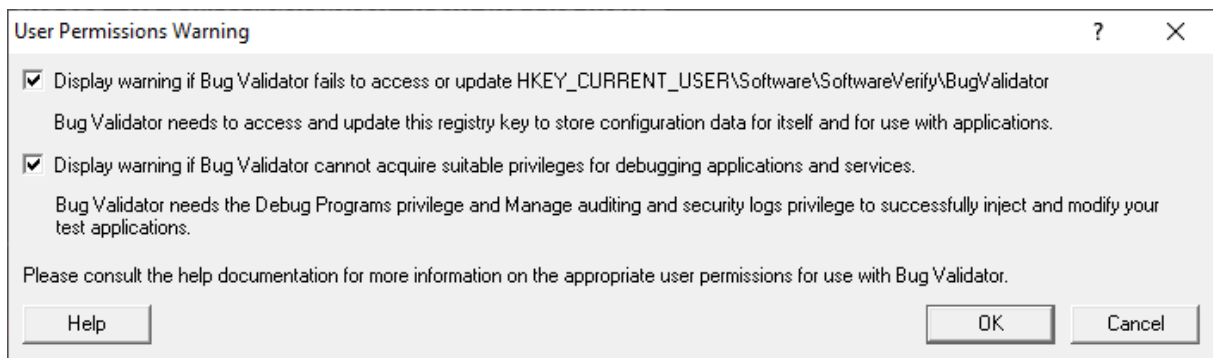
 **Settings menu** ➤ **Load Settings...** ➤ load a previously saved settings file

3.9.4 User Permissions Warnings

Bug Validator displays warning dialogs when errors occur accessing the Registry and/or obtaining debugging privileges. These warnings are enabled by default, but can be enabled or disabled as desired. The User Permissions Warnings dialog is used to enable or disable these warnings.

To display the User Permissions Warnings dialog.

 **Settings menu** ➤ **User Permissions Warning...** ➤ the User Permissions Warning dialog is displayed.

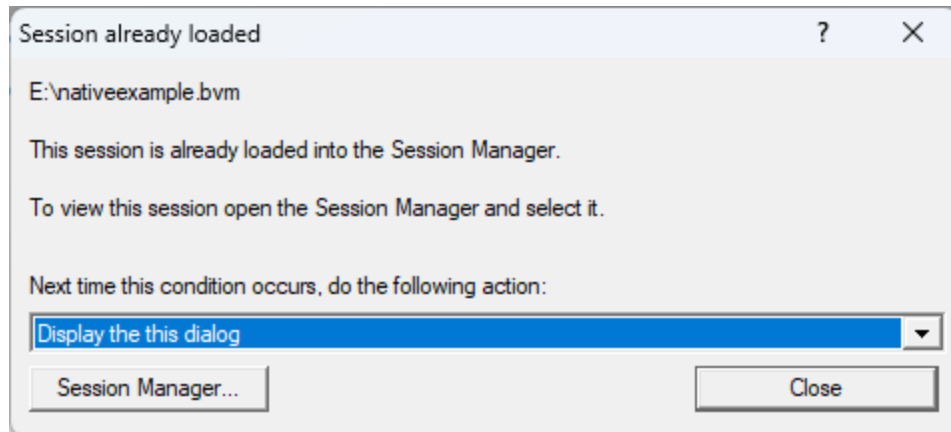


Select or deselect the check boxes appropriate to the warnings you wish to receive and click OK to accept the changes. The Help button displays the User Permissions help topic.

You may also want to read this topic relating to creating Power User accounts for Windows XP.

3.9.5 Session Already Loaded Dialog

The Session Already Loaded dialog is displayed when you try to load a session that is already present in the Session Manager.



Sessions loaded into Bug Validator can be closed by many different actions:

- **File Close** > user manually closing the session
- **Load Session** > user manually loading a session
- **Launch** > launching any application (native, .Net, etc) from the Bug Validator user interface
- **Inject** > injecting into any application from the Bug Validator user interface
- **Wait for Application** > waiting for any application and injecting into it when it launches, from the Bug Validator user interface
- **Command line** > Bug Validator launches an application from the command line
- **Bug Validator started from the API** > any use of the API that starts Bug Validator

When a session is closed in this manner, the session remains loaded into the Session Manager, unless it is purged based on the Session Manager settings.

This allows sessions to be kept in memory so that they can be inspected later.

Because of this arrangement, it is possible that you may try to load a session that is already loaded into the session manager.

When this happens, you have a choice of actions, allowing you to customize Bug Validator's behaviour.

- **Display the Session Already Loaded dialog** > display the Session Already Loaded dialog. This is equivalent to an improved version of the experience prior to October 2025.
- **Display the Session Manager** > display the Session Manager.
- **Display the loaded session** > display the session that is already in memory without loading it from disk.
- **Unload the loaded session, then load it from disk** > this option unloads the session found in memory, the loads the session from disk.
This option is provided to allow you to be sure you have the session you want loaded - in case you have the same data in two differently named files.

Default behaviour

Prior to October 2025, the default behaviour was **Display the Session Already Loaded dialog**.

From October 2025, the default behaviour is **Unload the loaded session, then load it from disk**.

- **Session Manager...** ➤ display the Session Manager

3.10 Managers

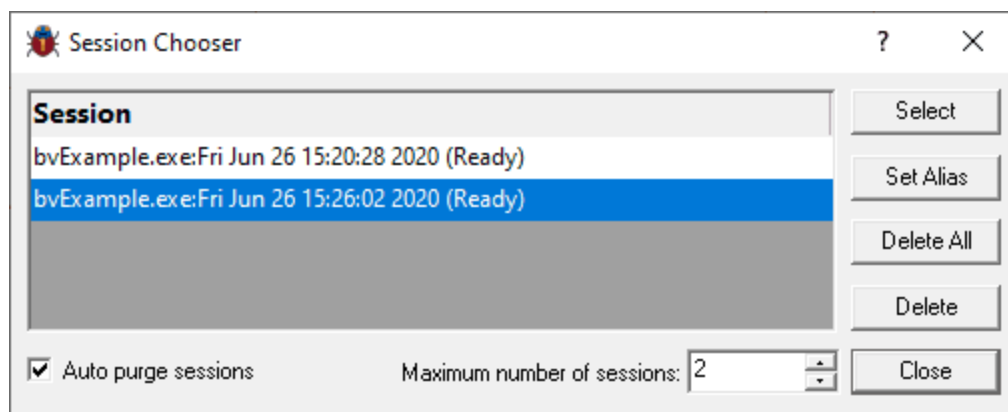
3.10.1 Session Manager

Managing multiple sessions

Bug Validator can manage multiple sessions at once.

As well as the actively running session, open sessions may include those run since Bug Validator started, or reloaded sessions that had been saved earlier.

 **Managers** menu ➤ **Session Manager...** ➤ shows the **Session Chooser** dialog below, highlighting the current session



Each time a session is started or loaded it is added to this list, using the name of the executable program and the date and time the session started.

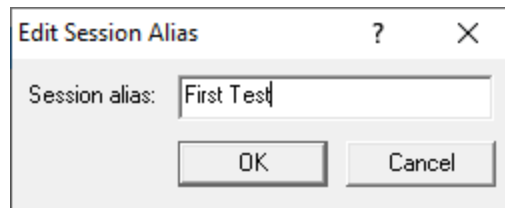
Managing the sessions

- **Select** ➤ makes the selected entry the *current* session, i.e. the one for which data will be displayed



Some tab views may update immediately, others may need a manual refresh

- **Set Alias...** ➤ opens the **Edit Session Alias** dialog so you can give the session a more useful name



- **Delete** ➤ removes the selected session

You can't delete a session that is actively collecting data.

- **Delete All** ➤ removes all the loaded sessions

If one of the session is actively collecting data, this will be disabled.

- **Close** ➤ closes the dialog (as opposed to closing any selected sessions!)

Limiting the number of sessions

You can choose to limit the maximum number of sessions open at once. Once the maximum is reached, then each time a new session is added, the oldest session may automatically be removed:

- **Auto purge sessions** ➤ ensures that the number of loaded sessions is limited to the maximum (below)
- **Maximum number of sessions** ➤ sets the maximum number of sessions allowed if auto-purge is on

3.11 Deploy

Bug Validator allows you to generate KVI files that can be used at a customer site with the Bug Validator Client software tool.

KVI files allow Bug Validator Client to monitor your application's execution history at customer sites without the need to supply PDB and MAP files to the customer site.

Additionally, Bug Validator Client can be distributed to your customers at no cost, whereas Bug Validator cannot be distributed to your customers.

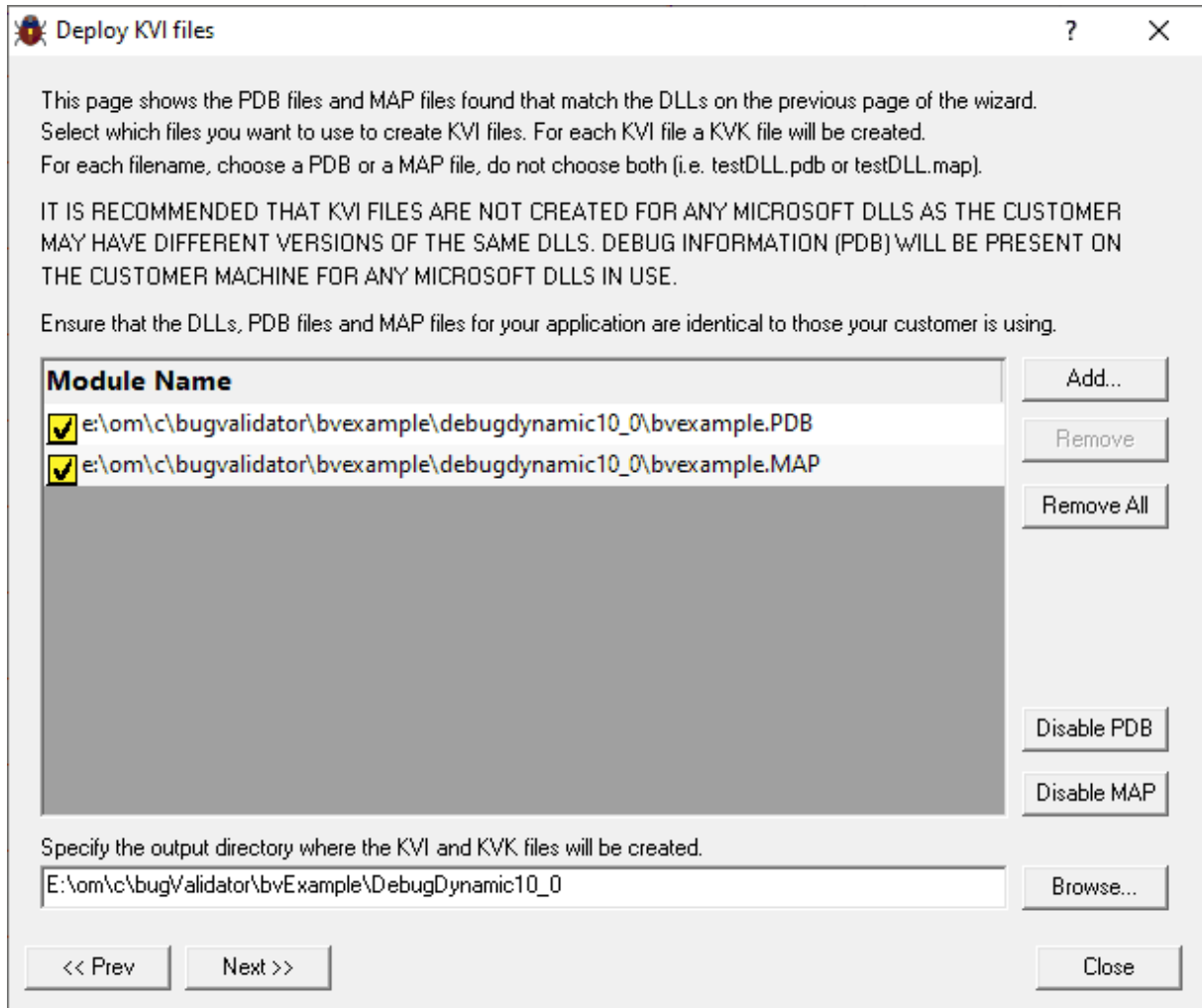
To start the process of generating KVI files

 **Deploy** menu ➤ **Create KVI files...** ➤ shows the Deploy KVI files wizard



Select the application you want to deploy, and then click the **Detect Dependent DLLs** button. Bug Validator detects the DLLs that will be used by the application, and displays the DLLs in the scrolled list. You can add DLLs to the list or remove DLLs from the list using the **Add**, **Remove** and **Remove All** buttons. You may want to add DLLs that are loaded via LoadLibrary(), and you may wish to remove DLLs that are 3rd party DLLs.

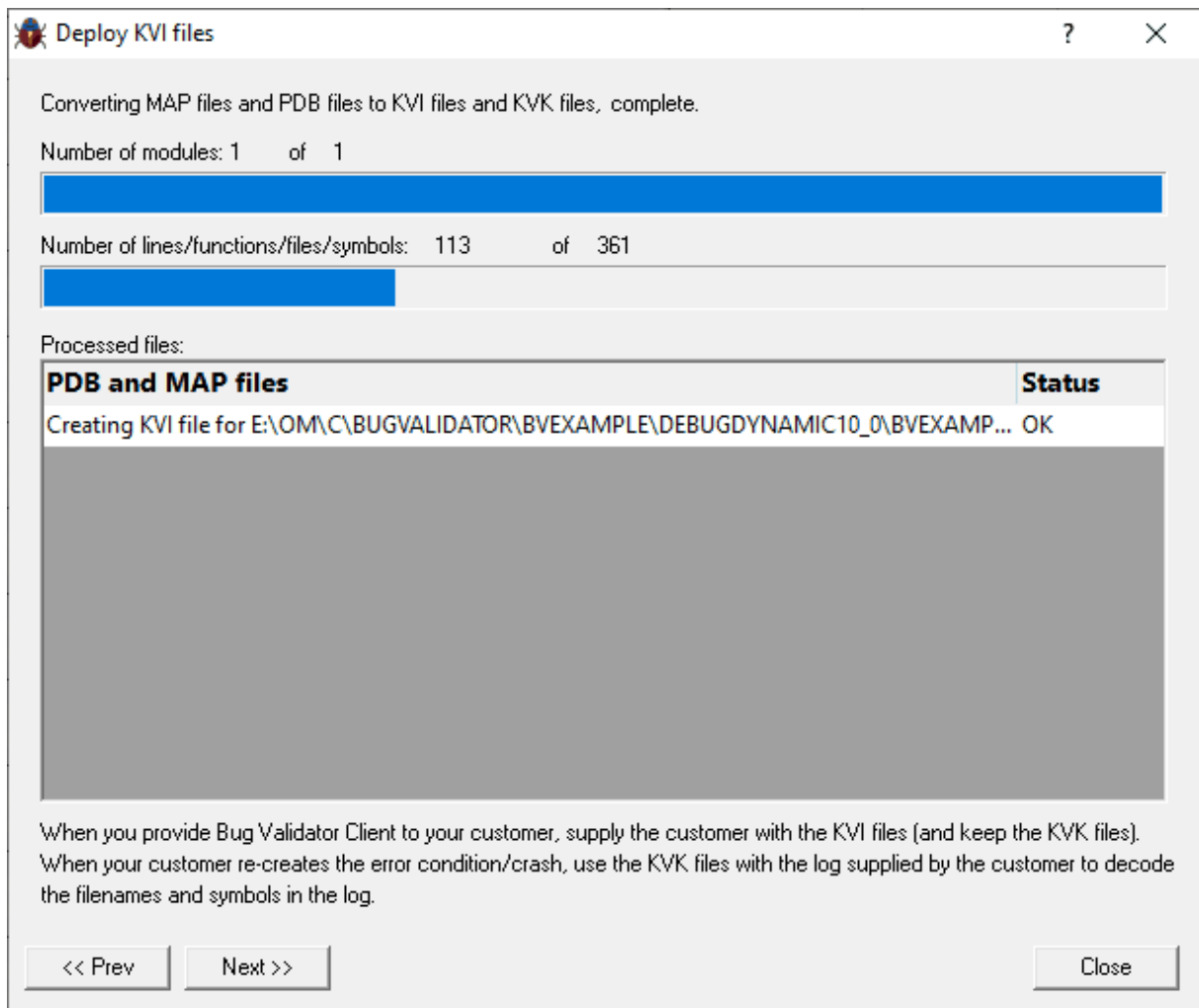
Click the **Next >>** button to move to the next page of the wizard.



The next page of the wizard displays all the PDB and MAP files that could be found for the files specified on the previous page of the wizard. Some DLLs will have both a PDB and MAP file specified. When this happens you must choose which file to use. Microsoft DLLs are always displayed with both the PDB and MAP files disabled. This is because the customer machine may have different versions of these DLLs are KVI files should not be generated when that is the case. To add or remove files from the list use the **Add**, **Remove** and **Remove All** Buttons. The **Disable PDB** button disables all PDB entries in the list. The **Disable MAP** button disables all MAP entries in the list.

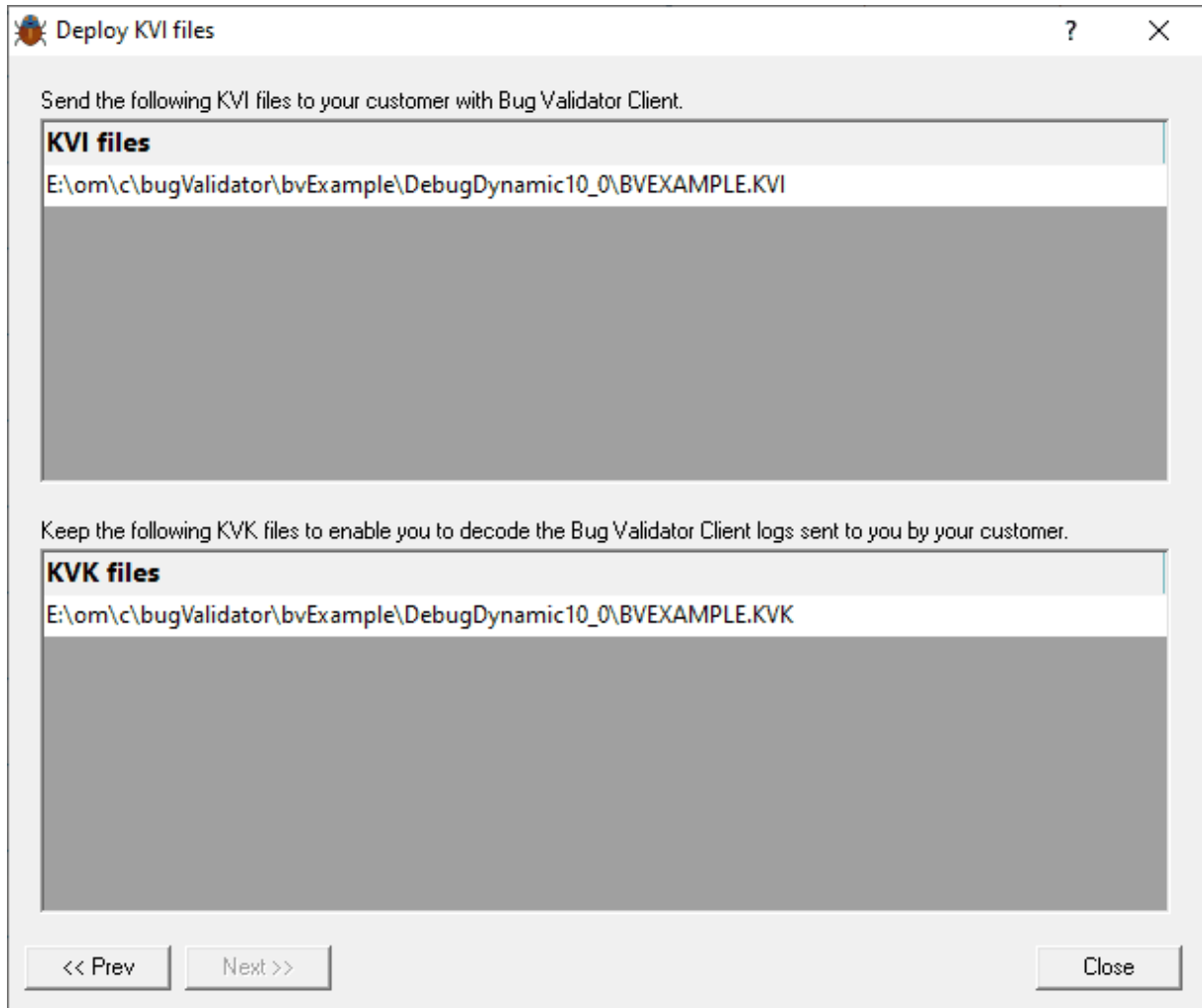
Specify where the generated KVI and KVK files should be saved.

Click the **Next >>** button to move to the next page of the wizard.



The next page of the wizard generates the KVI and KVK files. The progress of the KVI file generation is shown on the progress bars, with the status of each individual file shown in the list. MAP files may fail because they do not contain line number information. PDB files may fail because they refer to a different version of the DLL than the DLL that is being processed.

Click the **Next >>** button to move to the next page of the wizard.



The final page of the wizard shows the generated files. The KVI files should be sent to the customer with instructions about downloading Bug Validator. The KVK files should not be sent to the customer. The KVK files should be kept somewhere safe so that they can be used to decode the logs sent to you by the customer.

Currently Bug Validator Client is out-of-date compared to Bug Validator. Until we have updated Bug Validator Client you can use Bug Validator at the client site with KVI files, although this is not ideal as it does expose the symbolic information to the customer. That risk is your decision to take - it is your responsibility to decide if this is a risk you wish to take.

3.12 Tools

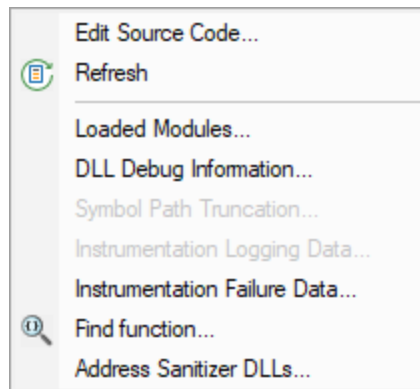
Tools

The Tools menu provides access to a few different tools including a couple not found on the Tools toolbar:

- A list of the modules loaded by your target application
- A list of the debug information status of modules loaded by your application
- A log of files, classes, functions, methods, or modules not instrumented, and reasons why not



Click on a menu item in the picture of the Tools Menu below to find out more:



3.12.1 Edit Source Code...

Source code editing

The editing settings let you set an editor of your choice to view or edit source code. Bug Validator's built-in editor is one of those options.

The built-in editor can be started in several ways:

-  Double click on a source code fragment (e.g. in the Execution History)
-  popup menu > **Edit Source Code...**
-  **Tools** menu > **Edit Source Code...**

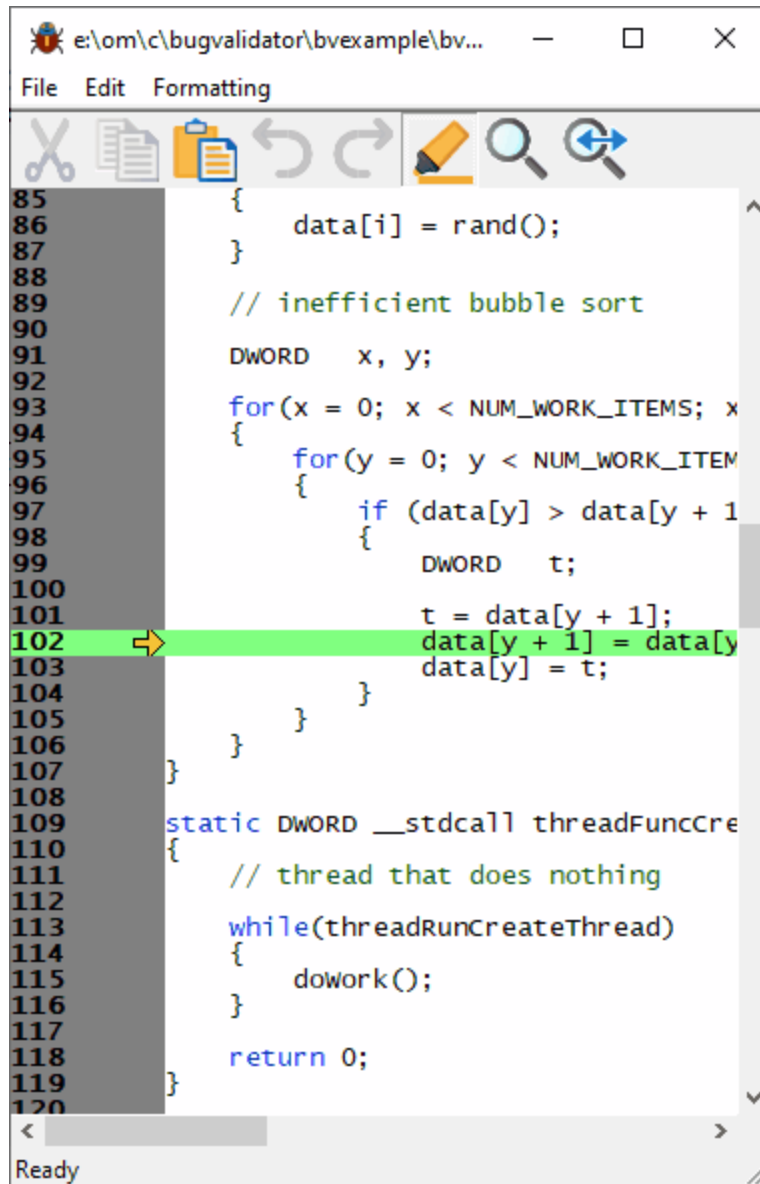
Using the built-in editor

The built-in editor supports the basic operations expected for editing source code:

The highlighting is identical to that in the source code views of the main tabs:

- The lines in green (for this colour scheme) have been visited

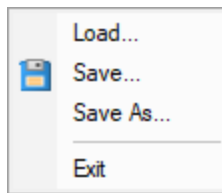
- Lines that have not been *visited* are displayed in pink
- Lines with a green tick next to them indicate that they have been successfully hooked
- Lines that could not be *hooked* have a red cross against them
- An arrow indicates the source code line of interest when the source code editor was displayed



```
85 {
86     data[i] = rand();
87 }
88
89 // inefficient bubble sort
90
91 DWORD    x, y;
92
93 for(x = 0; x < NUM_WORK_ITEMS; x
94 {
95     for(y = 0; y < NUM_WORK_ITEM
96     {
97         if (data[y] > data[y + 1
98         {
99             DWORD    t;
100
101             t = data[y + 1];
102             data[y + 1] = data[y]
103             data[y] = t;
104         }
105     }
106 }
107
108
109 static DWORD __stdcall threadFuncCre
110 {
111     // thread that does nothing
112
113     while(threadRunCreateThread)
114     {
115         dowork();
116     }
117
118     return 0;
119 }
120
```

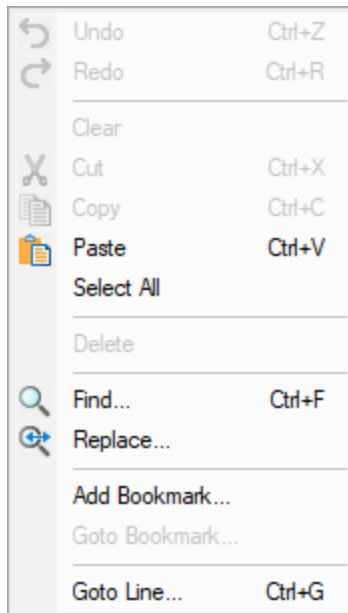
File menu

The file options need no explanation:



Edit menu

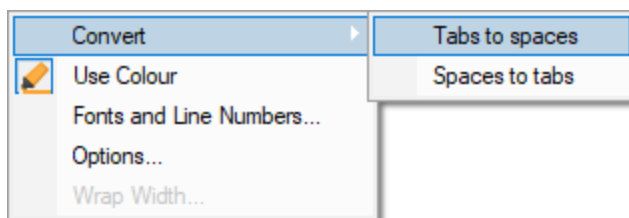
All the following edit options should also be familiar:



Undo/Redo is unlimited by default, but this can be changed in the options below.

Formatting menu

The formatting menu has general display and editing options



- **Convert > Tabs to spaces >** turns all tabs into spaces
- **Convert > Spaces to tabs >** turns all spaces into tabs
- **Use Colour >** toggles the colour coded display
- **Fonts and Line Numbers... >** change text colours, fonts and line numbers

Options Colour Dialog [X]

Default Text Colours		Line Number Colours	
Text Colour	black [v] [c]	Text Colour	black [v] [c]
Background Colour	white [v] [c]	Background Colour	gray [v] [c]
Select Text Colour	white [v] [c]	☒ Display Line Numbers	☐ Display Background Image
Select Background Colour	black [v] [c]		
Default Text Font		Line Number Font	
Font	Lucida Console [v]	Font	Lucida Console [v]
Height	[v]		
☐ Bold ☐ Italic ☐ Strikeout ☐ Underline		☒ Bold ☐ Italic ☐ Strikeout ☐ Underline	
[OK]		[Cancel]	

- **Options...** ➤ set tab length and other options

Edit Settings [X]

☒ Display Left Symbols	☒ Allow Line Collapsing
☒ Display Right Symbols	Line Collapse Colour silver [v] [c]
☒ Display Tooltips	☐ Allow Quoted Text Colouring
☒ Allow vertical scrollbar	Quoted Text Colour teal [v] [c]
☒ Allow horizontal scrollbar	☐ Allow Punctuation Colouring
☒ Allow Drag Open	Punctuation Colour olive [v] [c]
☒ Allow Drag Close	
☒ Allow Line Highlighting	
Character Handling	Undo / Redo
Tab length 4 [v]	Maximum number of undo operations [v]
☒ Add tabs as spaces	☒ Store movement undos ☒ All
☒ Replace tabs with spaces when deleting	☒ Store editing undos
☒ Translate \r to \n	
☒ Wrap cursor at end of column	
[OK] [Cancel]	

- **Wrap Width...** ➤ changes the column width at which lines will wrap in the display

Status bar

The status bar shows help text at the bottom as you hover over menu and toolbar options.

To the right of the status bar are insert mode, column number and line number.

Line collapsing

You can temporarily collapse sections of code as follows:

-  **Left click** in the margin to start the section ➤ **Drag** to define the length ➤ **Release** to set the end of the section

Click anywhere on the resulting indicator to collapse, and on the + to expand a section.

Expanded:

```
799 }
800
801 void CTeststakApp::doMemoryLeak4(DWORD dummyParam)
802 {
803     #if DO_MEMORY_LEAK
804         char *ptr;
805
806         ptr = MY_NEW char [456];
807     #endif
808 }
809
```

Collapsed:

```
799 }
800
801 + void CTeststakApp::doMemoryLeak4(DWORD dummyParam){#if DO_MEMORY_LEAK
809
810 void CTeststakApp::doMemoryLeak1(DWORD dummyParam)
811 {..
```

 Line collapsing is *temporary* and not remembered between edit sessions.

3.12.2 Refresh

Refreshing data

You have the option in some views to automatically update the view at an interval of your choice.

Sometimes you need to refresh the data when *you* want to, especially while inspecting the data.

Most views have a local refresh button, which updates the data.

The same function is found in the Tools menu, as well as an option to update all views at once:

 **Tools** menu > **Refresh** > refresh the data displayed only on the current tabbed view

Or use the **Refresh** icon on the Tools toolbar.



3.12.3 Loaded Modules

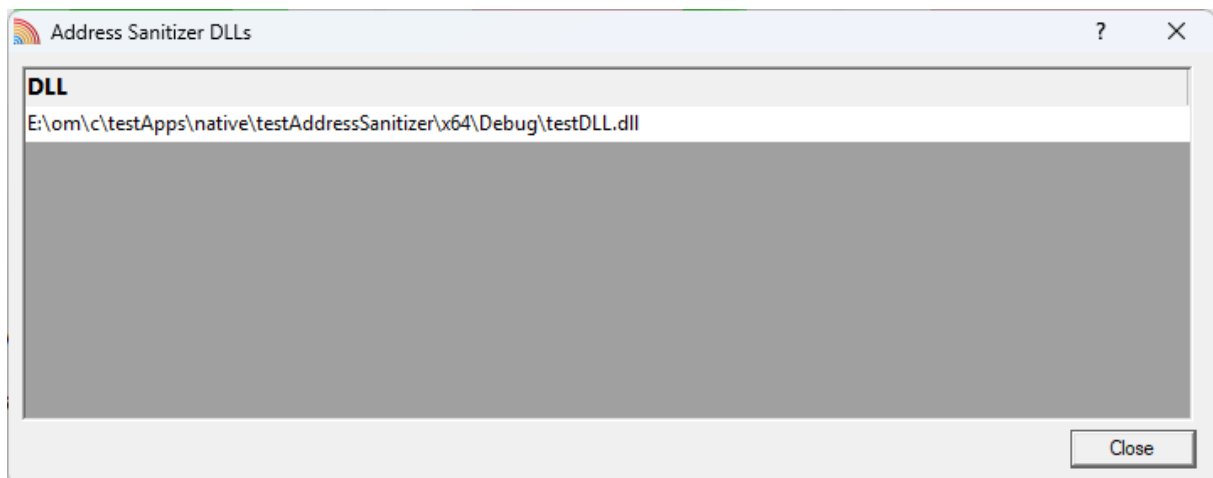
Viewing the loaded modules

You can view a list of the modules which are loaded by your target application.

 **Tools** menu > **Loaded Modules...** > shows the **Loaded Modules** dialog

The dialog shows:

- the **Address** space occupied by the module (DLL or EXE)
- the type of **Code** in the module (native, managed, mixed mode or resources only)
- the type of **Build** - is the code debug or release?
- the **Path** the module was loaded from



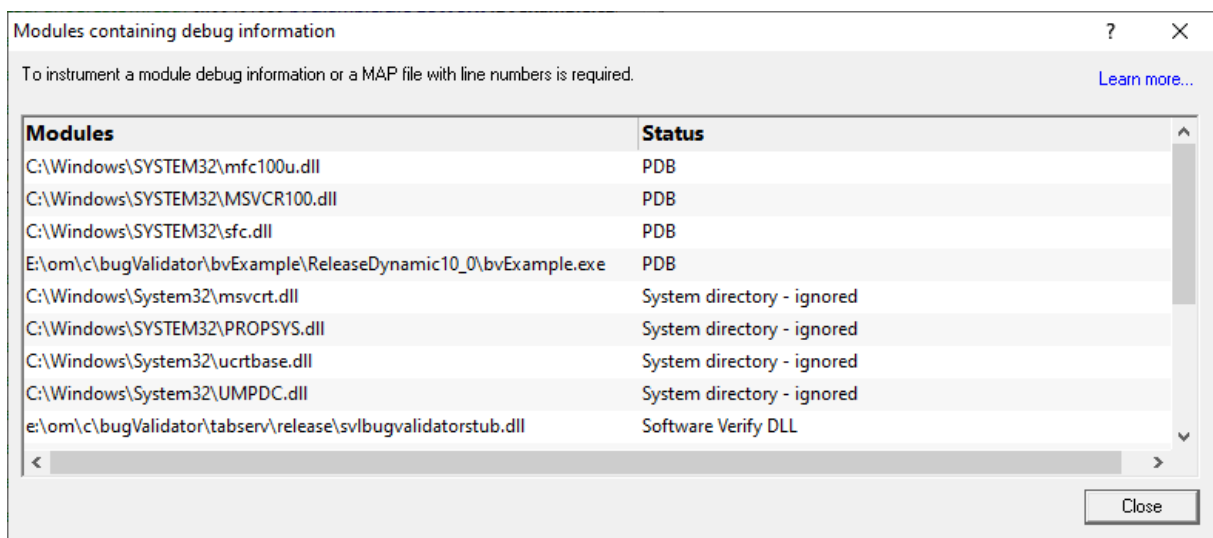
3.12.4 DLL Debug Information

Viewing the DLL debug information

If you are having problems collecting coverage data for a particular EXE/DLL the problem may be that the debug information that is required to perform the instrumentation of the software cannot be found.

You can view a list of the debug information status of modules loaded by your target application.

 **Tools** menu > **DLL Debug Information...** > shows the DLL Debug Information dialog below



The dialog shows:

- the path from which **Modules** (DLL or EXE) were loaded
- the debug **Status** (below)
- if any symbol server is not reachable (offline or doesn't exist) a message will be shown in red at the bottom of the dialog. You can edit the symbol server definitions here.

Debug status

There are various reasons why a module may not have its debug information read.

The dialog shows a comment or reason in the status column. Examples might be:

- PDB or MAP if the debug information was found and used

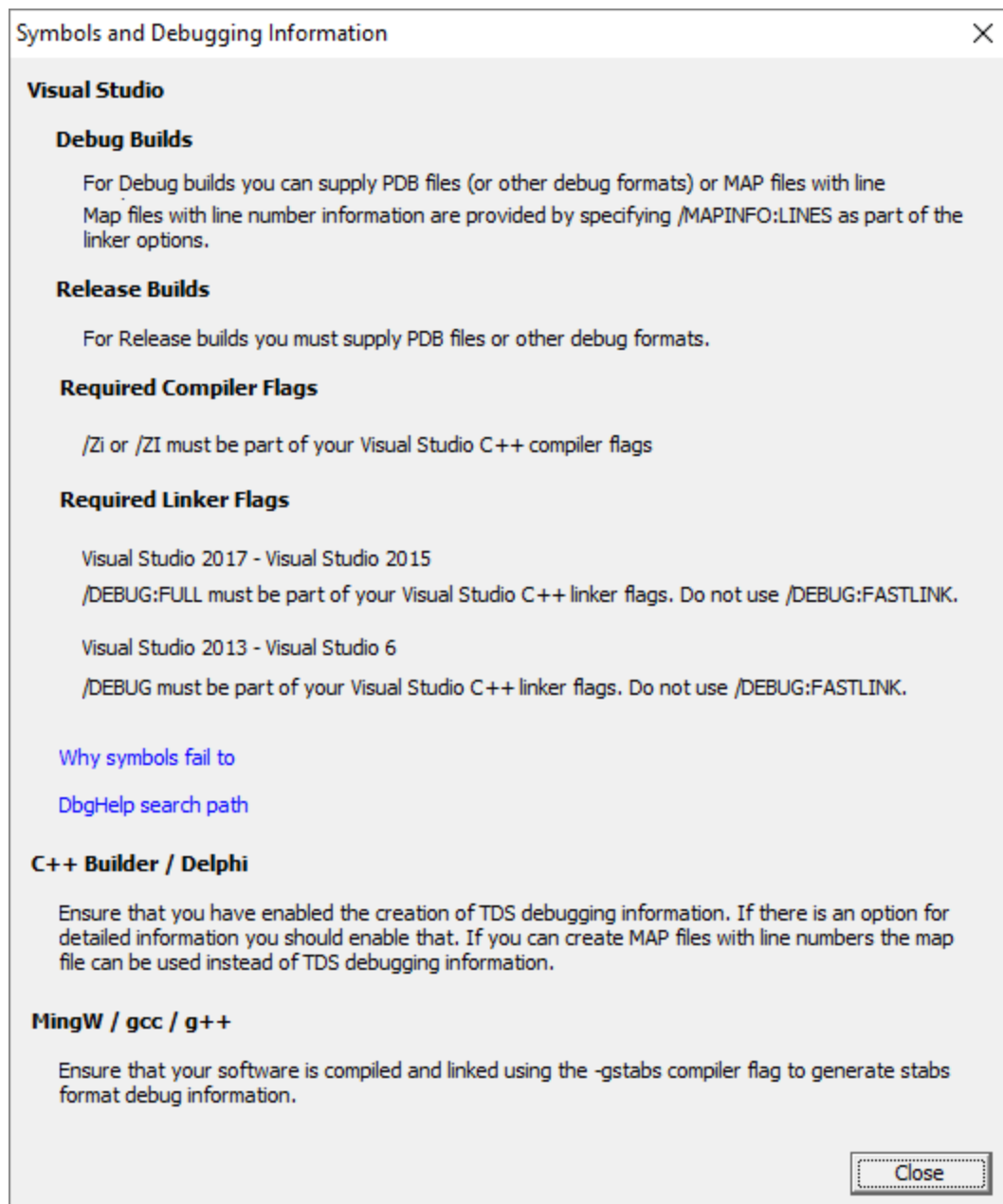
- Debug information not present
- A reason for being ignored
- Module is a part of the C Run-time Library (CRT) or Standard Template Library (STL)
- Location is a system directory
- Ignored due to Hooked DLLs advanced settings
- File is a Software Verify own module
- Module has been specified as a 3rd party
- No executable code is contained
- The module only has GUI resources

More information about PDB and MAP files

Clicking on the **Learn more...** link at the top right of the dialog shows some more details with additional links to topics in this help.



Click the links below to see read more in our frequently asked questions.

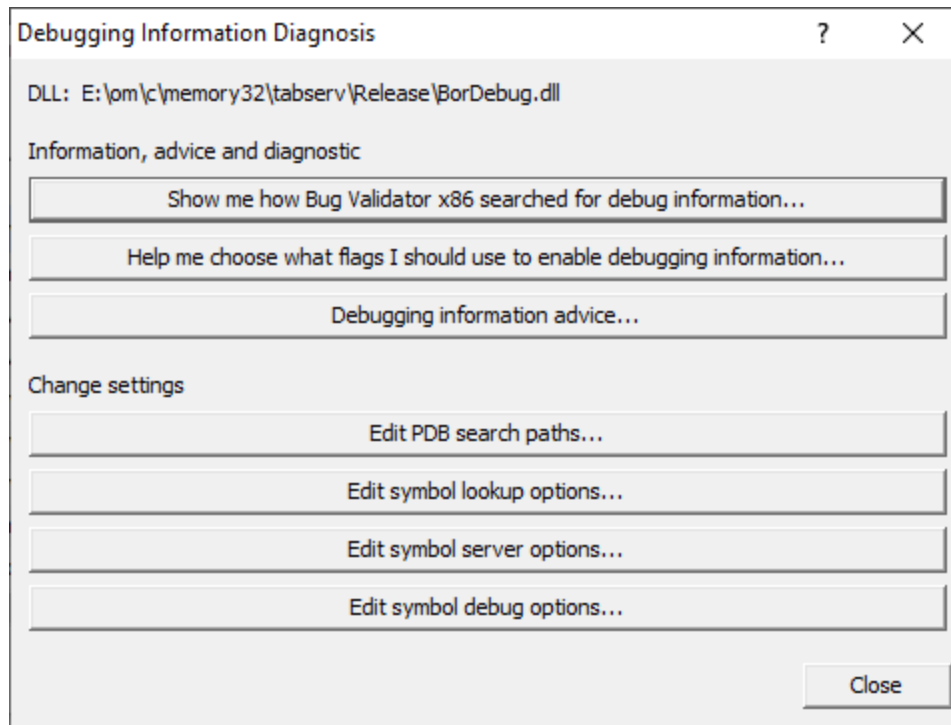


Finding out more using the Debugging Information Diagnosis Dialog

When debug information is not present for a given module the DLL Debug Information dialog (above) may display a button in the Status column to show the Debugging Information Diagnosis dialog.

