BUGTRAP for WIN32 & .NET

DEVELOPER'S GUIDE

Save developer's time and customer's budget!

CONTENTS

1 Preface

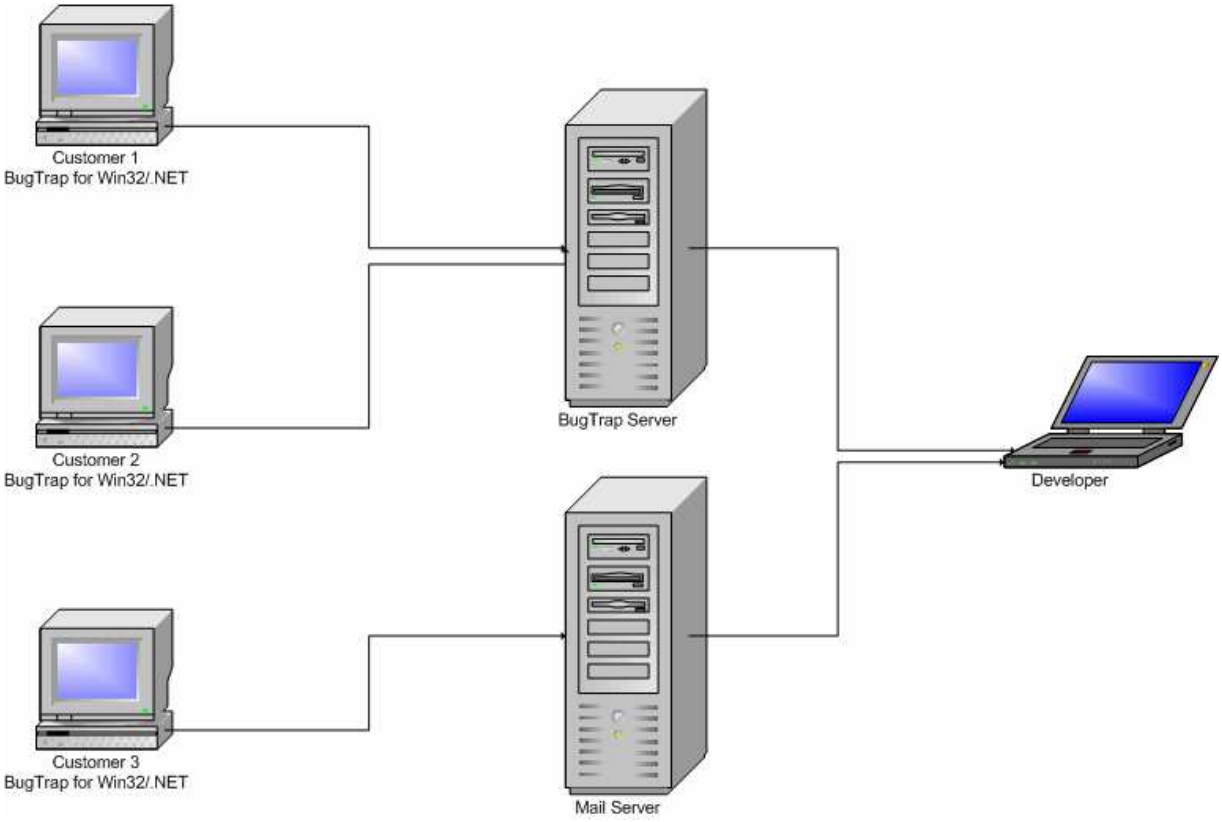
Some time ago I was working on a multi-tier application with quite complex logic. The application was handling medical information and it was important to correctly synchronize data in any circumstances. I put extra code to make the application as stable as possible, added automatic backups and self-recovery. Do you think it solved all problems?
- No, I was still searching for a tool to handle problems seen by customers remotely. How could I assist them and debug a problem if I live on another side of the globe? Eventually I found excellent Jim Crafton's article about a tool capable of intercepting unhandled errors. That was a solution!
Unfortunately, original BlackBox was not customizable, it didn't support minidump files, Unicode strings and it didn't have any server. In spite of these limitations it was an excellent starting point because I knew exactly what kind of tool I need. I started working on my own tool in hope to make flexible, customizable and powerful solution.