The dialog shows:

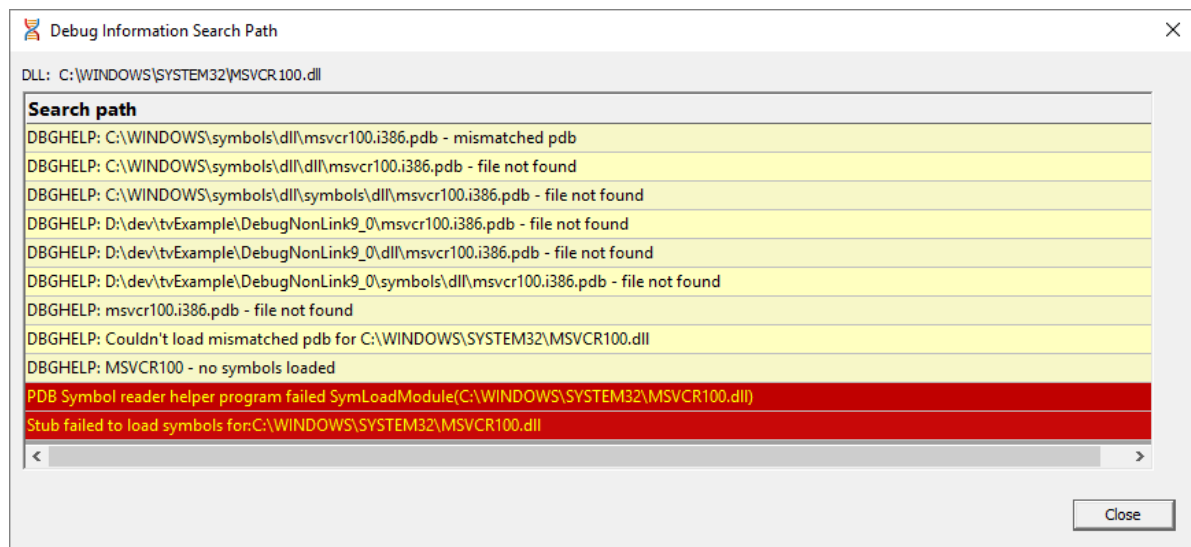
- Information, advice, and diagnostic help
- Quick links to change settings



The information options include:

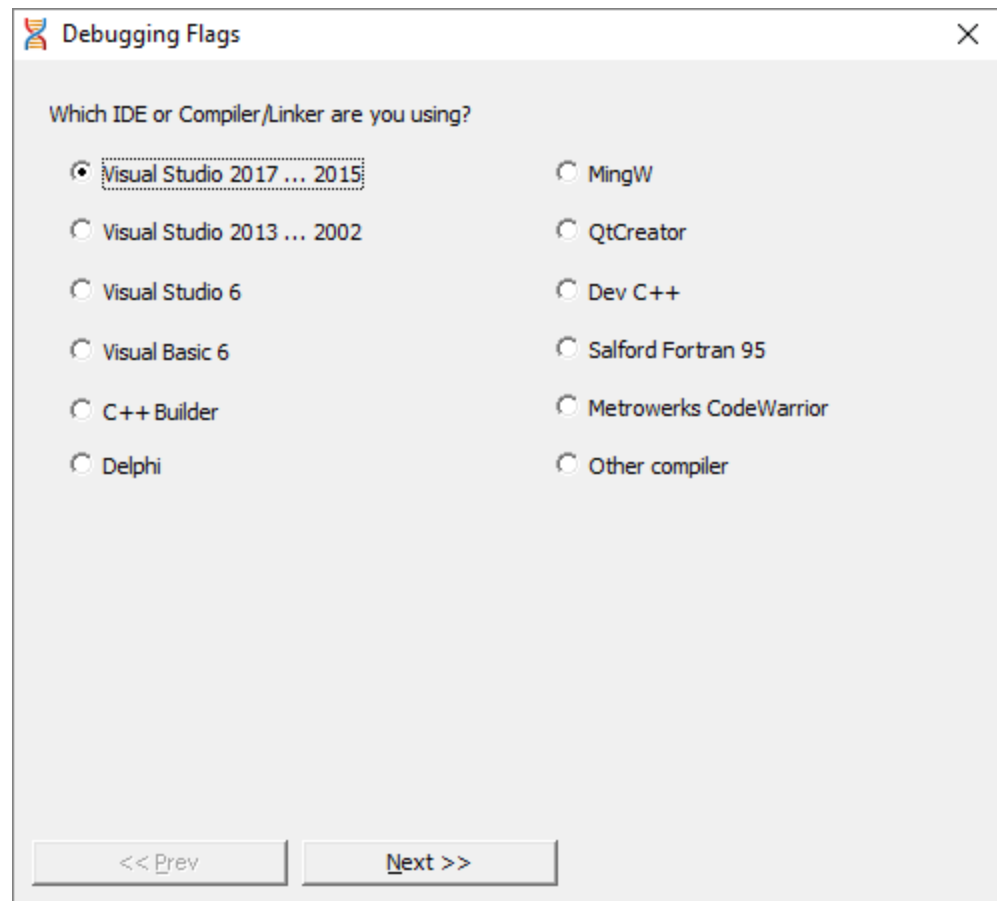
- **Show me how debug information was searched for...** ➤ shows the Debug Information Search Path dialog

This information is extracted from the Diagnostic tab and shows only the relevant information for the module selected in the DLL Debug Information dialog.

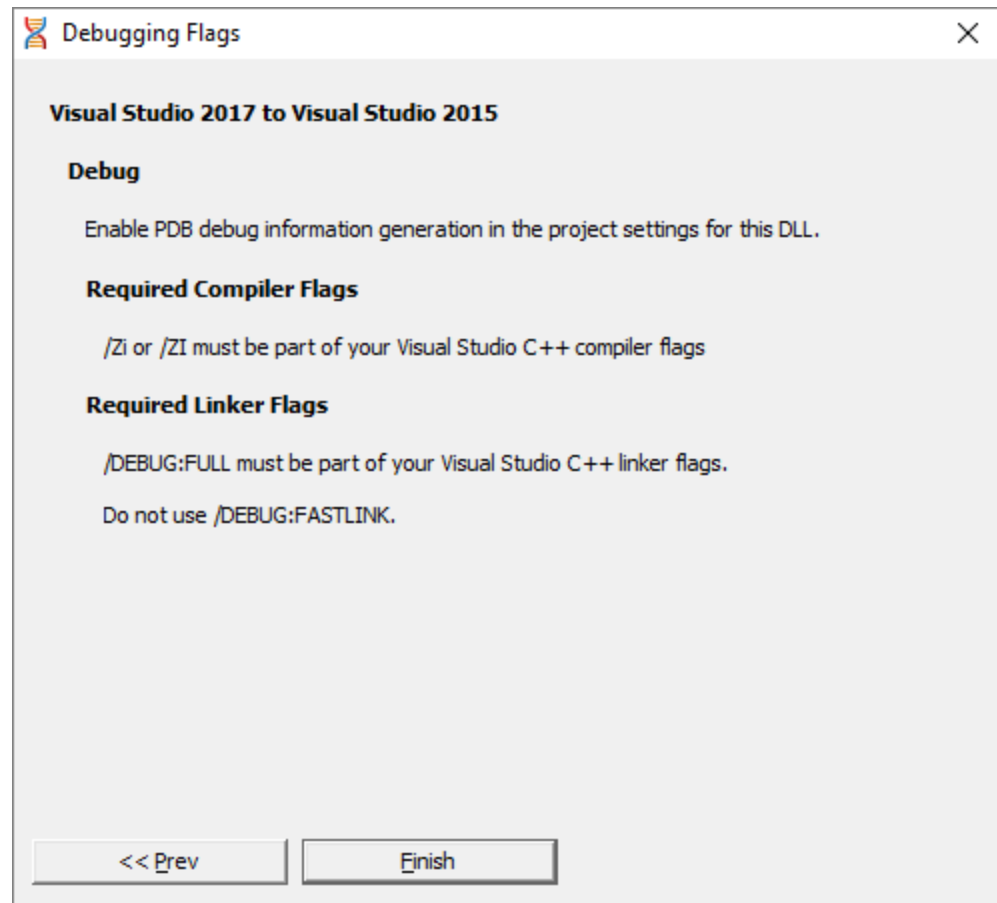


- **Help me choose what flags...** ➤ shows the Debugging Flags wizard

Use the wizard to first select the compiler or linker you're using



Next >> ➤ Provides the relevant debug compiler and linker flags. An example for Visual Studio 2017 to 2015 is below:



- **Debugging information advice...** ➤ shows the Symbols and Debugging Information dialog above.

The options for changing settings include quick links to the following pages from the Settings Dialog

- **Edit PDB search paths...** ➤ shows the File Location settings page for PDB files.
- **Edit symbol lookup options...** ➤ shows the Symbol Lookup settings page
- **Edit symbol server options...** ➤ shows the Symbol Servers settings page
- **Edit symbol debug options...** ➤ shows the Symbol Misc settings page

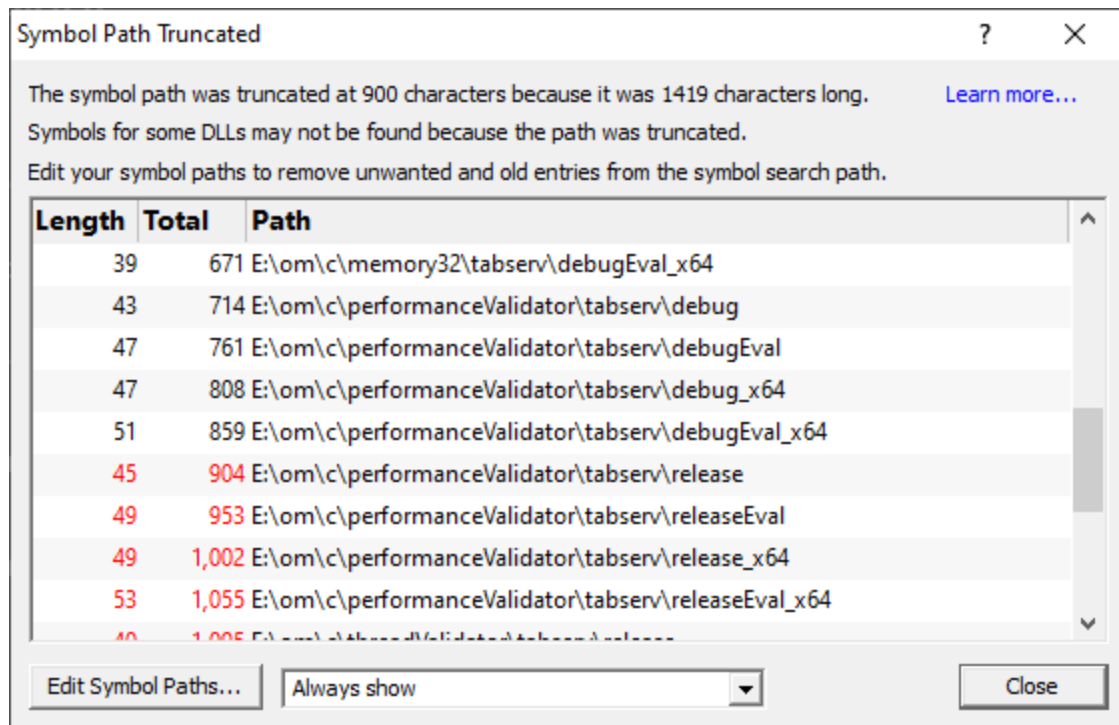
3.12.5 Symbol Path Truncation

The Symbol Path Truncated warning dialog

The symbol path truncated warning dialog is displayed to warn you when the symbol path is too long.

You can also display this dialog from the Tools menu.

 **Tools** menu > **Symbol Path Trunction...** > shows the symbol path truncation dialog below



- **Edit Symbol Paths...** > shows the file locations dialog so that you can edit the paths used for subsequent runs of the program.

You can choose when this dialog is displayed.

- **Always show** > The dialog is always shown when the symbol path is too long.
- **Show when path changes** > The dialog is shown when the symbol path is too long, but only if the symbol path is different than last time this warning was shown.
- **Never show** > The dialog is never shown.

Whether this dialog is displayed or not there is always a warning message written to the diagnostic window when the symbol path is truncated.

The display lists each path with it's length (including the unshown ';' path separator) and the total length so far so that you can see which paths exceed the truncation point (length and total displayed in red).

Any paths that don't exist on this computer are displayed in red.

Why is this dialog displayed?

You may see a Symbol Path Truncated warning dialog in some rare circumstances.

This dialog is displayed when the symbol path that has been calculated to pass to DbgHelp.dll to load Microsoft debugging symbols (found in .PDB files) is too long.

If the symbol path has been truncated because it is too long it is possible this may mean that some symbol searches will fail, resulting in failure to load some symbols. We display this dialog so that you are aware that the symbol path is too long and would benefit from editing to make the symbol path shorter.

Passing a symbol path that is too long to DbgHelp.dll will cause the program being tested to end with an `EXCEPTION_INVALID_CRUNTIME_PARAMETER` C runtime error. This happens because internally DbgHelp.dll is using a fixed length array to format a string. To prevent this fatal termination of the program we limit the length of the path passed to DbgHelp.dll.

Typically if a path that is long enough to cause this problem is passed to DbgHelp it's because the number of paths in the calculated path contain paths not relevant to finding symbols for the test program. We use the Symbol Path Truncated warning dialog to show you the calculated paths so that you can work out which paths to delete.

The calculated symbol paths come from several places:

- File locations PDB paths
- Symbol server symbol storage directories
- Symbol handling environment variables

Fixing the symbol path

For this example, we are testing the program `E:\om\c\3RD_SRC\cdplayer\Release\cdplayer.exe`

In the image shown above you can see that seven paths exceed the truncation limit, one of the 7 paths doesn't exist.

To work out what to do we need to do several actions:

1. Looking at the environment variable settings shows that none of the environment variables are being used. We do not need to consider the content of these environment variables.
2. Examining the symbol servers shows that `C:\Users\Admin` is a local symbol storage location. We should keep this path.
3. We should delete the path that doesn't exist: `E:\om\c\testApps\testStdinStdoutRedirectEx\Release`. We do this using the file locations dialog by clicking **Edit Symbol Paths...** then click **Delete invalid**.
4. Examining the paths in the file locations dialog we can identify any paths not relevant to the program we are testing. In this case the following paths are not relevant and can be deleted.

```
E:\om\c\testApps\testStdinStdoutRedirect\Release
E:\om\c\testApps\testAppTheReadsFromStdinAndWritesToStdout\Release
E:\om\c\testApps\testSimpleMemoryLeak\Release
e:\om\c\3rd_src\cppunit-1.12.1\examples\cppunittest\release
```

e:\om\c\3rd_src\cppunit-1.12.1\examples\cppunittest\releasedll

3.12.6 Instrumentation Logging Data

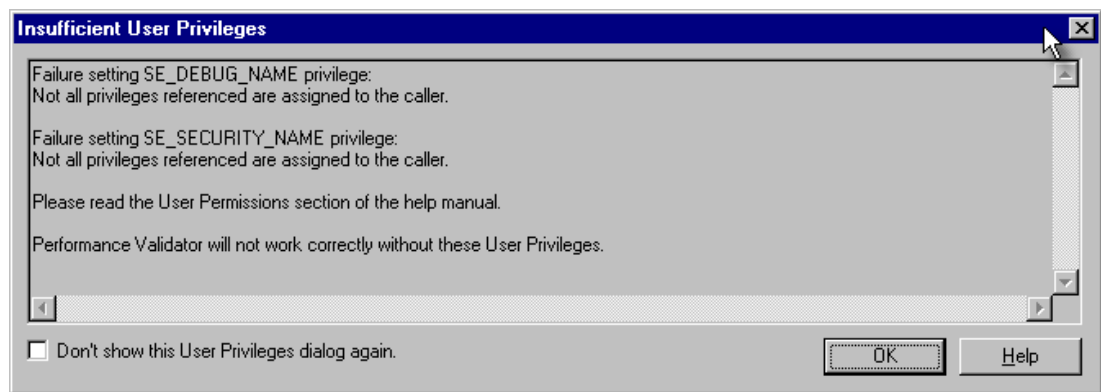
Viewing the instrumentation logging data

If you are having problems instrumenting a particular class or method, you may want to view logging information about which DLLs, classes and methods are instrumented by Bug Validator.

Instrumentation is controlled by filters that affect which DLLs are instrumented and which source files, classes and methods are instrumented.

To enable the collection of instrumentation logging data you need to

 **Tools** menu > **Instrumentation Logging Data...** > shows the Instrumentation logging data dialog below



The dialog shows:

- log order
- the name of the item that hasn't been instrumented
- the reason why each item wasn't instrumented

Example reasons why an item might not be instrumented include the following:

- MFC file ignored
- Microsoft C Runtime file ignored
- Microsoft DLL ignored
- CRT DLL ignored

- Software Verify's own DLL ignored
- File extension excluded by hook source file type filters
- Files excluded by source files filters

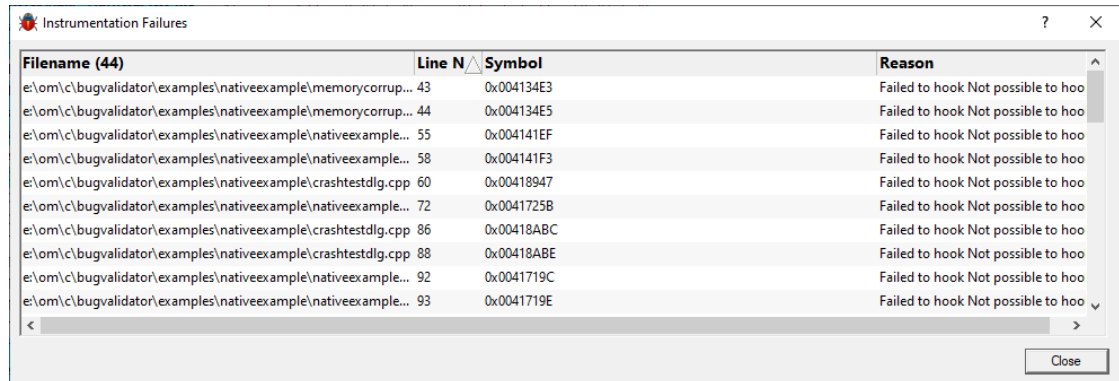
3.12.7 Instrumentation Failure Data

Instrumentation Failure Data

It can be very useful to know which functions in your code failed instrumentation.

You can view a list of the functions that have failed instrumentation via the Tools menu.

 **Tools** menu ➤ **Instrumentation Failure Data...** ➤ shows the Instrumentation Failures dialog



Filename (44)	Line N	Symbol	Reason
e:\om\c\bugvalidator\examples\nativeexample\memorycorrup...	43	0x004134E3	Failed to hook Not possible to hoo
e:\om\c\bugvalidator\examples\nativeexample\memorycorrup...	44	0x004134E5	Failed to hook Not possible to hoo
e:\om\c\bugvalidator\examples\nativeexample\nativeexample...	55	0x004141EF	Failed to hook Not possible to hoo
e:\om\c\bugvalidator\examples\nativeexample\nativeexample...	58	0x004141F3	Failed to hook Not possible to hoo
e:\om\c\bugvalidator\examples\nativeexample\crashtestdlg.cpp	60	0x00418947	Failed to hook Not possible to hoo
e:\om\c\bugvalidator\examples\nativeexample\nativeexample...	72	0x0041725B	Failed to hook Not possible to hoo
e:\om\c\bugvalidator\examples\nativeexample\crashtestdlg.cpp	86	0x00418ABC	Failed to hook Not possible to hoo
e:\om\c\bugvalidator\examples\nativeexample\crashtestdlg.cpp	88	0x00418ABE	Failed to hook Not possible to hoo
e:\om\c\bugvalidator\examples\nativeexample\nativeexample...	92	0x0041719C	Failed to hook Not possible to hoo
e:\om\c\bugvalidator\examples\nativeexample\nativeexample...	93	0x0041719E	Failed to hook Not possible to hoo

The dialog shows:

- the file name of the item that hasn't been instrumented
- the line number of the item that hasn't been instrumented
- the symbol name of the item that hasn't been instrumented
- the reason why each item wasn't instrumented

Example reasons why an item might not be instrumented include the following:

- Disallow computed unconditional jmp
- Failed to disassemble
- Found privileged instruction

3.12.8 Finding functions

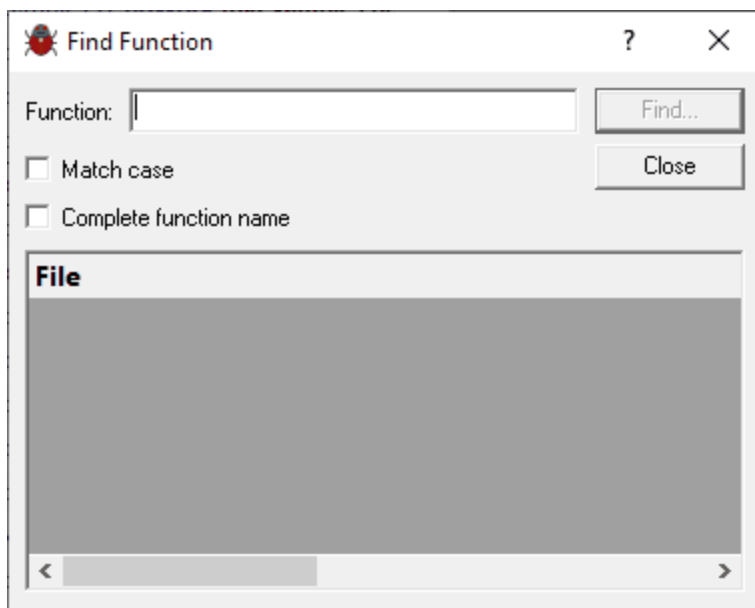
Finding functions

Bug Validator allows you to search for objects using a function name comparison against function names found in callstacks of allocated objects.

To display the find function dialog

 **Tools** menu > **Find function...** > shows the Find function dialog

Or use the **Find function** icon on the Tools toolbar.



Search Criteria

To perform a search, a function name is required.

Function

Type the name of the function to be searched for in the **Function** field.

Match case

If the search should be case sensitive, select the **Match case** check box.

If the search should be case insensitive, do not select the **Match case** check box.

Complete function name

If the search should only match complete function names, select the **Complete function name** check box.

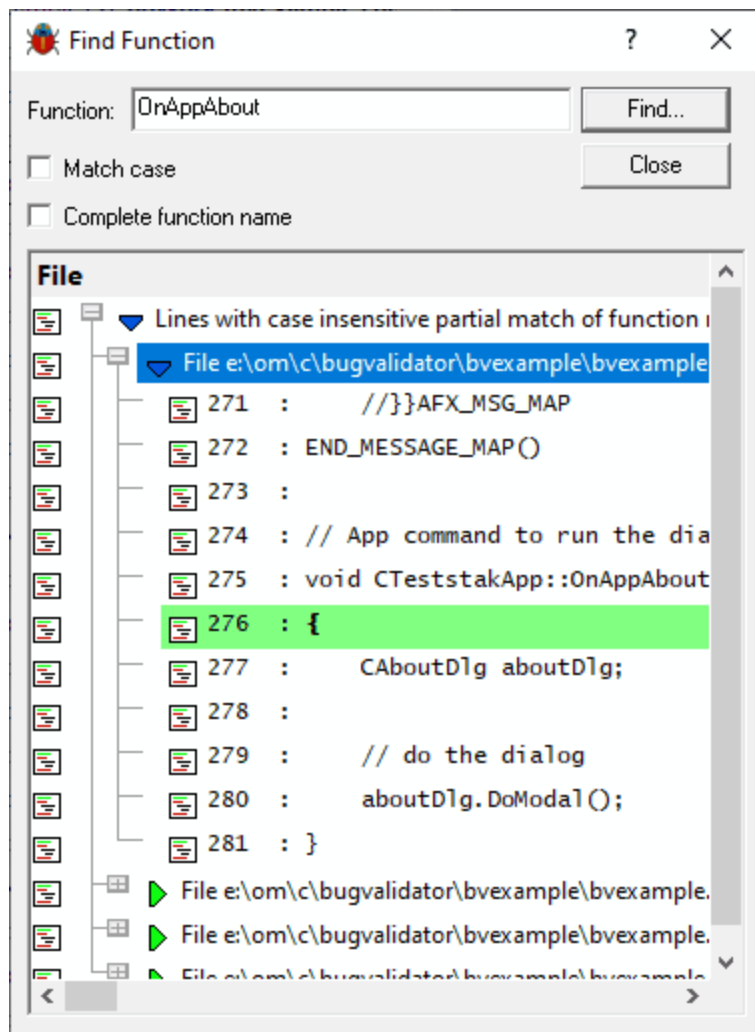
If the search should match any function name that has the specified name as part of its name, do not select the **Complete function name** check box.

For the purposes of complete function name, note that for C++ methods, you must specify `classname::methodname`.

Find

To find any objects with the specified criteria, click the **Find** button. The results are displayed in the **Objects** scrolled list.

The picture shown below shows the results of search for OnAppAbout in the sample application. The filename is shown in yellow, indicating that some parts of the file have been visited. The lines that are part of the function OnAppAbout are displayed as branches of the file entry. These lines are shown in pink, indicating that they have not been visited. One of the lines has been expanded to show the source code for the line (and its surrounding lines)..



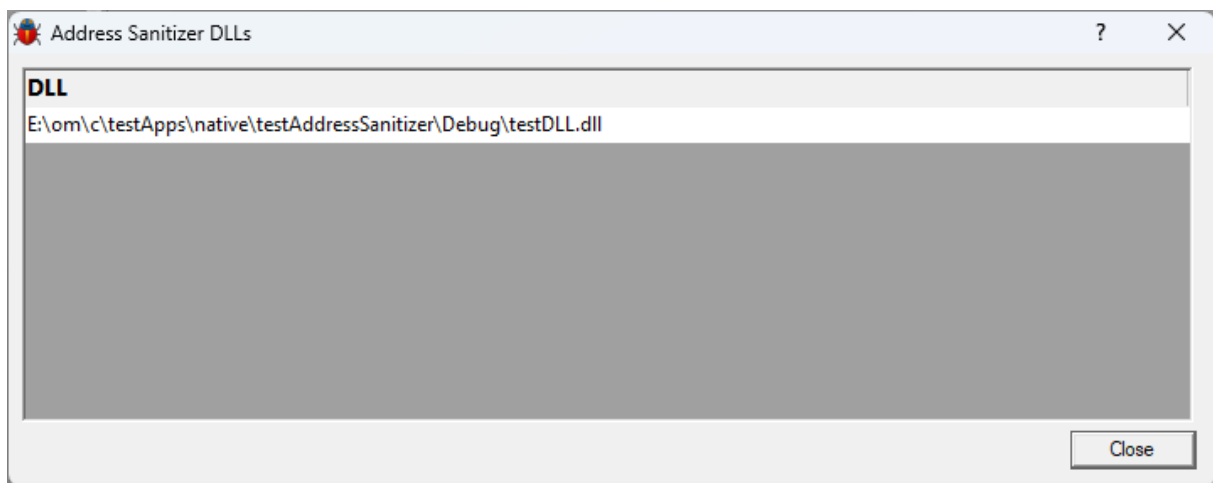
3.12.9 Address Sanitizer DLLs

You can view a list of the any DLLs that use Address Sanitizer which are loaded by your target application.

 **Tools** menu > **Address Sanitizer DLLs...** > shows the **Address Sanitizer DLLs** dialog

The dialog shows:

- the **Path** the DLL was loaded from



Address Sanitizer Compatibility

Bug Validator is not compatible with Address Sanitizer.

Bug Validator will prevent the launching of an executable that is dependent on Address Sanitizer.

However, it is still possible for Bug Validator to profile applications that use Address Sanitizer.

This can happen when Bug Validator:

- launches an application that is not dependent on Address Sanitizer and that application loads a DLL via LoadLibrary() that does use Address Sanitizer
- injects into an application that uses Address Sanitizer
- waits for an application that uses Address Sanitizer
- the Bug Validator API is used to load the Bug Validator profiler into an application that uses Address Sanitizer

When this happens:

- the diagnostic tab will record a "Uses Address Sanitizer" entry
- the Address Sanitizer DLLs dialog will also show that DLL

3.13 Software Updates

This topic covers the three items on the Software Updates menu:

- checking for software updates
- configuring your update schedule
- renewing your software maintenance
- setting your software update credentials
- setting the software update directory

Software updates

If you've been notified of a new software release to Bug Validator or just want to see if there's a new version, this feature makes it easy to update.

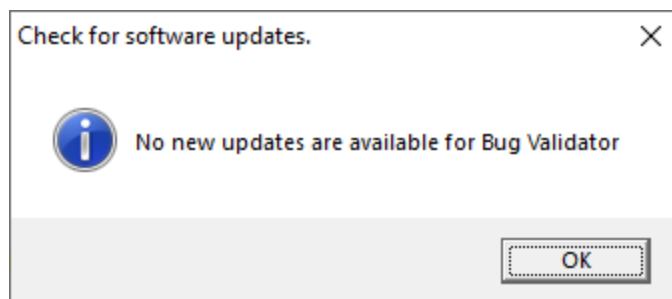
 **Software Updates** menu > **Check for software updates** > checks for updates and shows the software update dialog if any exist

An internet connection is needed to be able to make contact with our servers.



Before updating the software, close the help manual, and end any active session by closing target programs.

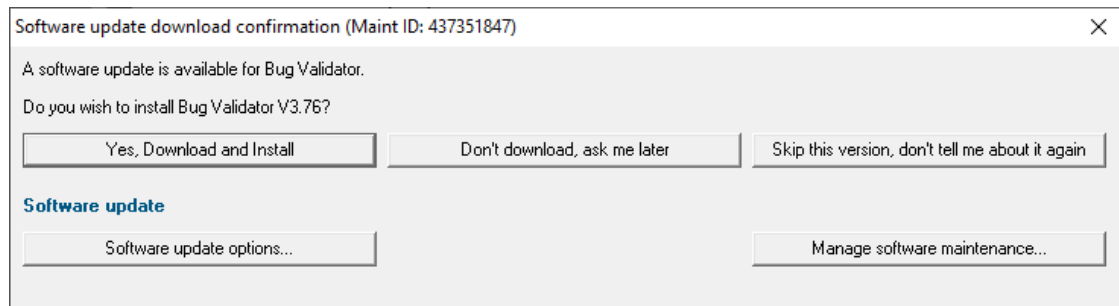
If no updates are available, you'll just see this message:



Note that evaluation and beta versions cannot be updated.

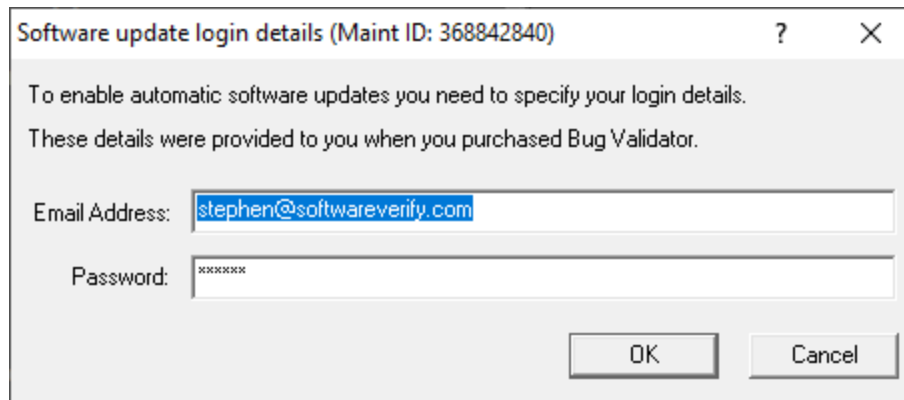
Software Update dialog

If a software update is available for Bug Validator you'll see the software update dialog, unless your maintenance has expired.

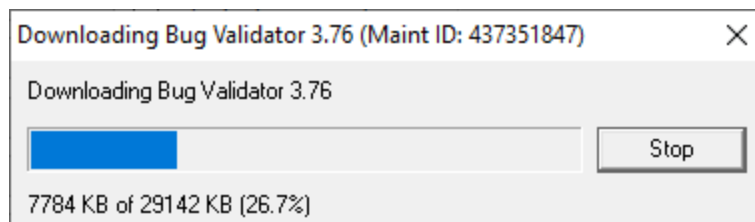


- **Download and install** ➤ prompts you for log-in details if not known, and then downloads the update, showing progress

 You may be asked for your log-in credentials, which you'll have received when you purchased Bug Validator.



Once logged in, the download will start:



Once the update has downloaded, Bug Validator will close, run the installer, and restart.

You can stop the download at any time, if necessary.

- **Don't download...** ➤ Doesn't download, but you'll be prompted for it again next time you start Bug Validator
- **Skip this version...** ➤ Doesn't download the update and doesn't bother you again until there's an even newer update
- **Software update options...** ➤ edit the software update schedule

- **Manage software maintenance...** ➤ opens your browser ready for maintenance renewal

Problems downloading or installing?

If for whatever reason, automatic download and installation fails to complete:

- Log in to <https://www.softwareverify.com/betadownload.php> with the details provided when you signed up for the Bug Validator Beta
- Download the latest installer manually, via one of the .exe, .xyz or .zip files that are available

Make some checks for possible scenarios where files may be locked by Bug Validator as follows:

- Ensure any open sessions are completed
- Ensure any target programs started by Bug Validator are closed
- Ensure Bug Validator and its help manual is also closed
- Ensure any error dialogs from the previous installation are closed

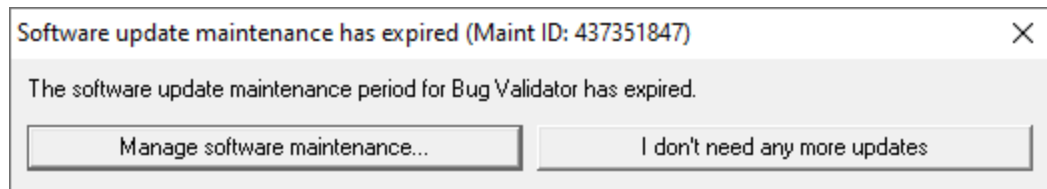
Have your license details handy as you may need to copy information into the license dialog.

You should now be ready to run the new version.

Software maintenance expiry

If the software maintenance period has expired you won't be able to automatically update Bug Validator as above.

Instead, you'll see the software update maintenance expiry dialog:



You can manage your software maintenance or choose to stop receiving any more software updates.

Software update schedule

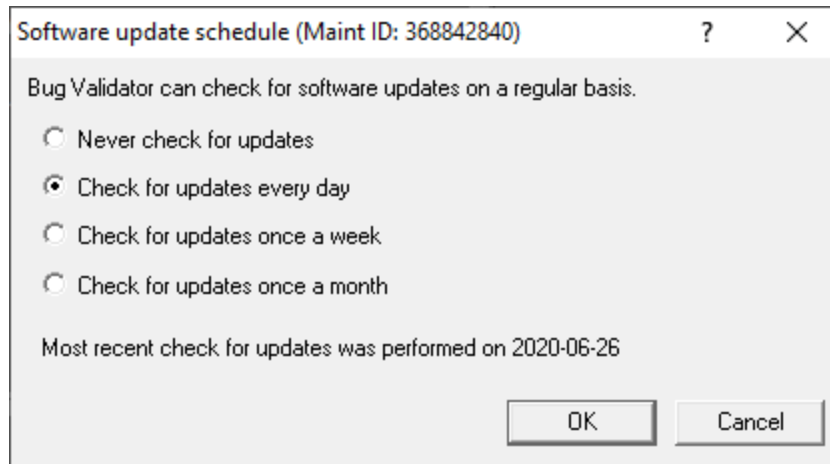
Bug Validator can automatically check to see if a new version of Bug Validator is available for downloading.

 **Software Updates** menu ➤ **Configure software updates** ➤ shows the software update schedule dialog

The update options are:

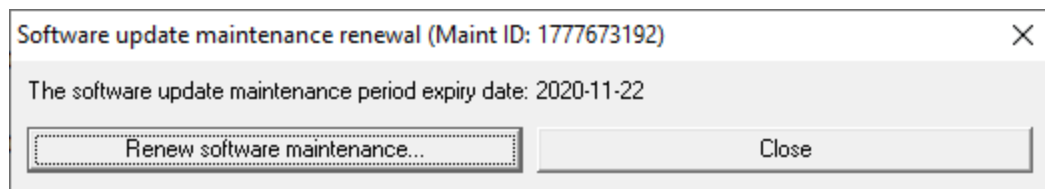
- never check for updates
- check daily (the default)
- check weekly
- check monthly

The most recent check for updates is shown at the bottom.



Managing software maintenance

 **Software Updates** menu > **Renew software updates** > shows the software update maintenance renewal dialog



Your maintenance expiry date is shown. If you don't need to do anything just **Close** the dialog.

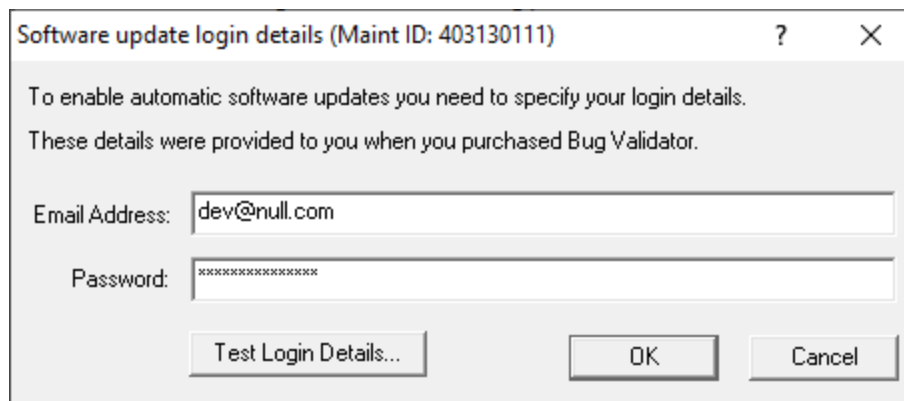
- **Renew software maintenance** > Opens your browser, logging you in to our website  from which you can purchase maintenance



Managing software update credentials

You can configure your software update credentials within the application.

 **Software Updates** menu > **Set software update credentials** > shows the Software update login details dialog

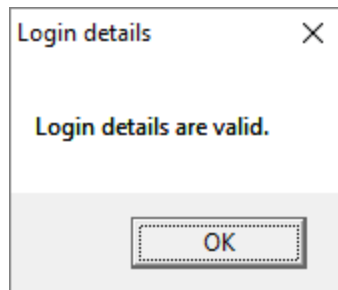


The text will be shown in red if the email address looks incorrectly formatted.

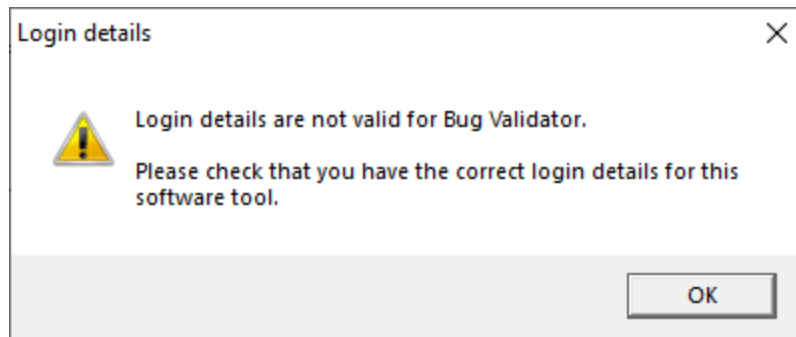
Testing the login details checks they're valid:

- **Test login details** > check your entered details are valid (requires an internet connection)

Valid details will be confirmed:



Invalid details may mean you entered credentials for another application in the Validator suite, or they could have been entered incorrectly.



You should have received the correct credentials when you purchased Bug Validator, or with any software update emails.

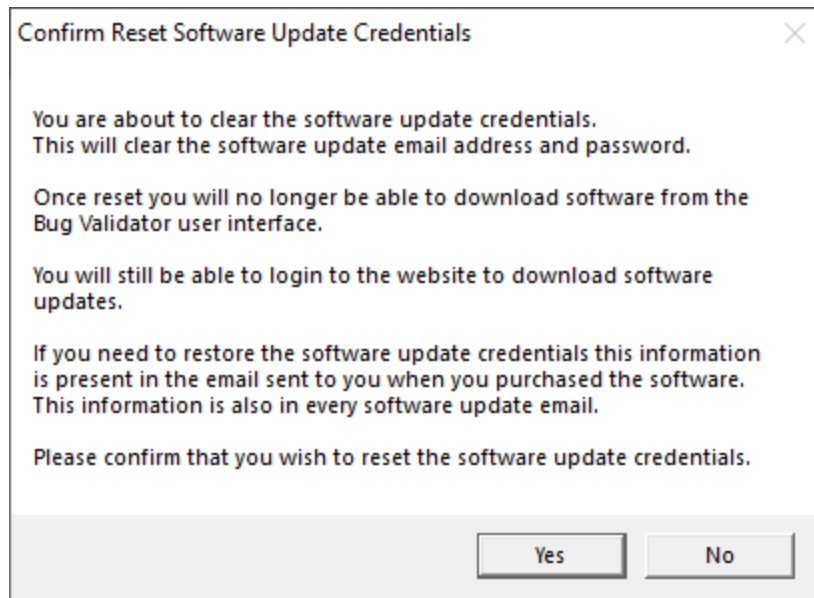
If you experience problems, check with your system administrator or contact Software Verify.

If you need to clear the update credentials, you can do this directly from the menu.

 **Software Updates** menu > **Reset software update credentials** > clears the email and password details stored in the application

You will be asked to confirm the reset. After resetting the credentials, no software updates will occur.

If you later need to restore your credentials, you should have received that information when you purchased Bug Validator, or with any software update emails.

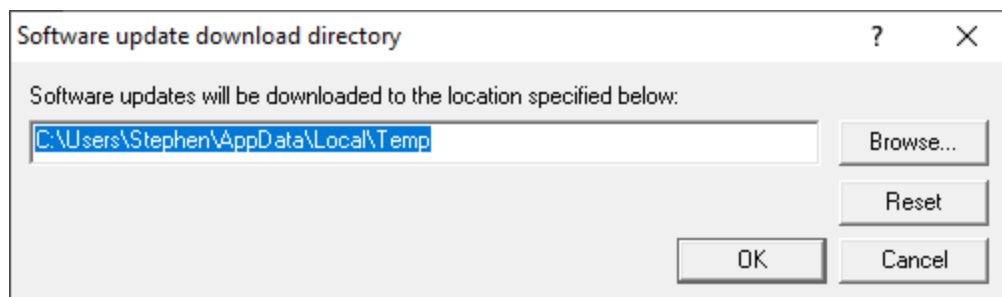


Software update directory

It's important to be able to specify where software updates are downloaded to because of potential security risks that may arise from allowing the `TMP` directory to be executable. For example, to counteract security threats it's possible that account ownership permissions or antivirus software blocks program execution directly from the `TMP` directory.

The `TMP` directory is the default location but if for whatever reason you're not comfortable with that, you can specify your preferred download directory. This allows you to set permissions for `TMP` to deny execute privileges if you wish.

 **Software Updates** menu > **Set software update directory** > shows the Software update download directory dialog



An invalid directory will show the path in red and will not be accepted until a valid folder is entered.

Example reasons for invalid directories include:

- the directory doesn't exist
- the directory doesn't have write privilege (update can't be downloaded)
- the directory doesn't have execute privilege (downloaded update can't be run)

 When modifying the download directory, you should ensure the directory will continue to be valid. Updates may no longer occur if the download location is later invalidated.

- **Reset** > reverts the download location to the user's `TMP` directory

The default location is `c:\users\[username]\AppData\Local\Temp`

3.14 Loading, Saving, Exporting, Closing

Working with sessions

Sessions with Bug Validator can be saved to and loaded from a file so that you can:

- share the session with a colleague
- examine the session at a later date
- compare the session with another session
- create baseline sessions for use in regression tests

Sessions can be even exported in HTML and XML formats.

Closing a session

When you've finished working with a session, it can be closed.

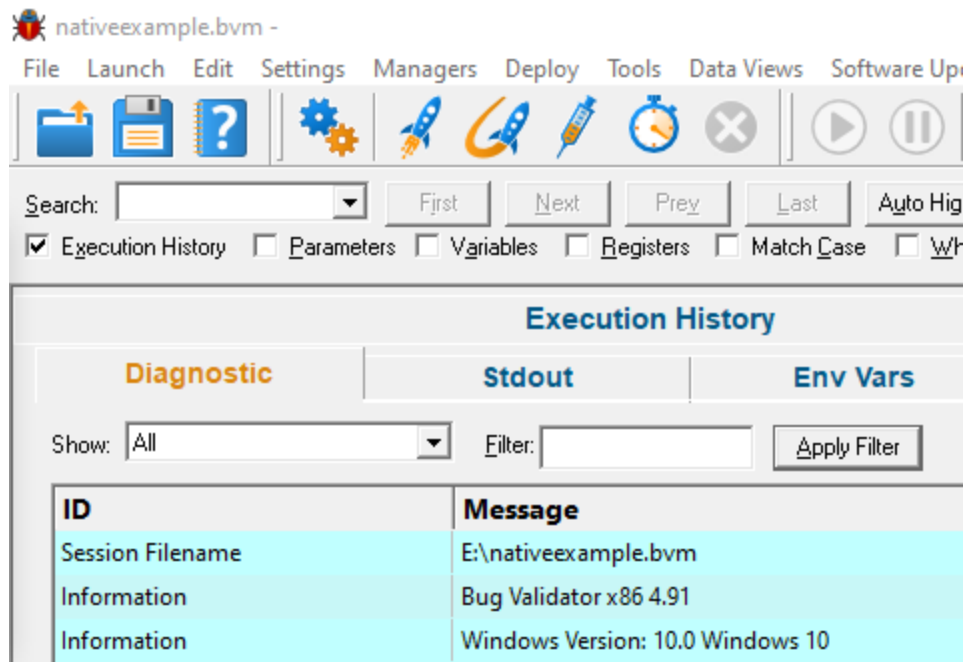
 **File menu** > **Close Session...** > closes the session, clearing the displays

Closing a session may happen automatically if you start a new session and the session count limit is 1.

If the maximum session count allows, closed sessions still appear in the Session Manager, where they can be reopened or deleted.

Session Filename

The session filename is displayed as the first line of the diagnostic data on the Diagnostic tab.



3.14.1 Loading and Saving Sessions

Loading sessions

Load a session using any of the following options.

 **File menu** > **Open Session...** > open a previously saved session from file (*.bvm)

Or click on the Open Session icon on the standard toolbar.



Or use the shortcut:

 +  Open session

If you have a limit of 1 session to be open at a time, any open session will be closed first, otherwise you can open multiple sessions at a time.

Saving sessions

Save a session using any of the following options.

 **File menu** > **Save Session...** > saves all the session data to a file (*.bvm), prompting for a file name if necessary

 **File menu** > **Save As...** > saves the session to a new file

Or click on the Save Session icon on the standard toolbar.



Or use the shortcut:

 +  Save session

Unlike exports, there are no options here, as all session data is saved.

3.14.2 Exporting

Exporting to HTML or XML

Exporting sessions allows you to use external tools to analyse or view session data for whatever reasons you might need.

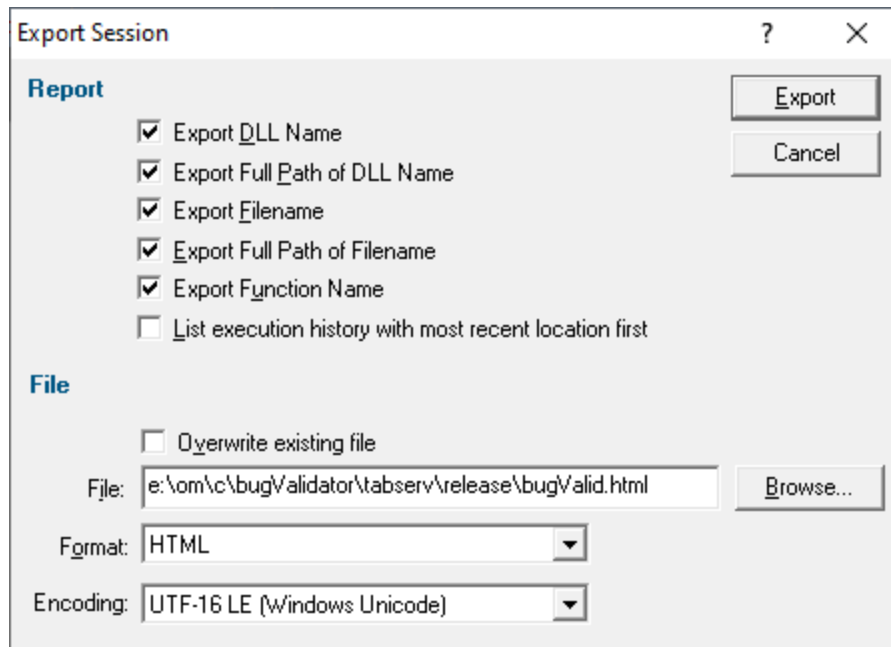
You can export to HTML or XML format:

 **File menu** > **Export Session...** > Choose an **HTML** or **XML** Report > shows the Export Session dialog below

Exporting is not saving

You can't import session data.

Use save and load if you want to save session data for loading back into Bug Validator at a later date.



Report

Select which data you want to export.

- DLL information.
- Filename information.
- Function information.
- Order of data exported.

If HTML export is chosen, the export is in the form of a table. If you require a specific HTML format for session exports, we recommend exporting an XML report and using that to generate the HTML report to your requirements.

File section

- **Overwrite existing file** ➤ check if you *don't* want to be warned about overwrites
- **File** ➤ type the filename or **Browse** to a location
- **Format** ➤ set whether exporting HTML or XML

Defaults to the original menu option selected, but included here to more easily export one format and then the other.



If you are exporting XML data you will want to know the XML export tags that are used.

- **Encoding** ➤ set whether UTF-16 LE, UTF-8 or ASCII encoding. By default the exported file is saved in the Windows Unicode format UTF-16 little endian. You can also save in UTF-8 and ASCII. ASCII has no byte order mark at the start of the file.

Ready to export?

Use the export button at top right when you're ready to go

- **Export** > export the session data

3.14.2.1 XML Export Tags

This section describes which XML tags are used to export the session data from a Bug Validator session.

The start of an exported XML file lists a few details about Bug Validator:

<VALIDATORVERSION>Version of Bug Validator**</VALIDATORVERSION>**

<VALIDATORDATE>Date Bug Validator was built**</VALIDATORDATE>**

<VALIDATORTIME>Time Bug Validator was built**</VALIDATORTIME>**

<TITLE>Name of executable**</TITLE>**

Execution history is listed in the following tag pairs.

<EXECUTIONDATA></EXECUTIONDATA>

The tags found inside the above tag pairs are shown below. Note that all hexadecimal numbers are written with leading zeros and a leading 0x.

<THREAD>Thread id**</THREAD>**

<MODULE>Name of DLL/EXE**</MODULE>**

<FILENAME>Filename of source file**</FILENAME>**

<LINE>Decimal line number**</LINE>**

<ADDRESS>Hexadecimal address**</ADDRESS>**

<FUNCTIONNAME>Function name and byte offset**</FUNCTIONNAME>**

3.15 Starting your target program

Starting options

There are seven ways to start a target program and have Bug Validator collect data from it.

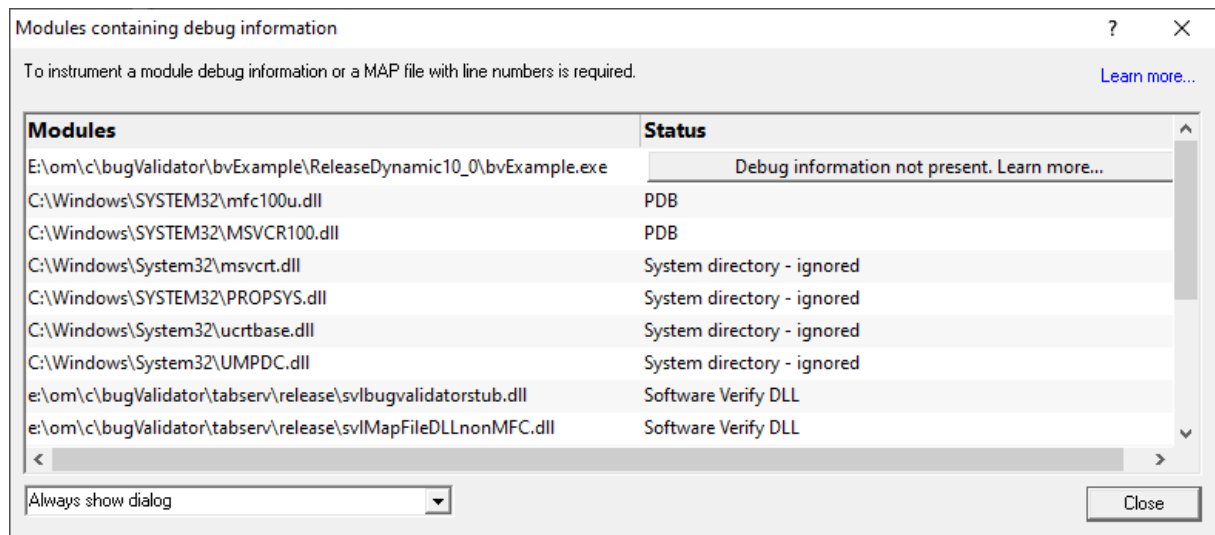
- Launch your program in a specified directory, with as many command line arguments as you want

- Inject Bug Validator into an already running program
- Wait until a specific program starts to run before attaching to it - e.g. for programs started as an OLE server
- Monitor a service
- Monitor IIS and ISAPI
- Use the Native API to start Bug Validator from code that you control
- Start Bug Validator from the command line, allowing you to automate your use of Bug Validator

Modules without PDB files and without MAP files

For your application to be processed for execution history tracking, each module that you want to have processed must have a PDB file with debug data, or a MAP file with line number data. Map files for Release builds cannot have line number data as the /MAPINFO:LINEs option does not work in Release builds.

For modules that are not system files, Bug Validator will warn you if a module did not have any appropriate line number information. The warning is shown as a dialog listing the module names that did not have the appropriate information.



If you do not wish to see this warning dialog again, select the **Do not show this dialog again** check box.

IMPORTANT.

Due to daylight saving times it is possible for a MAP file to have an embedded timestamp that is different than the DLL timestamp by an hour.

In these situations Bug Validator will not recognise the MAP as valid. The solution to this problem is to rebuild the application.

3.15.1 Launching the program

Launching the application

Having Bug Validator launch your program is the most common way to start up

When you're ready to start running a target program

 **File** menu > **Start Application...** > Shows the launch program wizard or dialog below

or click on the launch icon on the session toolbar.



or use the shortcut

 **F4** Start application

➔ You can easily re-launch the most recently run program.

User interface mode

There are two interface modes used while starting a program

- Wizard mode guides you through the tasks in a linear fashion
- Dialog mode has all options contained in a single dialog

All the options are the same - just in different places

In this section we'll cover the Wizard mode first and the Dialog mode later.

The start application wizard

On first use, the wizard appears with fields cleared, but here's an example with fields set:

Start application wizard ? X

Select an application to start.

Previously started applications are shown in the list at the bottom of the wizard, with their startup directory and command line arguments. If your application is displayed in the list, you can select it repeat the launch of the application.

Application (*.exe or *.bat):

Application to monitor (*.exe):

Command Line Arguments: Launch count:

Startup Directory:

Environment Variables (override global environment variables)

File to supply to stdin (leave blank for none):

File to supply to stdout (leave blank for none):

Click the Reset button to clear the list. ☐ Full Path ☒ Image Name

Admin	Application	Arguments	Directory	Environment
	testApp.exe		E:\om\c\dbgHelpBrowser\testApp\...	
	bvExample.exe		E:\om\c\bugValidator\bvExample\...	
	dbgHelpBrowser.exe		E:\om\c\dbgHelpBrowser\Release\...	
	ColInitializeTest.exe		E:\om\c\testApps\ColInitializeTest\...	

<< Prev

Enter the details for your program, or if you want to run a previous program select it from the application list to repopulate the details.

After entering details click **Next >>** for the next page of the wizard.

Page 1: Entering details

- **Application to start** ➤ type or **Browse** to set the program name to launch

You can also choose a batch file and the first executable started in the batch file will be launched.

You can also choose a powershell script and the first executable started in the powershell script will be launched.

Manually typing a path will show red text until a valid path is entered, after which the text becomes black.

- **Application to Monitor** ➤ select the application that you wish to monitor.

The values displayed in this combo will consist of:

- <<Any application that is launched>>
- The name of the application specified in the **Application to Launch** field.
- Any valid filenames that match the list of filenames specified in the application to monitor settings.

The default selection for **Application to Monitor** will be always be the same as **Application to Launch** unless there are valid application to monitor filenames added to the control and one of those filenames is marked as a default selection.

➤ click **Edit...** to edit the Application to Monitor settings.

*If you are not sure what to set this field to leave it set to the same value as **Application to Launch**.*

Learn more about Application to Monitor.

- **Launch Count** ➤ select the launch count.

This is reset to 1 every time the **Application to Monitor** field selection changes.

If Application to Launch is the same as Application to Monitor this field is set to 1 and is not editable.

If you are not sure what to set this field to leave it set to 1.

Learn more about Launch Count.

- **Command Line Arguments** ➤ enter program arguments exactly as passed to the target program
- **Startup Directory** ➤ enter or click **Dir...** to set the directory for the program to start in

When setting your target program, this will default to the location of the executable

- **Environment Variables** ➤ click **Edit...** to set any additional environment variables before your program starts

These are managed in the Environment Variables Dialog.

- **File to supply to stdin** ➤ optionally enter or **Browse** to set a file to be read and piped to the standard input of the application

- **File to supply to stdout** ➤ optionally enter or **Browse** to set a file to be written with data piped from the standard output of the application

Page 1: Using details from a previous run

The list at the bottom of the wizard shows previously run programs.

Selecting an item in the list populates all the details above as used on the last run for that program.

You can still edit those details before starting.

- **Full path** ➤ shows the full path to the executable in the list
- **Image Name** ➤ shows the short program name without path
- **Delete** ➤ removes a selected program from the list
- **Reset** ➤ clears all details in the wizard - including the list of previously run applications below

Page 2: Data collection and redirection

- **Collect data from application** ➤ If it's the startup procedure you want to validate, obviously start collecting data from launch.

Depending on your application, and what you want to validate, you may want to start collecting data immediately, or do it later.

If your program has a complex start-up procedure, initialising lots of data, it may be much faster *not* to collect data until the program has launched.

➡ See the section on controlling data collection for how to turn collection on and off after launch.

- **Redirect standard output** ➤ Controls redirection of stdout and stderr

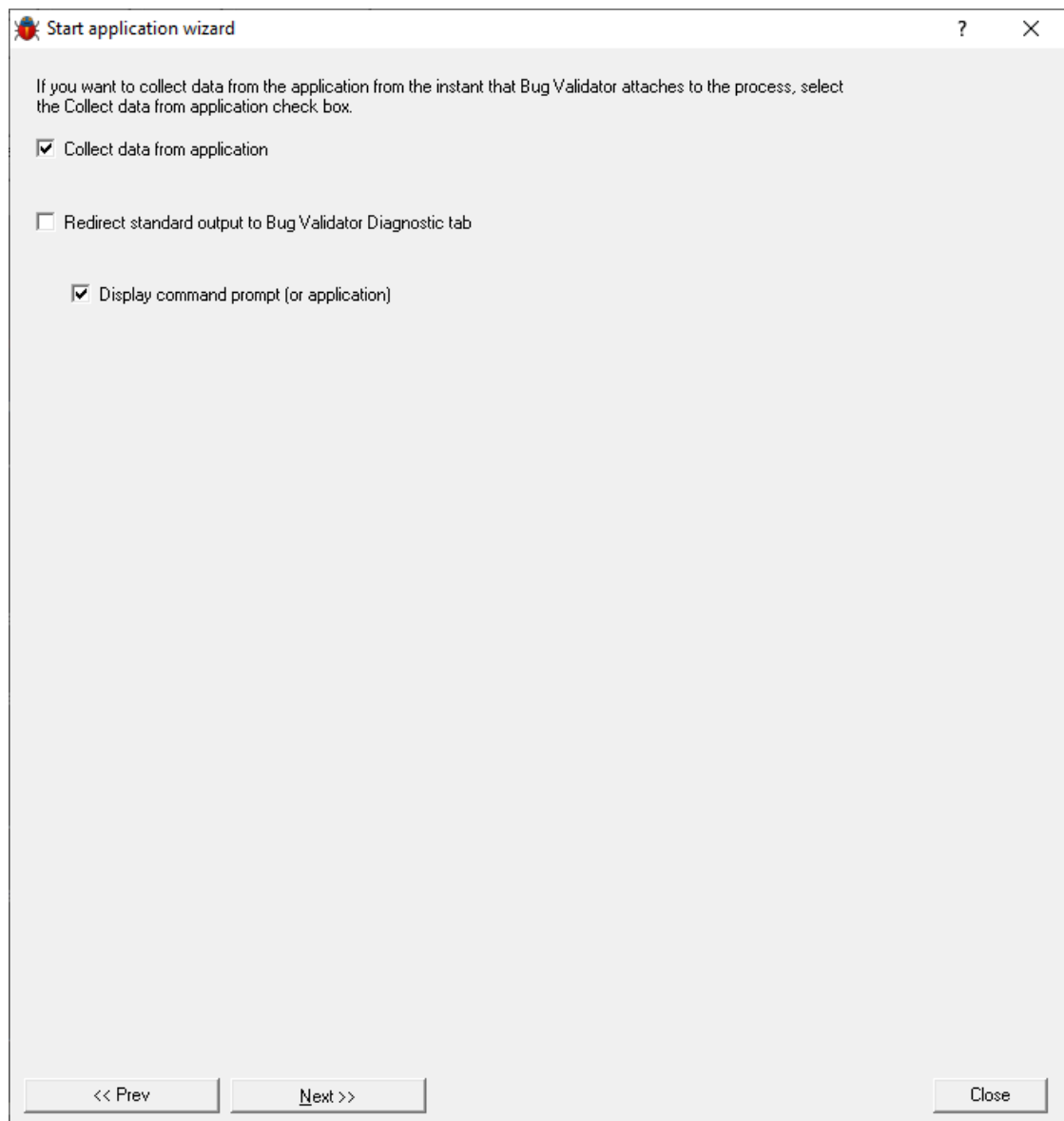
Use this option if you want to collect the output of stdout and stderr for later analysis.

Be aware that if the output of the program under test generates a lot of data via stdout or stderr that this data will need to be stored in memory and could exhaust Bug Validator's memory.

- **Display command prompt** ➤ Shows or hides the launched application.

If you are collecting stdout and stderr you may not be interested in viewing the application (or the command prompt if it is a console application). This provides you the option to hide the application when it is running.

Be aware that if you hide a command prompt you will not be able to type anything into the application.

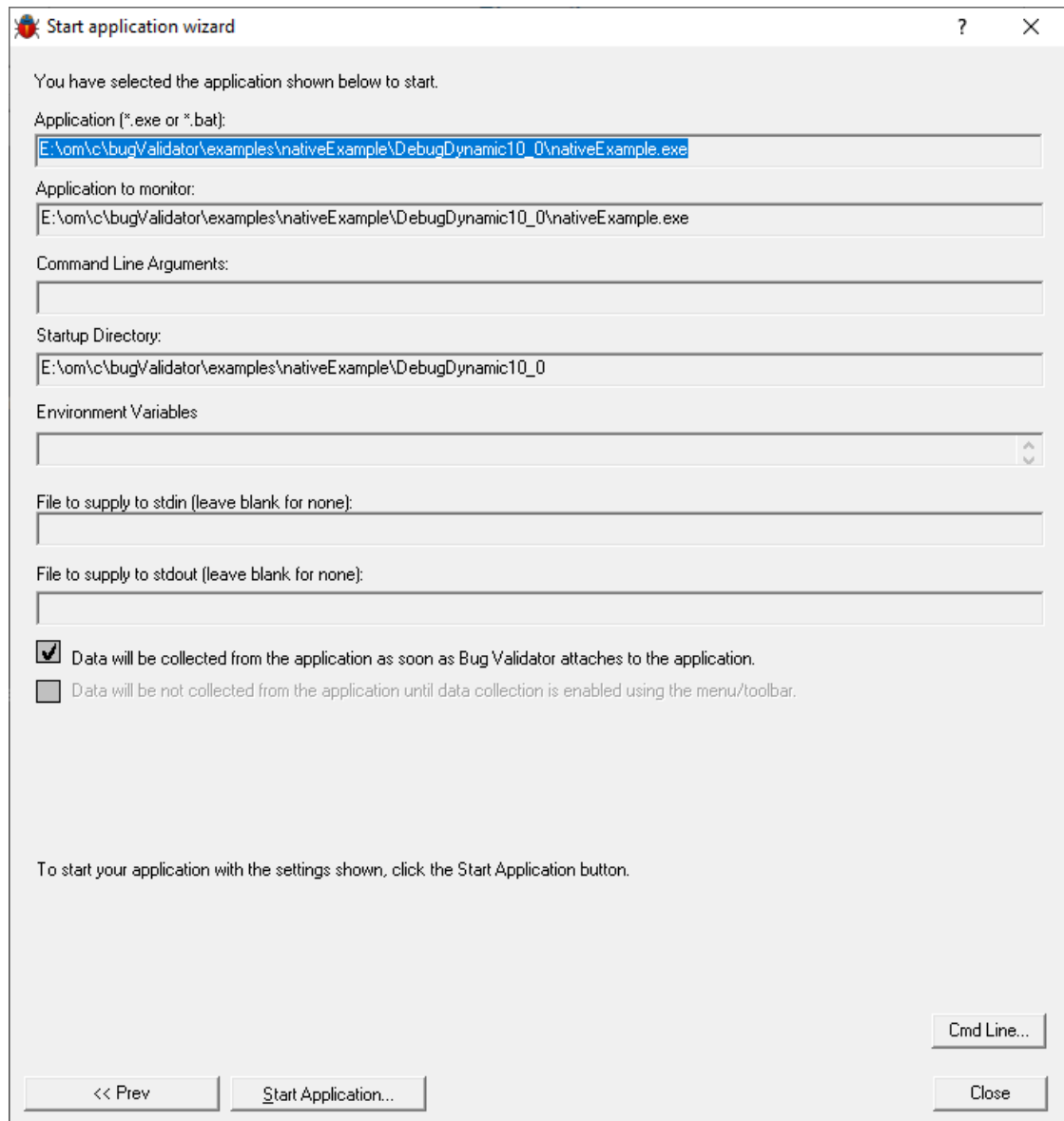


Page 3: Summary and starting your program

The last page is just a summary of the options you have chosen.

If you're happy with the settings, go ahead:

- **Start Application...** ➤ start your program and attach Bug Validator to it
- **Cmd Line...** ➤ display the command line builder



The image shows a Windows-style dialog box titled "Start application wizard". It contains several text input fields and checkboxes. The first field, "Application (*.exe or *.bat):", contains the path "E:\om\vc\bugValidator\examples\nativeExample\DebugDynamic10_0\nativeExample.exe". The second field, "Application to monitor:", contains the same path. The "Command Line Arguments:" field is empty. The "Startup Directory:" field contains "E:\om\vc\bugValidator\examples\nativeExample\DebugDynamic10_0". The "Environment Variables" field is empty with a dropdown arrow. The "File to supply to stdin (leave blank for none):" field is empty. The "File to supply to stdout (leave blank for none):" field is empty. There are two checkboxes: the first is checked and labeled "Data will be collected from the application as soon as Bug Validator attaches to the application.", and the second is unchecked and labeled "Data will be not collected from the application until data collection is enabled using the menu/toolbar.". At the bottom, there is a message: "To start your application with the settings shown, click the Start Application button." and three buttons: "<< Prev", "Start Application...", and "Close". A "Cmd Line..." button is also present on the right side.

Start application wizard

You have selected the application shown below to start.

Application (*.exe or *.bat):
E:\om\vc\bugValidator\examples\nativeExample\DebugDynamic10_0\nativeExample.exe

Application to monitor:
E:\om\vc\bugValidator\examples\nativeExample\DebugDynamic10_0\nativeExample.exe

Command Line Arguments:

Startup Directory:
E:\om\vc\bugValidator\examples\nativeExample\DebugDynamic10_0

Environment Variables

File to supply to stdin (leave blank for none):

File to supply to stdout (leave blank for none):

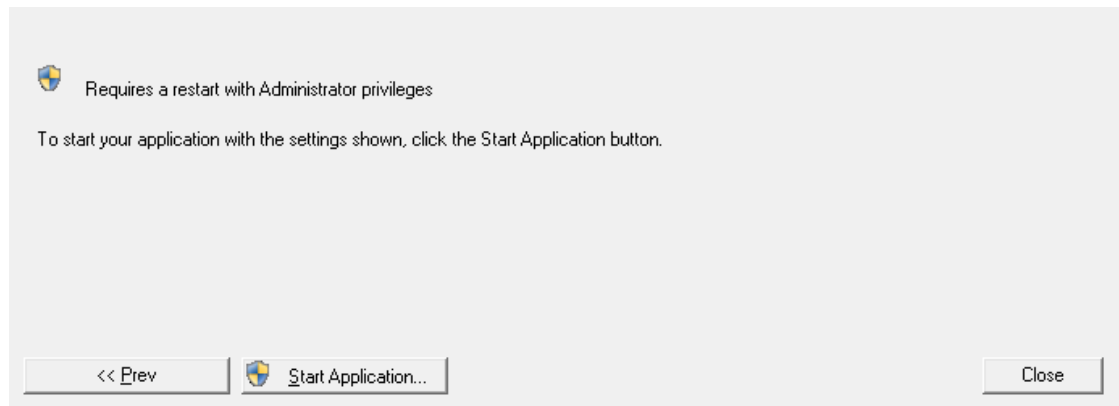
☒ Data will be collected from the application as soon as Bug Validator attaches to the application.
☐ Data will be not collected from the application until data collection is enabled using the menu/toolbar.

To start your application with the settings shown, click the Start Application button.

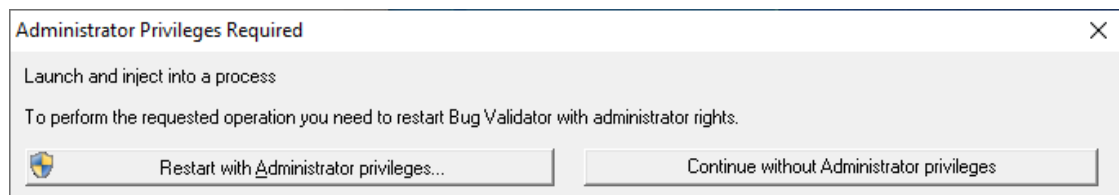
Cmd Line...

<< Prev Start Application... Close

If administrator privileges are required you'll be reminded of the need to restart here:



- **Start Application...** ➤ shows the Administrator Privileges Required confirmation dialog before restarting.



Dialog mode

In Dialog mode, all the settings are in one dialog which looks very much like the first page of the launch wizard above.

At the top are the options to collect stdout (this will be displayed on the diagnostic tab) and to start collecting data immediately.

- **Launch** ➤ start your program and attach Bug Validator to it
- **Cmd Line...** ➤ display the command line builder

Double clicking a program in the list will also start it immediately.

Start an application and inject Validator into the process

☒ Collect data from application ☐ Collect Stdout

Application (*.exe or *.bat):
 E:\om\c\bug\validator\examples\nativeExample\DebugDynamic10_0\nativeExample.exe

Application to monitor (*.exe):
 E:\om\c\bug\validator\examples\nativeExample\DebugDynamic10_0\nativeExample.exe

Arguments: Launch count: 1

Startup Directory:
 E:\om\c\bug\validator\examples\nativeExample\DebugDynamic10_0

Environment Variables (override global environment variables)

File to supply to stdin (leave blank for none):

File to supply to stdout (leave blank for none):

Previously started applications (double click to relaunch) ☒ Full Path ☐ Image Name

Admin	Application	Arguments	Directory	Environment
	E:\om\c\bug\validator\examples\nativeExample\DebugDynamic10_0\nativeExample.exe		E:\om\c\bug\validator\examples\nativeExample\DebugDynamic10_0	

Stdout and Stderr Collection

If a file is supplied for stdin and / or stdout, file redirection will take place. Stdout will still be visible on the command prompt.

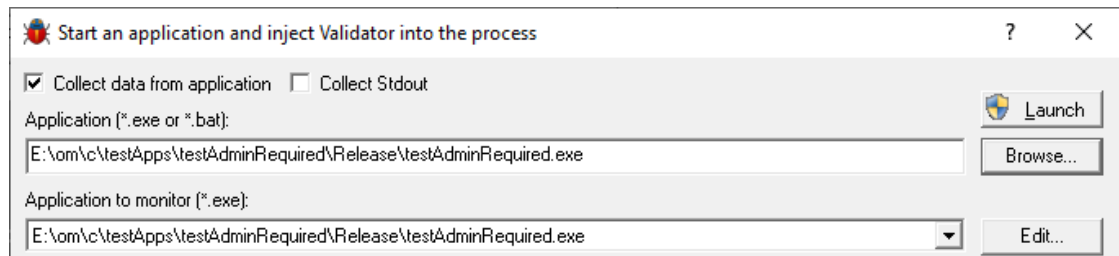
If **collect stdout** (or **redirect stdout** on the launch wizard) is selected, stdout will be collected by Bug Validator and displayed on the Diagnostic tab. Stdout will not be visible on the command prompt.

Administrator privileges in dialog mode

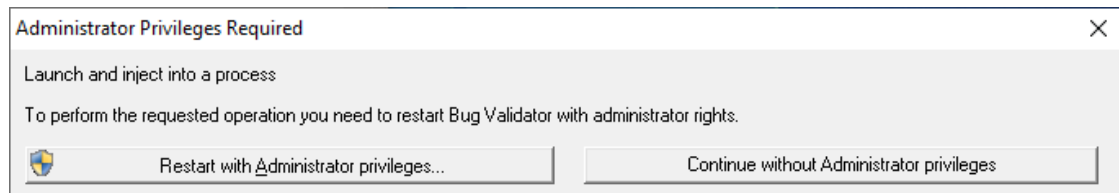
The following applies only if you did *not* start Bug Validator in administrator mode.

Anywhere you see the icon indicates that a restart with administrator privileges will be required to proceed.

This will be shown if the target application has a manifest with requireAdministrator or highestAvailable settings.



- **Launch** ➤ shows the Administrator Privileges Required confirmation dialog before restarting.



If you started Bug Validator in administrator mode, you won't see any of these warnings, and everything will behave as normal.

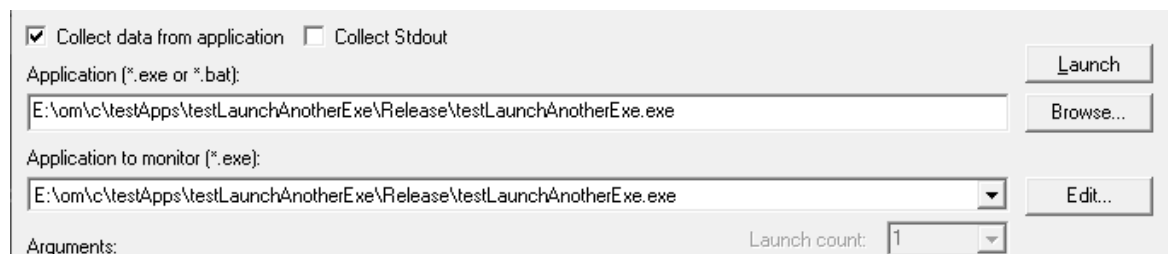
How do I use Application to Monitor and Launch Count?

The three fields **Application to Launch**, **Application to Monitor** and **Launch Count** work together to control which application gets monitored by Bug Validator.

To best explain how this works we will use an example application testLaunchAnotherExe.exe which launches some applications testLaunchApp1.exe, testLaunchApp2.exe, testLaunchApp3.exe.

Flow tracing for testLaunchAnotherExe.exe

- Set Application to Launch to testLaunchAnotherExe.exe
- Set Application to Monitor to testLaunchAnotherExe.exe
- Set Launch Count to 1.



Flow tracing for any application launched by testLaunchAnotherExe.exe

- Set Application to Launch to testLaunchAnotherExe.exe
- Set Application to Monitor to <<Any application that is launched>>
- Set Launch Count to 1.

To do the next two examples you would also need to use the Applications to Monitor settings to add a definition for testLaunchAnotherExe and the applications it launches.

Flow tracing for testLaunchApp1.exe

- Set Application to Launch to testLaunchAnotherExe.exe
- Set Application to Monitor to testLaunchApp1.exe
- Set Launch Count to 1.

Flow tracing for the third run of testLaunchApp2.exe

- Set Application to Launch to testLaunchAnotherExe.exe
- Set Application to Monitor to testLaunchApp2.exe
- Set Launch Count to 3.

3.15.2 Re-Launching the program

Re-launching the application

It's very easy to start another session using the most recently run program and settings:

 **File** menu > **Re-Start Application...** > starts the most recently launched program

or click on the re-launch icon on the session toolbar.



or use the shortcut

F5 Re-start application

No wizards or dialogs appear, so be ready for the application to start right away.

➔ In the general questions see *Why might Inject or Launch fail?* for troubleshooting launch problems.

There is no difference between wizard and dialog interface mode when re-launching.

3.15.3 Injecting into a running program

Injecting into a running program

Bug Validator attaches to a running process by injecting the stub into the process so it can start collecting data.

Choose one of these methods of starting the injection:

 **File** menu > **Inject...** > shows the Attach to Running Process wizard or dialog below

or click on the Inject icon on the session toolbar.



or use the shortcut

F3 Inject into running application

Injecting into a service?

If your process is a service, Bug Validator won't be able to attach to it.

Services can't have process handles opened by third party applications, even with Administrator privileges.

In order to work with services, you can use the NT service API and monitor the service

User interface mode

There are two interface modes used while starting a program

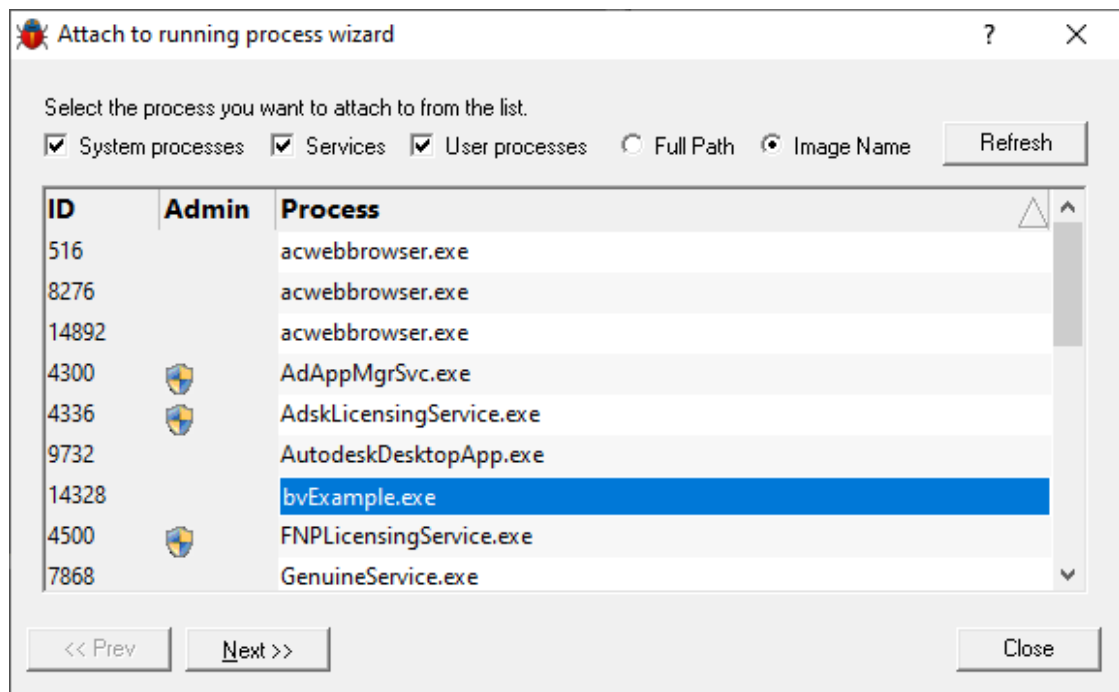
- Wizard mode guides you through the tasks in a linear fashion
- Dialog mode has all options contained in a single dialog

All the options are the same - just in slightly different places

In this section we'll cover the Wizard mode first and the Dialog mode later.

The attach to running process wizard

The first page of the wizard shows a list of running system and user processes.



Choose the process and click **Next >>** for the next page of the wizard.

Page 1: Choosing the process

- **System processes / Services / User processes** ➤ show either of system or services or user processes in the list, or both
- **Full path** ➤ shows the full path to the process executable in the list
- **Image Name** ➤ shows the short program name without path

- **Refresh** ➤ update the list with currently running processes

Clicking on the headers of the list will sort them by ID or by name using the full name or short name, depending on what's displayed.

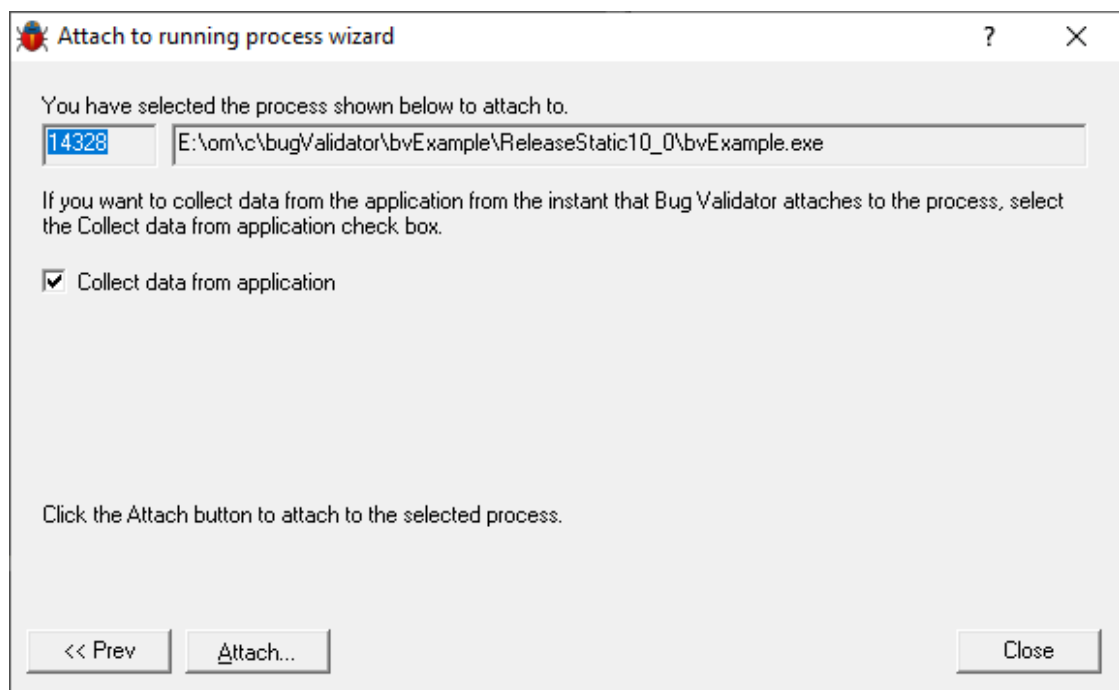
Page 2: Data collection

Depending on your application, and what you want to validate, you may want to start collecting data as soon as injection happens, or do it later.

If your program has a complex start-up procedure, initialising lots of data, it may be much faster *not* to collect data until the program has launched.

If it's the startup procedure you want to validate, obviously start collecting data from launch.

➔ See the section on controlling data collection for how to turn collection on and off after launch.



Summary and starting your program

The second page confirms the process you have selected to inject into, and prompts you to attach:

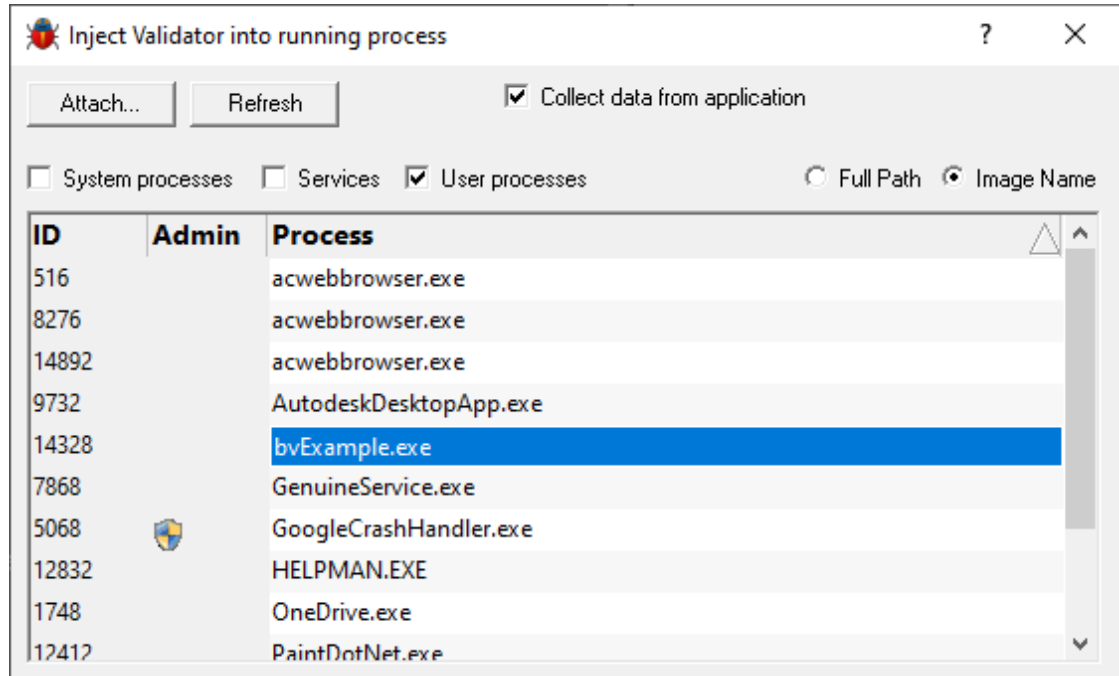
- **Attach...** ➤ injects Bug Validator into the specified process, showing progress status

➔ In the general questions see *Why might Inject or Launch fail?* for troubleshooting launch problems.

Dialog mode

In Dialog mode, all the settings are in one dialog which looks very much like the first page of the wizard above.

The option to start collecting data is at the top, as is the **Attach...** button



3.15.4 Waiting for a program

Waiting for a program

Waiting for a program is essentially the same as injection except that instead of injecting into a running program, Bug Validator watches for the process starting up and *then* injects.

 If the process is a service, Bug Validator won't be able to attach to it as services can't have process handles opened by third party applications, even with Administrator privileges.

Choose one of these methods of waiting:

 **File** menu > **Wait for Application...** > shows the Wait for application wizard or dialog below

or click on the Wait (timer) icon on the session toolbar.



or use the shortcut

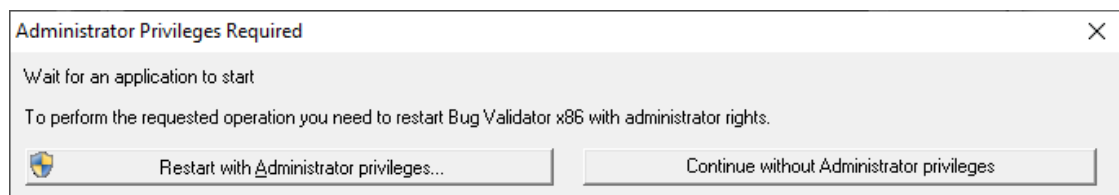
F2 Wait for application

Administrator privileges

The following applies only if you did *not* start Bug Validator in administrator mode.

If the application you want to wait for is running with Administrator privileges, Bug Validator will also need to run with Administrator privileges.

When choosing the 'wait for program' method described in this topic, a restart of Bug Validator with administrator privileges will be required to proceed.



Waiting for a service?

If your process is a service, Bug Validator won't be able to attach to it.