2 Overview

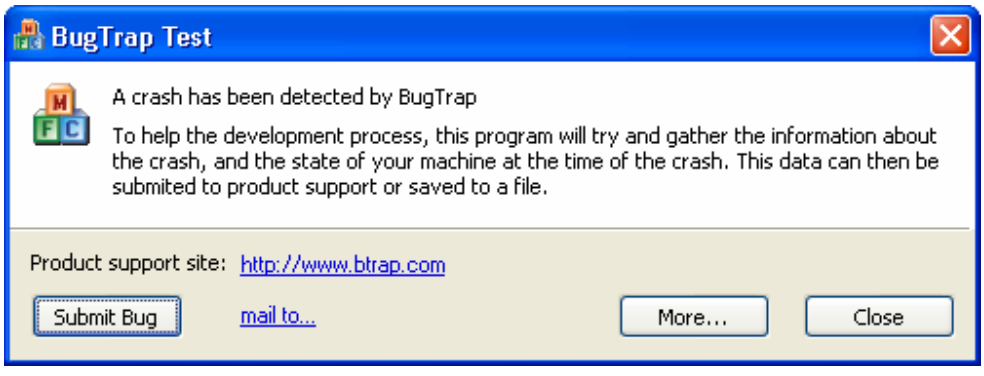
Usually it's very frustrating to receive a message from your customer saying that your program doesn't work. Most users may not let you know what's incorrect in your application and which piece of code is wrong. Windows has built-in handler for unhandled errors, however this default handler might be useless when error happens on customer side because you rarely want to send your error report to Microsoft:

Default Win32 error handler

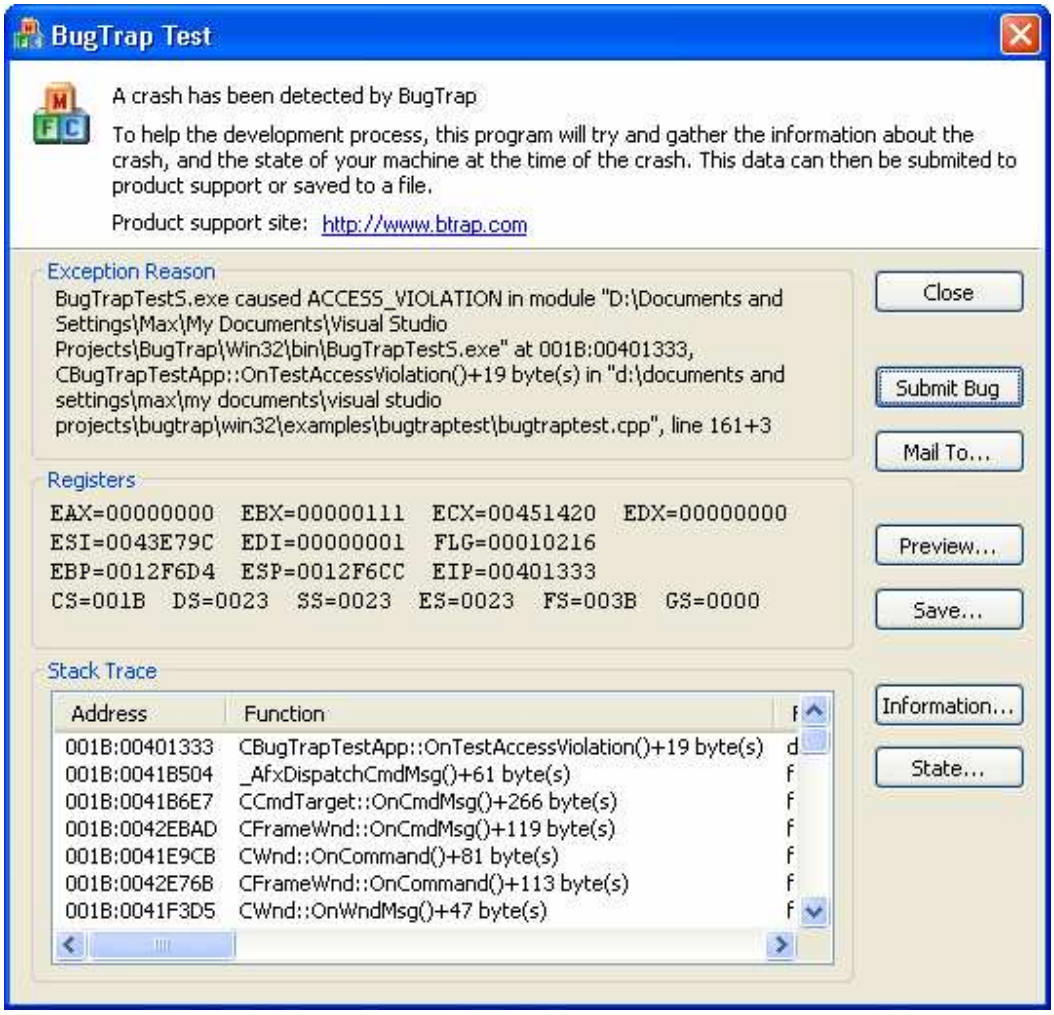
BugTrap solves this problem by overriding default error handler. BugTrap gathers error details such as address, call stack and computer environment. It's also possible to add arbitrary number of custom log files with additional information to the default error report using built-in or external logging functions.
BugTrap may save error reports on disk or automatically deliver error reports to developer's computer by e-mail, over HTTP or fast low-level TCP-based network protocol. BugTrap server automatically manages error reports repository and notifies developers about new errors.



BugTrap stores error description in log and minidump files. Minidump files may be opened in Microsoft Visual Studio .NET and in WinDbg. BugTrap package also includes CrashExplorer utility that can extract symbolic information from MAP and PDB files.
There is a special BugTrap version for .NET applications. .NET version can handle exceptions in pure .NET applications as well as it can handle mixed .NET assembles (managed/unmanaged assembles) written in C++.



Simplified dialog



Dialog with error details

3 BugTrap for Win32

3.1 Adding BugTrap to Win32 application

BugTrap is redistributed as dynamic-link library (DLL). Two versions of BugTrap DLL are available: ANSI version and Unicode version.

DLL name	Character encoding *
BugTrap.dll	ANSI multi-byte character strings
BugTrapU.dll	Unicode strings

* In theory, ANSI version of BugTrap should correctly handle multi-byte character strings and Unicode version of BugTrap should correctly handle surrogate character sequences, but it has not been tested in such environments.

It is recommended to use Unicode strings in new applications targeting Windows NT/2000/XP platforms. Unicode applications not only better deal with national character sets, but offer better speed. For instance, BugTrap encodes report and log files in UTF-8 format. While there is direct and quite simple mapping between Unicode and UTF-8 encoded characters, ANSI strings require additional conversions to/from Unicode. These conversions affect performance of XML parser, log generator and network communications.
The code bellow adds BugTrap support to Win32 application:

```
#include "BugTrap.h"

#pragma comment(lib, "BugTrap.lib")      // Link to ANSI DLL
// #pragma comment(lib, "BugTrapU.lib")  // Link to Unicode DLL

static void SetupExceptionHandler()
{
      BT_InstallSehFilter();
      BT_SetAppName(_T("Your application name"));
      BT_SetSupportEMail(_T("your@email.com"));
      BT_SetFlags(BTF_DETAILEDMODE | BTF_EDITMAIL);
      BT_SetSupportServer(_T("localhost"), 9999);
      BT_SetSupportURL(_T("http://www.your-web-site.com"));
}
```

`SetupExceptionHandler()` function may be called from `InitInstance()` or `main()` function depending on the type of your application.

Note: you may omit `BT_SetAppName()` and `BT_SetAppVersion()` calls if your application includes version info block. BugTrap can retrieve application name and version number from application resources.