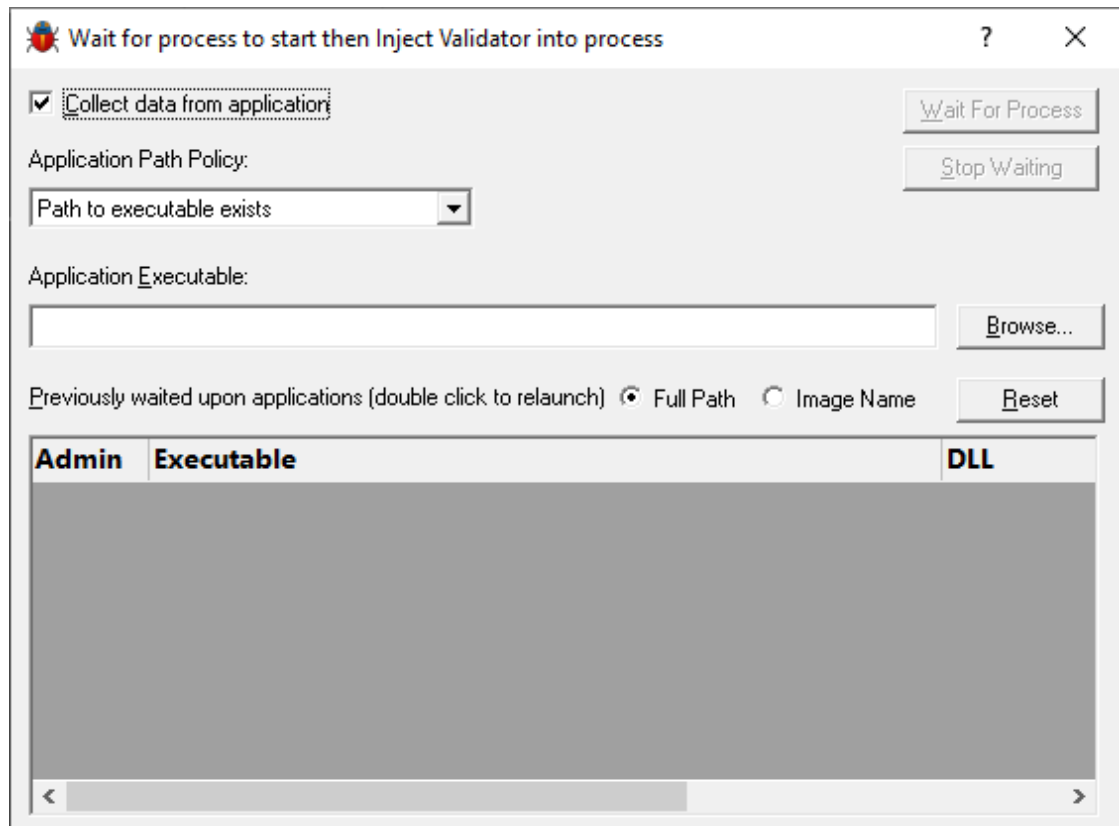
Services can't have process handles opened by third party applications, even with Administrator privileges.

In order to work with services, you can use the NT service API and monitor the service

The wait for application dialog

The wait for application dialog lets you specify the application or choose one that you've waited for previously.

If you choose a previously waited for application the Application Path Policy will be set to the same value as used with that application.



Data collection

Depending on your application, and what you want to validate, you may want to start collecting data as soon as injection has happened, or do it later.

If your program has a complex start-up procedure, initialising lots of data, it may be much faster *not* to collect data until the program has launched.

If it's the startup procedure you want to validate, obviously start collecting data from launch.

➔ See the section on controlling data collection for how to turn collection on and off after launch.

Specifying the application

- **Application Path Policy** ➤ specify how the specified executable is treated
 - Path to executable exists ➤ the executable will be checked that it exists and is appropriate for Bug Validator to work with
 - Path to executable is created dynamically ➤ most pre-wait checks are not performed - use this if the path the executable is on does not exist at the time you start waiting for the process to start
- **Application to wait for** ➤ type or **Browse** to set the application name to launch

Alternatively, select a previously waited for application from the list.

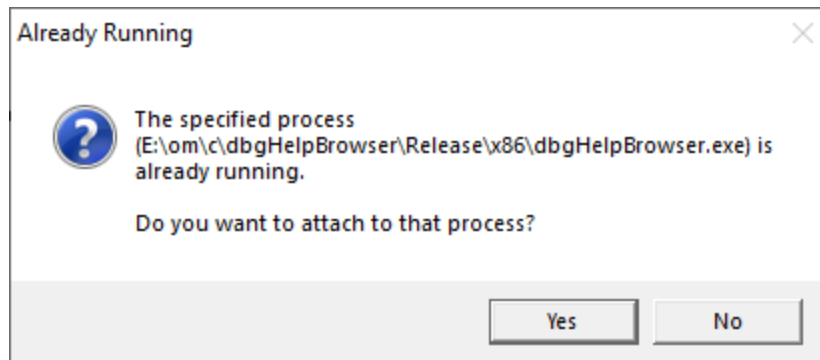
- **Full path** > shows the full path to the process executable in the list
- **Image Name** > shows the short program name without path
- **Reset** > clears the list

Waiting for an application

- **Wait For Process** > start waiting for the specified executable to start
- **Stop Waiting** > stop waiting for the specified executable to start

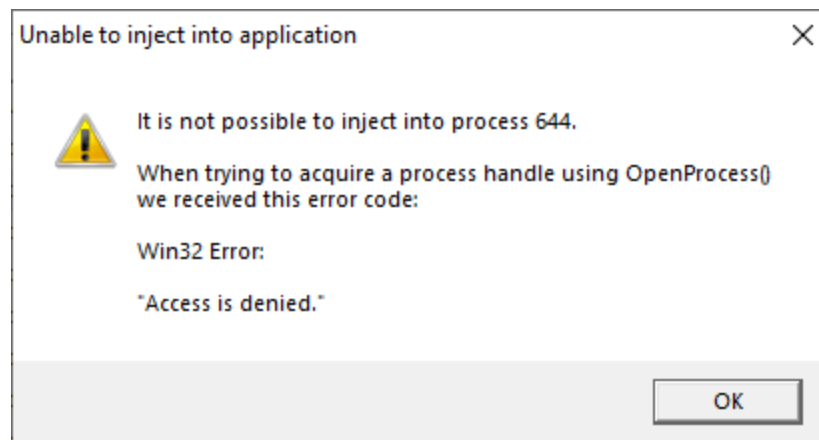
What could go wrong?

The program you're waiting for might already be running, in which case you'll be given the option to cancel or attach to the existing process:



Timing issues are inherit with native injecting into a program as it starts up.

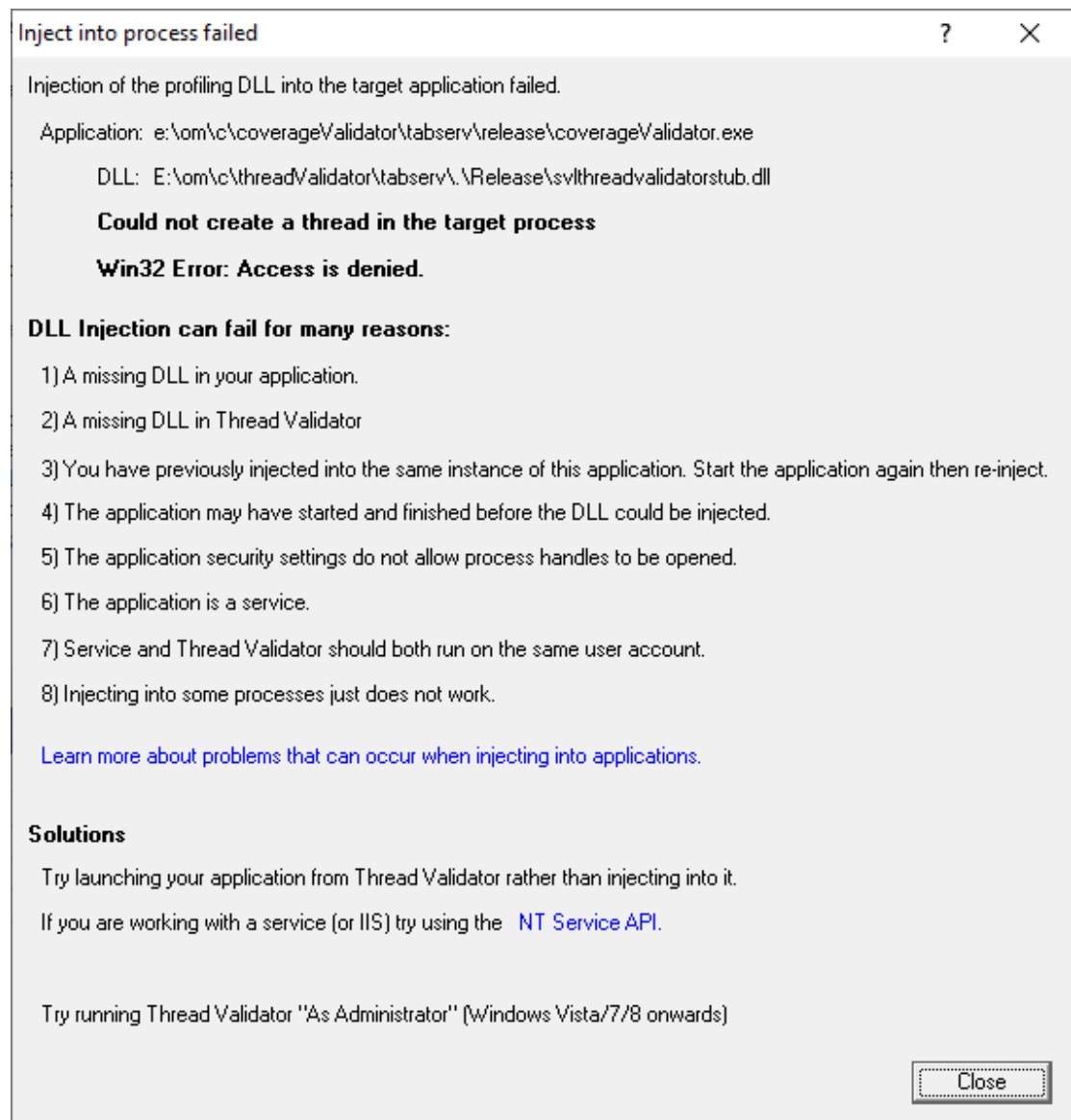
This could cause the injection to fail in unpredictable ways and you may see dialogs like that below:



One case when this dialog can occur is if the program needs to run at an elevated privilege and is waiting for the user to give permission via the UAC dialog.

Injection may fail for different reasons and you might see the following information dialog showing:

- messages relating to the specific failure
- a selection of reasons why failure might be occurring
- some possible solutions to the problem



Sometimes retrying a few times might catch a better moment for attaching to the process.

➔ In the general questions see *Why might Inject or Launch fail?* for troubleshooting launch problems.

3.15.5 Monitor a service

Monitoring a service

Monitoring a service works for:

- native services
- .Net services
- mixed mode services.

Native Services

If you are working with native services you must use the NT Service API in your service *as well as* using the Monitor a service method below.

.Net Services

Bug Validator won't attach until some .Net code is executed.

If there is native code being called prior to the .Net code, Bug Validator won't monitor that code, only the native code called after the first .Net code that is called.

To monitor any native code called *prior* to your .Net code, use the NT Service API.

 When working with Bug Validator and services, you still start the service the way you normally do - e.g. with the service control manager.

The code that you have embedded into your service then contacts Bug Validator, which you should have running before starting the service.

To start monitoring a service:

 **File** menu > **Monitor a service...** > shows the Monitor a service dialog below

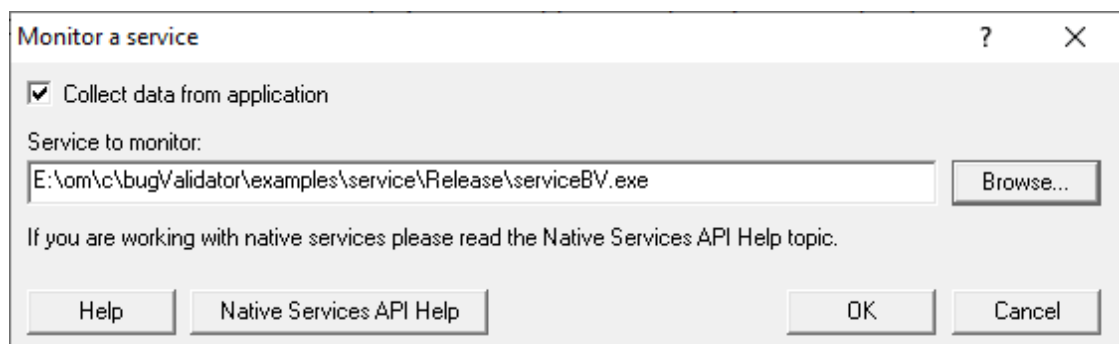
or use the shortcut

 Monitor a service

The monitor a service dialog

First ensure the service is installed, but not running.

Set the service to monitor, choose whether to start collecting data right away, and click OK.



- **Service to monitor** > type or **Browse** to set the service name to monitor
- **OK** > waits for the service to start before injecting into it

Start the service in the normal manner, e.g. from the control panel, the command line or programmatically.

Data collection

Depending on your application, and what you want to validate, you may want to start collecting data as soon as injection has happened, or do it later.

If your program has a complex start-up procedure, initialising lots of data, it may be much faster *not* to collect data until the program has launched.

If it's the startup procedure you want to validate, obviously start collecting data immediately.

➔ See the section on controlling data collection for how to turn collection on and off after launch.

Examples

Example demonstrating how to monitor a service.

Example demonstrating how to monitor an application launched from a service (how to monitor any application running on a service account).

3.15.6 IIS

3.15.6.1 Monitor IIS and ISAPI

Monitoring ISAPI

Monitoring ISAPI works for:

- Native ISAPI extensions.

Native ISAPI

If you are working with native ISAPI you must use the NT Service API in your service *as well as* using the Monitor ISAPI method below.

An outline of how to work with IIS is described in the topic Working with IIS.

To start monitoring ISAPI:

 **Launch** menu ➤ **IIS** menu ➤ **Monitor IIS and ISAPI...** ➤ shows the Monitor ISAPI dialog below

or use the shortcut

 **F7** Monitor IIS and ISAPI

The monitor ISAPI dialog

Set the dll to monitor, the web root, the IIS process, an optional web browser to use and an optional url to launch, and click OK.

- **ISAPI DLL** ➤ type or **Browse** to set the ISAPI DLL that we're monitoring
- **IIS web** ➤ type or **Browse** to set the web root for the IIS website we're working with
- **IIS process to monitor** ➤ select the IIS process we're working with
- **Web Browser** ➤ select the web browser that you're going to use to load the web page
- **URL to open in browser** ➤ type the web page and arguments you want to load to cause the ISAPI to be loaded in IIS
- **OK** ➤ resets IIS, setups all the variables, copies DLLs and settings into the web root and starts the web browser to load the specified web page

 IIS is a protected process and can only execute, read and write files in specific directories. That's why Bug Validator copies data to the web root so that it can be read, written or executed.

3.15.6.2 Resetting and stopping IIS

Resetting IIS

 **Launch** menu > **Services** > **Reset IIS** > resets Internet Information Server (stops it and starts it again).

The session is not discarded when IIS is reset.

Stopping IIS

 **Launch** menu > **Services** > **Stop IIS** > stops Internet Information Server.

The session is not discarded when IIS is stopped.

3.16 Stopping your target program

Stopping the application

You can stop or kill your program at any time using the task manager, or debugger.

You can also stop your program from within Bug Validator.

 **File** menu > **Abandon Application...** > stop the target program

or click on the red cross icon on the session toolbar.



The target program is ended using `ExitProcess()` from inside the stub.

Since the session is discarded, using Bug Validator to stop the target program is usually quicker and more convenient than external stop methods.

➔ You can easily re-launch the program again using the same settings as before.

3.17 Command Line Builder

The command line builder helps you create command lines with valid options.

The command line builder is a two stage process, the first stage helping you choose how you want to build the command line, and the second stage actually building the command line based on the choices in the first stage.

Command Line Builder

How would you like to build your command line?

☒ I'll build my own command line

☐ Use a predefined template for common command line tasks

Native & .Net: Run & Save session

-program "c:\myProgram.exe" -saveSession "c:\myResults\session1.bvm" -hideUI

☐ Use an existing command line I use to launch my program

☐ Use an existing Bug Validator command line

☐ Use an existing Bug Validator command file

Browse...

Next Cancel

There are five options for building your command line:

- **I'll build my own** ➤ you'll build your command line from scratch, with no predefined options
- **Use a predefined template** ➤ choose from a list of predefined command lines that you can customize

The predefined templates cover a range of tasks you may want to perform from the command line. These include running sessions, saving sessions, exporting to HTML and XML.

- **Use an existing command line** ➤ use the command line you use to start the tool you want to collect profiling data for

Example: `e:\dev\paintpot\release\paintpot.exe e:\testimages\venn.png -invert -mirror -phaseChange`

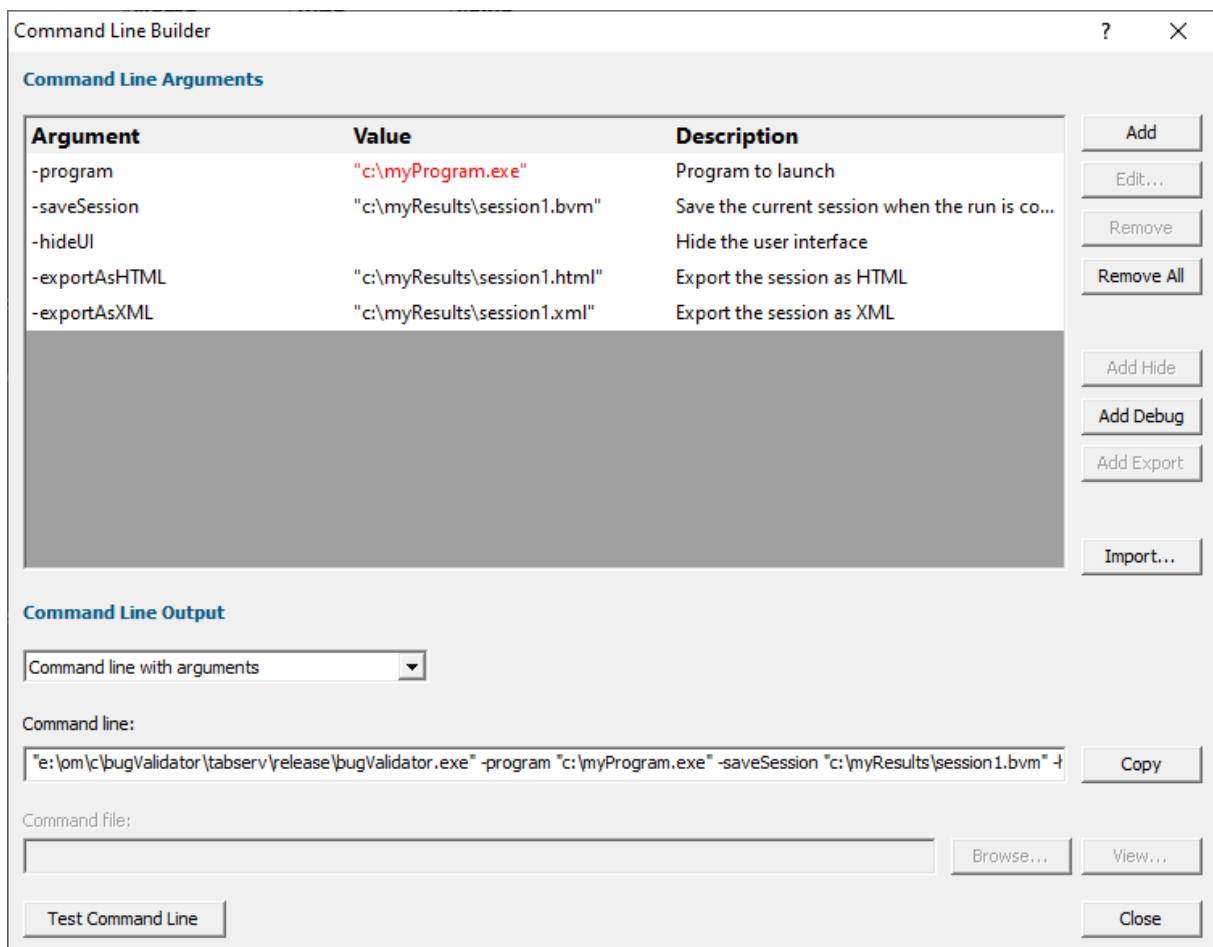
- **Use an existing Bug Validator command line** ➤ use an existing Bug Validator command line and customize that

Example: `-program e:\dev\paintpot\release\paintpot.exe -hideUI -exportAsHTML e:\testResults\gantt.html -allArgs e:\testimages\gantt.png -inflate:3`

- **Use an existing Bug Validator command file** ➤ use an existing Bug Validator command file and customize that

Example: `-commandFile e:\commandFiles\paintpot_gantt.cf`

When you have made your choice the **Next** button moves you to the customization part of the command line builder.



The image above shows the command line builder populated with one of the predefined template choices. You can see a few entries refer to directories and files that do not exist on disk (they are red).

These are items you will need to customize to match the program you are testing.

Any entries that will only exist after they have been created by the test will also be shown in red.

Editing

To edit an argument, double click the argument. A combo box will display a list of valid arguments you can choose.

To edit a value, double the value. If the argument type has a list of known values a combo box will be provided, directories will display a directory chooser, files will display a file chooser, numbers will only allow numeric editing. All other values will be edited as text.

- **Add** ➤ add a new argument to the grid
- **Remove** ➤ remove the selected item
- **Remove All** ➤ removes all items in the grid
- **Add Hide** ➤ adds a -hideUI argument which will cause Bug Validator to hidden when running. Bug Validator will close after the target program finishes running
- **Add Debug** ➤ adds various arguments which will cause Bug Validator to display error messages if there are problems with the command line.
- **Add Export** ➤ adds export options that will cause Bug Validator to export html and/or xml reports after the target program finishes running
- **Import...** ➤ you can import a command file, the contents of which will replace all the items in the grid

Command Line Output

There are two command line output styles.

- **Command line with arguments** ➤ generates a command line containing all arguments and values shown in the grid
- **Command line with command file** ➤ generates a command file containing all arguments and values shown in the grid, and a command line that references the command file

When this option is chosen the command file edit field and the **Browse...** and **View...** buttons are enabled, allowing you to specify a command file name, and to view it's contents.

If a command file has not been specified when this option is selected you will be prompted to select a name for the command file.

When the command file name is selected the command file will be created with the arguments and values shown in the grid.

- **Copy** ➤ copies the command line to the clipboard so that you can paste the command line in cmd prompts, batch files and automated scripts (Jenkins etc)
- **Browse...** ➤ opens a Windows file dialog to allow you to specify the command file location

- **View...** ➤ opens the command file using the Windows shell, this allows you to view the command file in your favourite editor

Testing

If you wish to test the command line, you have two options:

- **Manual test** ➤ use the **Copy** button to copy the command line, then paste it into a cmd prompt and press return.
- **Test Command Line** ➤ a new instance Bug Validator is started with the specified command line.

3.18 Data Collection

Collecting data

Once you've launched or injected into a program, you can stop and start data collection whilst the program is running.

This is a high level switch that controls *all* data collection, regardless of any other settings.

With data collection off, the target program runs at close to normal speed.

Temporarily turning off collection can be a good idea if you need to take actions to get the program into the right state for validation.

You can also turn data off from the start and only turn it on when you need it.

Starting and stopping data collection

 **File** menu ➤ **Start collecting data...** ➤ starts collecting data immediately

or click on the green arrow icon on the session toolbar to start collecting.



 **File** menu ➤ **Stop collecting data...** ➤ stops collecting

or click on the orange pause icon on the session toolbar to stop collecting.

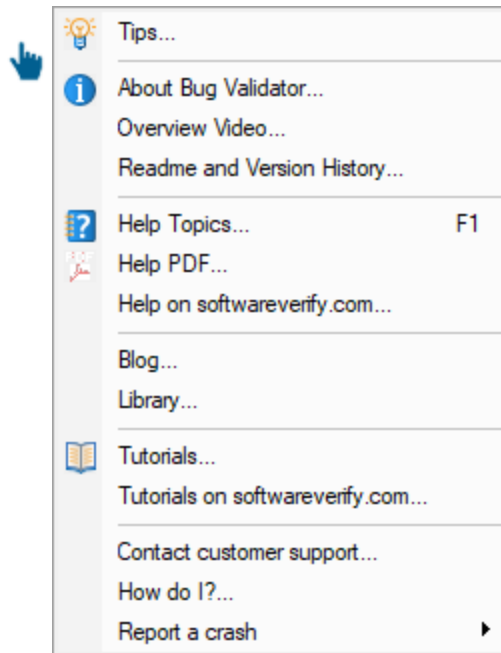


3.19 Help

The help menu

The help menu provides access to useful help, tips and tutorials.

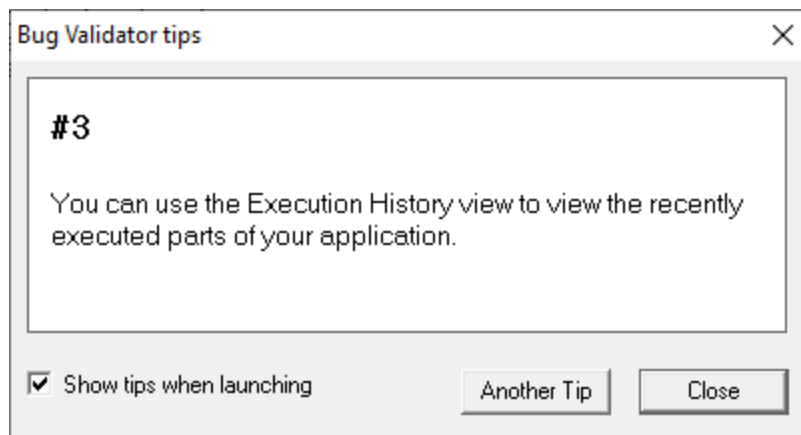
Each item is covered briefly below, in menu order.



Tips

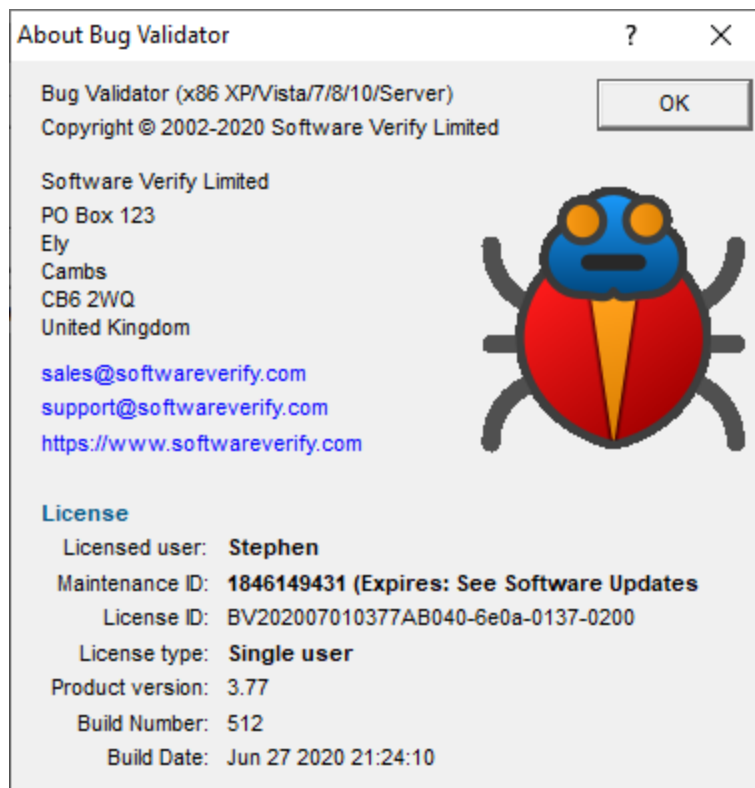
 **Help** menu > **Tips...** > shows the tip dialog where you can browse tips in random order

Here you can also choose whether to display the tips dialog while launching programs.



About box

 **Help** menu > **About Bug Validator...** > shows contact and copyright information, as well as details of your license



Overview video

 **Help** menu > **Overview video...** > displays the Bug Validator overview video.

Readme and version history

 **Help** menu > **Readme...** > opens the readme.html (from your installation) in your browser.

The readme file contains all the latest information about Bug Validator including:

- basic information about getting started and where to go for support
- known issues
- version history

To see what's changed since the version you have installed see the latest version history [🔗](#).

Help HTML

 **Help** menu > **Help Topics...** > shows the HTML help dialog

You might be reading this right now!.

Or click on the question mark icon on the standard toolbar:



 The **F1** key also shows the help, but has the added bonus of jumping directly to the page relevant for the current view or dialog.

 We occasionally get reports of customers seeing exception errors while viewing the HTML help. Unfortunately, we don't have a solution for this!

Help PDF

 **Help** menu > **Help PDF** > shows the PDF version of this help. This will open in your web browser on most computers.

PDF help for *all* our software tools are online [🔗](#).

Help on softwareverify.com

 **Help** menu > **Help on softwareverify.com** > shows the online version of this help [🔗](#)

Blog

 **Help** menu > **Blog** > shows the Software Verify Blog [🔗](#).

Library

 **Help** menu > **Library** > shows the Software Verify Library - all our best articles organised into related topics  for easy access.

Tutorials

The tutorials are intended to guide you through learning how to use aspects of Bug Validator.

Latest tutorials are available online  in the form of short videos and examples covering popular topics.

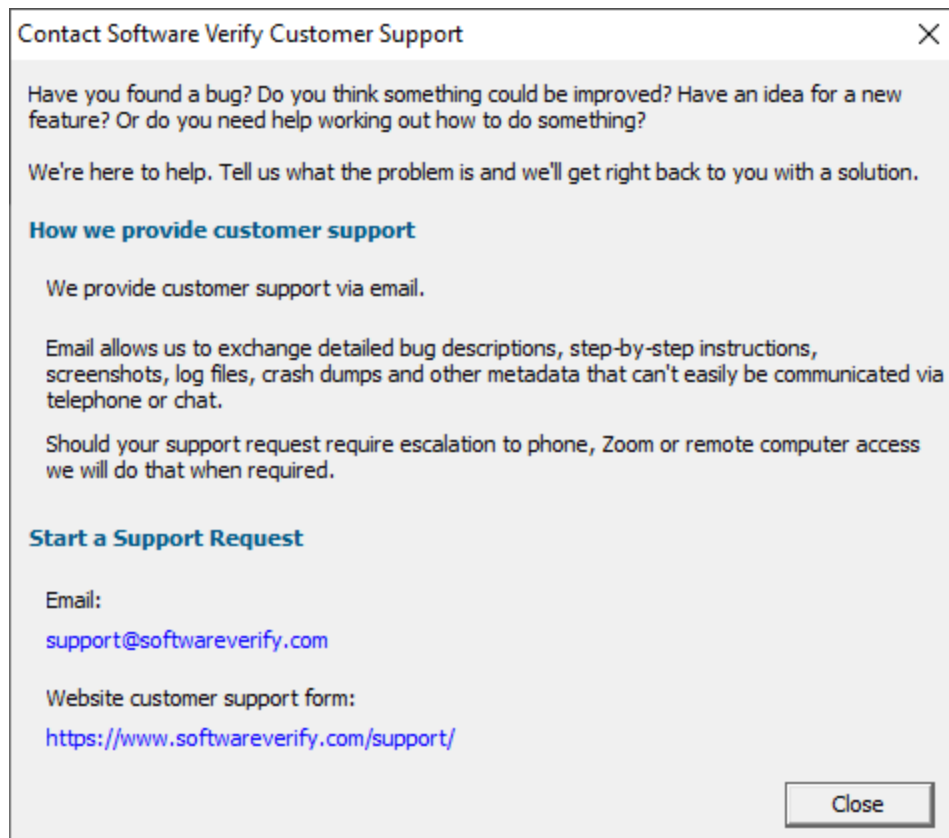
 **Help** menu > **Tutorials** > simply selects the Tutorials tab to show a list of the tutorials

Double click on the row of a tutorial in the list to open it in a browser.

 **Help** menu > **Tutorials on softwareverify.com...** > opens the online tutorials  in a browser

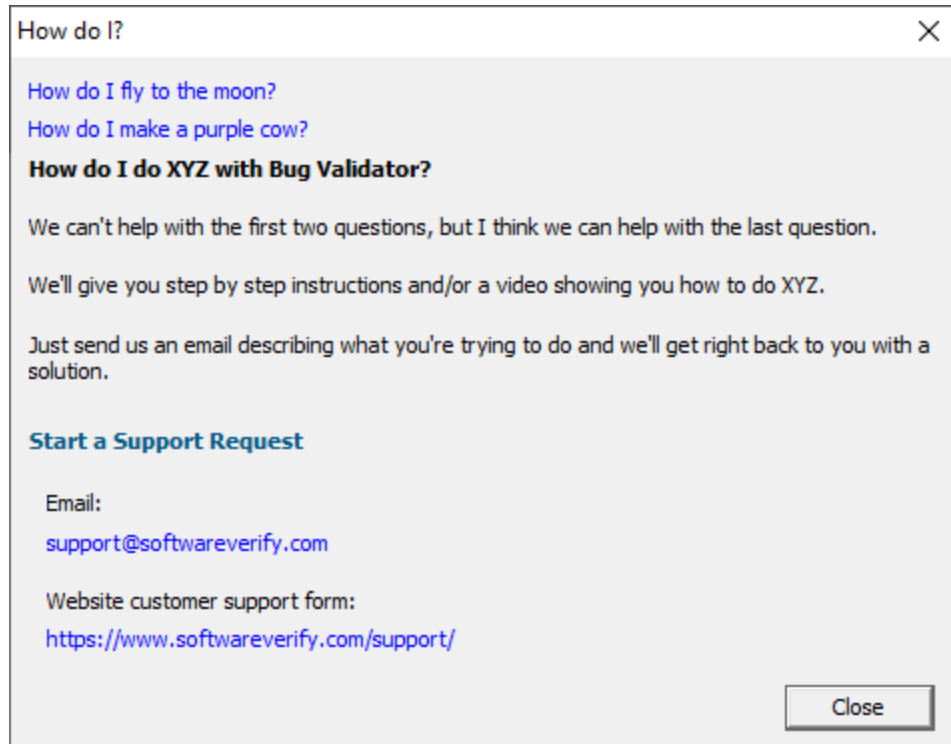
Contact customer support

 **Help** menu > **Contact customer support** > shows the Contact customer support  dialog.



How do I?

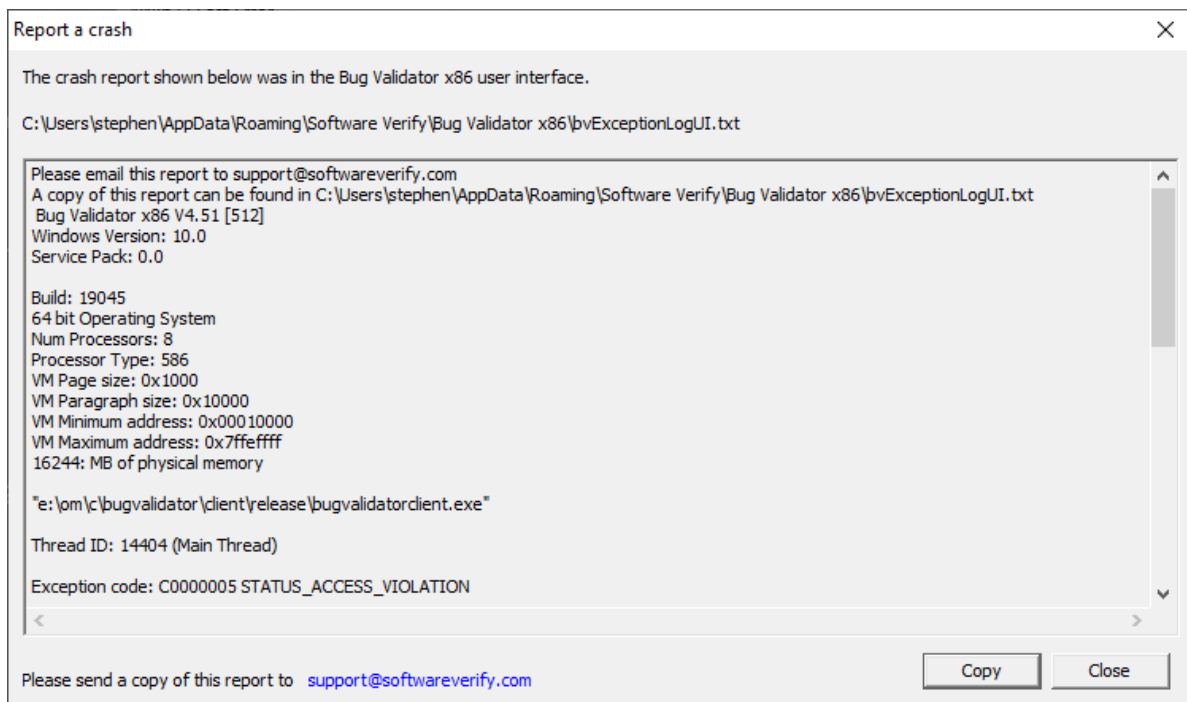
 **Help** menu > **How do I?** > shows the How do I? dialog.



Report a crash

 **Help** menu > **Report a crash** > displays the options for reporting a crash.

If an exception report for the Bug Validator user interface, or an exception report for an application that Bug Validator was monitoring is available it will be displayed with options to copy it to the clipboard and contact customer support at Software Verify.



Part

IV

4 Command Line Interface

Bug Validator provides a command line interface to allow you to perform automated critical section data collection.

To run 32 bit bug validator run `c:\Program Files (x86)\Software Verify\Bug Validator x86\bugValidator.exe`

Command line argument usage

There are a few basic rules to remember when using the command line arguments:

- separate arguments by spaces
- quote arguments if they contain spaces
- some arguments are only useful in conjunction with others
- some arguments are incompatible with others

If your command line is very long, consider using **-commandFile** to specify a command file for your arguments.

Unrecognised arguments

Any unrecognised arguments found on the command line are simply ignored, whether or not they are prefixed with a hyphen.

Arguments intended for your program will not conflict with the Bug Validator arguments in this manual as you should use **-arg** (or **-allArgs**) to redirect them to your program.

Need some help building the command line?

If you find creating command lines from nothing to be a bit daunting we've created a Command Line Builder tool to help you build command lines.

You'll still need to complete some details, but the builder will help prevent you making some mistakes.

4.1 Example Command Lines

Typical command line examples

The following examples demonstrate a few different scenarios in which you might want to use Bug Validator via the command line.

To run 32 bit bug validator run `c:\Program Files (x86)\Software Verify\Bug Validator x86\bugValidator.exe`

Example 1 - running a session and saving the results

This example starts the application, showing no progress dialog whilst attaching to the process.

On completion, the resulting session is saved, and some tabs are refreshed.

The last tab refreshed is displayed, resulting in the ActiveObjects tab being the current tab.

-program "c:\myProgram.exe" -saveSession "c:\myResults\session1.bvm" -displayUI -refreshSummary

A brief explanation of each argument:

Option	Argument	Description
-program	"c:\myProgram.exe"	The target program to launch
-saveSession	"c:\myResults\session1.bvm"	After the application finishes, the session should be saved in this file
-displayUI		Show the user interface during the thread test
-refreshSummary		This tab should be refreshed when the test completes

Example 2 - running a session and saving and exporting the results

This example starts the application, showing no progress dialog whilst attaching to the process.

On completion, the resulting session is saved, and some tabs are refreshed.

The last tab refreshed is displayed, resulting in the ActiveObjects tab being the current tab.

-program "c:\myProgram.exe" -saveSession "c:\myResults\session1.bvm" -exportAsHTML "c:\myResults\session1.html" -displayUI -refreshSummary

A brief explanation of each argument:

Option	Argument	Description
-program	"c:\myProgram.exe"	The target program to launch

-saveSession	"c: \myResults\session1.bvm "	After the application finishes, the session should be saved in this file
-exportAsHTML	"c: \myResults\session1.htm "	After the application finishes, the exported data should be saved in this file
-displayUI		Show the user interface during the thread test
-refreshSummary		This tab should be refreshed when the test completes

4.2 Environment variables

Environment variables can be referenced on the command line.

This allows you to set an environment variable outside of Bug Validator (cmd prompt, batch file, etc) and then reference it on the command line.

For example:

```
-program %BUILD_DIR%\testProgram.exe
```

If the BUILD_DIR environment variable is set to e:\dev\debug the above would evaluate to -
program e:\dev\debug\testProgram.exe

What if I can't set an environment variable?

There are situations where it isn't desirable, or possible to set the environment variable value prior to starting Bug Validator.

In those situations you can set the environment variable on the command line using -setenvironment.

```
-setenvironment BUILD_DIR=e:\dev\debug -program %BUILD_DIR%\testProgram.exe
```

Problems with environment variable substitution

If you are running from a command prompt, or batch file, or any process that will handle environment variable substitution using %ENV_VAR% you will find that referencing the environment variable on the command line won't work when using -setenvironment, because by the time Bug Validator sees the command line the %ENV_VAR% values have already been substituted.

To get around this, using \$ENV_VAR\$ instead of %ENV_VAR%.

```
-setenvironment BUILD_DIR=e:\dev\debug -program $BUILD_DIR$\testProgram.exe
```

-setenvironment

Set environment variables for Bug Validator, as a series of name/value pairs.

Use this option once for each environment variable you wish to set.

Usage of **-setenvironment** for any environment variable must appear on the command line prior to any reference to that environment variable on the command line.

 To pass quotes along with the string, escape a pair of inner quotes like the example below

Examples:

```
-setenvironment APP_FLAG=ON;  
-setenvironment "APP_FAG=ON;"  
-setenvironment "APP_COMMS=ON; APP_DEBUG=OFF;"  
-setenvironment "APP_MSG=\"A quoted string with spaces\";"  
-setenvironment BUILD_DIR=e:\dev\debug
```

Note that this is not the same as `-environment`, which allows you to specify environment values that you can pass to the program being launched.

4.3 Development environment

The following options allow you to specify the development environment you used to build the application being tested.

-devIDE

Specifies the development environment or compiler used to build the application being tested.

These values correspond to the values on the Symbol Lookup part of the settings dialog.

- clang
- codeWarrior
- cppBuilder
- delphi
- devC++
- mingw
- other
- rust
- salfordFortran
- visualBasic6
- visualStudio. If you specify this you also need to specify **-devVisualStudioVersion** or **-devVisualStudioYear**
- visualStudioCustomDLL
- visualStudioDontSet

Examples:

-devIDE visualStudio
-devIDE delphi

-devVisualStudioVersion

If **-devIDE** has been specified as **visualStudio** then **-devVisualStudioVersion** can be used to specify the version of Visual Studio.

The version of Visual Studio needs to be installed on the machine for this to work.

The following versions are valid:

- 6
- 7
- 7.1
- 8
- 9
- 10
- 11
- 12
- 14
- 15
- 16
- 17

Examples:

-devVisualStudioVersion 6
-devVisualStudioVersion 10
-devVisualStudioVersion 17

*If you use **-devVisualStudioVersion** then you don't need to use **-devVisualStudioYear**.*

-devVisualStudioYear

If **-devIDE** has been specified as **visualStudio** then **-devVisualStudioYear** can be used to specify the version of Visual Studio.

The version of Visual Studio needs to be installed on the machine for this to work.

The following years are valid:

- 1996
- 2002
- 2003
- 2005
- 2008
- 2010
- 2012
- 2013
- 2015

- 2017
- 2019
- 2022

Examples:

```
-devVisualStudioYear 1996  
-devVisualStudioYear 2010  
-devVisualStudioYear 2022
```

*If you use **-devVisualStudioYear** then you don't need to use **-devVisualStudioVersion**.*

4.4 Target Program & Start Modes

Resetting global settings

-resetSettings

Forces Bug Validator to reset all settings to the default state, except for any configured colours and the UI Global Hook settings which must be reset manually.



If using this option, it's recommended that you list this *first* on your command line or in your command file.

Specifying the target application

The following options let you launch a program (with various start-up modes), inject into a running program or wait for a program to start before attaching.

Launching a program

-program

Specifies the full file system path of the executable target program to be started by Bug Validator, *including any extension*.

Not compatible with **-injectName**, **-injectID**, **-waitName** or **-monitorAService**.

See **-arg** below to pass arguments to your program, and **-directory** to set where it runs.

Examples:

```
-program c:\testbed.exe  
-program "c:\new compiler\version2\testbed.exe"
```

If you specify the file *without* a path then:

- If you used **-directory** to set a startup directory then the filename in that directory is used if it exists

- Otherwise, the directories in the PATH environment variable are used to look for the filename

-programToMonitorEXE

-programToMonitor

-programToMonitor has been replaced by **-programToMonitorEXE**. **-programToMonitor** will be honoured to provide backward compatibility.

Specifies the full path of the program from which the data is collected, but *does not change* which process is initially launched. *Include the extension*.

This program will be monitored by Bug Validator only when the program specified using **-program** starts it.

If no path is specified, the first child process that has the same name will be monitored.

To monitor *any* program that is launched specify `<<Any>>` as the program argument. In batch files and powershell scripts you will need to quote this to get it accepted by the file parser.

See **-programToMonitorLaunchCount** to change which instance of the program is monitored.

Only valid in conjunction with **-program**.

Examples:

```
-programToMonitorEXE c:\testbed-child-process.exe
-programToMonitorEXE "c:\new compiler\version2\testbedChildProcess.exe"
-programToMonitorEXE testbed-child-process.exe
-programToMonitorEXE "<<Any>>"
```

```
-program c:\testbed.exe -programToMonitorEXE c:\testbed-child-process.exe
```

In this last example `c:\testbed.exe` is launched but not monitored. Only when `testbed.exe` launches a child process `c:\testbed-child-process.exe` is that child process monitored.

-programToMonitorDLL

This option provides a qualifying DLL to identify different .Net Core processes, which are typically launched using the same .Net Core runtime. *Include the dll extension*.

Only valid in conjunction with **-program** and **-programToMonitorEXE**.

Examples:

```
-programToMonitorDLL c:\test\dotNetCoreApp.dll
```

```
-program c:\testbed.exe -programToMonitorEXE "c:\program files\dotnet\dotnet.exe" -
programToMonitorDLL c:\test\dotNetCoreApp.dll
```

In this last example `c:\testbed.exe` is launched but not monitored. Only when `testbed.exe` launches a child process `c:\program files\dotnet\dotnet.exe` to run the application `c:\test\dotNetCoreApp.dll` is that child process monitored.

-programToMonitorLaunchCount

Specify the n^{th} invocation of the child program specified by `-programToMonitor` which is to have its data collected.

Any value which is invalid (including anything less than 1) will default to 1.

Only valid in conjunction with `-programToMonitor` and consequently also `-program`.

Examples:

```
-programToMonitorLaunchCount 1  
-programToMonitorLaunchCount 34
```

```
-program c:\testbed.exe -programToMonitor c:\testbed-child-process.exe -  
programToMonitorLaunchCount 1
```

In the above example `c:\testbed.exe` is launched but not monitored. As soon as `testbed.exe` launches a child process `c:\testbed-child-process.exe` then that child process is monitored.

If the value 1 was changed to a 2, then only the second invocation of `c:\testbed-child-process.exe` would get monitored, with the first invocation being ignored.

-environment

Environment variables for program, as a series of name/value pairs. Not to be confused with `-setenvironment`.

Use this option once for each environment variable you wish to set.

 To pass quotes along with the string, escape a pair of inner quotes like the example below

Only valid with: `-program`

Examples:

```
-environment APP_FLAG=ON;  
-environment "APP_FAG=ON;"  
-environment "APP_COMMS=ON; APP_DEBUG=OFF;"  
-environment "APP_MSG=\"A quoted string with spaces\";"
```

-arg

Passes the following element on the command line to the target program.

`-arg` can be used multiple times, or you can use `-allArgs`



To pass quotes along with the string, escape a pair of inner quotes like the example below

Only valid with: -program

Examples:

```
-arg myProgram.exe  
-arg "c:\Program Files\myApp\myProgram.exe"  
-arg "-in infile -out outfile"  
-arg "\"a quoted string\""
```

-allArgs

Passes the remainder of the command line (after -allArgs) to the program being launched.

Unlike -arg above, there is no need to escape the quotes as the content is passed verbatim.

Only valid with: -program

Example:

-allArgs anything put here is passed to the target program "even stuff in quotes" is passed

-directory

Sets the working directory in which the program is executed. If -directory is not specified the program is run in its current directory.

Only valid with: -program

Examples:

```
-directory c:\development\  
-directory "c:\research and development\"
```

-stdin

Specifies a file to be read and piped to the standard input of the application being tested.

If the filename contains spaces, the filename should be quoted.

An error occurs if the file does not exist. See **-ignoreMissingStdin** to avoid this error.

Examples:

```
-stdin c:\settings\input.txt  
-stdin "c:\reg tests settings\input.txt"
```

-stdout

Specifies a file to be written with data piped from the standard output of the application being tested.

If the filename contains spaces, the filename should be quoted.

An error occurs if the file does not exist. See **-ignoreMissingStdout** to avoid this error.

Examples:

```
-stdout c:\settings\output.txt  
-stdout "c:\reg tests results\output.txt"
```

-ignoreMissingStdin

If this flag is specified and the file specified by **-stdin** does not exist, no error is reported.

-ignoreMissingStdout

If this flag is specified and the file specified by **-stdout** does not exist, no error is reported.

Injecting into a program

-injectName

Sets the name of the process for Bug Validator to attach to.

Not compatible with **-program**, **-injectID**, **-waitName** or **-monitorAService**.

Examples:

```
-injectName c:\testbed.exe  
-injectName "c:\new compiler\version2\testbed.exe"
```

-injectID

Sets the numeric id of a process for Bug Validator to attach to.

Not compatible with **-program**, **-injectName**, **-waitName** or **-monitorAService**.

Example:

```
-injectID 1032
```

Waiting for a program

-waitNameEXE

-waitName

-waitName has been replaced by **-waitNameEXE**. **-waitName** will be honoured to provided backwards compatibility.

Sets the name of a process that Bug Validator will wait for.

When the named process starts Bug Validator will attach to the process.

Not compatible with -program, -injectName, -injectID or -monitorAService.

Examples:

```
-waitNameEXE c:\testbed.exe  
-waitNameEXE "c:\new compiler\version2\testbed.exe"
```

-waitNameDLL

Sets the name of a process DLL that Bug Validator will wait for.

When the named process starts Bug Validator will attach to the process.

Examples:

```
-waitNameDLL c:\dotNetApp.dll  
-waitNameDLL "c:\new compiler\version2\dotNetApp.dll"
```

For use with -waitNameEXE when you want to wait for .Net Core applications.

```
-waitNameEXE "c:\program files\dotnet\dotnet.exe" -waitNameDLL "c:  
\testApps\dotNetCoreApp\release\dotNetCoreApp.dll"
```

Monitoring a service

-monitorAService

Sets the full file system path of a service *including any extension*.

Bug Validator will wait for the service to start and attach to it.

Not compatible with -program, -injectName, -injectID or -waitName.

Examples:

```
-monitorAService c:\service.exe  
-monitorAService "c:\new compiler\version2\service.exe"
```

Data Collection

-collectData

Enables or disables the collection of flow tracing data

Examples:

-collectData:On
-collectData:Off

-collectStdout

Enables or disables the collection of standard output (stdout)

Examples:

-collectStdout:On
-collectStdout:Off

4.5 User interface visibility

User interface visibility

You can choose to hide or show Thread Validator during the test, as well as the window of the target application.

-displayUI

Forces the Bug Validator user interface to be displayed during the test.

This is useful for debugging a command line session that is not working, for example inspecting the Diagnostic tab for messages related to the test.

You wouldn't normally use this option when running unattended thread tests.

-doNotInteractWithUser

Never display dialog boxes in the target application that is being profiled.



This applies even for warning and error dialog boxes.

The intended use for this option is for when you are running command line sessions on unattended computers and you have automated processes that may kill the Bug Validator user interface if something goes wrong. Actions such as this then cause the stub to recognise the user interface has gone away and display an error warning.

-hideUI

Hides the Bug Validator user interface during the test.

-launchAppHide

-launchAppHidden (for backwards compatibility only)

Hides the target application during the test.



Depending on your application, this may not work and may not even be suitable.

This is equivalent to setting the `wShowWindow` member of the `STARTUPINFO` struct to `SW_HIDE` when using the Win32 `CreateProcess()` function.

It's useful if you're testing console applications that have no user interaction, as it prevents the console/command prompt from being displayed.

For GUI applications this option very much depends on how your application works.

For interactive applications, it clearly has no use, but for some, hiding the GUI may help prevent various windows messages from being processed.

Typically, for complex applications, it's better to design this capability into your application and control it via a command line, which can be passed in from Bug Validator via the **-arg** option.

-launchAppShow

Shows the target application during the test.

This is equivalent to setting the `wShowWindow` member of the `STARTUPINFO` struct to `SW_SHOW` when using the Win32 `CreateProcess()` function.

-launchAppShowMaximized

-launchAppShowMinimized

-launchAppShowMinNoActive

-launchAppShowNA

-launchAppShowNoActivate

-launchAppShowNormal

As well as the previous two options to show or hide the target application during the test there are other options equivalent to values that can be used in the `STARTUPINFO` struct.

The options are equivalent to the setting the `wShowWindow` member to the following values

Option	<code>wShowWindow</code> member	Launched application is shown...
-		
-launchAppShowMaximized	<code>SW_SHOWMAXIMIZED</code>	Maximized and activated
zed		

- launchAppShowMinimized	SW_SHOWMINIMIZED	Minimized and activated
- launchAppShowMinNoActive	SW_SHOWMINNOACTIVE	Minimized and not active
-launchAppShowNA	SW_SHOWNA	Shown at current size and position but not activated
- launchAppShowNoActivate	SW_SHOWNOACTIVATE	Show at most recent size and position but not activated
- launchAppShowNormal	SW_SHOWNORMAL	Show at original size and position and activated

-showCommandPrompt
-hideCommandPrompt

Causes any launched console window to be shown or hidden during the test.

-showErrorsOnCommandPrompt

If an error occurs when launching the application, the error will be reported on the command line.

Refreshing the interface after test completion

You can run automated tests that leave the user interface open after completion,

The following options are used to automatically refresh the main data tabs in Bug Validator once a test is complete.

-refreshSummary

4.6 Session Management

Session management

The following options let you control the sessions during testing

-numSessions

Sets the number of sessions that can be loaded at once.

This is equivalent to the same setting in the session manager and can't be less than 1.

Example:

```
-numSessions 2
```

-saveSession

Saves the session data when all data has finished being collected from the target program.

Bug Validator 32 and 64 bit use the file extension .bvm and .bvm_64 respectively.

A missing or incorrect filename extension will be corrected automatically

Examples:

```
-saveSession c:\results\testMacro1.bvm  
-saveSession "c:\test results\testMacro1.bvm_x64"  
-saveSession c:\results\testMacro1
```

-sessionLoad

-sessionLoad loads a previously created session to be merged with the data from the session being *recorded*.

These options might be used when a series of tests have already been performed and their sessions saved.

Bug Validator 32 and 64 bit use the file extension .bvm and .bvm_64 respectively.

A missing or incorrect filename extension will be corrected automatically

Examples:

```
-sessionLoad c:\results\testMacro1.bvm  
-sessionLoad "c:\test results\testMacro1.bvm"  
-sessionLoad c:\results\testMacro1
```



Ensure your session manager is configured to hold at least 2 or 3 sessions or use **-numSessions** to specify how many sessions to use.

4.7 Session Export options

Session export format - HTML or XML

```
-exportAsHTML  
-exportAsXML
```

Export the session data as an HTML or XML file when Bug Validator has finished collecting data from the target program.

If you merge the current session with another session, the exported HTML will be for the *merged* session.

If you disable merging with the current session the export will be for the *unmerged* session.

Example:

```
-exportAsHTML c:\results\html\testMacro1.html  
-exportAsXML "c:\test results\xml\testMacro1.html"
```

Session export encoding - HTML or XML

These options allow you to export the session data as UTF-16, UTF8 or ASCII. UTF-16 and UTF-8 will add a byte order mark (BOM) to the start of the exported file.

-exportAsHTML_BOM

The exported HTML will be exported with the appropriate format.

```
-exportAsHTML_BOM ASCII  
-exportAsHTML_BOM UTF8  
-exportAsHTML_BOM UTF16
```

-exportAsXML_BOM

The exported XML will be exported with the appropriate format.

```
-exportAsXML_BOM ASCII  
-exportAsXML_BOM UTF8  
-exportAsXML_BOM UTF16
```

4.8 Filter and Hook options

Source file hooks

-sourceFileFilterHookFile

Points to a file specifying the source files to be hooked for the test.

If the filename contains spaces, the filename should be quoted.

 The settings dialog Source Files Filter tab provides options for creating a ready made source file filter hook file by using the **Export...** button on the dialog.

Examples:

```
-sourceFileFilterHookFile c:\settings\testMacroFiles.bvxf  
-sourceFileFilterHookFile "c:\reg tests settings\testMacroFiles.bvxf"
```

Source file filter format

The first line of text in the DLL hooks file is *one* of the following:

- Rule:DoNotHook ➤ Source files that follow *will not* be hooked. All other files will be hooked
- Rule:DoHook ➤ Source files that follow *will* be hooked. All other files will not be hooked

 Capitalization is important.

The remaining lines list one source file (or source directory) per line.

The file is named with a path or without a path, i.e. the same way that Coverage Validator discovers the path.

Example:

Most files are listed with a full path but constituent files of MFC might be listed without a path.

```
Rule:DoNotHook  
"c:\program files\software verification\coverage validator\examples\nativeExample  
appmodule.cpp
```

Example:

Using paths with and without spaces:

```
Rule:DoHook  
"E:\OM\C\coverage Validator\examples\nativeExample"  
  
Rule:DoHook  
E:\OM\C\coverageValidator\examples\nativeExample
```

Example:

Using environment variables.

```
Rule:DoHook  
%ENV_VAR%\examples\nativeExample
```

Using wildcards.

```
Rule:DoHook  
c:\test\*\nativeExample
```

The file can be ANSI or UNICODE text and paths with spaces do not need quotes.

4.9 File Locations

File Locations

When using the command line it's convenient to store settings and options in files that can be easily referenced.

Those files include:

- Global settings files
- File locations for source, PDB or MAP files
- DLL hook files

Each of these file types can be saved or exported from Bug Validator.

The **-settings** option is used to specify the settings to be used for the test. If the filename contains spaces, the filename should be quoted. This option is the same as **-loadSettings** and is provided for backwards compatibility.

Loading global settings from a file

Global settings are usually stored in the registry, but you can save a specific set of settings for use in thread tests:

- **Settings** menu > **Save settings...**

-loadSettings

-settings

Points to a previously saved settings file to be used for the test.

Examples:

```
-loadSettings c:\settings\testMacro1.bvs  
-loadSettings "c:\flowTraceSettings\testMacro1.bvs"
```



The **-settings** option is identical to **-loadSettings** and is provided for backwards compatibility

File locations for source, PDB or MAP files

File location files can be easily generated by exporting file locations from the File Locations page of the settings dialog.

-fileLocations

Specify a plain text file listing file locations to be used during testing. See the format of the file below.

Each set of file types (one per line) is preceded by a header line in the file.

- **[Files]** > Source files
- **[Third]** > Third party source files
- **[PDB]** > PDB files
- **[MAP]** > MAP files

Example:

-fileLocations c:\flowTraceTests\testFileLocations1.bvxf

Example file:

```
[Files]
c:\work\project1\
[Third]
d:\VisualStudio\VC98\Include
[PDB]
c:\work\project3\debug
c:\work\project3\release
[MAP]
c:\work\project3\debug
c:\work\project3\release
```

Files listing DLLs to hook

DLL hook files can be easily generated by exporting DLL hooks from the Hooked DLLs page in the Filters section of the settings dialog.

-dllHookFile

Points to a file listing the DLLs to be hooked for the test.

Examples:

```
-dllHookFile c:\settings\testMacroDLLs.bvx
-dllHookFile "c:\flowTraceSettings\testMacroDLLs.bvx"
```

The first line of text in the DLL hooks file is one of the following:

- Rule: DoNot Hook > DLLs marked as enabled *will not* be hooked. All other DLLs will be hooked
- Rule: DoHook > DLLs marked as enabled *will* be hooked. All other DLLs will not be hooked
- Rule: HookAll > All DLLs will be hooked regardless of the settings in the list



Capitalization is important.

The remaining lines list one DLL filename or folder path and an enabled state on each line.

Example:

```
Rule:DoNotHook
nativeExample.exe enable=FALSE
MFC42D.DLL enable=TRUE
MSVCRTD.dll enable=TRUE
KERNEL32.dll enable=TRUE
ole32.dll enable=TRUE
```

Example:

```
Rule:DoHook
E:\OM\C\bugValidator\examples\nativeExample\DebugNonLink enable=TRUE
```

Example:

```
Rule:DoHook
"E:\OM\C\bugValidator\examples\nativeExample with
spaces\DebugNonLink" enable=TRUE
```

Example:

```
Rule:DoHook
%ENV_VAR%\DebugNonLink enable=TRUE
```

Here, the environment variable `ENV_VAR` is used to replace the text `%ENV_VAR%` in the path definition.

The file can be ANSI or UNICODE text and paths with spaces do not need quotes.

4.10 Command Files

Using a command file

If your command line is very long, consider using **-commandFile** to specify a command file for your arguments.

-commandFile

Specify a file from which to read the command line arguments.

Useful when command lines become unwieldy or longer than the windows command size limits.

Use `--` to insert comments into the file, including when commenting out option.

Examples:

```
-commandFile c:\flowTraceTests\testMacro1.cf  
-commandFile "c:\flowTraceTests\testMacro1.cf"
```

Example command file

```
-hideUI  
-program c:\testbed\testApp.exe  
  
-- arguments for application  
-arg argumentOne  
-arg argumentTwo  
-arg "-s wobble"  
-directory c:\testbed\test1  
-settings c:\testbed\settings_test1.bvs  
  
-- do export and save of the results  
-exportAsHTML c:\testbed\results\test1.html  
-saveSession c:\testbed\results\test1.bvm
```

For any argument that can be supplied to a command in a command file, you can also specify an environment variable substitution.

```
-directory %DIR%  
-program %DIR%\testProgram.exe
```

The environment variables must have been set prior to starting Bug Validator.

You cannot specify a command with an environment variable substitution.

4.11 Help, Errors & Return Codes

The following options may help with using and debugging the command line driven automated regression testing.

Command line help

```
-help  
-?
```

Command line help is printed on the standard output.

-echoArgsToUser

Show the arguments that were supplied to the program.

Debugging command driven testing

If you're having problems with using the command line, check the following, try displaying error messages using the option below, and look at the exit return codes.

- separate command line arguments with spaces
- all command line options that include spaces need to have quotes around them
- some arguments are only useful in conjunction with others - check notes against each option
- some arguments are incompatible with others - check notes against each option

-showErrorsWithMessageBox

Forces errors to be displayed using a message box when running from the command line.

This can be very useful when debugging a command line that does not appear to work correctly.

Exit return codes

Bug Validator returns the following status codes when running from the command line. Some of these status codes are for internal use, or are obsolete (retired products).

- **0** > All ok
- **-1** > Unknown error. An unexpected error occurred starting the runtime
- **-2** > Application started ok. You should not see this code returned
- **-3** > Application failed to start. E.g. runtime not present, not an executable or injection dll not present,
- **-4** > Target application is not an application
- **-5** > Don't know what format the executable is, cannot process it
- **-6** > Not a 32 bit application
- **-7** > Not a 64 bit application
- **-8** > Using incorrect MSVCR(8|9).DLL that links to CoreDLL.dll (incorrect DLL is from WinCE)
- **-9** > Win16 app cannot start these because we can't inject into them
- **-10** > Win32 app - not used
- **-11** > Win64 app - not used
- **-12** > .Net application
- **-13** > User bailed out because app not linked to MSVCRT dynamically
- **-14** > Not found in launch history
- **-15** > DLL to inject was not found
- **-16** > Startup directory does not exist
- **-17** > Symbol server directory does not exist
- **-18** > Could not build a command line
- **-19** > No runtime specified, cannot execute script (or Java) (obsolete)
- **-20** > Java arguments are OK - not an error (obsolete)
- **-21** > Java agentlib supplied that is not allowed because Java Bug Validator uses it (obsolete)
- **-22** > Java xrun supplied that is not allowed because Java Bug Validator uses it (obsolete)
- **-23** > Java cp supplied that is not allowed because Java Bug Validator uses it (obsolete)
- **-24** > Java classpath supplied that is not allowed because Java Bug Validator uses it (obsolete)
- **-25** > Firefox is already running, please close it (obsolete)
- **-26** > Lua runtime DLL version is not known (obsolete)
- **-27** > Not compatible software
- **-28** > InjectUsingCreateProcess, no DLL name supplied

- -29 > InjectUsingCreateProcess, Unable to open PE File when inspecting DLL
- -30 > InjectUsingCreateProcess, Invalid PE File when inspecting DLL
- -31 > InjectUsingCreateProcess, No Kernel32 DLL
- -32 > InjectUsingCreateProcess, NULL VirtualFree() from GetProcAddress
- -33 > InjectUsingCreateProcess, NULL GetModuleHandleW() from GetModuleHandleW
- -34 > InjectUsingCreateProcess, NULL LoadLibraryW() from LoadLibraryW
- -35 > InjectUsingCreateProcess, NULL FreeLibrary() from FreeLibrary
- -36 > InjectUsingCreateProcess, NULL VirtualProtect() from GetProcAddress
- -37 > InjectUsingCreateProcess, NULL VirtualFree() from GetProcAddress
- -38 > InjectUsingCreateProcess, unable to find DLL load address
- -39 > InjectUsingCreateProcess, unable to write to remote process's memory
- -40 > InjectUsingCreateProcess, unable to read remote process's memory
- -41 > InjectUsingCreateProcess, unable to resume a thread
- -42 > UPX compressed - cannot process such executables
- -43 > Java class not found in CLASSPATH
- -44 > Failed to launch the 32 bit svGetProcAddressHelperUtil.exe
- -45 > Unknown error with svGetProcAddressHelperUtil.exe
- -46 > Couldn't load specified DLL into svGetProcAddressHelperUtil.exe
- -47 > Couldn't find function in the DLL svGetProcAddressHelperUtil.exe
- -48 > Missing DLL argument svGetProcAddressHelperUtil.exe
- -49 > Missing function argument svGetProcAddressHelperUtil.exe
- -50 > Missing svGetProcAddressHelperUtil.exe
- -51 > Target process has a manifest that requires elevation
- -52 > svInjectIntoProcessHelper_x64.exe not found
- -53 > svInjectIntoProcessHelper_x64.exe failed to start
- -54 > svInjectIntoProcessHelper_x64.exe failed to return error code
- -55 > getImageBase() worked ok
- -56 > ReadFile() failed in getImageBase()
- -57 > NULL pointer when trying to allocate memory
- -58 > CreateFile() failed in getImageBase()
- -59 > ReadProcessMemory() failed in getImageBase()
- -60 > VirtualQueryEx() failed in getImageBase()
- -61 > Bad /appName argument in svInjectIntoProcessHelper_x64.exe
- -62 > Bad /dllName argument in svInjectIntoProcessHelper_x64.exe
- -63 > Bad /procId argument in svInjectIntoProcessHelper_x64.exe
- -64 > Failed to OpenProcess in svInjectIntoProcessHelper_x64.exe
- -65 > A DLL that the .exe depends upon cannot be found
- -66 > A stdin file was specified, but Validator could not open it
- -67 > A stdout file was specified, but Validator could not open it
- -68 > Failed to create the child output pipe
- -69 > Failed to create a duplicate of the output write handle for the std error write handle. This is necessary in case the child application closes one of its std output handles
- -70 > Failed to create the child input pipe
- -71 > Failed to create a duplicate output read temporary file
- -72 > Failed to create a duplicate input write temporary file
- -73 > User was trying to launch a service as an application that was linked to TV APIs. User cancelled when informed of this fact
- -74 > Returned by Bug Validator if user performs a baseline comparison and memory leaks are detected
- -75 > Shutdown and restart as 32 bit Bug Validator
- -76 > Shutdown and restart as 64 bit Bug Validator
- -77 > Entry point in executable is NULL
- -78 > Application is .Net Core

- **-79** ➤ Entry point is for a .Net application
- **-80** ➤ VirtualAllocEx() returned NULL
- **-81** ➤ InjectUsingCreateProcess, NULL GetLastError() from GetProcAddress
- **-82** ➤ Starting application. Used internally by our automated test harness. Shown here for completeness. You should never encounter this error code
- **-83** ➤ Not implemented yet in this product
- **-84** ➤ Not supported in this product
- **-85** ➤ An executable or DLL uses Address Sanitizer and the Validator is not compatible with Address Sanitizer
- **-86** ➤ An attempt to launch, inject into, or wait for an IIS executable was made. The user was shown links to documentation demonstrating the correct way to work with IIS

4.12 Command Line Reference

Command line reference

The following alphabetical list provides a convenient look-up for all the command line arguments used in automated regression testing.

Option	Description
-?	Print command line help on the standard output.
-allArgs	Pass the remainder of the command line to the program being launched.
-arg	Pass command line arguments to the target program. Can be used multiple times.
-collectData	Turn data collection on or off
-collectStdout	Turn collection of stdout on or off
-commandFile	Specify a file from which to read the command line arguments.
-devIDE	Specify the development environment used to be the target program.
-devVisualStudioVersion	Specify which version of Visual Studio by year.
-devVisualStudioYear	Specify which version of Visual Studio by version number.
-directory	Set the working directory in which the program is executed.
-displayUI	Force the Bug Validator user interface to be displayed during the test.
-dllHookFile	Points to a file listing the DLLs to be hooked for the test.
-doNotInteractWithUser	Never display dialog boxes in the target application that is being profiled.
-echoArgsToUser	Show the arguments that were supplied to the program.
-environment	Environment variables for program, as a series of name/value pairs

-exportAsHTML	Export the session data as an HTML or XML file when Bug Validator has finished collecting data from the target program.
-exportAsXML	
-exportAsHTML_BOM	Specify the file encoding for the exported file
-exportAsXML_BOM	
-fileLocations	Specify a plain text file listing file locations to be used during testing. See the format of the file below.
-help	Print command line help on the standard output.
-hideCommandPrompt	Any launched console window will be hidden during the test.
-hideUI	Hide the Bug Validator user interface during the test.
-injectID	Set the numeric (decimal) id of a process for Bug Validator to attach to.
-injectName	Set the name of the process for Bug Validator to attach to.
-launchAppHide	Hide the target application during the test.
-launchAppShow	Show the target application during the test.
-launchAppShowMaximized	Show the target application maximized and activated.
-launchAppShowMinNoActive	Show the target application minimized and activated.
-launchAppShowMinimized	Show the target application minimized and not active.
-launchAppShowNA	Show the target application at current size and position but not activated.
-launchAppShowNoActivate	Show the target application at most recent size and position but not activated.
-launchAppShowNormal	Show the target application at original size and position and activated.
-loadSession	Load a previously created session to be merged with the data from the session being recorded.
-loadSettings	Points to a previously saved settings file to be used for the test.
-monitorAService	Specify the full file system path to the service to monitor, including any extension. The service is not started by Bug Validator but by an external means.
-numSessions	Set the number of sessions that can be loaded at once.
-program	Specify the full file system path of the executable target program to be started by Bug Validator, including any extension.
-programToMonitor	Changes which program the data is collected from but does not change which process Bug Validator initially launches.
-programToMonitorLaunchCount	Specify the n^{th} invocation of the programToMonitor which is to have its data collected.
-refreshSummary	Automatically refresh the Summary once a test is complete.

-resetSettings	Forces Bug Validator to reset (nearly) all settings to the default state.
-saveSession	Save the session data when all data has finished being collecting from the target program.
-sessionLoad	Loads a previously created session to be merged with the data from the session being <i>recorded</i>
-setenvironment	Environment variables for Bug Validator, as a series of name/value pairs
-settings	Points to a previously saved settings file to be used for the test.
-showCommandPrompt	Any launched console window will be shown during the test.
-showErrorsOnCommandPrompt	If an error occurs when launching the application, the error will be reported on the command line.
-showErrorsWithMessageBox	Force errors to be displayed using a message box when running from the command line.
-sourceFileFilterHookFile	Points to a file specifying the source files to be hooked for the test.
-waitName	Name a process that Bug Validator will wait for.

To run 32 bit bug validator run `c:\Program Files (x86)\Software Verify\Bug Validator x86\bugValidator.exe`

4.13 Troubleshooting

Running from the command line can cause some problems, often because you can't be sure that what you put on the command line did what you thought would do.

Ensure the arguments supplied are what you expected.

-echoArgsToUser

If you are testing a console application, make sure you can see it.

-showCommandPrompt

If an errors occur when processing the command line, make sure you can see those.

-showErrorsOnCommandPrompt

-showErrorsWithMessageBox

Look on the diagnostic tab to ensure the diagnostic data collected makes sense.

If you've got `-hideUI` in your command line, comment it out temporarily (make it `-xhideUI` so that it's not recognised).

What if the tool hangs?

If you're running from the command line, most likely you'll be running from a cmd prompt, or possibly powershell.

We've only ever had one customer report a hang with any of our tools when running from the command line.

We eventually found the problem, and it wasn't with the software tool.

The problem was that they were running the tool in hidden mode (-hideUI) from a command prompt and for unknown reasons the tool would never exit.

When they added a simple change to their command the problem went away.

They added **cmd /c** to the start of their command line. This opens a new command prompt and instructs it to launch the command line and wait for it to exit.

Problem command line:

```
"c:\program files (x86)\Software Verify\Bug Validator\bugValidator.exe" -program c:\testProgram.exe
```

Working command line:

```
cmd /c "c:\program files (x86)\Software Verify\Bug Validator\bugValidator.exe" -program c:\testProgram.exe
```

Part

V

5 API

The Bug Validator API

There are some features of Bug Validator that are useful to call directly from your program.

Bug Validator has an API that makes this possible; just include `svlBVAPI.c` and `svlBVAPI.h` to your codebase. There is no library to link to, dlls to copy.

Working with services?

If you are working with services you to attach Bug Validator to a service and to start Bug Validator, you should use the NT Service API, not the functions in this API.

All the other functions in this API can be used with applications and with services.

Unicode or ANSI?

All the API functions are provided in Unicode and ANSI variants where strings are used. We've also provided a character width neutral `#define` in the same fashion that the `Windows.h` header files do.

For example the function for naming a thread is provided as `bvSetThreadNameA()`, `bvSetThreadNameW()` with the character width neutral `bvSetThreadName()` mapping to `bvSetThreadNameW()` for unicode and `bvSetThreadNameA()` for ANSI.

In this document we're going to use `TCHAR` like the `Window.h` header files do.

Deploying on a customer machine

You can use the API without incurring any dependency on Bug Validator.

If Bug Validator is not installed on the machine the software runs on, nothing will happen.

This allows you to add the Bug Validator API to your software without need to have a separate builds for use with and without Bug Validator.

Loading the Profiler

For most use cases won't need to load the profiler, as the profiler will have been loaded when your launched your program from Bug Validator, or when you injected into your program using Inject or Wait For Application.

However if you're running your program from outside of Bug Validator and want to load the profiler from inside your program you can use `bvLoadProfiler()` to do that. You'll then need to call `bvStartProfiler()` to start it.

Starting the Profiler

To start the profiler from your API code you need to call the function `bvStartProfiler()` from your code before you call any API functions. Ideally you should call this function as early in your program as possible.

If you prefer to start the profiler from the user interface or command line you can omit the `bvStartProfiler()` call. You can leave it present if you wish to start Bug Validator from your program.

Naming threads

You can name threads using the `bvSetThreadNameA()` and `bvSetThreadNameW()` functions.

Turning data collection on and off

You can turn data collection on and off using the `bvSetCollect()` function.

You can use `bvGetCollect()` to inspect the data collection status.

5.1 Source Files

The source files can be found in the `API` directory in the Bug Validator install directory.

C / C++ Source files

The source files can be found in the `API` directory in the Bug Validator install directory.

```
svlBVAPI.h  
svlBVAPI.c
```

Just add these files to your project and build.

If you are using precompiled headers you will need to disable them for `svlBVAPI.c`.

5.2 Native API Reference

Unicode or ANSI?

All the API functions are provided in Unicode and ANSI variants where strings are used. We've also provided a character width neutral `#define` in the same fashion that the `Windows.h` header files do.

For example the function for naming a thread is provided as `bvSetThreadNameA()`, `bvSetThreadNameW()` with the character width neutral `bvSetThreadName()` mapping to `bvSetThreadNameW()` for unicode and `bvSetThreadNameA()` for ANSI.

In this document we're going to use `TCHAR` like the `Window.h` header files do.

All the API functions are declared as `extern "C"`, so they can be used by C users and C++ users.

To use these functions `#include` `svlBVAPI.h` into your code.

You do not need to link with any libraries to use the Native API.

bvLoadProfiler

Loads the profiler DLL into memory, but does not start the profiler.

For most use cases won't need to load the profiler, as the profiler will have been loaded when your launched your program from Bug Validator, or when you injected into your program using Inject or Wait For Application.

However if you're running your program from outside of Bug Validator and want to load the profiler from inside your program you can use `bvLoadProfiler()` to do that. You'll then need to call `bvStartProfiler()` or `bvStartProfilerWithOwnGUI()` to start it.

```
extern "C"
int bvLoadProfiler();
```

Returns:

```
TRUE    Successfully loaded BV DLL into target application.
FALSE   Failed to load the BV DLL into target application.
        Check that the PATH environment variable points to the Bug Validator
        install directory contains svlBugValidatorStub*.dll.
```

Do not use this function if you are working with services, use the **NT Service API**.

bvStartProfiler

To start the profiler from your API code you need to call the function `bvstartProfiler()` from your code before you call any API functions. Ideally you should call this function as early in your program as possible.

This will look for an existing Bug Validator user interface and send profiling results to that. If no Bug Validator user interface is running, `bvStartProfiler()` will start a new Bug Validator user interface.

```
extern "C"
int bvStartProfiler();
```

Returns:

```
TRUE    Successfully started BV profiler.
FALSE   Failed to start the BV profiler.
```

If you prefer to start the profiler from the user interface or command line you can omit the `bvStartProfiler()` call. You can leave it present if you wish to start Bug Validator from your program.

Do not use this function if you are working with services, use the ***NT Service API***.

bvStartProfilerWithOwnGUI

This function is the same as `bvStartProfiler()` except that this always starts a new Bug Validator user interface to receive the profiling results.

Use this function if you want to profile multiple processes at the same time (one Bug Validator user interface per process being profiled).

```
extern "C"
int bvStartProfilerWithOwnGUI();
```

Returns:

```
TRUE    Successfully started BV profiler.
FALSE   Failed to start the BV profiler.
```

If you prefer to start the profiler from the user interface or command line you can omit the `bvStartProfilerWithOwnGUI()` call. You can leave it present if you wish to start Bug Validator from your program.

Do not use this function if you are working with services, use the ***NT Service API***.

bvSetThreadName()

Sets the name of a thread.

```
extern "C"
void bvSetThreadName(DWORD threadID
                    const TCHAR *name);
```

bvSetThreadNameA()

Sets the name of a thread.

```
extern "C"
void bvSetThreadNameA(DWORD threadID
                    const char *name);
```

bvSetThreadNameW()

Sets the name of a thread.

```
extern "C"
void bvSetThreadNameW(DWORD threadID
                    const wchar_t *name);
```


bvSetCollect()

Enables or disables data collection - i.e. whether data is sent to Bug Validator from your target application.

```
extern "C"  
void bvSetCollect(int enable); // TRUE to enable, FALSE to disable
```

bvGetCollect()

Returns whether data collection is on.

```
extern "C"  
int bvGetCollect(); // Returns TRUE or FALSE
```

5.3 Calling the API via GetProcAddress

Calling API functions using GetProcAddress

If you don't want to use the svBVAPI.c/h files you can use `GetProcAddress()` to find the interface functions in the Bug Validator DLL.

The interface functions have different names and do not use C++ name mangling, but have identical parameters to the API functions.

To determine the function name take any native API name, replace the leading **bv** with **api**. For example `bvSetThreadNameW()` becomes `apiSetThreadNameW()`;

Example usage

```
typedef void (*bvSetThreadNameW_FUNC)(const TCHAR *name);  
  
HMODULE getValidatorModule()  
{  
    HMODULE hModule;  
  
    hModule = GetModuleHandle(_T("svlBugValidatorStub6432.dll")); // 32 bit DLL with 64 bit Bug  
    if (hModule == NULL)  
        hModule = GetModuleHandle(_T("svlBugValidatorStub_x64.dll")); // 64 bit DLL with 64  
    if (hModule == NULL)  
        hModule = GetModuleHandle(_T("svlBugValidatorStub.dll")); // 32 bit DLL with 32  
  
    return hModule;  
}
```

```

HMODULE    hMod;

// get module, will only succeed if Bug Validator launched this app or is injected into this a

hMod = getValidatorModule();
if (hMod != NULL)
{
    // Bug Validator is present, lookup the function and call it to set the thread name

    bvSetThreadNameW_FUNC    pFunc;

    // "apiSetThreadNameW" is equivalent to linking against "bvSetThreadNameW"

    pFunc = (bvSetThreadNameW_FUNC)GetProcAddress(hMod, "apiSetThreadNameW");
    if (pFunc != NULL)
    {
        (*pFunc)(threadName);
    }
}

```

API functions and their GetProcAddress names

For any API functions not listed, try looking up the name in `svlBugValidatorStub.dll` using `depends.exe` or PE File Browser.

 Show API functions and GetProcAddress names

API Name	GetProcAddress() Name
<code>bvLoadProfiler</code>	
<code>bvIsValidatorPresent</code>	
<code>bvStartProfiler</code>	
<code>bvSetThreadNameA</code>	<code>apiSetThreadNameA</code>
<code>bvSetThreadNameW</code>	<code>apiSetThreadNameW</code>
<code>bvSetCollect</code>	<code>apiSetCollect</code>
<code>bvGetCollect</code>	<code>apiGetCollect</code>
<code>bvShutdownBugValidat</code>	<code>apiShutdownBugValidator</code>

Other exported functions

You may see some other functions exported from `svlBugValidatorStub.dll(_x64).dll`.



These other functions are for Bug Validator's use. Using them may damage memory locations and/or crash your code. Best not to use them!

5.4 Convenience functions

Convenience functions

One convenience function is provided that will start the Bug Validator GUI (if it is not already running), then load the Bug Validator execution tracer into your process and start tracing it.

```
extern "C"  
int loadValidatorIntoApplication();
```

Returns:

```
TRUE    Successfully loaded BV DLL into target application and successfully  
started the profiler.  
FALSE   Failed to load the BV DLL or failed to start the profiler.
```

To use this function `#include loadValidatorIntoApplication.h` into your code.

Source Files

The source files can be found in the `API` directory in the Bug Validator install directory.

```
loadValidatorIntoApplication.h  
loadValidatorIntoApplication.c
```

Just add these files to your project and build.

You do not need to link with any libraries to use these convenience functions.

Part

VI

6 Working with IIS and Services



When working with NT services your account must have the appropriate privileges described in the User Permissions topic.

Attaching to your service

To use Bug Validator with NT Services you need to link a small library to your application and call two functions in the library.

The NT Service API

The NT Service API is provided to enable Bug Validator to work with services.

The API works just as well with normal applications, and the same considerations outlined here also apply generally.

When the NT Service API is used, source code symbols are acquired in the stub and sent to the Bug Validator user interface.

Monitoring the service

When working with Bug Validator and services using the NT Service API you don't start the service using Bug Validator.

Instead, you start the service the way you normally start the service - e.g. with the service control manager.

The code that you have embedded into your service then contacts Bug Validator, which you should have running *before* starting the service.

Once you've exercised your service and stopped it, Bug Validator will show the usual execution trace information. You can refresh the GUI at any time during the service's execution.

Examples and help

We provide some Example Service Source Code to demonstrate how to embed the service code into your service.

If you have problems using Bug Validator with services, please contact us at support@softwareverify.com.

6.1 NT Service API

The Bug Validator stub service libraries

The NT Service API is very simple, consisting of functions to load, start and unload the Bug Validator DLL.

We have also provided some debugging functions to help you debug the implementation of the NT Service API because getting data into and out of services is not always straightforward.

The stub service libraries used for this are shown in the following table:

	32 bit Bug Validator
32 bit service	svlBVStubService.lib
	svlBVStubServiceMD.lib
	svlBVStubServiceMT.lib

All the functions exported from these libraries are exported as `extern "C"` so that C and C++ users can use them.

Library name suffixes

The MD suffix indicates the library was built with the /MD compiler switch.
The MT suffix indicates the library was built with the /MT compiler switch.

Directory Name: 2010 or 2012?

Visual Studio 6 to Visual Studio 2010

If you are using Visual Studio 2010 or earlier, use libraries from a directory with 2010 in the directory name.

Visual Studio 2010 to Visual Studio 2022

If you are using Visual Studio 2012 or later, use libraries from a directory with 2012 in the directory name.

Header files

The header files can be found in the `svlBVStubService` directory in the Bug Validator install directory.

The headers file provide an error enumeration and the NT Service API functions.

```
svlBVStubService.h
svlServiceError.h
```

Linker Problems

Some linkers cannot link the stub service library file. If you have this problem see What do I do if I cannot use `svlBVStubService.lib`?

Loading the Bug Validator DLL into your service

To load the Bug Validator stub dll `svlBugValidatorStub.dll` into your service, call `svlBVStub_LoadBugValidator()`, *do not call* `LoadLibrary()`.

Shutting down the Bug Validator DLL from your service.

To shutdown Bug Validator's monitoring of the service, call `svlBVStub_ShutdownBugValidator()`.

This sends the shutting down notification and removes any hooks for your process.

Calling this function is optional. You can stop your service without calling this function.

Unloading the Bug Validator DLL from your service.

To unload the Bug Validator stub dll, call `svlBVStub_UnloadBugValidator()`, *not* `FreeLibrary()`.

Calling this function is optional. You can stop your service without calling this function.

Setting a service notification callback

Once you have successfully loaded the Bug Validator DLL you can setup a service callback so that the service control manager can be kept updated during the process of starting the validator.

When a service is starting, Windows requires the service to inform the Service Control Manager (SCM) that is starting at least every ten seconds.

Failure to do so results in Windows concluding that the service has failed to start, and the service is terminated.

Instrumenting your service may well take *more* than 10 seconds, depending on the complexity and size of your service.

The solution is for *Bug Validator* to periodically call a user supplied callback from which you can regularly inform the SCM of the appropriate status.

You can set the service callback with `svlBVStub_SetServiceCallback(callback, userParam)`.

Usage

Here is an example callback which ignores the userParam.

```

void serviceCallback(void *userParam)
{
    static DWORD dwCheckPoint = 1;

    ssStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
    ssStatus.dwServiceSpecificExitCode = 0;

    ssStatus.dwControlsAccepted = 0;

    ssStatus.dwCurrentState = dwCurrentState;
    ssStatus.dwWin32ExitCode = dwWin32ExitCode;
    ssStatus.dwWaitHint = dwWaitHint;

    ssStatus.dwCheckPoint = dwCheckPoint++;

    // Report the status of the service to the service control manager.

    return SetServiceStatus(ssStatusHandle, &ssStatus);
}

```

Once your service is running (rather than starting) your service callback should set the appropriate running status `SERVICE_RUNNING` rather than `SERVICE_START_PENDING`.

```

if (!ReportStatusToSCMgr(SERVICE_RUNNING,          // service state
                        NO_ERROR,                  // exit code
                        0))                         // wait hint
{
    dwErr = GetLastError();
    if (bLogging)
        svlBVStub_writeToLogFileW(L"ReportStatusToSCMgr:5\r\n");
    goto cleanup;
}

```

An alternative solution is to prevent the service callback from being called once the service status has been set to running.

```
svlBVStub_SetServiceCallback(NULL, NULL);.
```

Starting Bug Validator DLL in your service

To start Bug Validator inspecting your service call `svlBVStub_StartBugValidator()`.

Setting a filename for all logging data to be written to

To set the filename for all debugging/logging information to be written to call `svlBVStub_setLogFileName()`.

Deleting the logfile

To delete the log file call `svlBVStub_deleteLogFile()`.

Writing text to the logfile

To write a standard ANSI character string to the log file call `svlBVStub_writeToLogFileA(text)`. The ANSI string will be converted to Unicode prior to writing to the log file.

To write a Unicode character string to the log file call `svlBVStub_writeToLogFileW(text)`.

Writing error code descriptions to the logfile

To write a human readable description of the SVL_SERVICE_ERROR error code to the log file call `svlBVStub_writeToLogFile(errCode)`.

Writing GetLastError code descriptions to the logfile

To write a human readable description of the Windows error code to the log file call `svlBVStub_writeToLogFileLastError(errCode)`.

Dumping the PATH environment to the logfile

To write the contents of the PATH environment variable to the log file call `svlBVStub_dumpPathToLogFile()`.

This can be useful if you want to know what the search path is when trying to debug why a DLL wasn't found during an attempt to load the Validator DLL.

6.1.1 Changes to the NT Service API

API changes - February 2018

To make the API easier to use with services we made the following changes:

- Changed the API by adding many debugging functions to allow you to easily log information.
- We also extended the error enumeration to provide additional error status values.
- We also split the function of loading and starting Bug Validator into two functions - a *load* function and a *start* function.
- We split the functionality so that you could setup a service callback prior to calling the *start* function.