3.2 Redistributing BugTrap for Win32

BugTrap is compatible with MS Windows 95/98/Me/NT/2000/XP. It requires `shlwapi.dll` that's installed with MS Internet Explorer 4.0 on Windows 95 or Windows NT 4.0. Windows 98/Me and Windows 2000/XP already have required system libraries.
BugTrap uses DbgHelp library which is redistributed as `dbghelp.dll`. This DLL is included in MS Windows 2000 and later. To use this DLL on earlier systems, such as Windows NT

4.0 or Windows 98, you should redistribute `dbghelp.dll` with your application. To obtain the latest version of `dbghelp.dll`, download Debugging Tools for Windows.
It's recommended to put the most recent version of `dbghelp.dll` to the same folder with BugTrap DLL otherwise some functions may be disabled. BugTrap always attempts to load recent version of `dbghelp.dll` from its folder. If it cannot find `dbghelp.dll` in that folder, it attempts to load `dbghelp.dll` from Windows system folder.

3.3 Error analysis for Win32 applications

Usually it's desirable to get source file name, function name and line number information from error address because such information can greatly simplify further error analysis and correction. There are several approaches to get this information:

a) symbolic information in PDB files when available;
b) minidump files;
c) a utility that performs post-mortem MAP and PDB files analysis.

Let's discuss every approach.

3.3.1 Symbolic information and PDB files

A program database (PDB) file holds debugging and project state information that allows incremental linking of a Debug configuration of your program. A PDB file is created when you compile a C/C++ program with `/ZI` or `/Zi` or a Visual Basic/C# .NET program with `/debug`.
BugTrap automatically uses PDB file if available when it encounters a problem. PDB file must be located in the same directory with EXE file to be found. BugTrap automatically displays source file names, function names and line numbers for call stack entries in the main window when it finds an appropriate PDB file. If application's PDB file cannot be found on customer's computer, BugTrap displays hexadecimal addresses with no symbolic information. These addresses can be analyzed later on developer's computer.
Debug information in PDB file doesn't affect the size and speed of your program, Visual Studio only saves a path to PDB file in EXE file. You may enable PDB file generation for Release configuration in your project and redistribute PDB file to your customers along with program EXE file.
However PDB files have a couple of disadvantages:

a) they are quite large, for example, PDB file for 300KB application may require 3MB disk space;
b) though PDB files don't include application source code, many developers won't redistribute PDB files with their applications because PDB files may simplify reverse engineering.

Usually it's better to redistribute PDB files to application testers and quality assurance, but don't include PDB files in public releases.

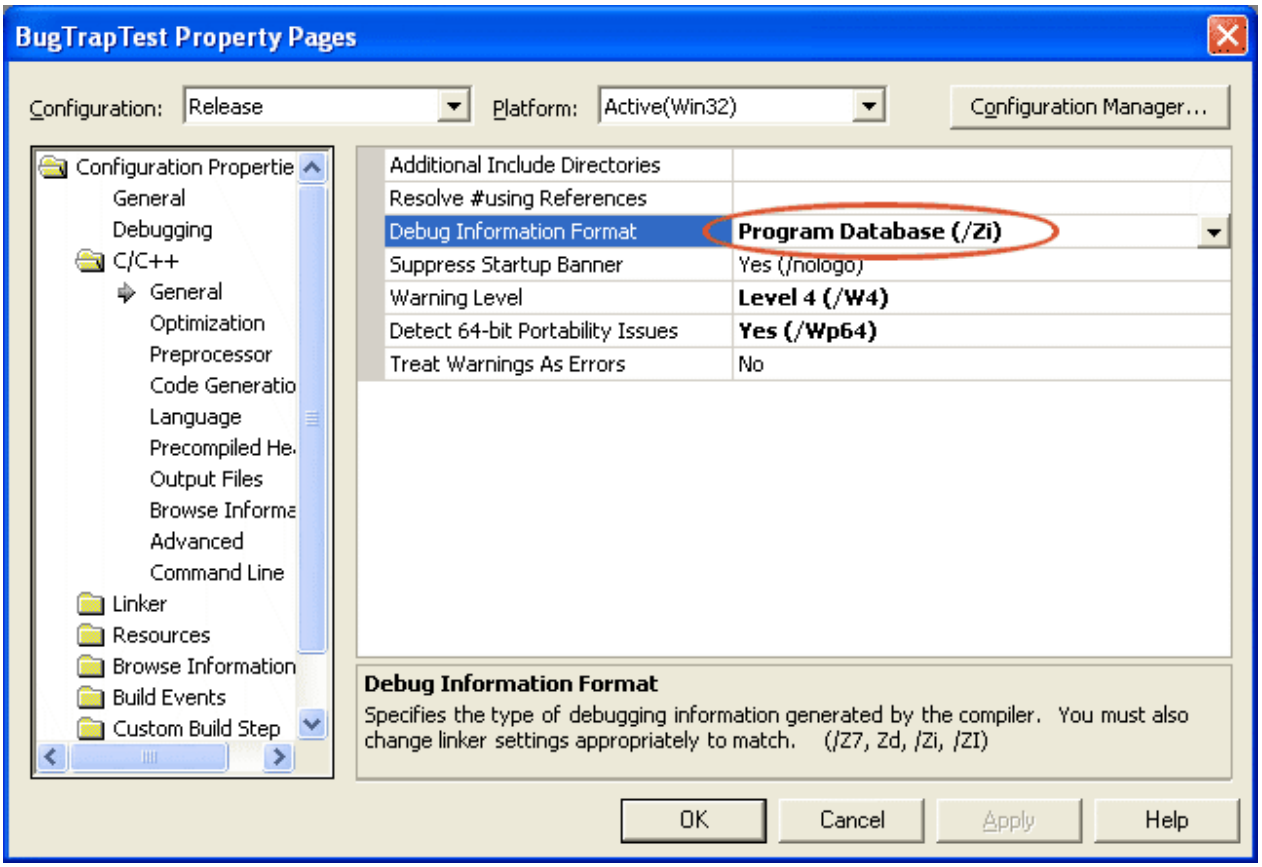
Hopefully it is not necessarily to redistribute PDB files with your application in order to take advantage of this technology. CrashExplorer can extract symbolic information from PDB files and merge it with raw error log on developer's computer. PDB files can be stored locally for all public releases and used later for generating human readable error reports.