The service callback allows the service control manager to be informed that the service is still active during time consuming operations, such as starting Bug Validator when the service is non-trivial in scope.

Failure to inform the service control manager results in the service being killed by the service control manager because it thinks the service has hung.

This change in the API is to ensure you get better results from using our software.

What do you need to do to move from the old API to the new API?

Change all SVL_ERROR declarations to SVL_SERVICE_ERROR.

Your previous startup code probably looked like this:

```
SVL_ERROR errCode;

errCode = svlBVStub_LoadBugValidator();
```

Change it to this:

```
SVL_SERVICE_ERROR errCode;

errCode = svlBVStub_LoadBugValidator();

errCode = svlBVStub_SetServiceCallback(serviceCallback, NULL);

errCode = svlBVStub_StartBugValidator();

errCode = svlBVStub_StartBugValidatorForIIS();
```

The serviceCallback would look something like this:

```
void serviceCallback(void *userParam)
{
    static DWORD dwCheckPoint = 1;

    ssStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
    ssStatus.dwServiceSpecificExitCode = 0;

    ssStatus.dwControlsAccepted = 0;

    ssStatus.dwCurrentState = dwCurrentState;
    ssStatus.dwWin32ExitCode = dwWin32ExitCode;
    ssStatus.dwWaitHint = dwWaitHint;
    ssStatus.dwCheckPoint = dwCheckPoint++;

    // Report the status of the service to the service control manager.

    return SetServiceStatus(ssStatusHandle, &ssStatus);
}
```

In the code above we have omitted error handling.

To see how to use the new logging function with error handling please examine the source code **service.cpp** in the ex

Important.

Once your service is running (rather than starting) your service callback should set the appropriate running status SERVICE_RUNNING rather than SERVICE_START_PENDING.

```

if (!ReportStatusToSCMgr(SERVICE_RUNNING,      // service state
                        NO_ERROR,              // exit code
                        0))                     // wait hint
{
    dwErr = GetLastError();
    if (bLogging)
        svlBVStub_writeToLogFileW(L"ReportStatusToSCMgr:5\r\n");
    goto cleanup;
}

```

An alternative solution is to prevent the service callback from being called once the service status has been set to running.

```
svlBVStub_SetServiceCallback(NULL, NULL);.
```

6.1.2 NT Service API Reference

The API consists of the following functions.

SVL_SERVICE_ERROR Enumeration

Various error states can happen when working with the API. These error states are represented by the SVL_SERVICE_ERROR enumeration.

```

typedef enum _svlServiceError
{
    SVL_OK,                                     // Normal behaviour
    SVL_ALREADY_LOADED,                        // Stub DLL already loaded into serv
    SVL_LOAD_FAILED,                          // Failed to load stub DLL into serv
    SVL_FAILED_TO_ENABLE_STUB_SYMBOLS,        // Loaded DLL, but failed to enable
    SVL_NOT_LOADED,                           // Couldn't unload DLL because could
    SVL_FAIL_UNLOAD,                          // Couldn't unload DLL because could
    SVL_FAIL_TO_CLEANUP_INTERNAL_HEAP,        // Couldn't get the internal stub he
    SVL_FAIL_MODULE_HANDLE                    // Couldn't get the stub DLL handle
    SVL_FAIL_SETSERVICECALLBACK,             // Couldn't call the set service cal
    SVL_FAIL_COULD_NOT_FIND_ENTRY_POINT,      // Couldn't find the DLL entry point
    SVL_FAIL_TO_START,                       // Failed to start the Validator
    SVL_FAIL_SETSERVICECALLBACKTHRESHOLD,   // Couldn't call the set service cal
    SVL_FAIL_PATHS_DO_NOT_MATCH,              // Path to service in env vars doesn
    SVL_FAIL_INCORRECT_PRODUCT_PREFIX,        // Wrong validator
    SVL_FAIL_X86_VALIDATOR_FOUND_EXPECTED_X64_VALIDATOR, // Found wrong bit depth validator
    SVL_FAIL_X64_VALIDATOR_FOUND_EXPECTED_X86_VALIDATOR, // Found wrong bit depth validator
    SVL_FAIL_DID_YOU_MONITOR_A_SERVICE_FROM_VALIDATOR, // Looks like Monitor A Service wasn
    SVL_FAIL_ENV_VAR_NOT_FOUND,               // Env Var not found
    SVL_FAIL_VALIDATOR_ENV_VAR_NOT_FOUND,     // Env Var identifying validator not
    SVL_FAIL_VALIDATOR_ID_NOT_SPECIFIED,      // Validator process not specified
    SVL_FAIL_VALIDATOR_ID_NOT_A_PROCESS,      // Validator process identified does
    SVL_FAIL_VALIDATOR_NOT_FOUND,             // Validator process identified does
} SVL_SERVICE_ERROR;

```

svlBVStub_LoadBugValidator

```

extern "C"
SVL_SERVICE_ERROR svlBVStub_LoadBugValidator();

```

To load the Bug Validator stub `svlBugValidatorStub.dll` into your service, use `svlBVStub_LoadBugValidator()`, *not* `LoadLibrary()`.

This loads the DLL and sets up a few internal variables in the DLL to ensure that symbols are sent from the stub to the Bug Validator user interface.

This is necessary because the Bug Validator user interface can't open a process handle to a service and so is unable to get symbols from the process.

To solve this, symbols are sent from the stub to the user interface as needed.

If you just call `LoadLibrary()` on the DLL, symbols will *not* be sent to the Bug Validator user interface and you won't get meaningful function names in your stack traces.

This function can be used when monitoring 32 bit services or applications with Bug Validator.

Example usage.

svlBVStub_StartBugValidator

```
extern "C"  
SVL_SERVICE_ERROR svlBVStub_StartBugValidator();
```

To start Bug Validator inspecting the service call `svlBVStub_StartBugValidator()`.

svlBVStub_StartBugValidatorForIIS

```
extern "C"  
SVL_SERVICE_ERROR svlBVStub_LoadBugValidatorForIIS();
```

To start Bug Validator inspecting IIS call `svlBVStub_StartBugValidatorForIIS()`.

Example usage.

svlBVStub_ShutdownBugValidator

```
extern "C"  
SVL_SERVICE_ERROR svlBVStub_ShutdownBugValidator();
```

To stop Bug Validator inspecting the service call `svlBVStub_ShutdownBugValidator()`.

This sends the shutting down notification and removes any hooks for your process.

Calling this function is optional. ***You can stop your service without calling this function.***

svlBVStub_UnloadBugValidator

```
extern "C"  
SVL_SERVICE_ERROR svlBVStub_UnloadBugValidator();
```

To unload Bug Validator call `svlBVStub_UnloadBugValidator()`, ***do not call*** `FreeLibrary()`.

Calling this function is optional. ***You can stop your service without calling this function.***

svlBVStub_SetServiceCallback

```
extern "C"  
SVL_SERVICE_ERROR svlBVStub_SetServiceCallback(serviceCallback_FUNC callback,  
                                                void* userParam);
```

`svlBVStub_SetServiceCallback` is used to setup a service callback that is used to inform the Windows service control manager.

`userParam` is a value you can supply which will then be passed to the callback every time the callback is called during instrumentation.

Why is a service callback needed?

When a service is starting, Windows requires the service to inform the Service Control Manager (SCM) that is starting at least every ten seconds.

Failure to do so results in Windows concluding that the service has failed to start, and the service is terminated.

Instrumenting your service may well take *more* than 10 seconds, depending on the complexity and size of your service.

The solution is for *Bug Validator* to periodically call a user supplied callback from which you can regularly inform the SCM of the appropriate status.

We strongly recommend that you setup a service callback. Not setting a service callback can result in failure of your service.

Debugging functions

The following functions are provided to help you log information about the progress, success or failure of the NT Service API attaching Bug Validator to your service.

We strongly recommend that you use these logging functions so that you can understand why Bug Validator might fail to connect to a service.

To see example usage of these debugging functions please look in `service.cpp` in the `examples\service` directory in the Bug Validator install directory.

svlBVStub_setLogFileName

```
extern "C"  
void svlBVStub_setLogFileName(const wchar_t* fileName);
```

Call `svlBVStub_setLogFileName` to set the name of the filename used for logging.

This function must be called before you can use any of the other debugging functions.

Setting this filename also sets the filename used by some of these API functions - you will find additional logging data from those functions that will help debug any issues with the service.

svlBVStub_deleteLogFile

```
extern "C"  
void svlBVStub_deleteLogFile();
```

This function deletes the log file.

svlBVStub_writeToLogFileA

```
extern "C"  
void svlBVStub_writeToLogFileA(const char* text);
```

This function writes a standard ANSI character string to the log file.

The ANSI string will be converted to Unicode prior to writing to the log file.

svlBVStub_writeToLogFileW

```
extern "C"  
void svlBVStub_writeToLogFileW(const wchar_t* text);
```

This function writes a Unicode character string to the log file.

svlBVStub_writeToLogFile

```
extern "C"  
void svlBVStub_writeToLogFile(SVL_SERVICE_ERROR errCode);
```

This function writes a human readable description of the SVL_SERVICE_ERROR error code to the log file.

svlBVStub_writeToLogFileLastError

```
extern "C"  
void svlBVStub_writeToLogFileLastError(DWORD errCode);
```

This function writes a human readable description of the Windows error code to the log file.

The errCode parameter is the error code returned from GetLastError().

svlBVStub_dumpPathToLogFile

```
extern "C"
```

```
void svlBVStub_dumpPathToLogFile();
```

This function writes the contents of the PATH environment variable to the log file.

This can be useful if you want to know what the search path is when trying to debug why a DLL wasn't found during an attempt to load the Validator DLL.

6.1.3 Troubleshooting

Troubleshooting - Service fails to start

If a service takes too long to start the service control manager kills the service.

The way to stop this is for a service to call `ReportStatusToSCMgr()` to tell the service control manager that the service is still OK.

Bug Validator can't do this for you as the call requires some data from any earlier call you have made.

The solution is that you provide a callback using `svlBVStub_SetServiceCallback()` that Bug Validator can call during the process of attaching to the service, and you can call the appropriate function.

Example code to set the callback:

```
errCode = svlBVStub_SetServiceCallback(serviceCallback,           // the
callback                                                         NULL);                  // some
user data (we don't have any, so set NULL)
if (bLogging)
{
    if (errCode != SVL_OK)
    {
        svlBVStub_writeToLogFileW(L"Setting service callback failed.
\r\n");
        svlBVStub_writeToLogFile(errCode);
    }

    svlBVStub_writeToLogFileW(L"Starting Bug Validator\r\n");
}
```

Example callback:

```
static void serviceCallback(void *userParam)
{
    // just tell the Service Control Manager that we are still busy
    // in this example userParam is not used
    //
    // note that prior to the Validator loading it's DLL ssStatus.dwCurrentState
    must have been initialised, most likely to SERVICE_START_PENDING
    // you could pass a fixed value here, but it would need to change once the
    service has finished starting up so that you don't unintentionally change the service
    state
```

```
// when this callback is called. This callback is called whenever
instrumentation happens (when a DLL is loaded). Thus you can't assume this is only
called during service startup,
// it may also get called later in the service lifetime.

    ReportStatusToSCMgr(ssStatus.dwCurrentState,    // service state
                        NO_ERROR,                  // exit code
                        3000);                      // wait hint
}
```

We strongly recommend that you set a service callback. It won't harm your program and it will remove any likelihood of your service being killed by the service control manager.

Troubleshooting - Service starts, Bug Validator gets no data

If you have problems getting Bug Validator to monitor your service you'll need to find out what's failing.

Until Bug Validator loads correctly and successfully connects to the graphical user interface you have no way of knowing what is happening.

The solution is to set a log file that Bug Validator can write status messages to. You can also write your own status messages to this log file.

Set the log file using `svlBVStub_setLogFileName`. Write to it using `svlBVStub_writeToLogFile()`, `svlBVStub_writeToLogFileA()`, `svlBVStub_writeToLogFileW()`.

Then when things are not working as expected take a look at the log file to see the errors. The Bug Validator will often suggest what the problem is.

We strongly recommend that you configure the log file and use it when working with services. It has saved us a lot of time.

6.2 Working with IIS

Configuring IIS for use with ISAPI

We assume that you are familiar with IIS. This is not a topic we can provide advice for.

That said, we wrote a blog article about configuring IIS for use with ISAPI.

Example ISAPI

We have provided an example ISAPI extension configured for use with Bug Validator.

This example is provided as source code and project files. You will need to build it yourself, you may need to change an include path to find the appropriate headers. The resulting ISAPI will need to be copied to your website for testing and the website configured to allow the ISAPI to execute (please see the above mentioned blog article for details on that).

You can find the example ISAPI in the **isapiExample** folder in the Bug Validator installation directory.

Using Bug Validator with IIS

IIS is a service application. It runs as one of the more restricted applications on Microsoft Windows.

Bug mapped files created by IIS cannot be opened by user mode programs (Bug Validator, for example). DLLs, executables and files cannot be opened by IIS except if they are in directories which IIS has access to. These are security measures intended to make your computer secure from attack.

These security measures make it hard for tools like Bug Validator to work.

- We have to communicate settings information to Bug Validator via text file
- All DLLs and helper programs we want to use need to be copied to the web root (or a subdirectory within the web root) so that they can be used.
- We need to have our own data transport because our usual high speed memory mapped data transport is not available.

It's also not possible to launch IIS or inject into a running IIS instance.

The only way to work with IIS is by using the NT Service API, and using the `svlBVStub_StartBugValidatorForIIS()` function instead of `svlBVStub_StartBugValidator()`.

We've provided some example code to show you how to attach and detach from your ISAPI extension.

Workflow

1) Start monitoring your ISAPI by using the Monitor ISAPI dialog.

 **Launch** menu > **IIS** menu > **Monitor ISAPI...**

2) When you have finished interacting with the web pages that use the ISAPI component shutdown IIS, wait for Bug Validator's status to indicate "Ready" and examine the results.

 **Launch** menu > **IIS** menu > **Stop IIS**

6.3 Example Source Code

Service Example

Example demonstrating how to monitor a service.

Also see the example service that ships with Bug Validator.

You can find this in the `\examples\service` directory in the Bug Validator install directory.

Also see the example service and child process that ships with Bug Validator.

You can find this in the `\examples\serviceWithAChildProcess` directory in the Bug Validator install directory.

IIS Example

Example demonstrating how to monitor an ISAPI DLL.

Also see the example ISAPI DLL that ships with Bug Validator.

You can find this in the `\examples\isapiExample` directory in the Bug Validator install directory.

6.3.1 Example Service Source Code

When you use the functions to load and unload Bug Validator from your service, it is important that you put the function calls in the correct place in your software. Typically, this means that Bug Validator is loaded as the first action in the **service_main()** function and unloaded just before the service control manager is informed of the stopped status.

The source code shown below shows an example **service_main()** function in a service, demonstrating example places to load and unload Bug Validator. The source code shown below also includes a comment about problems with the way services are stopped and what may be displayed in a debugger if this happens.

An example NT service can be found here.

An example client for the service can be found here.

An example utility for controlling if the service uses Bug Validator can be found here.

```
void serviceCallback(void *userParam)
{
    // just tell the Service Control Manager that we are still busy
    // in this example userParam is not used

    static DWORD dwCheckPoint = 1;

    ssStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
    ssStatus.dwServiceSpecificExitCode = 0;

    ssStatus.dwControlsAccepted = 0;

    ssStatus.dwCurrentState = dwCurrentState;
    ssStatus.dwWin32ExitCode = dwWin32ExitCode;
    ssStatus.dwWaitHint = dwWaitHint;
    ssStatus.dwCheckPoint = dwCheckPoint++;

    // Report the status of the service to the service control manager.

    return SetServiceStatus(ssStatusHandle, &ssStatus);
}
```

```

void WINAPI service_main(DWORD dwArgc, LPTSTR *lpszArgv)
{
    if (bLogging)
    {
        svlBVStub_setLogFileName(SZLOGFILENAME);
        svlBVStub_deleteLogFile();
    }

    // register our service control handler:

    sshStatusHandle = RegisterServiceCtrlHandler(TEXT(SZSERVICENAME), service_ctrl);
    if (sshStatusHandle != 0)
    {
        DWORD    dwErr = 0;

        // **BV_EXAMPLE** start

        if (bBugValidator)
        {
            // load Bug Validator

            if (bLogging)
            {
                svlBVStub_writeToLogFileW(_T("About to load Bug Validator\r\n"));
            }

            SVL_SERVICE_ERROR    errCode;

            errCode = svlBVStub_LoadBugValidator();
            if (bLogging)
            {
                if (errCode != SVL_OK)
                {
                    DWORD    lastError;

                    lastError = GetLastError();
                    svlBVStub_writeToLogFileW(_T("Bug Validator load failed. \r\n"));
                    svlBVStub_writeToLogFileLastError(lastError);
                    svlBVStub_writeToLogFile(errCode);

                    svlBVStub_dumpPathToLogFile();
                }
                else
                {
                    svlBVStub_writeToLogFileW(_T("Bug Validator load success. \r\n"));
                }
            }

            // setup a service callback so that the Service Control Manager knows the service
            // is starting up even if instrumentation takes longer than 10 seconds (which it will
            // for a non-trivial application)

            if (bLogging)
                svlBVStub_writeToLogFileW(_T("Setting service callback Bug Validator\r\n"));

            errCode = svlBVStub_SetServiceCallback(serviceCallback, // the callback
                                                    NULL);           // some user data (we don't have)

            if (bLogging)
            {
                if (errCode != SVL_OK)

```

```

        {
            svlBVStub_writeToLogFileW(_T("Setting service callback failed. \r\n"));
            svlBVStub_writeToLogFile(errCode);
        }

        svlBVStub_writeToLogFileW(_T("Starting Bug Validator\r\n"));
    }

    errCode = svlBVStub_StartBugValidator();
    if (bLogging)
    {
        if (errCode != SVL_OK)
        {
            DWORD    lastError;

            lastError = GetLastError();
            svlBVStub_writeToLogFileW(_T("Starting Bug Validator failed. \r\n"));
            svlBVStub_writeToLogFileLastError(lastError);
            svlBVStub_writeToLogFile(errCode);
        }

        svlBVStub_writeToLogFileW(_T("Finished loading Bug Validator\r\n"));
    }
}
else
{
    if (bLogging)
        svlBVStub_writeToLogFileW(_T("Not using Bug Validator, DLL will not be loaded\r\n"));
}

// **BV_EXAMPLE** end

// SERVICE_STATUS members that don't change in example

ssStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
ssStatus.dwServiceSpecificExitCode = 0;

// report the status to the service control manager.

if (ReportStatusToSCMgr(SERVICE_START_PENDING, // service state
                        NO_ERROR,                // exit code
                        3000))                    // wait hint
{
    if (bLogging)
        svlBVStub_writeToLogFileW(_T("About to start service\r\n"));

    // do work

    dwErr = ServiceStart(dwArgc, lpszArgv);

    // finished doing work

    if (bLogging)
        svlBVStub_writeToLogFileW(_T("Started service\r\n"));
}

// **BV_EXAMPLE** start

if (bBugValidator)
{

```

```
// unload Bug Validator here
// IMPORTANT.
// Because of the way services work, you can find that this thread which is trying to gra
// BugValidator is ripped from under you by the operating system. This prevents Bug Valid
// removing all its hooks successfully. If Bug Validator does not remove all of its hooks
// because this happens, then you may get a crash when the service stops.
//
// A callstack for such a crash is shown below. If you see this type of crash you need to
// unload Bug Validator somewhere else. The stack trace may be different, but a fundamen
// code calling through doexit(), exit() and ExitProcess()
//
//NTDLL! 77f64e70()
//SVLBUGVALIDATORSTUB!
//MSVCRT! 78001436()
//MSVCRT! 7800578c()
//DBGHELP! 6d55da25()
//DBGHELP! 6d55de83()
//DBGHELP! 6d53705d()
//DBGHELP! 6d51cc69()
//DBGHELP! 6d51f6e8()
//DBGHELP! 6d524ebf()
//DBGHELP! 6d52a7b0()
//DBGHELP! 6d52b00a()
//DBGHELP! 6d526487()
//DBGHELP! 6d5264d7()
//DBGHELP! 6d5264f7()
//SVLBUGVALIDATORSTUB!
//SVLBUGVALIDATORSTUB!
//SVLBUGVALIDATORSTUB!
//SVLBUGVALIDATORSTUB!
//SVLBUGVALIDATORSTUB!
//SVLBUGVALIDATORSTUB!
//SVLBUGVALIDATORSTUB!
//SVLBUGVALIDATORSTUB!
//SVLBUGVALIDATORSTUB!
//MSVCRT! 78001436()
//MSVCRT! 780057db()
//KERNEL32! 77f19fdb()
//SVLBUGVALIDATORSTUB! ExitProcess hook
//doexit(int 0x00000000, int 0x00000000, int 0x00000000) line 392
//exit(int 0x00000000) line 279 + 13 bytes
//mainCRTStartup() line 345
//KERNEL32! 77f1b9ea()

svlBVStub_UnloadBugValidator();

// try to report the stopped status to the service control manager.

if (sshStatusHandle)
    ReportStatusToSCMgr(SERVICE_STOPPED, dwErr, 0);

// tried putting the call to svlBVStub_UnloadBugValidator(); here but often the thread
// was pulled from under it by the operating system

return;
}
}
```

6.3.2 Example ISAPI Source Code

Where to put your code

When you use the functions to load and unload Bug Validator from your service, it is important that you put the function calls in the correct place in your ISAPI extension.

Typically, this means that Bug Validator is:

- loaded as the first action in the `GetExtensionVersion()` function of your ISAPI extension.
- unloaded in the `TerminateExtension()` function of your ISAPI extension.

Example source code

The source code shown below shows an example `GetExtensionVersion()` and an example `TerminateExtension()` used in an ISAPI, demonstrating where to load and unload Bug Validator.

This example code logs errors. We strongly recommend that you do this in your example. Because IIS is a protected process that can't communicate to the outside world except via HTTP/HTTPS when anything fails during the loading and start of Bug Validator the only means we have of communicating that failure to you is via the log file. Please use the log file, it will make debugging any mistakes very much easier, simpler and quicker than any other method.

This process is almost identical to working with a regular service, except that `svlBVStub_StartBugValidator()` is replaced with `svlBVStub_StartBugValidatorForIIS()`.

This example assumes the web root is located `C:\\testISAPIWebsite`

 Show the C++ example ISAPI functions

```
#include "svlBVStubService.h"
#include "svlServiceError.h"

BOOL WINAPI GetExtensionVersion(HSE_VERSION_INFO *pVer)
{
    // some setup work to define what the extension is

    pVer->dwExtensionVersion = HSE_VERSION;
    strncpy(pVer->lpszExtensionDesc, "Validate ISAPI Extension",
HSE_MAX_EXT_DLL_NAME_LEN);

    // load Validator here

    svlBVStub_setLogFileName(L"C:\\testISAPIWebsite\\svl_BV_log.txt");
    svlBVStub_deleteLogFile();

    SVL_SERVICE_ERROR    errCode;
#ifdef IS6432
    // x86 with x64 GUI
    errCode = svlBVStub_LoadBugValidator6432();
#else
    // #ifdef IS6432
    // x86 with x86 GUI
```

```

        // x64 with x64 GUI
        errCode = svlBVStub_LoadBugValidator();
    #endif // #ifdef IS6432
    if (errCode != SVL_OK)
    {
        DWORD    lastError;

        lastError = GetLastError();
        svlBVStub_writeToLogFileW(L"Bug Validator load failed. \r\n");
        svlBVStub_writeToLogFileLastError(lastError);
        svlBVStub_writeToLogFile(errCode);

        svlBVStub_dumpPathToLogFile();
    }
    else
    {
        svlBVStub_writeToLogFileW(L"Bug Validator load success. \r\n");

        errCode = svlBVStub_StartBugValidatorForIIS();
        if (errCode != SVL_OK)
        {
            DWORD    lastError;

            lastError = GetLastError();
            svlBVStub_writeToLogFileW(L"Starting Bug Validator failed.
\r\n");

            svlBVStub_writeToLogFileLastError(lastError);
            svlBVStub_writeToLogFile(errCode);
        }

        svlBVStub_writeToLogFileW(L"Finished starting Bug Validator\r\n");
    }

    return TRUE;
}

BOOL WINAPI TerminateExtension(DWORD    dwFlags)
{
    // unload Validator here

    svlBVStub_UnloadBugValidator();

    return TRUE;
}

```

Part

VII

7 Examples

Bug Validator is a complex product. Whilst we have made every effort to add useful and easy to use features, we realise that some examples of how to use Bug Validator will be helpful. In addition we have provided an example application which has examples of most of the types of error that Bug Validator can detect. This section will use the example application to produce conditions that can be monitored in the examples.

All the example projects are supplied in source code form. You will need to build the examples before you can use them.

Using Visual Studio Express?

You might find you can't build the example application with Express versions of Visual Studio because it doesn't provide all the necessary libraries.

If that's the case, try searching for the missing libraries in one of the freely available Windows SDKs [from the Microsoft website](#).

 If you use Visual Studio Express to build your *own* application, Bug Validator will still work with it just fine.

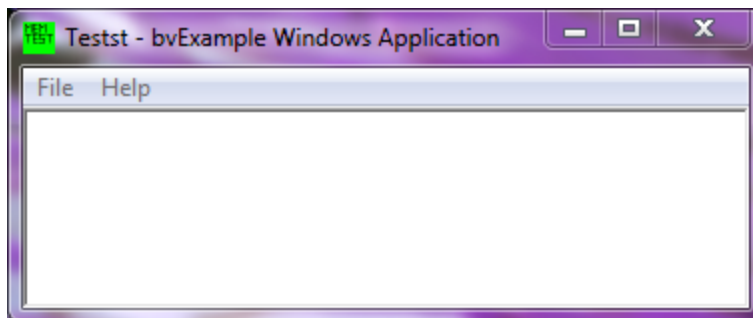
7.1 Example Application

Bug Validator is shipped with an example application that exhibits many of the programming errors that Bug Validator detects. The sample program is called **nativeExample.exe**. Full source code and a Microsoft® Visual Studio® project are supplied. The example program can also be linked with the Bug Validator API to demonstrate some uses of the Bug Validator API.

Information about building the example application can be found [here](#).

Use the program to generate thread errors, waits, potential deadlocks and deadlocks. Use Bug Validator to monitor the behaviour of nativeExample.exe as you use it.

The example program looks like this.



File

The **File** menu has one entry, **Exit**, which closes the application

Please examine the program source to see the nature of the program errors. Brief descriptions are given for each menu entry, but the real detail is in the source code. Some functions have detailed explanations of what is going wrong in the example function and what the likely consequences of this are.

7.1.1 Building the example application

The example project can be found in the **nativeExample** subdirectory in the directory where Bug Validator was installed. If the directory is not present, reinstall your software and choose custom or full installation.

Open the file **nativeExample.dsp** using Microsoft® Developer Studio® 6.0. There are two configurations, one Debug and one Release. Build the application by choosing **Batch Build** on the **Build** menu and then click the **Build** button.

7.2 Example NT Service

Bug Validator is shipped with an example service that demonstrates how to call the two functions required to use Bug Validator with NT Services. Full source code and a Microsoft® Visual Studio® project are supplied.

Information about building the example service can be found here.

The example service performs the following tasks when it is started:

- Loads the Bug Validator stub DLL into the service
- Deliberately leaks some memory so that you can see this in the Bug Validator user interface.
- Performs the normal work of the service until the service is stopped.
- Unloads the Bug Validator stub DLL from the service.
- Informs the service control manager that a stop is pending.

7.2.1 Building the example service

The example project can be found in the **service** subdirectory in the directory where Bug Validator was installed. If the directory is not present, reinstall your software and choose custom or full installation.

Open the file **service.dsp** using Microsoft® Developer Studio® 6.0.

There are two configurations, Debug and Release. The **Debug** and **Release** configurations are linked to the **svlTVStubService.lib** and demonstrate the use of the NT Service API.

The service has the name **SVL TV Simple Service** in the control panel services dialog.

The service provides the following command line options:

- **-install**. Install the service.

- **-remove**. Uninstall the service.
- **-start**. Start the service.
- **-stop**. Stop the service.
- **-debug**. Run as a console application for debugging.
- **-?**. Display the help message.
- **-help**. Display the help message.

Open a cmd prompt in administrator mode, navigate to the location of the service executable, and use one of these commands to install, remove, start, stop the service.

Examples:

```
serviceBV.exe -install  
  
serviceBV.exe -start  
  
serviceBV.exe -stop  
  
serviceBV.exe -remove
```

7.2.2 Building the example client

The example project can be found in the **serviceClient** subdirectory in the directory where Bug Validator was installed. If the directory is not present, reinstall your software and choose custom or full installation.

Open the file **serviceClient.dsp** using Microsoft® Developer Studio® 6.0.

There are two configurations, Debug and Release.

The application takes two commands:

- **-help**. Displays the help message
- **-string**. Sends the (optionally quoted) text appears after -string to the service. If the service is running the service will return the string in reverse order.

For example: `serviceClient.exe -string "The quick brown fox"` returns "xof nworb kciug ehT"

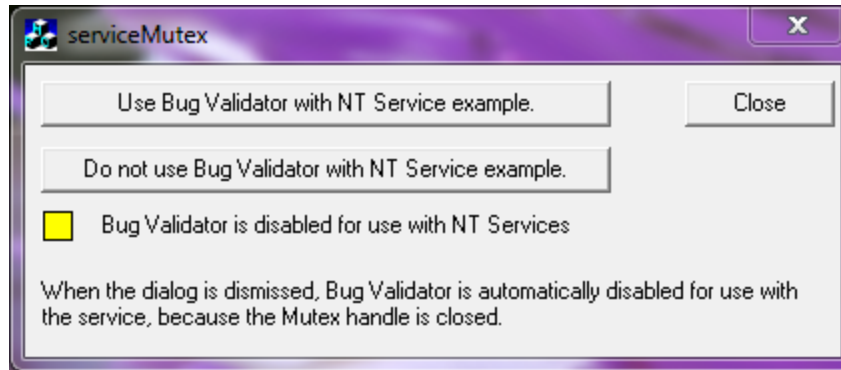
7.2.3 Building the example service utility

The example project can be found in the **serviceMutex** subdirectory in the directory where Bug Validator was installed. If the directory is not present, reinstall your software and choose custom or full installation.

Open the file **serviceMutex.dsp** using Microsoft® Developer Studio® 6.0.

There are two configurations, Debug and Release.

The utility provides a dialog box interface to allow the control over the creation of a mutex object with the name specified in the **service.h** header file. If the service is started with the mutex created, the service loads Bug Validator. If the service is started and the mutex does not exist, the service does not load Bug Validator. This allows you to control if Bug Validator is used without rebuilding your service. If you don't like uses mutexes in this way, you could change the code in the service and the utility to communicate this fact through shared memory, or a registry setting.



7.2.4 Monitoring the service

Once the example service and example client has been built, the next step is to test them using Bug Validator.

Installing the service

If you haven't installed the service, do the following:

- open an administrator mode cmd prompt
- navigate to the directory containing the serviceBV.exe to install
- serviceBV.exe -install

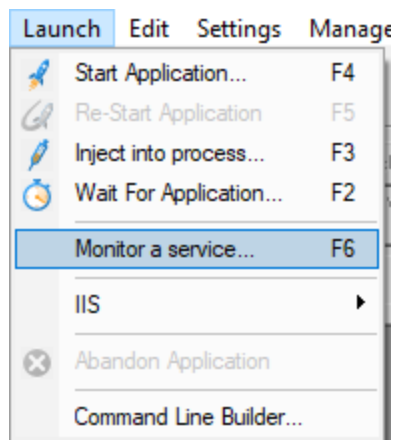
Monitoring the service

Prerequisites

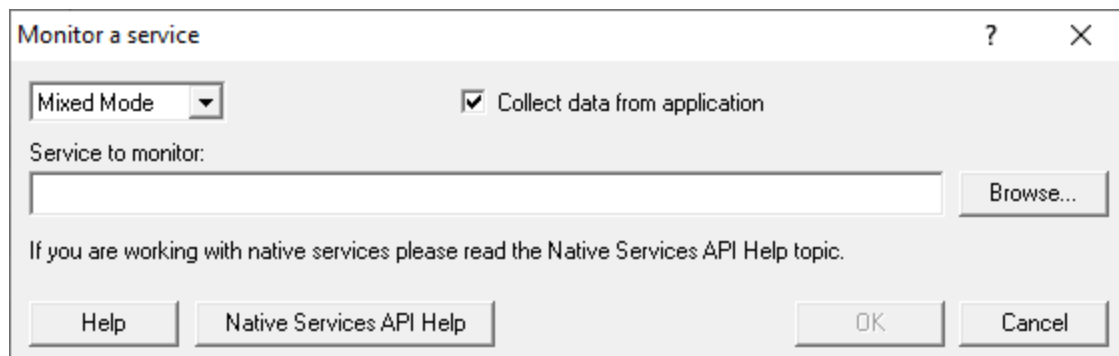
- example service has been installed, but not started (if service has been started, stop the service)
- example service and example client have been built

The following process is used to monitor the application launched by the service:

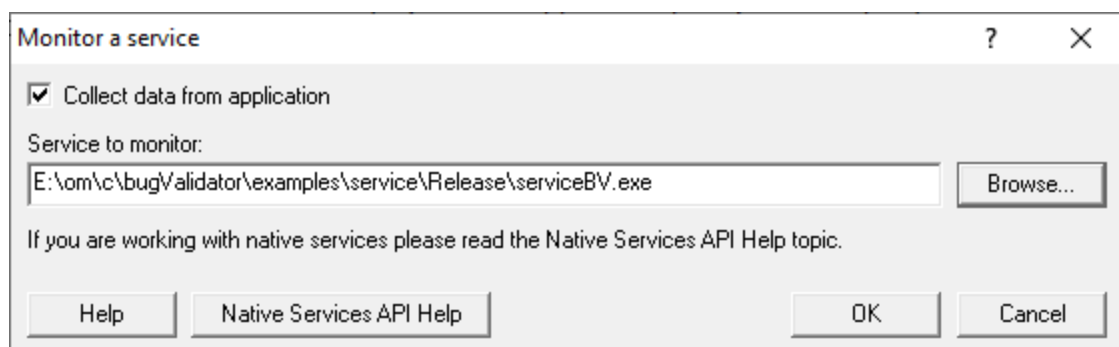
- From the Launch menu choose `Services > Monitor a Service...`



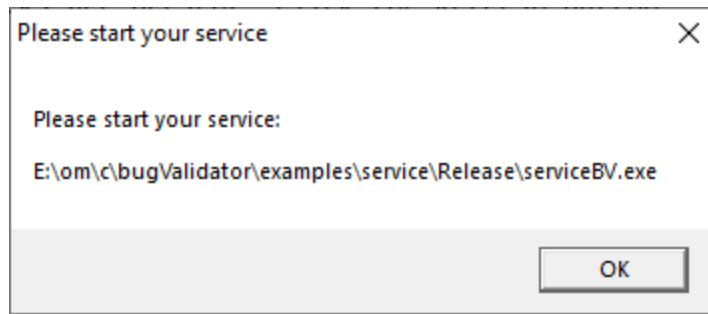
- The Monitor a service dialog is displayed



- Use **Browse...** to open the file chooser dialog and choose the service that will be monitored by Bug Validator.



- Click **OK**
- Bug Validator sets up a variety of parameters then displays a dialog box asking you to start you service. Click **OK** to dismiss the dialog



- Start your service. For the example `serviceBV.exe` do the following
 - open an administrator mode cmd prompt
 - navigate to the directory containing the `serviceBV.exe` to start
 - `serviceBV.exe -start`
 - `serviceBV.exe` starts will be monitored by Bug Validator
- The target application contacts Bug Validator
- Data is collected until the service finishes executing
- Bug Validator displays the results

7.3 Example Application Launched from a Service

The example Application launched from a Service

This pair of projects create an application that is launched from a service.

The purpose of this example is to show how to monitor the application that is launched from the service. This is also the same process for monitoring an application launched by an application launched from a service.

This process is subtly different to the method for working with services (see the example service for that).

Service

The service project is `serviceWithAChildProcess.vcxproj`

The following tasks are performed when the service is started:

- the test application is launched from the service

Application

The application project is `serviceChildProcess.vcxproj`

The application's first task is to load Bug Validator into the application.

- Loads the Bug Validator stub DLL into the application
- Configures the NT Service API to communicate to Bug Validator
- Does some work that can be monitored by Bug Validator
- Exits

Implementation Details

For implementation details see `attachToBugValidator()`; in `serviceChildProcess.cpp`.

The application will need to link to the NT Service API, for example `..\..\..\svlBVStubService\release_2010\svlBVStubService.lib` (for a release EXE/DLL).

Important. Call `attachToBugValidator()` as close to the start of your application as possible, before any threads have been created.

➔ Read more about working with NT Services.

7.3.1 Building the service and application

Example solution files

The example solution can be found in the `examples\serviceWithAChildProcess` subdirectory in the directory where Bug Validator was installed.

If the directory is not present, reinstall your software and choose custom or full installation.

Example project files

The example projects can be found in the subdirectories in the directory where Bug Validator was installed.

`examples\serviceWithAChildProcess\serviceWithAChildProcess`

- **serviceWithAChildProcess.vcproj** ➤ for Microsoft® Visual Studio / .net

`examples\serviceWithAChildProcess\serviceChildProcess`

- **serviceChildProcess.vcproj** ➤ for Microsoft® Visual Studio / .net

Configurations

There are a small number of configurations in each project:

- **Debug / Release** ➤ dynamically links to the `svlBVStubService.lib` demonstrating use with the NT Service API

Using the service

The service is named **SVL *** BV Child Process** in the control panel services dialog (** changes depending on the build configuration), and provides the following command line options:

- **-install** ➤ Install the service
- **-remove** ➤ Uninstall the service
- **-start** ➤ Start the service
- **-stop** ➤ Stop the service
- **-debug** ➤ Run as a console application for debugging
- **-?** ➤ Display the help message
- **-help** ➤ Display the help message

Open a cmd prompt in administrator mode, navigate to the location of the service executable, and use one of these commands to install, remove, start, stop the service.

Examples:

```
serviceWithAChildProcess.exe -install  
  
serviceWithAChildProcess.exe -start  
  
serviceWithAChildProcess.exe -stop  
  
serviceWithAChildProcess.exe -remove
```

7.3.2 Monitoring the application launched from the service

Once the example service and example application are built, the next step is to test them using Bug Validator.

Installing the service

If you haven't installed the service, do the following:

- open an administrator mode cmd prompt
- navigate to the directory containing the `serviceWithAProcess.exe` to install

- serviceWithAProcess.exe -install

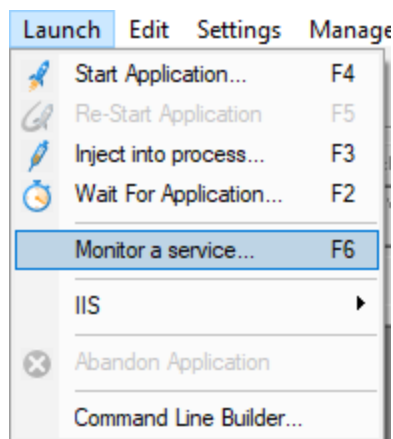
Monitoring the application launched by the service

Prerequisites

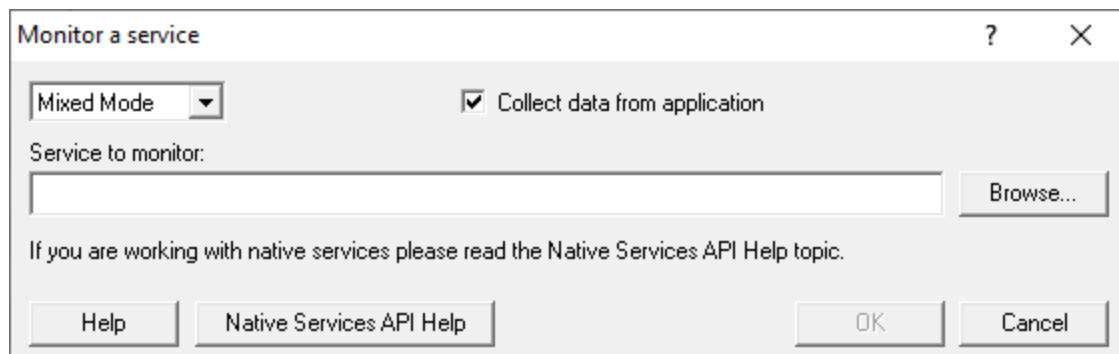
- example service has been installed, but not started (if service has been started, stop the service)
- example service and example application have been built (application must use the NT Service API as demonstrated in attachToBugValidator())
- example application executable is in the same directory as the example service (this is only a requirement for the example)

The following process is used to monitor the application launched by the service:

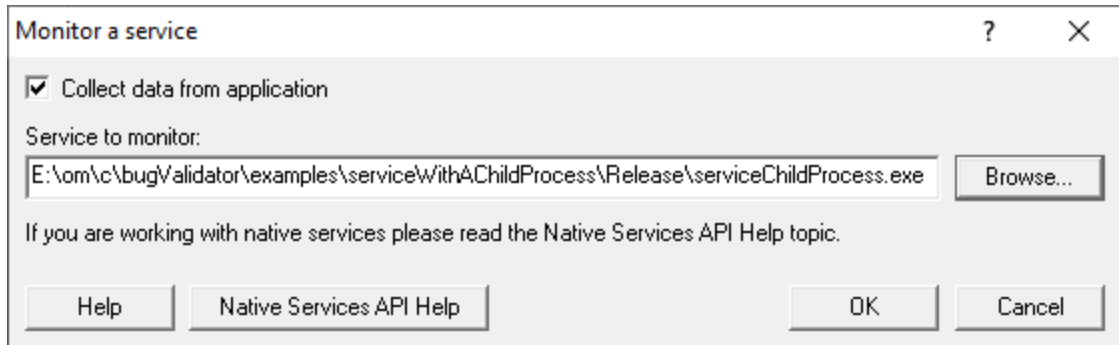
- From the Launch menu choose `Services > Monitor a Service...`



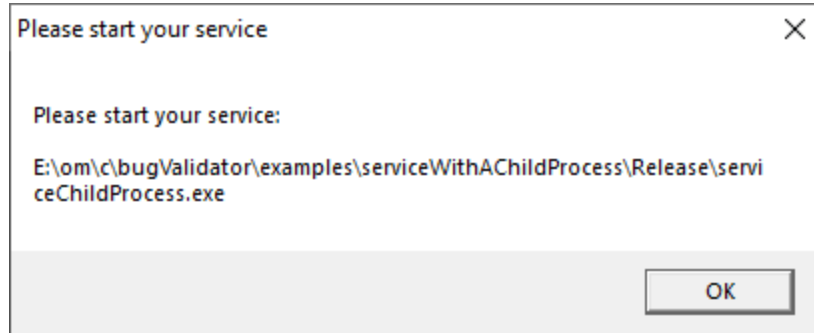
- The Monitor a service dialog is displayed



- Use **Browse...** to open the file chooser dialog and choose the application that will be monitored by Bug Validator. This is the application that is launched by the service. Do not choose the service



- Click **OK**
- Bug Validator sets up a variety of parameters then displays a dialog box asking you to start your service. Click **OK** to dismiss the dialog



- Start your service. For the example `serviceWithAChildProcess.exe` do the following
 - open an administrator mode cmd prompt
 - navigate to the directory containing the `serviceWithAProcess.exe` to start
 - `serviceWithAProcess.exe -start`
 - `serviceWithAProcess.exe` starts and launches the child process `serviceChildProcess.exe` that will be monitored
- The target application contacts Bug Validator
- Data is collected until the target process finishes executing
- Bug Validator displays the results

Part



8 Debug Information, Symbols, Filenames, Line Numbers

Depending on which IDE or compiler/linker combination the process to create debug information to ensure that you have symbols, filenames and line numbers is different.

This section shows you what to do to ensure you have symbols for your compiler and linker.

8.1 Visual Studio

Enabling debug information in Visual Studio has changed over the years depending on the version of Visual Studio you are using.

It's generally the same, but there have been some changes in recent versions that can cause confusion.

By default debug configurations create debug information, but for some versions of Visual Studio, release configurations do not create debug information.

You need to set both compiler and linker settings to get debug information. ***Setting just one or the other will not give you debug information you can use.***

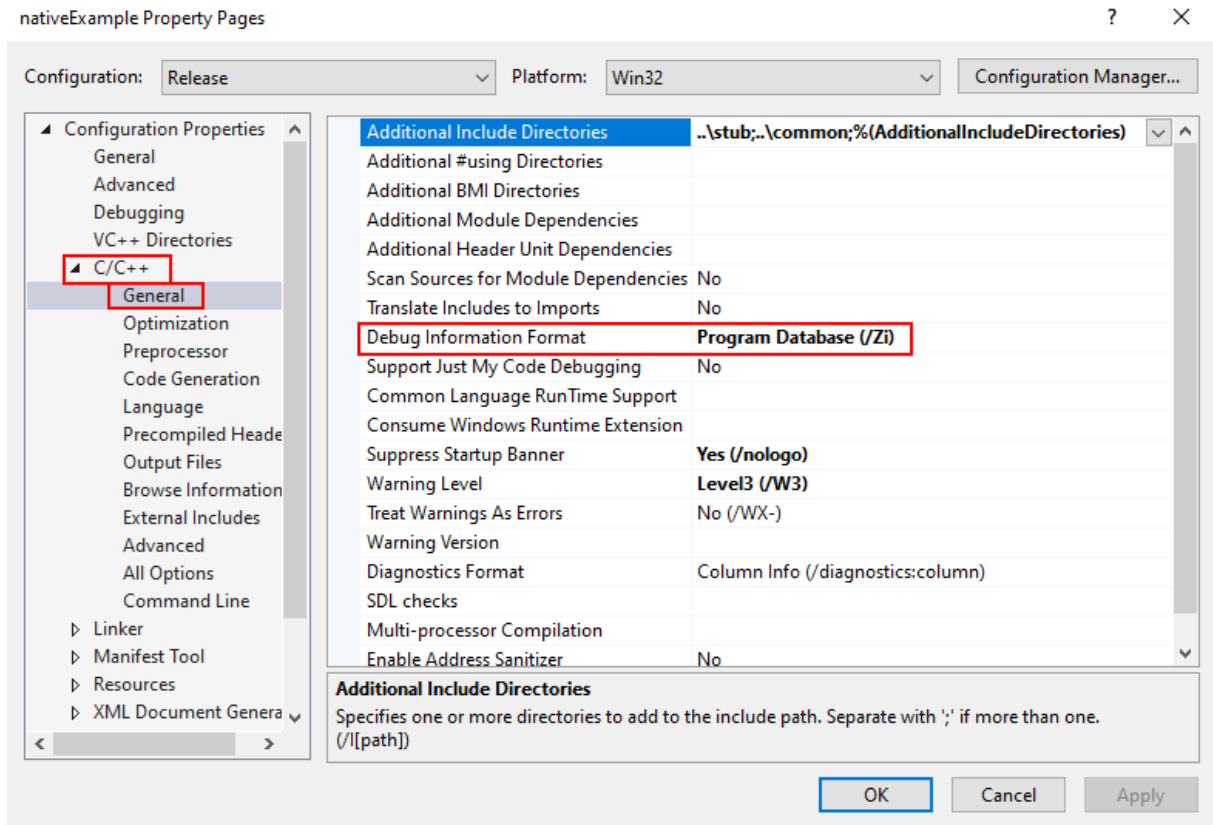
Configurations

In the help below we show you how to modify one configuration, for example Release | Win32.

You need to modify all configurations appropriately. Release, Debug, Win32, Win64 and any other configurations you are using.

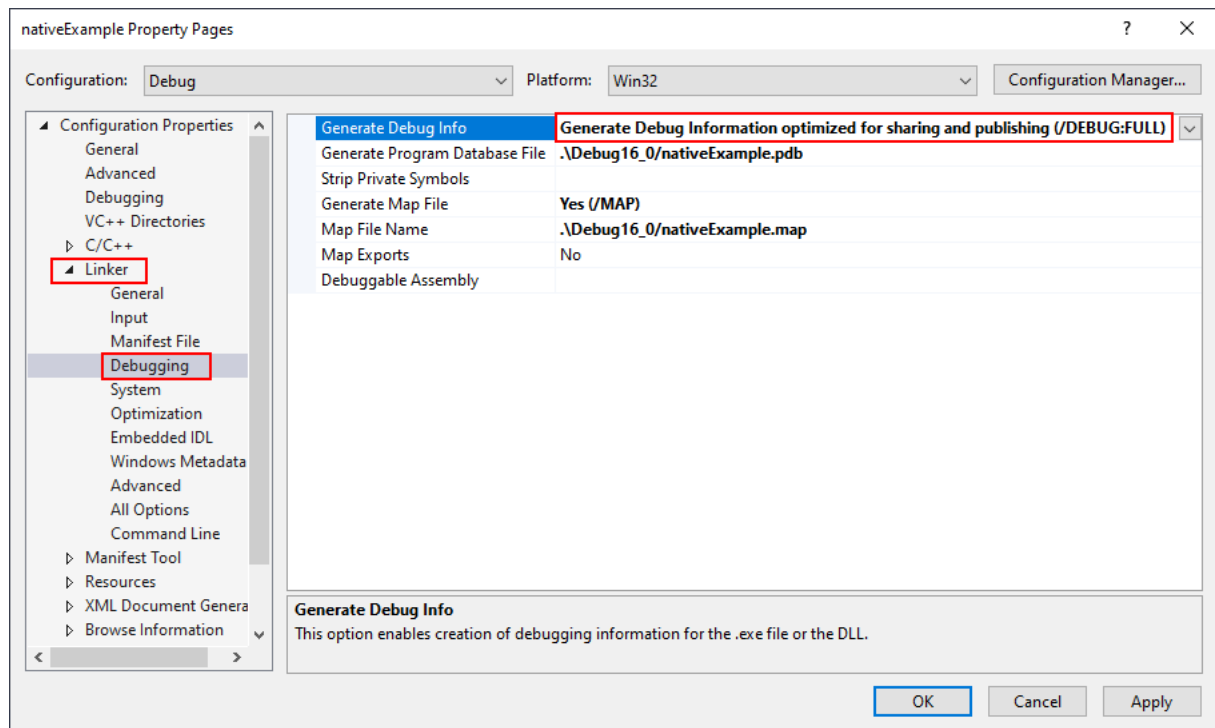
Visual Studio 2017 - 2021

Compiler Settings



Linker Settings

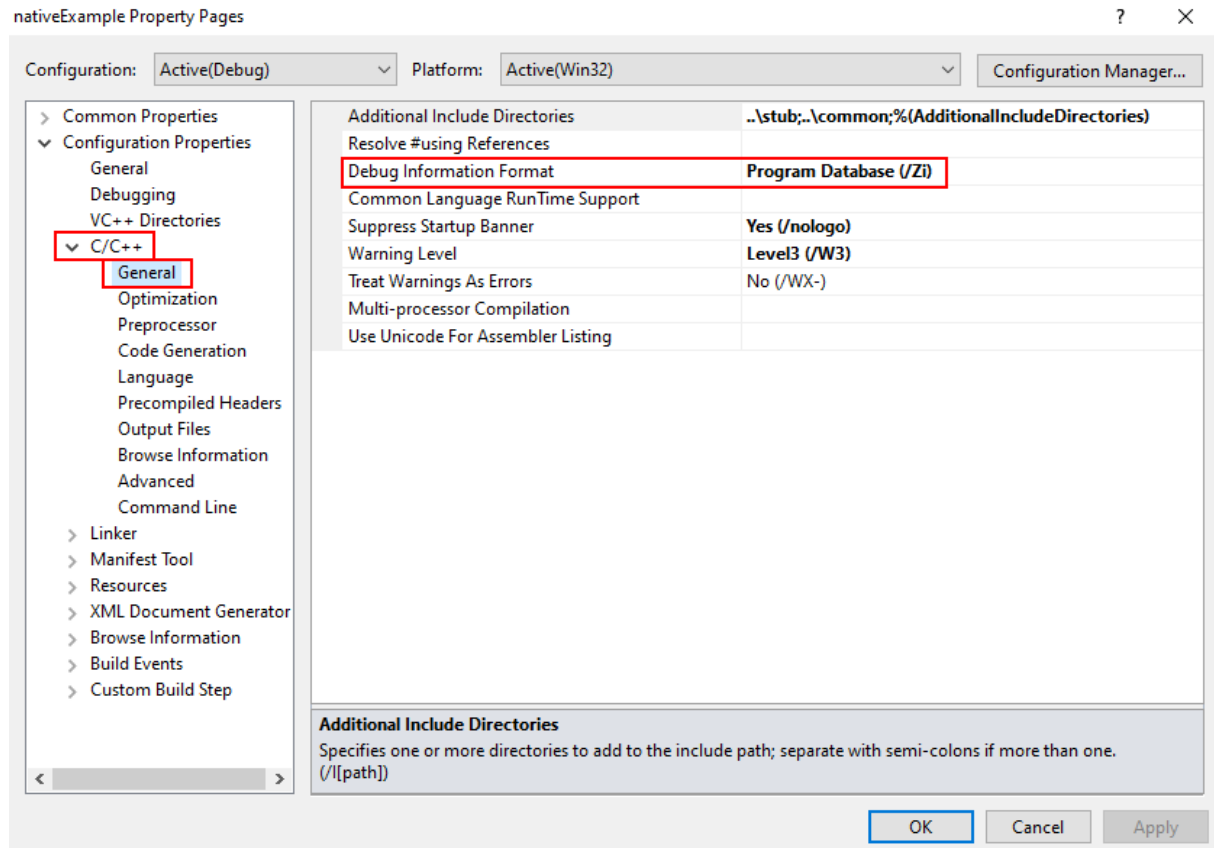
If you're building on a different machine to the machine you're working on (for example a build server), you should choose **/DEBUG:FULL**, not **/DEBUG** or **/DEBUG:FASTLINK**.



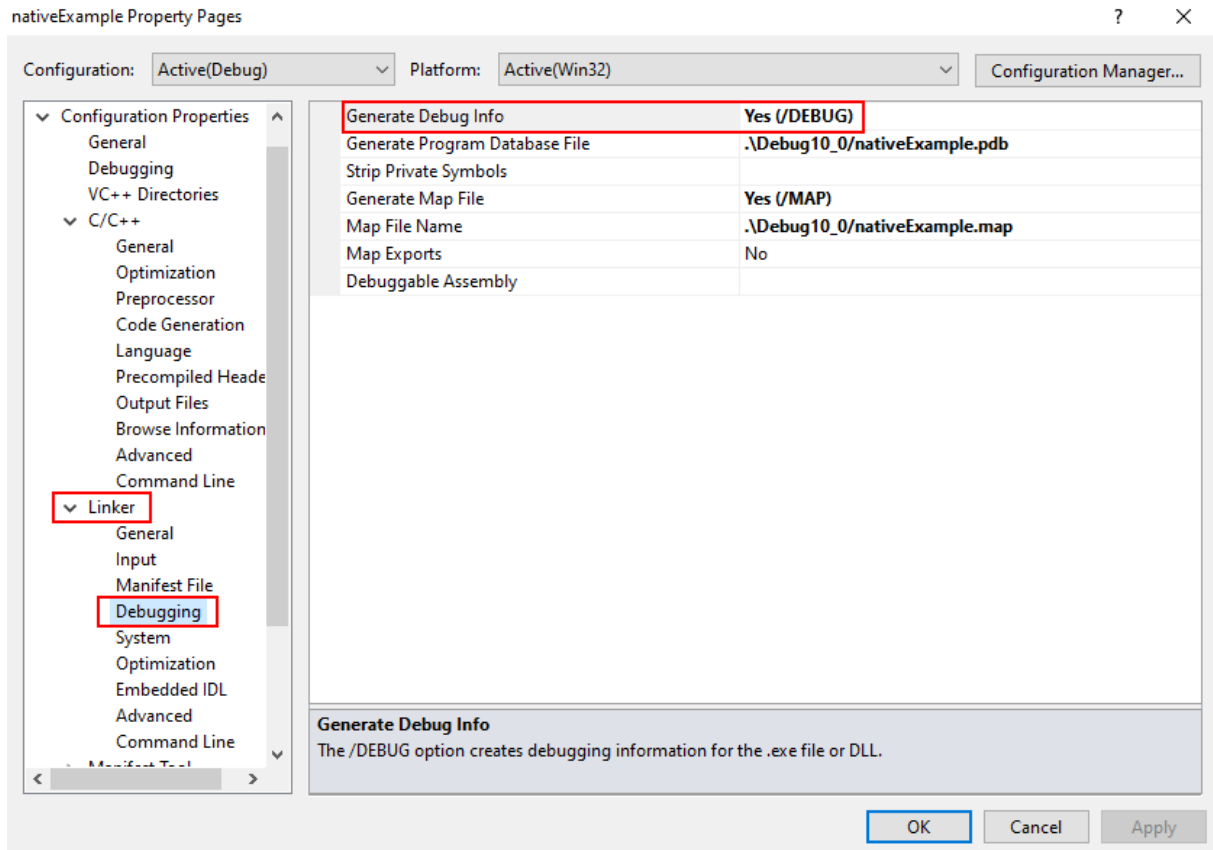
When you have edited the project options you need to rebuild the software for the options to take effect and create the debug information.

Visual Studio 2010 - 2015

Compiler Settings



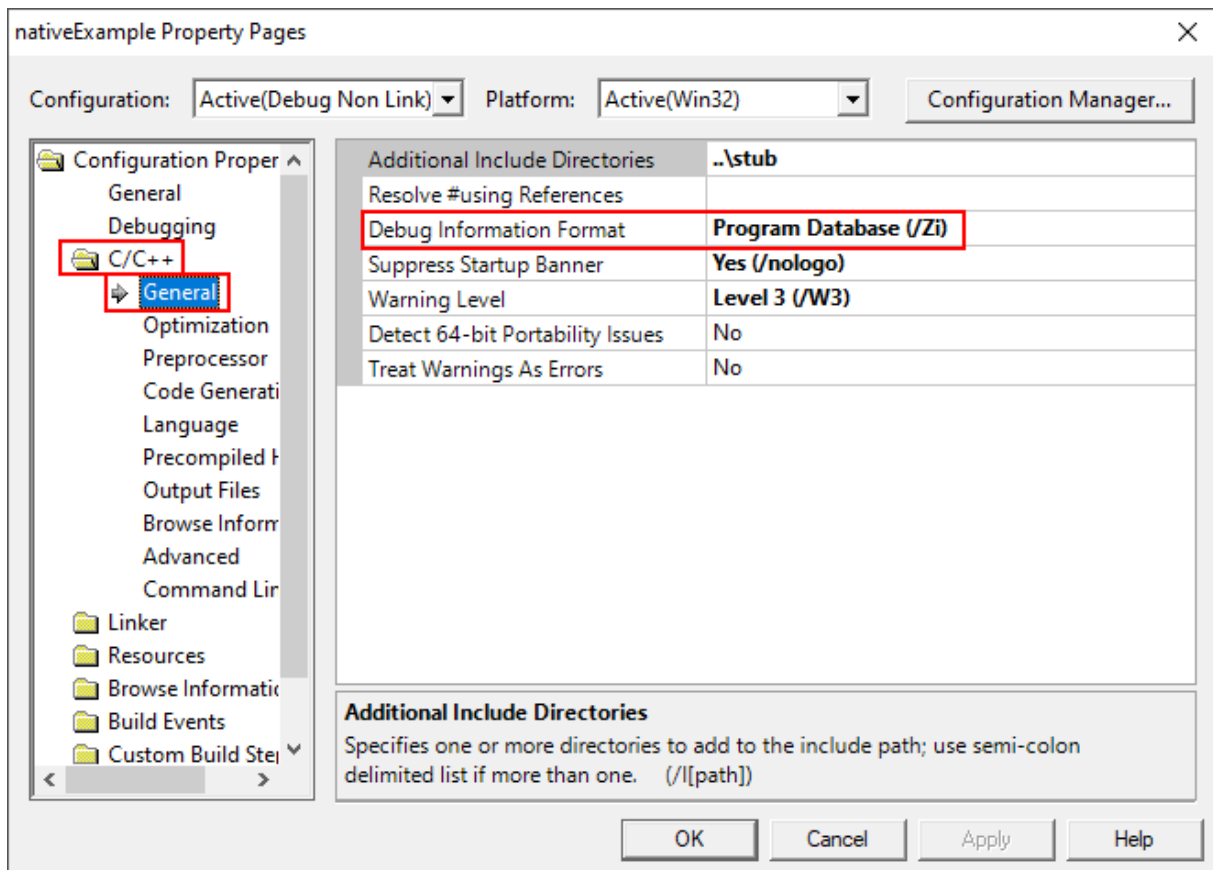
Linker Settings



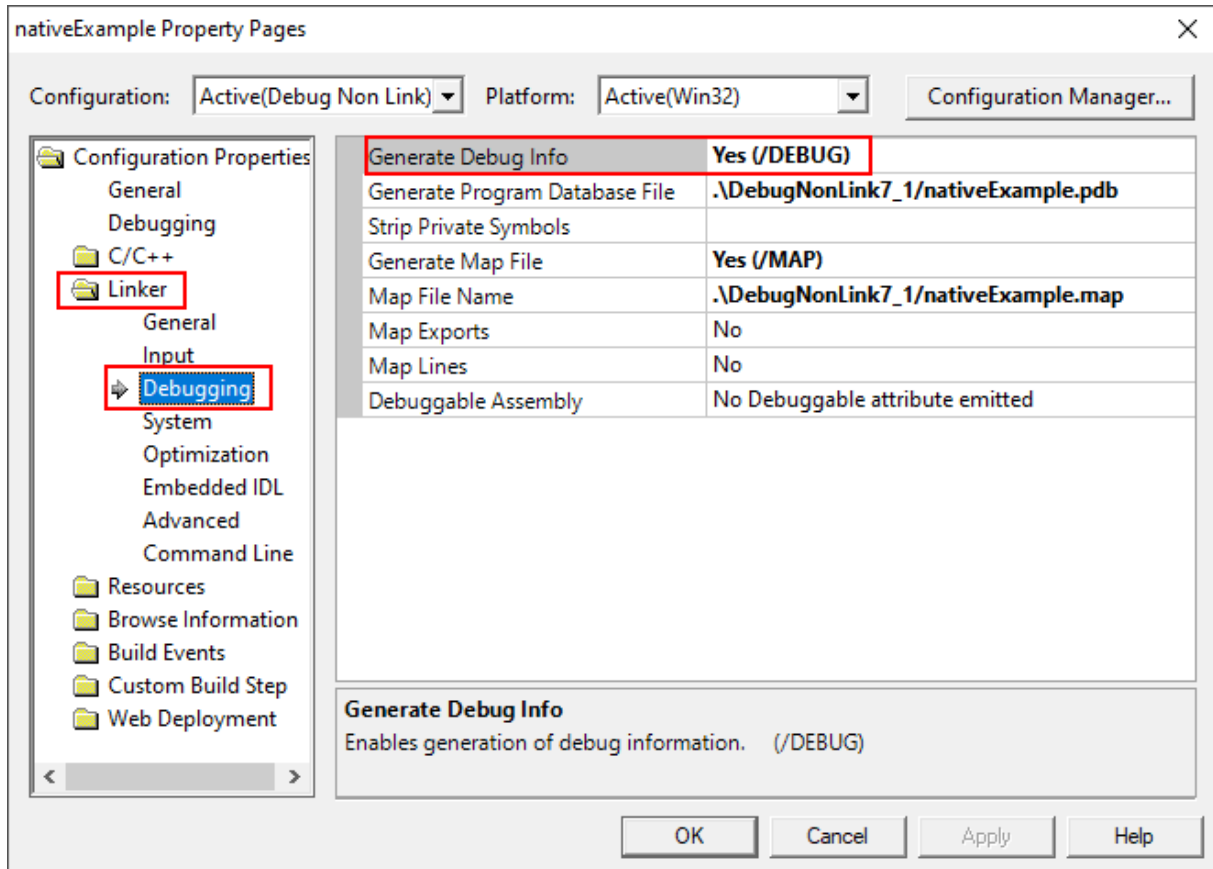
When you have edited the project options you need to rebuild the software for the options to take effect and create the debug information.

Visual Studio 2002 - 2008

Compiler Settings



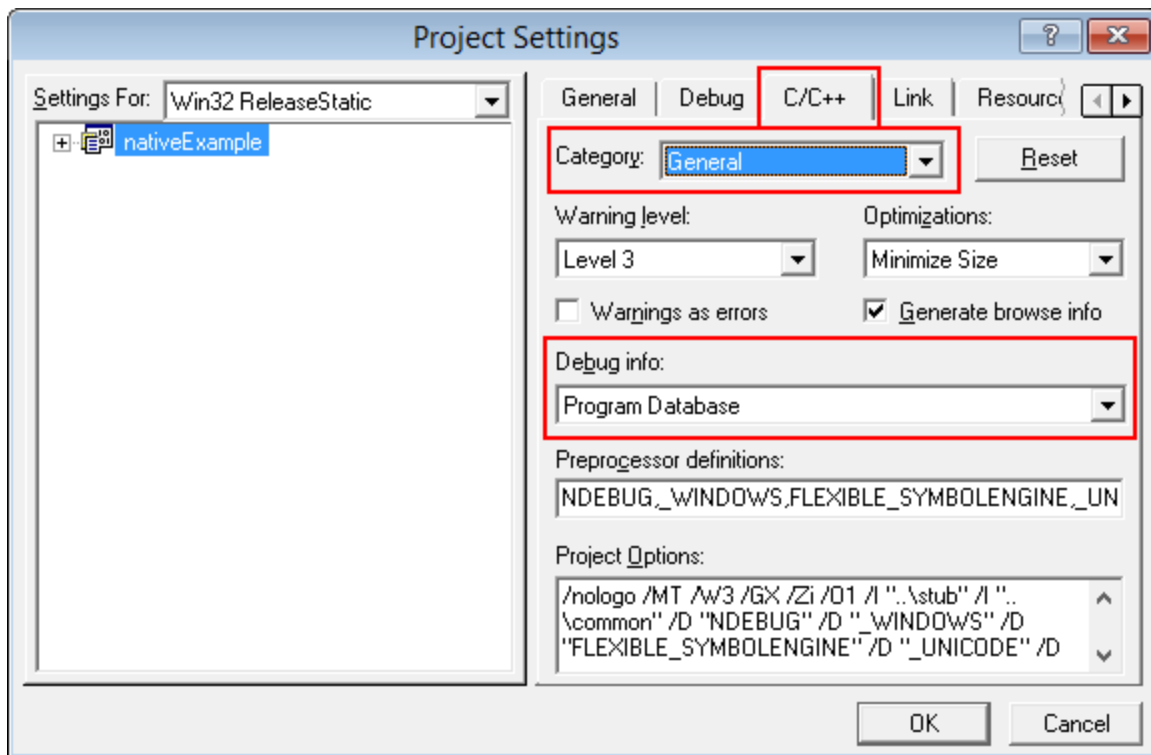
Linker Settings



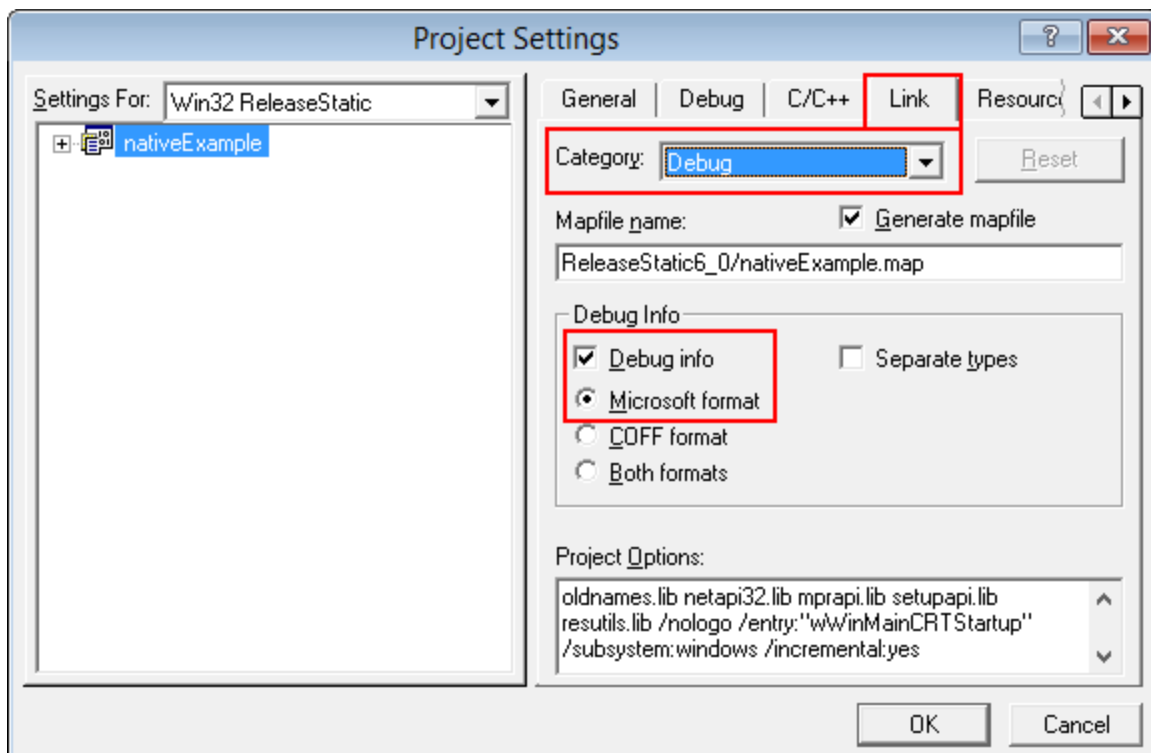
When you have edited the project options you need to rebuild the software for the options to take effect and create the debug information.

Visual Studio 6.0

Compiler Settings



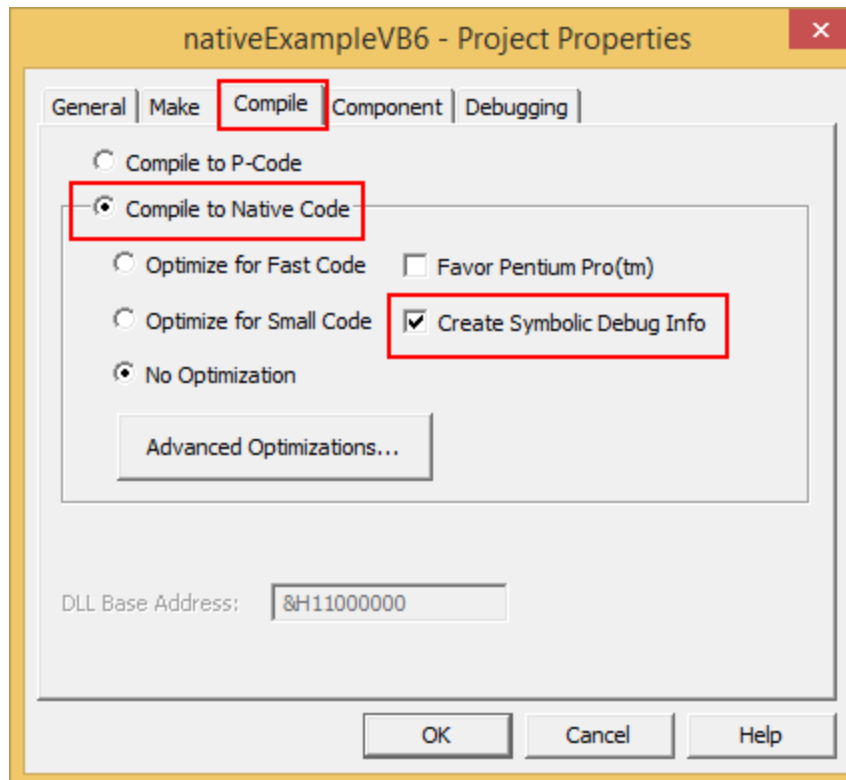
Linker Settings



When you have edited the project options you need to rebuild the software for the options to take effect and create the debug information.

8.2 Visual Basic 6

To get debug symbols for Visual Basic you need to open the **Properties** dialog box from the **Project** menu (you'll find it at the bottom of the menu).



When you have changed your project properties you need to build the application.

Go to the **File** menu and choose **Make <projectname.exe>**.

Part

IX

9 Frequently Asked Questions

This section lists the commonly asked questions about Bug Validator.

9.1 General Questions

Does Coverage Validator work with NT Services?

Absolutely. There is a help section on working with NT Services.

Why might Inject or Launch fail?

Not using CreateProcess

If you are launching your application with any option other than CreateProcess you are effectively using CreateRemoteThread to inject into the application you have just started running using CreateProcess.

The Inject and Wait for Application to Start functionality also use CreateRemoteThread to inject into an application.

For the reasons below, injection using CreateRemoteThread does not always work.

Common reasons for injection failure

- A missing DLL in your application
Check your application is complete.
- The target application is a .NET application or .NET service
Check your application or service is not written using .NET technology.
- A missing DLL in Bug Validator
Check Bug Validator is installed correctly.
- The application may have started and finished before the DLL could be injected
This only applies if you are *launching* the application.
- The application security settings do not allow process handles to be opened
- The application is a service and is running with different privileges than Bug Validator

If the application being injected into is a service it is recommended that the service and Bug Validator are both run on the same user account. See the topic on working with NT services.

Application Specific Reasons for Failure

A small percentage of applications/services will not allow any DLL to be injected into them.

The reasons for this are unknown, but our testing shows that the reason for failure to inject is a combination of application, operating system and hardware that causes an inconsistency during injection (we think it is a timing issue) that causes a failure.

Our tests show that on NT 4 about 1% of all applications fail to inject, 2% on Windows 2000 rising to 5% with Windows XP.

We expect that subsequent operating systems (Windows 2003 and Windows Vista) will have higher failure rates.

How do I clear the symbol cache?

Flush the symbol cache files:

- **Settings Menu > Edit Settings > Hook safety > Clean Instrumentation Cache > Scan and delete symbol cache files > Close > OK**

You may also want to disable the on-disk cache of PDB file symbols:

- **Settings Menu > Edit Settings > Hook safety > deselect Cache instrumentation data > OK**

I have an idea for a feature, can it be added to Coverage Validator?

We have tried to add as many features to Bug Validator that we thought would be useful to our users.

In fact, every feature in Bug Validator has been used to solve problems and bugs for clients who consult us, and in our own business, so we know the features we have are useful.

However, maybe we overlooked a feature that you would find very useful.

We'll happily consider most ideas for new features to Bug Validator. But no Quake, FlightSim or Flappy Bird Easter eggs though, sorry!

Please contact us to let us know your thoughts.

9.2 What file extensions does Bug Validator use for itself?

Bug Validator stores most of the configuration data it needs in the registry. However some data, such as the data hooks, coverage data and filter data is stored in files. This section describes the file extensions used by Bug Validator so that you can recognise such files. The file data structure is not described as it may change from version to version of the product.

Settings, Filters, Coverage

kvs Settings

Program Launch, Extensions

exe Program files

9.3 Crashes and error reports

☐ The program I'm trying to monitor keeps crashing, why?

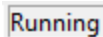
The following assumes your crash is one that only happens when using Bug Validator.

Here's a few scenarios in which your program *might* crash:

- **Third party DLLs are using system wide hooks**

Some DLLs from third party vendors use system wide hooks and do not interact with Bug Validator and the target program very well.

If you can identify such DLLs, prevent them being hooked by adding the DLL name to the Hooked DLLs page of the global settings dialog as in the example below.



- **Third party DLLs are using global hooks**

A global hook DLL from a third party vendor could be adversely affecting Bug Validator when hooking your program.

Read about handling global hooks on the Global Hooks page of the settings dialog.

Judging by multiple independent error reports, we believe there may be an incompatibility between Bug Validator and the global hooks that come with the Matrox G400 and the Matrox Millenium II PCI video cards released in the late 1990's.

- **There may be a bug in Bug Validator**

It happens. We've tried to make Bug Validator as robust as possible, but bugs and new scenarios do occur.

First, ensure that the crash never happens if you are not using Bug Validator.

Second, check all the suggestions above.

Then drop us a line sending details of the error and we'll try to reproduce the crash with a view to fixing any bugs found in as timely a manner as possible.

❏ Bug Validator gives an Unrecoverable Error?

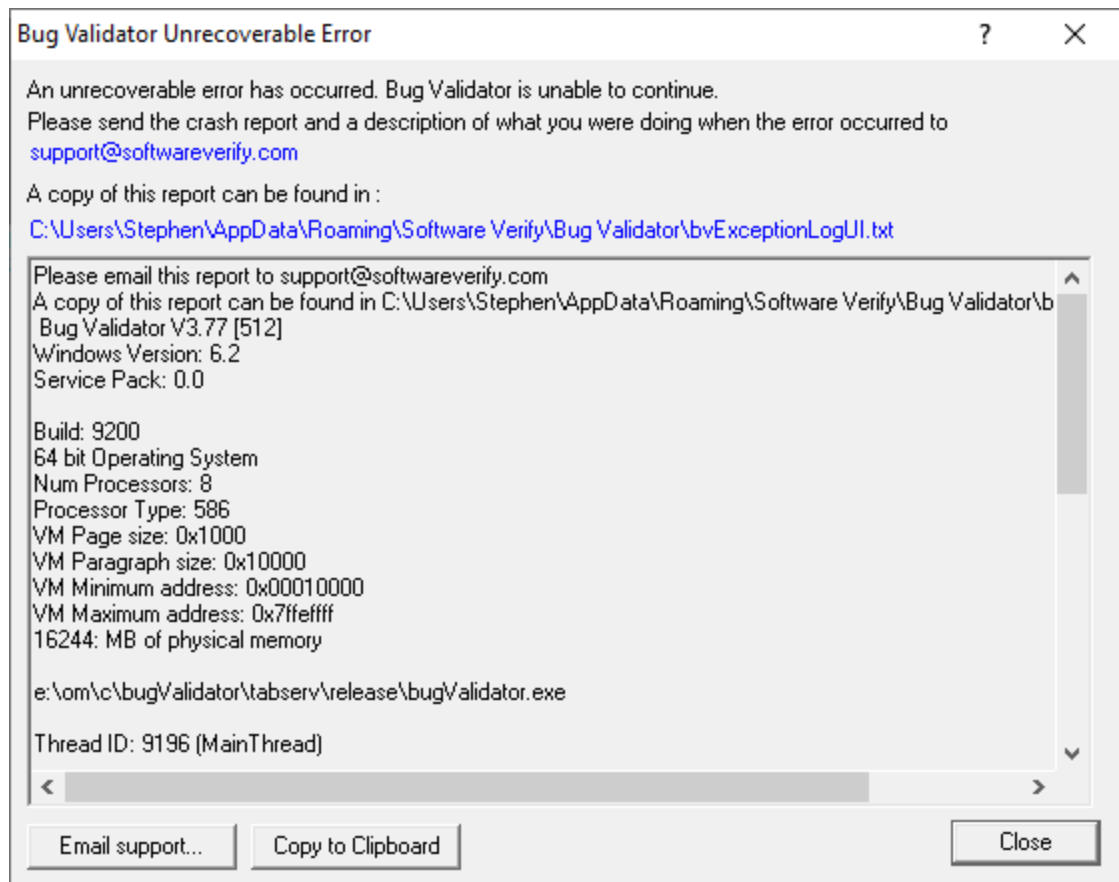
The Bug Validator Unrecoverable Error dialog is displayed when an unexpected internal error means Bug Validator cannot continue to execute.

A stack trace and register dump is shown and you can **Copy to Clipboard** so that the data can be sent to us with a description of the activities that caused the error.

We'll aim to fix any problems in as timely manner as possible.

The data shown in the dialog is also written to `c:\users\<username>\AppData\Roaming\Software Verify\Bug Validator\bvExceptionLogUI.txt`

The picture below shows a stack overflow exception report (we created the crash for this topic).



bvExceptionLogUI.txt

bvExceptionLogUI.txt details a crash in the Bug Validator user interface.

bvExceptionLog.txt

bvExceptionLog.txt details a crash in the target program that Bug Validator was monitoring. The crash may be in the target program or in Bug Validator's monitoring code.

What is in bvExceptionLog.txt?

In the event of a crash, the file `c:\users\<username>\AppData\Roaming\Software Verify\Bug Validator\bvExceptionLogUI.txt` contains information that identifies where Bug Validator was executing when it crashed.

The file contains a stack trace and register dump and is the same information that is displayed in the Unrecoverable Error dialog when a crash occurred.

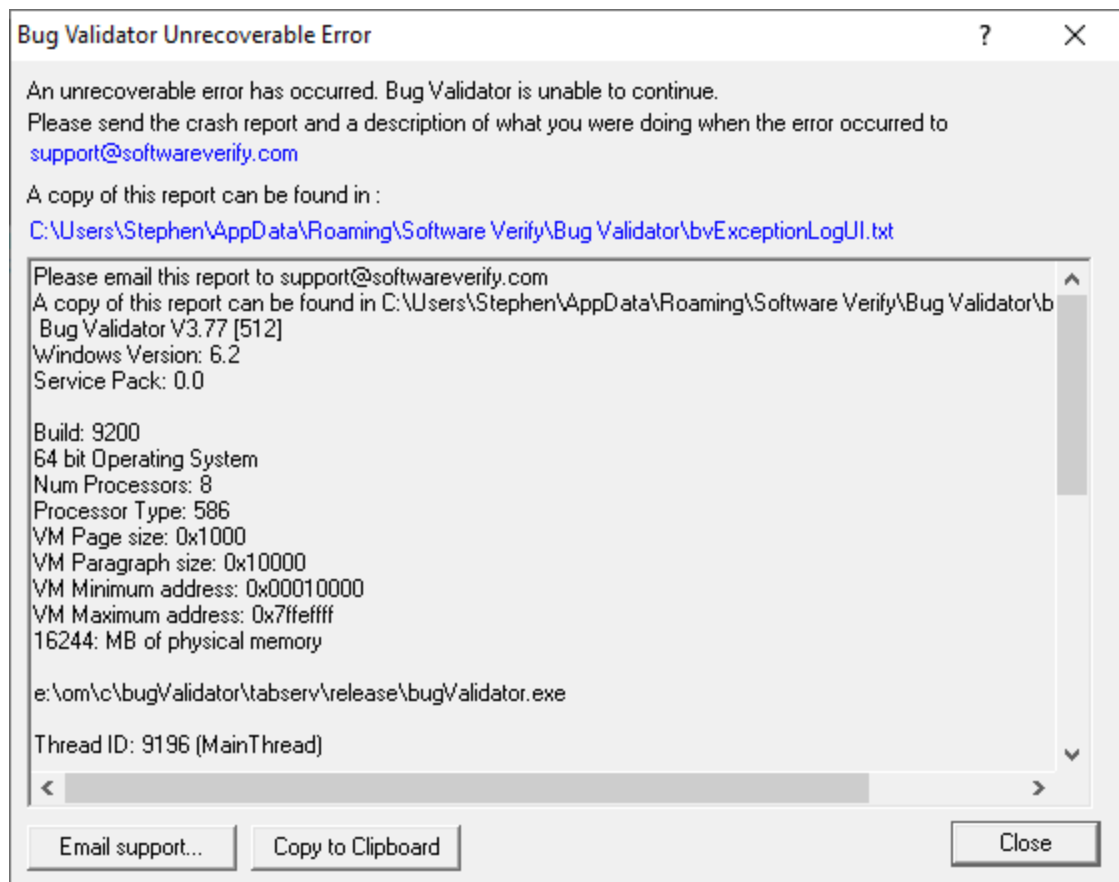
The file contains only the data for the most *recent* exception.

9.4 Bug Validator Unrecoverable Error

The Bug Validator Unrecoverable Error dialog is displayed when an internal error has occurred that Bug Validator did not expect. Bug Validator cannot continue to execute. A stack trace and register dump is shown so that the data can be sent with a description of the activities that caused the error to **support@softwareverify.com**. The data shown in the dialog is also written to `c:\users\<username>\AppData\Roaming\Software Verify\Bug Validator\bvExceptionLogUI.txt`.

The picture below show the exception report for a stack overflow error. The error shown below is artificial and was deliberately caused to allow this picture to be taken.

If you see this dialog, please copy the data and send it with a description of what you were doing with Bug Validator to support@softwareverify.com so that we can fix the bug that caused this error.



bvExceptionLogUI.txt

bvExceptionLogUI.txt details a crash in the Bug Validator user interface.

bvExceptionLog.txt

bvExceptionLog.txt details a crash in the target program that Bug Validator was monitoring. The crash may be in the target program or in Bug Validator's monitoring code.

9.5 What is in bvExceptionLogUI.txt?

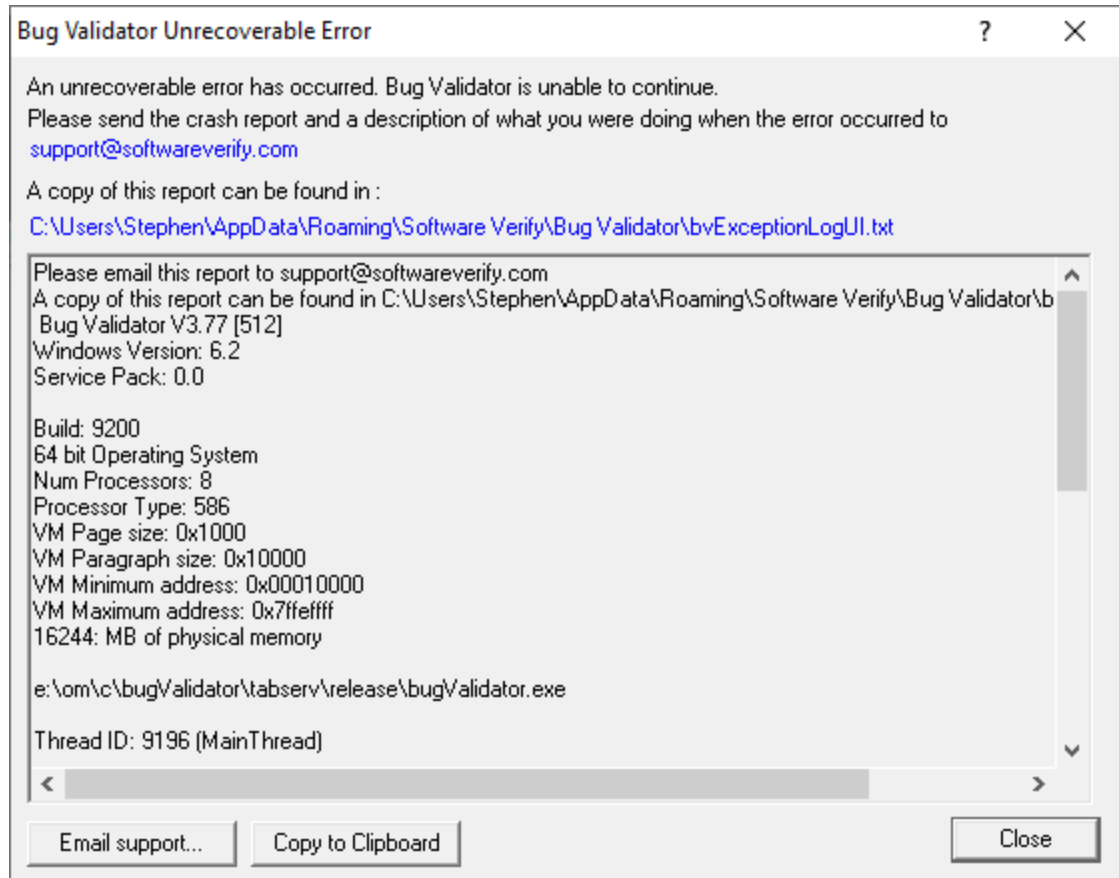
The file **c:\users\<username>\AppData\Roaming\Software Verify\Bug Validator\bvExceptionLogUI.txt** contains information that identifies where Bug Validator was executing when Bug Validator crashed.

This information is also displayed in the exception report dialog that is displayed when a crash occurs.

The file **c:\users\<username>\AppData\Roaming\Software Verify\Bug Validator\bvExceptionLogUI.txt** contains the data for the most recent exception.

The picture below show the exception report for a stack overflow error. The error shown below is artificial and was deliberately caused to allow this picture to be taken.

If you see this dialog, please copy the data and send it with a description of what you were doing with Bug Validator to support@softwareverify.com so that we can fix the bug that caused this error.



bvExceptionLogUI.txt

bvExceptionLogUI.txt details a crash in the Bug Validator user interface.

bvExceptionLog.txt

bvExceptionLog.txt details a crash in the target program that Bug Validator was monitoring. The crash may be in the target program or in Bug Validator's monitoring code.

9.6 How do I create a Power User on Windows XP?

Windows XP provides Power User accounts but does not make it easy to create a user with Power User privileges. To create a user with Power User privileges do the following:

- Create a **Limited User** account (we will call it "Test Limited User").
- Open **Control Panel** and set to Classic View.

- Open **Administrative Tools**.
- Open **Computer Management**.
- In the left hand pane expand **Local Users and Groups**.
- In the left hand pane select **Users**.
- In the right hand pane select the user account you created above ("Test Limited User").
- Right click on "Test Limited User" and choose **Properties** from the context menu.
- Select the **Member Of** tab.
- Click **Add**. A dialog box will be displayed. In the bottom edit box type **Power Users**. Click OK.
- Select the **Users** entry. Click **Remove**. Click OK.
- Close the **Computer Management** window.

Your Test Limited User is now a member of the Power Users group. Now is a good time to rename the account to something more appropriate.

9.7 Why does Bug Validator fail to load my symbols?

In a few cases Bug Validator will fail to load symbols for a DLL that you believe you have provided symbols for. This topic describes the possible causes. Please read the suggested course of action for each compiler.

Microsoft Developer Studio / Microsoft Visual Studio

Symbols are defined in PDB files. A PDB file has the same name as the DLL to which it refers, with a PDB extension instead of DLL or EXE. For example:

text.exe would have a PDB file called **test.pdb**.
features.dll would have a PDB file called **features.pdb**.

Bug Validator uses the Microsoft supplied DbgHelp.dll to perform all symbol handling activities. When searching for the symbols for a DLL, DbgHelp expects to find a PDB file with the correct checksum that matches the DLL. If DbgHelp finds a PDB file with a different checksum, DbgHelp will fail to load the symbols and the search for a symbol file for the DLL will end.

To ensure that the correct DLL and PDB file are found the following need to be TRUE.

- The DLL and PDB file have the same name, except for the extension.
- The first PDB file found with the appropriate name when traversing the PDB search path is the PDB file with the correct checksum.

If DbgHelp is failing to load your symbols (you can check the diagnostics tab for messages indicating such a failure), you need to check the following:

- Ensure that your EXE/DLL is compiled to include symbol information (these are different options to the linker options).
- Ensure that your EXE/DLL is linked to include symbol information (these are different options to the compiler options).
- You are using the most recent version of your DLL.
- You are using the correct version of your DLL (release DLL with release builds, debug DLL with debug builds).
- Verify the above when your application is running by checking the modules loaded by the application. You can check the modules by using the Loaded Modules dialog, or by inspecting the

diagnostics tab. You need to be sure that your application is not loading a different DLL with the same name from a different directory that is on the search path.

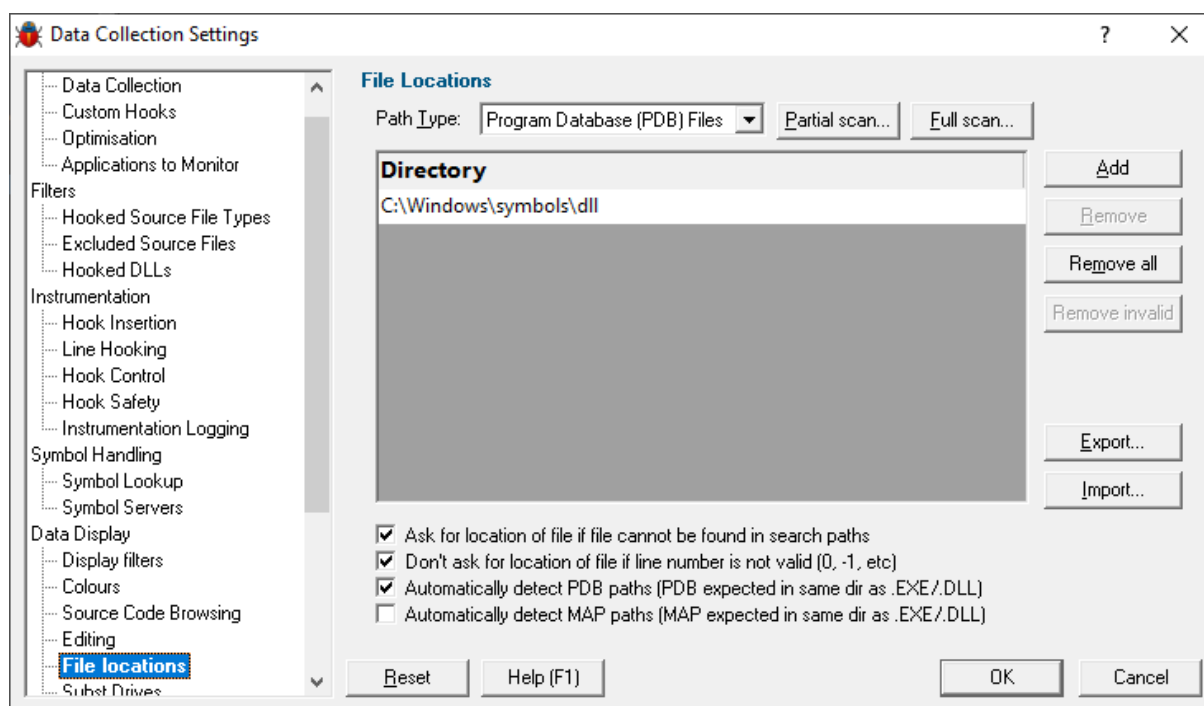
- Verify that there are no PDB files with the same file name that are on the PDB search path, except for the PDB file you expect to be used.
- Check the version of DbgHelp.dll used by your Visual Studio installation and the version of DbgHelp.dll distributed with Bug Validator. If the version of DbgHelp.dll used by Visual Studio is higher than the version distributed with Bug Validator it is possible that Microsoft have changed the PDB file format. This will result in Bug Validator being unable to read the symbols. To fix this:
 - Copy the DbgHelp.dll from Visual Studio to the Bug Validator installation directory.
 - When Bug Validator launches an application it copies Bug Validator's DbgHelp.dll to the directory of the executable to ensure that the DbgHelp.dll that is more recent than the default system32\debughelp.dll (which doesn't get updated by Windows Update). You need to remove these dlls, search for them and delete them (they will be in paths such as c:\myapplication\debug, etc).

Some symbolic information will not load for reasons unknown. In this circumstance, you should (after trying the above suggestions) try changing the location in which symbols are sourced, and also disabling the caching of symbols. Information on this topic is available [here](#).

If after checking all of the above you still have problems, please contact support@softwareverify.com.

Visual Studio 2005 (Visual Studio 8.0)

You may find that symbols for the MSVC80.DLL, MSVC80D.DLL, MF80.DLL, MFC80U.DLL, MFC80D.DLL and MFC80UD.DLL dlls are not loaded. The reason for this is that these symbols are stored in **c:\windows\symbols\dl** rather than with the DLLs themselves. This is due to the Windows.NET Side-by-Side (WinSxS) DLL/assembly loading. Add the path **c:\windows\symbols\dl** to the list of paths for Program Database Files on the File Locations tab.



You may need to restart Bug Validator to get valid symbols for MFC80(u)(d).dll if you have already recorded a session for which you did not get symbols. Alternatively you can do the following actions to clear the symbol cache.

- Open the Session Manager from the Managers menu.
- Delete All sessions.
- Close the Session Manager.
- Open the Settings Dialog (in Intermediate or Expert user interface mode).
- Go to the File Cache / Subst Drives tab.
- Click the Flush Cache button.
- Click OK.

You may also want to disable Bug Validator's on-disk cache of symbols read from PDB files. To do this take the following actions:

- Open the Settings Dialog (in Intermediate or Expert user interface mode).
- Go to the Symbol Lookup tab.
- Deselect the **Enable caching...** check box.
- Click OK.

Metrowerks CodeWarrior for Windows V8 & V9

Metrowerks symbolic information is embedded in the .exe/.dll as CodeView information. Please consult the documentation for CodeWarrior to include debug information (including filenames and line numbers) in the CodeView information.

If after checking all of the above you still have problems, please contact support@softwareverify.com.

Salford Software FORTRAN95

Salford FORTRAN95 symbolic information is embedded in the .exe/.dll as COFF (Common Object File Format) information, with some extensions proprietary to Salford Software (which they have kindly shared with Software Verify). Please consult the documentation for Salford FORTRAN95 to include debug information (including filenames and line numbers) in the COFF information.

If after checking all of the above you still have problems, please contact support@softwareverify.com.

9.8 How do I examine the DbgHelp symbol search path?

When symbols fail to load for modules built with compilers that generate PDB files (Microsoft C++ compilers and linkers, Intel performance compilers and linkers) it can be confusing why the failure has occurred.

There are typically three broad classes of success/failure when trying to load symbols in a PDB file:

1. a missing PDB file
2. a PDB file located in the wrong place (you think it's present but the debugging library can't find it)
3. an incorrect PDB file (you think it's correct but it's actually from the wrong build)

The diagnostic tab of Bug Validator displays many types of information that can help diagnose many problems when working with Bug Validator.

These information groups can be selected using the combo box at the top of the diagnostic display. The DbgHelp debug group allows us to examine where DbgHelp.dll looks for symbols. By examining the output we can identify if it is finding the PDB file we think it should be finding and if it likes the contents of any PDB file it finds.

The output for alternate modules is shown in alternate colours. This makes it easier for you to determine which output is for the DLL you are interested in. The output shown is the exact same output that DbgHelp.dll sends to its debugging stream. We have added nothing. We have removed nothing. We believe showing you the exact information DbgHelp.dll outputs is the best way to help you determine what actions to take to resolve any issue with symbol loading.

Below we show three examples using nativeExample.exe and nativeExample.pdb.

Correct symbol file for nativeExample.exe

Show:

DbgHelp debug

 Filter:

Apply Filter

ID	Message
DbgHelp Search Info	DBGHELP: Symbol Search Path: C:\WINDOWS\symbols\dlldata\Program Files (x86)\Software Verify\Bug Validator x86\examples\nativeExample\ReleaseDynamic10_0;
DbgHelp Search Info	DBGHELP: C:\WINDOWS\symbols\dlldata\nativeExample.pdb - file not found
DbgHelp Search Info	DBGHELP: C:\WINDOWS\symbols\dlldata\exe\nativeExample.pdb - file not found
DbgHelp Search Info	DBGHELP: C:\WINDOWS\symbols\dlldata\symbols\exe\nativeExample.pdb - file not found
DbgHelp Search Info	DBGHELP: nativeExample - private symbols & lines
DbgHelp Search Info	C:\Program Files (x86)\Software Verify\Bug Validator x86\examples\nativeExample\ReleaseDynamic10_0\nativeExample.pdb

DbgHelp searches in various places looking for nativeExample.pdb.

Eventually nativeExample.pdb is found in c:\Program Files (x86)\Software Verify\Bug Validator x86\examples\nativeExample\ReleaseDynamic10_0\nativeExample.pdb.

DbgHelp loads private symbols and lines. (The alternate outcome is that DbgHelp loads public symbols).

Outcome: Success. Symbols are loaded.

Missing symbol file for nativeExample.exe

Show:

DbgHelp debug

 Filter:

Apply Filter

ID	Message
DbgHelp Search Info	DBGHELP: C:\WINDOWS\symbols\dlldata\nativeExample.pdb - file not found
DbgHelp Search Info	DBGHELP: C:\WINDOWS\symbols\dlldata\exe\nativeExample.pdb - file not found
DbgHelp Search Info	DBGHELP: C:\WINDOWS\symbols\dlldata\symbols\exe\nativeExample.pdb - file not found
DbgHelp Search Info	DBGHELP: C:\Program Files (x86)\Software Verify\Bug Validator x86\examples\nativeExample\ReleaseDynamic10_0\nativeExample.pdb - file not found
DbgHelp Search Info	DBGHELP: nativeExample - no symbols loaded

DbgHelp has the search path set then searches in various places looking for nativeExample.pdb.

nativeExample.pdb never gets found on the search path. SymSrv then looks for additional locations for nativeExample.pdb. None are found.

DbgHelp does find some COFF symbols in the executable. Unfortunately COFF symbols do not contain filename or line number information.

Outcome: Failure. The PDB file could not be found. Some default symbols are loaded but are not of much use,

Resolution: Check the File Locations PDB paths to ensure that all the possible paths for PDB files are listed.

Incorrect symbol file for nativeExample.exe

Show:	DbgHelp debug	Filter:		Apply Filter
ID	Message			
DbgHelp Search Info	DBGHELP: Symbol Search Path: C:\WINDOWS\symbols\dl;C:\Program Files (x86)\Software Verify\Bug Validator x86\examples\nativeExample\ReleaseDynamic10_0;			
DbgHelp Search Info	DBGHELP: C:\WINDOWS\symbols\dl\nativeExample.pdb - file not found			
DbgHelp Search Info	DBGHELP: C:\WINDOWS\symbols\dl\exe\nativeExample.pdb - file not found			
DbgHelp Search Info	DBGHELP: C:\WINDOWS\symbols\dl\symbols\exe\nativeExample.pdb - file not found			
DbgHelp Search Info	DBGHELP: C:\Program Files (x86)\Software Verify\Bug Validator x86\examples\nativeExample\ReleaseDynamic10_0\nativeExample.pdb - mismatched pdb			
DbgHelp Search Info	DBGHELP: C:\Program Files (x86)\Software Verify\Bug Validator x86\examples\nativeExample\ReleaseDynamic10_0\exe\nativeExample.pdb - file not found			
DbgHelp Search Info	DBGHELP: C:\Program Files (x86)\Software Verify\Bug Validator x86\examples\nativeExample\ReleaseDynamic10_0\symbols\exe\nativeExample.pdb - file not found			
DbgHelp Search Info	DBGHELP: C:\Program Files (x86)\Software Verify\Bug Validator x86\examples\nativeExample\ReleaseDynamic10_0\nativeExample.pdb - mismatched pdb			
DbgHelp Search Info	DBGHELP: Couldn't load mismatched pdb for C:\Program Files (x86)\Software Verify\Bug Validator x86\examples\nativeExample\ReleaseDynamic10_0\nativeExample.exe			
DbgHelp Search Info	DBGHELP: nativeExample - no symbols loaded			

DbgHelp searches in various places looking for nativeExample.pdb.

Eventually nativeExample.pdb is found in c:\Program Files (x86)\Software Verify\Bug Validator x86\examples\nativeExample\ReleaseDynamic10_0\nativeExample.pdb.

DbgHelp attempts to load the symbols but fails because the symbols are for a different build of the software. The checksum inside the PDB file does not match the module.

DbgHelp does find some COFF symbols in the executable. Unfortunately COFF symbols do not contain filename or line number information.

Outcome: Failure. A PDB file was found but it was not the correct PDB file. Some default symbols are loaded but are not of much use,

Resolution:

- Ensure the PDB file found is the correct PDB file for the build. If you are copying builds from a build server be sure to copy the correct PDB files as well.
- Check the File Locations PDB paths to ensure that all the possible paths for PDB files are listed in the correct order so that if multiple paths have a PDB file with the same name that the correct PDB file is found first.

9.9 Why are some functions not hooked?

Bug Validator instruments your application by re-writing the prologue and epilogue of each function in your application, inserting code to monitor code Bug. Before inserting the code Bug Validator checks to ensure that it is safe to re-write the function prologue and epilogue. If it is not safe to re-write the function epilogue and prologue the function cannot be instrumented.

The following items can prevent the function from being hooked:

- Function too short to hook.
- Function has multiple exits.
- Function has jumps into epilogue.
- Function has jumps into prologue.

- Function cannot be disassembled.
- Instruction sequence cannot be hooked.

You can improve the likelihood of your function being hooked by enabling the check boxes on the Hook Control settings.

9.10 Why are some lines not hooked?

Bug Validator instruments the lines in your application by inserting code to recognise the execution of the start of every line in your application. Before inserting the code Bug Validator checks to ensure that it is safe to re-write the function lines. If it is not safe to re-write the function epilogue and prologue the function cannot be instrumented.

The following items can prevent the function from being hooked:

- Function too short to hook. The function must be at least 5 bytes in length.
- The code for the line is too short to hook. The code for the line must be at least 5 bytes in length.
- Function cannot be disassembled.
- Instruction sequence cannot be hooked.

You can improve the likelihood of the lines in each function being hooked by enabling the check boxes on the Hook Control settings.

9.11 Debug symbols and DbgHelp

Why does Coverage Validator fail to load my symbols?

In a few cases Bug Validator will fail to load symbols for a DLL that you believe you have provided symbols for.

This topic describes the possible causes. Please read the suggested course of action for each compiler.

Microsoft Visual Studio or Developer Studio

Symbols are defined in PDB files with the same name as the .exe or .dll to which it refers.

Bug Validator uses the Microsoft supplied DbgHelp.dll to perform all symbol handling activities.

Correct PDB name and location?

To ensure that the correct PDB is found to match a DLL the following must be true:

- The DLL and PDB file have the same name, except for the extension

For example test.pdb matches for test.dll or test.exe.

- The first matching PDB file in the PDB search path has the correct checksum

If DbgHelp finds a PDB file with a different checksum, loading symbols will fail but the search will still stop.

Verify that there are no PDB files with the same file name that are on the PDB search path, except for the PDB file you expect to be used.

You can check the DbgHelp symbol search path to troubleshoot symbol loading failures relating to the symbol search path.

Are compiler and linker producing symbols?

If DbgHelp is still failing to load your symbols, check the following:

- Your program is **compiled** to include symbol information
- Your program is **linked** to include symbol information

Linker options are different to the compiler options

Running correct version of DLL?

Check that you are using:

- The **most recent** version of your DLL
- The **correct build** version of your DLL

For example release DLL with release builds, debug DLL with debug builds

Checking for correctly loaded modules

When your application is running, check the modules being loaded by the application.

In Bug Validator, you can check the modules by using the Loaded Modules dialog, or by inspecting the Diagnostics tab.

You need to be sure that your application is not loading a different DLL with the same name from a different directory that is on the search path.

Correct version of DbgHelp.dll?

Try checking the version of DbgHelp.dll used by your Visual Studio installation and the version of DbgHelp.dll distributed with Bug Validator.

If the version used by Visual Studio is higher, it's possible Microsoft changed the PDB file format, making the symbols unreadable by Bug Validator.

To fix this:

- Copy the DbgHelp.dll from Visual Studio to the Bug Validator installation directory
- Remove any DbgHelp.dll from your application directory

When Bug Validator launches an application it copies Bug Validator's DbgHelp.dll to the directory of the executable.

This ensure that the DbgHelp.dll used is more recent than the default `system32\dbghelp.dll` which may not get updated.

You need to find and remove these dlls - e.g. `c:\myapplication\debug\DbgHelp.dll` etc.

If all else fails...

Sometimes symbolic information will not load for unknown reasons.

In this circumstance, after trying the above suggestions, try changing the location in which symbols are sourced.

You could also try flushing and disabling the caching of symbols.

If you still have problems, please contact us giving as much detail as possible, including what you've tried.

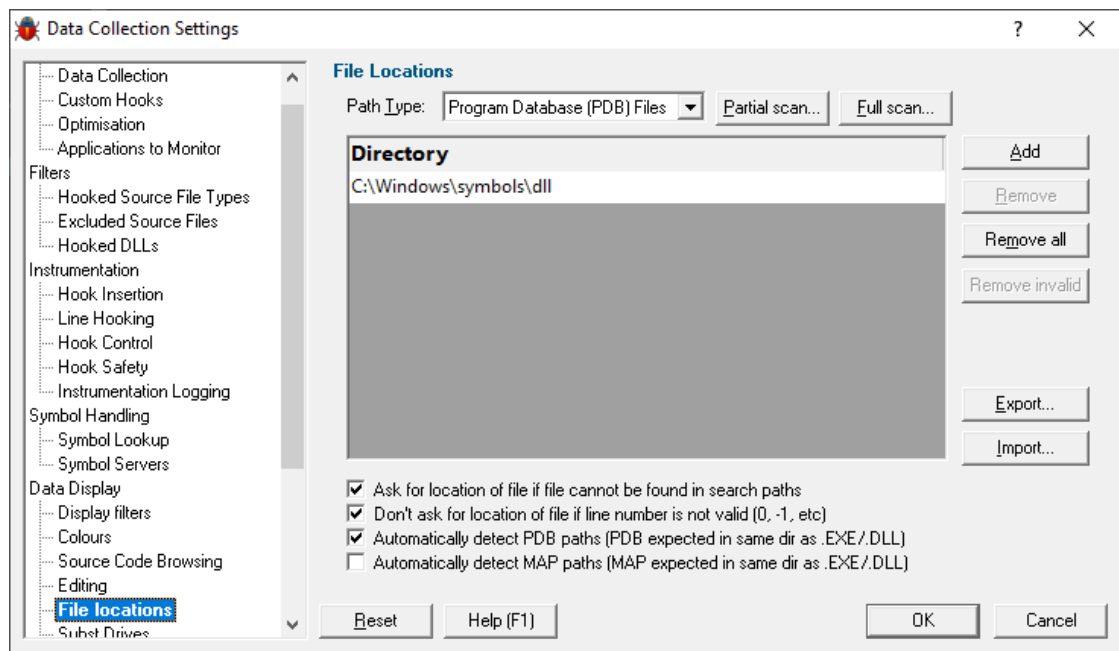
Visual Studio 2005 (8.0) and later versions

You may find that symbols for the `msvcr80.dll`, `msvcr80d.dll`, `mf80.dll`, `mfc80u.dll`, `mfc80d.dll` and `mfc80ud.dll` DLLs are not loaded.

The reason for this is that these symbols are stored in `c:\windows\symbols\dll` rather than with the DLLs themselves.

This is due to the Windows.NET Side-by-Side (WinSxS) DLL/assembly loading.

To resolve this, add the path `c:\windows\symbols\dll` to the list of paths for Program Database Files on the File Locations tab:



You may need to restart Bug Validator to get valid symbols for `MFC80 (u) (d) .dll` if you have already recorded a session for which you did not get symbols.

Alternatively follow the instructions in the question on how to clear the symbol cache:

Metrowerks CodeWarrior for Windows V8 / V9

Metrowerks symbolic information is embedded in the `.exe/.dll` as CodeView information.

Please consult the documentation for CodeWarrior in order to include debug information (including filenames and line numbers) in the CodeView information.

If you still have problems, please contact us giving as much detail as possible, including what you've tried.

Salford Software Fortran 95

Salford Fortran 95 symbolic information is embedded in the `.exe/.dll` as COFF (Common Object File Format) information, with some proprietary extensions to Salford Software (which they have kindly shared with us).

Please consult the documentation for Salford FORTRAN95 to include debug information (including filenames and line numbers) in the COFF information.

If you still have problems, please contact us giving as much detail as possible, including what you've tried.

MingW compiler

We recommend compiling your software with `-gstabs` to create stabs debugging information.

The `-gCoff` option is also supported, but this does create a lot of unnecessary symbols, making symbol parsing slower.

Troubleshooting DbgHelp.dll

Bug Validator uses the Microsoft Debugging DLL, DbgHelp.dll, copying the correct private version to your application's directory as your program is started.

However, there are cases where your application can be started independently, and you must ensure that your application uses the correct DbgHelp.dll.

Diagnostic error messages appear on the Diagnostics tab as in the example below detailing which version of DbgHelp.dll was expected and what was actually loaded.

DbgHelp.dll version	C:\Program Files (x86)\Software Verification\C++ Coverage Validator\cvExample\DebugNonLinkANSI_0\dbghelp.dll
DbgHelp.dll version	DbgHelp.dll version loaded into target: 6.3.16.1
DbgHelp.dll version	DbgHelp.dll version expected: 6.11.1.404
DbgHelp.dll version warning	DbgHelp.dll loaded has a lower version number than the DbgHelp.dll that ships with C++ Coverage Validator.
DbgHelp.dll version warning	This may cause failures when trying read debugging information (Symbols, Filenames, Line Numbers).
DbgHelp.dll version warning	DbgHelp.dll prior to 6.0 will not work properly. DbgHelp.dll 6.9 or better is preferred.
DbgHelp.dll version warning	For best results you need to ensure that C++ Coverage Validator's DbgHelp.dll is found on the \$PATH before the DbgHelp.dll that is being loaded.
DbgHelp.dll version warning	You can usually do this by putting the current directory (".") at the start of your \$PATH.

If you see any DbgHelp warning dialogs, or get diagnostic errors, ensure the correct DbgHelp.dll is used by:

- **Copy (don't move) DbgHelp.dll**

from: the Bug Validator install directory

to: the location of the application being tested (the same directory as the .exe).

Rerun your test.

- **Try updating the versions of DbgHelp.dll in:**

```
c:\windows\system32
```

and

```
c:\windows\system32\dllcache
```

Accept any Windows permission warnings if you try to do this.

Rerun your test.

If you *still* continue to have problems, please drop us a line via our support email.

How do I examine (and fix) the DbgHelp symbol search path?

It can be confusing to see why symbols fail to load for modules built with compilers that generate PDB files, e.g.: Microsoft, Intel.

There are typically three reasons for failure: the PDB file is...

- **missing**, for example it was not provided with the executable
- **in the wrong place**, so the debugging library can't find it
- **the wrong version**, for example from a different build

The diagnostic tab

The Diagnostic tab of Bug Validator displays lots of messages that can help diagnose many problems.

To show only DbgHelp debug information, use the message filter drop down at the top of the diagnostic tab. This lets you examine where DbgHelp.dll looks for symbols.

Examine the output to see if it's finding the PDB file you think it should, and if it rejects the contents of any PDB file it finds.

Output for alternate modules is shown in alternating coloursets, and the messages are the exact same output from the DbgHelp.dll debugging stream.

Examples of examining the diagnostics

Below we show three examples using bvExample.exe and bvExample.pdb from our example application.

- **Correct symbol file found**

DbgHelp first searches in various places looking for bvExample.pdb

DbgHelp Search Info	DBGHELP: C:\Windows\symbols\dlldata\cvExample.pdb - file not found
DbgHelp Search Info	DBGHELP: C:\Windows\symbols\dlldata\exe\cvExample.pdb - file not found
DbgHelp Search Info	DBGHELP: C:\Windows\symbols\dlldata\symbols\exe\cvExample.pdb - file not found

Depending on your machine, there may be other search paths included.

Finally bvExample.pdb is found in the same directory as the .exe file of the target program

DbgHelp Search Info	DBGHELP: cvExample - private symbols & lines
DbgHelp Search Info	C:\Program Files (x86)\Software Verification\C++ Coverage Validator\cvExample\DebugNonLinkANSI9_0\cvExample.pdb
Loaded symbols	Loaded PDB symbols for:C:\Program Files (x86)\Software Verification\C++ Coverage Validator\cvExample\DebugNonLinkANSI9_0\cvExample.exe

DbgHelp loads private symbols and lines, (the alternative being that DbgHelp loads public symbols).

Outcome:

Success. Symbols are loaded.

- **Missing symbol file**

As before, DbgHelp first searches in various places looking for bvExample.pdb

But, bvExample.pdb doesn't get found in the same directory as the .exe file of the target program.

DbgHelp Search Info	DBGHELP: c:\Program Files (x86)\Software Verification\C++ Coverage Validator\cvExample\DebugNonLinkANSB_0\cvExample.pdb - file not found

bvExample.pdb never gets found on the search path.

SymSrv might then look for additional locations for bvExample.pdb, but has no luck.

DbgHelp might find some COFF symbols in the executable, however these don't contain filename or line number information.

Finally all options are exhausted.

DbgHelp Search Info	DBGHELP: cvExample - no symbols loaded

Outcome:

Failure. The PDB file could not be found. Some default symbols are loaded but are not of much use.

Resolution:

Check the File Locations PDB paths to ensure that all the possible paths for PDB files are listed.

- **Incorrect symbol file**

As before, DbgHelp first searches in various places looking for bvExample.pdb

This time, bvExample.pdb *does* get found in the same directory as the .exe file of the target program.

DbgHelp tries to load the symbols but fails - the checksum inside the PDB file does not match the module.

This might be because the symbols are for a different build of the software, or it's an incorrectly named PDB file belonging to another program.

DbgHelp Search Info	DBGHELP: C:\Program Files (x86)\Software Verification\C++ Coverage Validator\cvExample\DebugNonLinkANSB_0\cvExample.pdb - mismatched pdb
DbgHelp Search Info	DBGHELP: C:\Program Files (x86)\Software Verification\C++ Coverage Validator\cvExample\DebugNonLinkANSB_0\exe\cvExample.pdb - file not found
DbgHelp Search Info	DBGHELP: C:\Program Files (x86)\Software Verification\C++ Coverage Validator\cvExample\DebugNonLinkANSB_0\symbols\exe\cvExample.pdb - file not found
DbgHelp Search Info	DBGHELP: c:\Program Files (x86)\Software Verification\C++ Coverage Validator\cvExample\DebugNonLinkANSB_0\cvExample.pdb - mismatched pdb
DbgHelp Search Info	DBGHELP: Couldn't load mismatched pdb for C:\Program Files (x86)\Software Verification\C++ Coverage Validator\cvExample\DebugNonLinkANSB_0\cvExample.exe

Finally all options are exhausted.

DbgHelp Search Info	DBGHELP: cvExample - no symbols loaded

Outcome:

Failure. A PDB file was found, but it was not the right one.

Resolutions:

Double check the PDB is the correct one for the build you are running.

When copying builds from another machine (or from a build server), make sure to copy the correct PDB as well.

Check the File Locations PDB paths to ensure that all the possible paths for PDB files are listed.

Check the order of those PDB paths in case there are multiple paths resulting in the wrong PDB being found first.

 How can I create a map file with line numbers

If you don't have the ability to use .PDB files for debug information, you may be able to use .MAP files with line information.

The following is only applicable to Debug builds. Map files for Release builds can't have line number data.

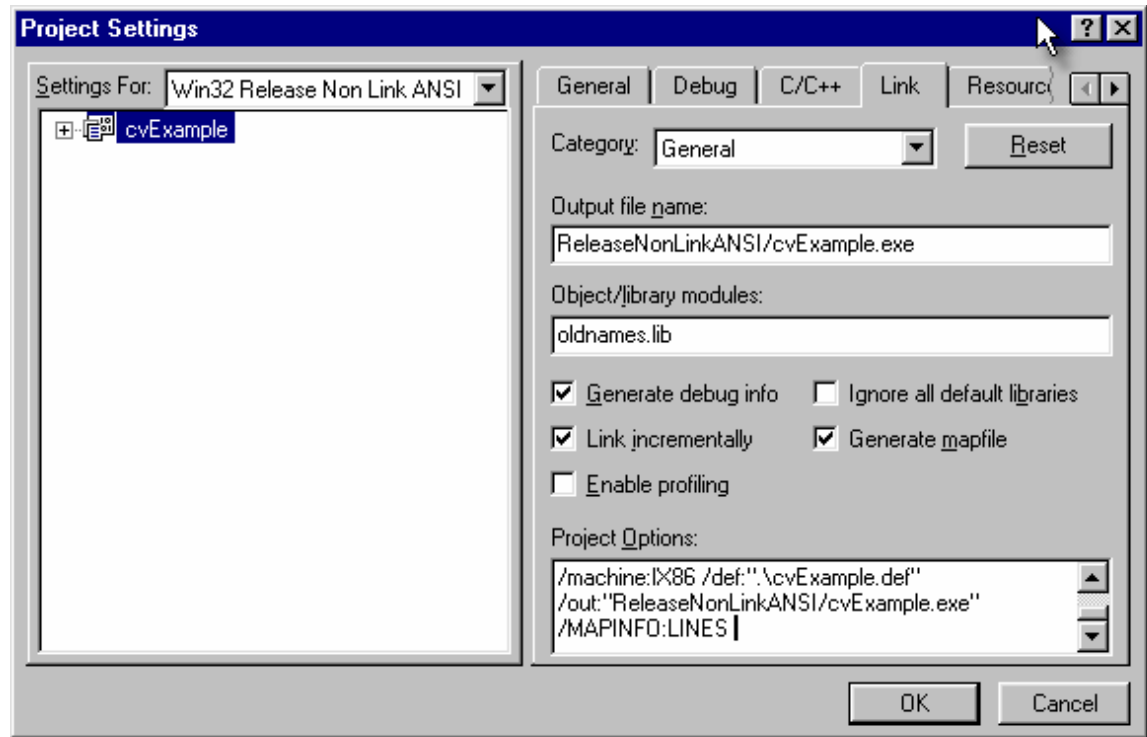
 Microsoft discontinued support for including line information in .MAP files with Visual Studio 8.0 (2005). There is no easy workaround to this.

To select the `/MAPINFO: LINES` option for Visual Studio 6.0 use the following steps. If you are using Visual Studio 7.0, 7.1 (i.e .NET 2002 or 2003) the project settings user interface is slightly different, but the basic principle remains the same.

In Visual Studio:

 **Project Menu** > **Settings...** > Select project > Shows project settings

The example image below shows project **bvExample**.



- **Generate mapfile** ➤ check option to request MAP file output
- **Project Options** ➤ add the text `/MAPINFO:Lines` to add line information to the file ➤ **OK**

Save your project workspace and build your project.

 Due to daylight saving times it is possible for a MAP file to have an embedded timestamp that is different than the DLL timestamp by an hour. In these situations Bug Validator will not recognise the MAP as valid. The solution to this problem is to rebuild the application.

9.12 Extensions, services and tools

 Including `stublib.h` in my project doesn't compile. Why?

You may encounter problems when including `stublib.h` in order to link directly with Bug Validator.

Include path problems

Ensure that your project **C preprocessor include paths** reference both of the **stub** and **stublib** subdirectories in the installation directory of Bug Validator.

For example, if Bug Validator is installed in:

```
C:\Program Files (x86)\Software Verify\Bug Validator
```

Then add the following paths for *all* configurations; Debug, Release, etc:

```
C:\Program Files (x86)\Software Verify\Bug Validator\stub
C:\Program Files (x86)\Software Verify\Bug Validator\stublib
```

Compiler errors

If you include `stublib.h`, your project must have included `windows.h` first, (or see below for an alternative).

If you fail to include `windows.h` then `stublib.h` will refer to some none-existent datatypes, causing compiler errors similar to the ones shown below.

Here's an example program that will not compile:

```
#include "stdafx.h"
#include "stublib.h"

int main(int argc, char* argv[])
{
    return 0;
}
```

 See the compiler errors from the above code

```
-----Configuration: testMV_allEnum - Win32
Debug-----
Compiling...
testMV_allEnum.cpp
c:\program files\software verification\bug validator\stub\allenum.h(70) : error
C2146: syntax error : missing ';' before identifier 'lRequest'
c:\program files\software verification\bug validator\stub\allenum.h(70) : error
C2501: 'LONG' : missing storage-class or type specifiers
c:\program files\software verification\bug validator\stub\allenum.h(70) : error
C2501: 'lRequest' : missing storage-class or type specifiers
c:\program files\software verification\bug validator\stub\allenum.h(71) : error
C2146: syntax error : missing ';' before identifier 'reserved3'
c:\program files\software verification\bug validator\stub\allenum.h(71) : error
C2501: 'DWORD' : missing storage-class or type specifiers
c:\program files\software verification\bug validator\stub\allenum.h(71) : error
C2501: 'reserved3' : missing storage-class or type specifiers
c:\program files\software verification\bug validator\stub\allenum.h(73) : error
C2143: syntax error : missing ';' before '*'
c:\program files\software verification\bug validator\stub\allenum.h(73) : error
C2501: 'BYTE' : missing storage-class or type specifiers
c:\program files\software verification\bug validator\stub\allenum.h(74) : error
C2501: 'dde_pbData' : missing storage-class or type specifiers
```

To fix this problem simply include `windows.h` before `stublib.h`

```
#include "stdafx.h"
#include <windows.h>      // new line to fix compile errors
#include "stublib.h"

int main(int argc, char* argv[])
{
    return 0;
}
```

Can't include windows.h?

If including `windows.h` is not an option, you can just define the following types:

```
#define LONG long
#define DWORD unsigned long
#define BYTE unsigned char
#define HANDLE void *
```

What do I do if I cannot use `svBVStubService.lib`?

You may find that you can't use **`svBVStubService.lib`** / **`svBVStubService_x64.lib`** because your linker doesn't understand the format of the lib file.

If that happens you can use the code below to compile the two functions that would be provided by those libraries.

See the header file

```
#ifndef _SVL_BVSTUB_SERVICE_H
#define _SVL_BVSTUB_SERVICE_H
```

```
#include "svlServiceError.h"

// IMPORTANT.
// If you use svlBVStub_LoadBugValidator() to load svlBugValidatorStub.dll into you
// application, you must also use svlBVStub_UnloadBugValidator() to unload the DLL
// your application being closed down. Failure to do so will almost certainly resul
// It does not matter how the application is closed down, you must ensure that you
// svlBVStub_UnloadBugValidator() to unload the DLL if you have loaded it.
//
// The DLL prepares itself in different ways and shuts itself down differently depe
// it is:-
// a) Directly linked to the application for use with the API or injected with Bug
// When the DLL is used in this manner to DLL expects to oversee and manage the
// shutdown.
// b) Loaded by using svlBVStub_LoadBugValidator().
// When the DLL is used in this manner to DLL expects to be removed prior to app
// and the behaviour of the DLL is undefined once you enter the program shutdown
//
// This difference in behaviour is intentional and is done to allow the use of
// services.

#ifdef __cplusplus
extern "C" {
#endif

SVL_SERVICE_ERROR svlBVStub_LoadBugValidator(serviceCallback_FUNC callback,
                                              void *userParam);

SVL_SERVICE_ERROR svlBVStub_UnloadBugValidator();

#ifdef __cplusplus
}
#endif

#endif
```

 See the implementation file

```

#include "svlBVStubService.h"

#include <windows.h>
#include <tchar.h>

// -NAME-----
// .DESCRIPTION.....
// .PARAMETERS.....
// .RETURN.CODES.....
// -----

static HMODULE hModule = NULL;

// -NAME-----
// .DESCRIPTION.....
// .PARAMETERS.....
// .RETURN.CODES.....
// -----

typedef void (*ENABLE_STUB_SYMBOL_FUNC) ();

SVL_SERVICE_ERROR svlBVStub_LoadBugValidator(serviceCallback_FUNC callback,
                                              void *userParam)
{
    SVL_SERVICE_ERROR errCode = SVL_OK;

    if (hModule == NULL)
    {
        hModule = LoadLibraryW(L"svlBugValidatorStub.dll"); // change this to svl
        if (hModule != NULL)
        {
            // DLL loaded, set the service callback function

            SETCALLBACK_FUNC setCallbackFunc;

            setCallbackFunc = (SETCALLBACK_FUNC)GetProcAddress(hModule, "apiSetService");
            if (setCallbackFunc != NULL)
            {
                (*setCallbackFunc)(callback, userParam);
            }

            // now start the profiler

            PROC *p;

            p = GetProcAddress(hModule, "startProfiler");
            if (p != NULL)
                (*p)();
        }
    }
}

```

```

        // now turn on provision of symbols by the stub

ENABLE_STUB_SYMBOL_FUNC    enableSymbolFunc;

enableSymbolFunc = (ENABLE_STUB_SYMBOL_FUNC)GetProcAddress(hModule, "apiEn
if (enableSymbolFunc != NULL)
{
    (*enableSymbolFunc)();
}
else
{
    errCode = SVL_FAILED_TO_ENABLE_STUB_SYMBOLS;
}
}
else
{
    errCode = SVL_LOAD_FAILED;
}
}
else
{
    errCode = SVL_ALREADY_LOADED;
}

return errCode;
}

//--NAME-----
//.DESCRIPTION.....
//.PARAMETERS.....
//.RETURN.CODES.....
//-----

typedef void (*UNLOAD_FUNC)();
typedef HANDLE (*GET_STUB_HEAP_FUNC)();

SVL_SERVICE_ERROR svlBVStub_UnloadBugValidator()
{
    SVL_SERVICE_ERROR    errCode = SVL_OK;

    if (hModule != NULL)
    {
        // get the stub heap before we shut down the DLL

        HANDLE            hStubHeap = NULL;
        GET_STUB_HEAP_FUNC    getHeapFunc;

        getHeapFunc = (GET_STUB_HEAP_FUNC)GetProcAddress(hModule, "apiGetInternalMVst
        if (getHeapFunc != NULL)
        {
            hStubHeap = (*getHeapFunc)();
        }

        // get the unload stub function

        UNLOAD_FUNC        unloadFunc;

        unloadFunc = (UNLOAD_FUNC)GetProcAddress(hModule, "apiShutdownBugValidator");
        if (unloadFunc != NULL)

```

```
{
    (*unloadFunc)();

    // get the function

    HMODULE    hModule;

    hModule = GetModuleHandleW(L"svlBugValidatorStub.dll");
    if (hModule != NULL)
    {
        // unload the stub

        FreeLibrary(hModule);

        // destroy the stub's heap (which was still in use whilst FreeLibrary())

        if (hStubHeap != NULL)
            HeapDestroy(hStubHeap);
        else
        {
            if (errCode == SVL_OK)
                errCode = SVL_FAIL_TO_CLEANUP_INTERNAL_HEAP;
        }
    }
    else
    {
        errCode = SVL_FAIL_MODULE_HANDLE;
    }
}
else
{
    errCode = SVL_FAIL_UNLOAD;
}

hModule = NULL;
}
else
    errCode = SVL_NOT_LOADED;

return errCode;
}
```

9.13 Working with Debuggers

You can attach a debugger to the target program while Bug Validator is monitoring the execution trace.

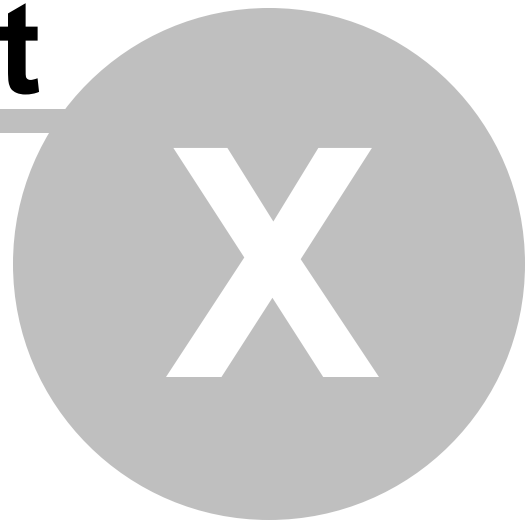
Windows Debugger API

Bug Validator does not use the Windows Debugger API, so there is no conflict with any debugger that uses the Windows Debugger API.

Exceptions

You may find that with the debugger attached from time to time access violations occur within the Bug Validator dlls (svl*.dll). This is most likely when Bug Validator is trying to access some memory within a section of code guarded with structured exception handling. The debugger will always be notified about these exceptions, even though Bug Validator's code is handling them correctly. The only thing you can do is disable the notification for these exceptions in the debugger settings.

Part



10 Copyright notices

The following copyright notices acknowledge any open source code used in this software tool.

10.1 Udis86

This software uses the library `svUdis86.dll` and `svUdis86_x64.dll`. These libraries are modified binary versions of the open source disassembler `udis86`.

`udis86` was hosted at <http://udis86.sourceforge.net/>

`udis86` is currently hosted at <https://github.com/vmt/udis86> although the current distribution (at the time of writing) appears to be missing some files required to compile.

The 1.7.0 version of the `udis` source code contains this copyright notice: Copyright (c) 2005, 2006, Vivek Mohan

The 1.7.2 version of the `udis` source code contains this copyright notice: Copyright (c) 2002-2009 Vivek Thampi

These copyright notices appear to conflict and the latter copyright notice completely ignores the claims set forth in the 1.7.0 copyright notice.

In accordance with the license terms in the 1.7.2 software we include this binary license.

```
* 1.7.2 Copyright (c) 2002-2009 Vivek Thampi
* All rights reserved.
*
* Redistribution and use in source and binary forms, with or without modification,
* are permitted provided that the following conditions are met:
*
*   * Redistributions of source code must retain the above copyright notice,
*     this list of conditions and the following disclaimer.
*   * Redistributions in binary form must reproduce the above copyright notice,
*     this list of conditions and the following disclaimer in the documentation
*     and/or other materials provided with the distribution.
*
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
* WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
* DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR
* ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
* (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
* ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
* (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
* SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Index

- D -

Diagnostic 40

- F -

Frequently Asked Questions 270

- I -

instrumentation 141

- L -

logging 141

- R -

Refresh 131

Reset 49

- S -

start a target program 157

stop 181