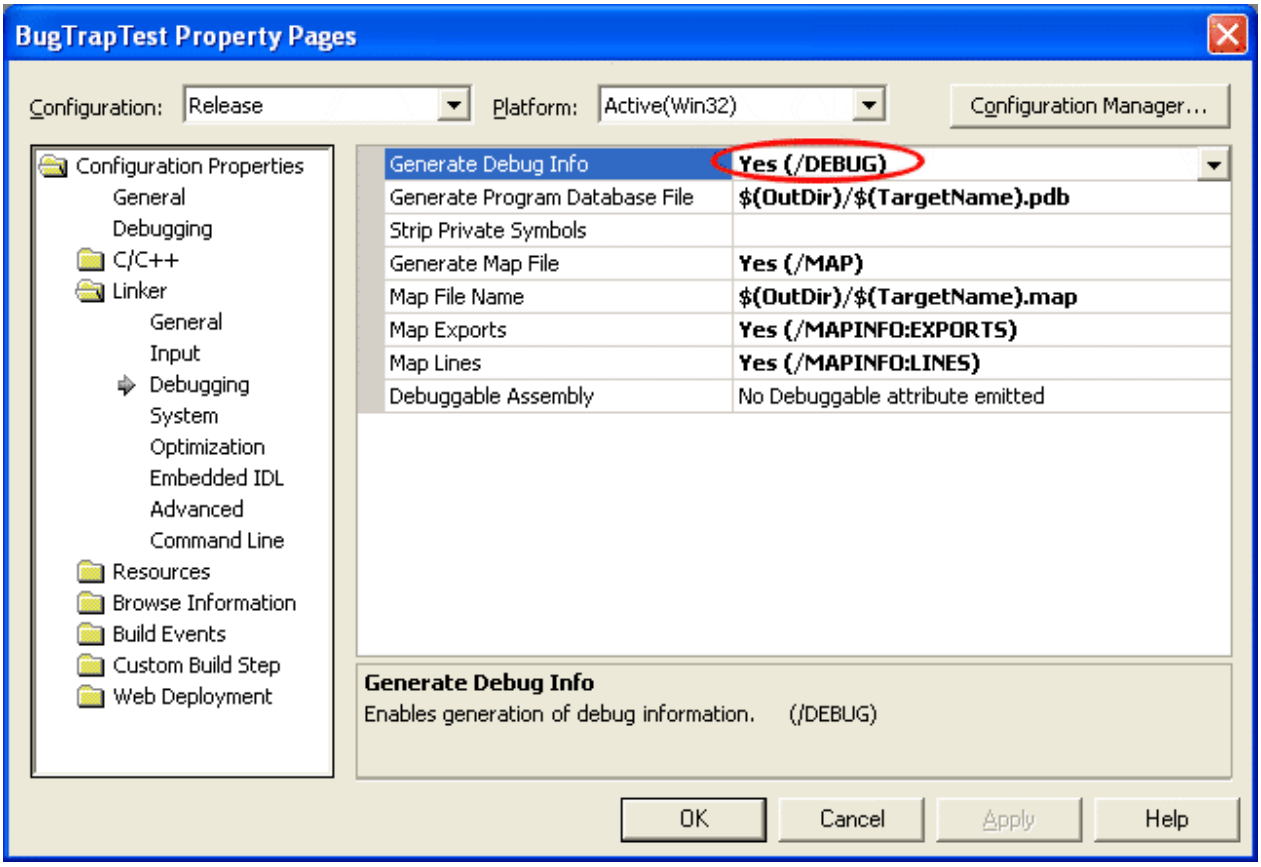
The following steps enable PDB files for Release configuration:

1. select Release configuration in "Project Settings" dialog;
2. select "Program Database" format of debug information on "C/C++\General" tab;
3. enable "Generate Debug Info" option.

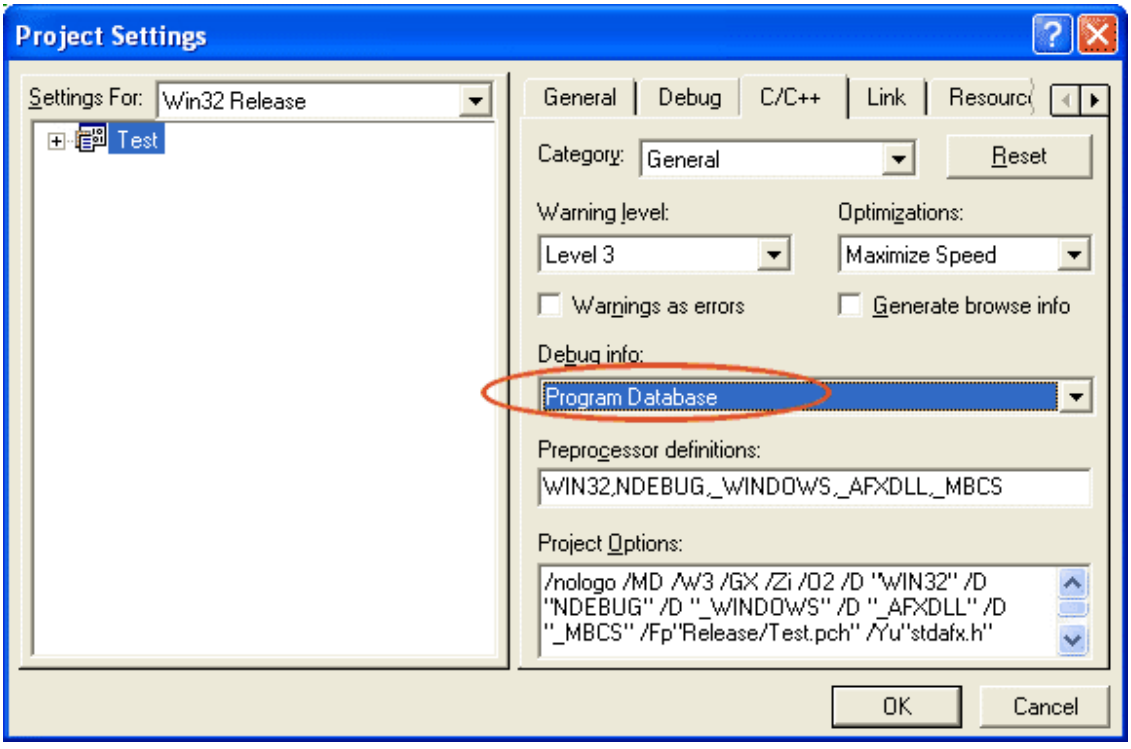
Use these pictures for the reference:

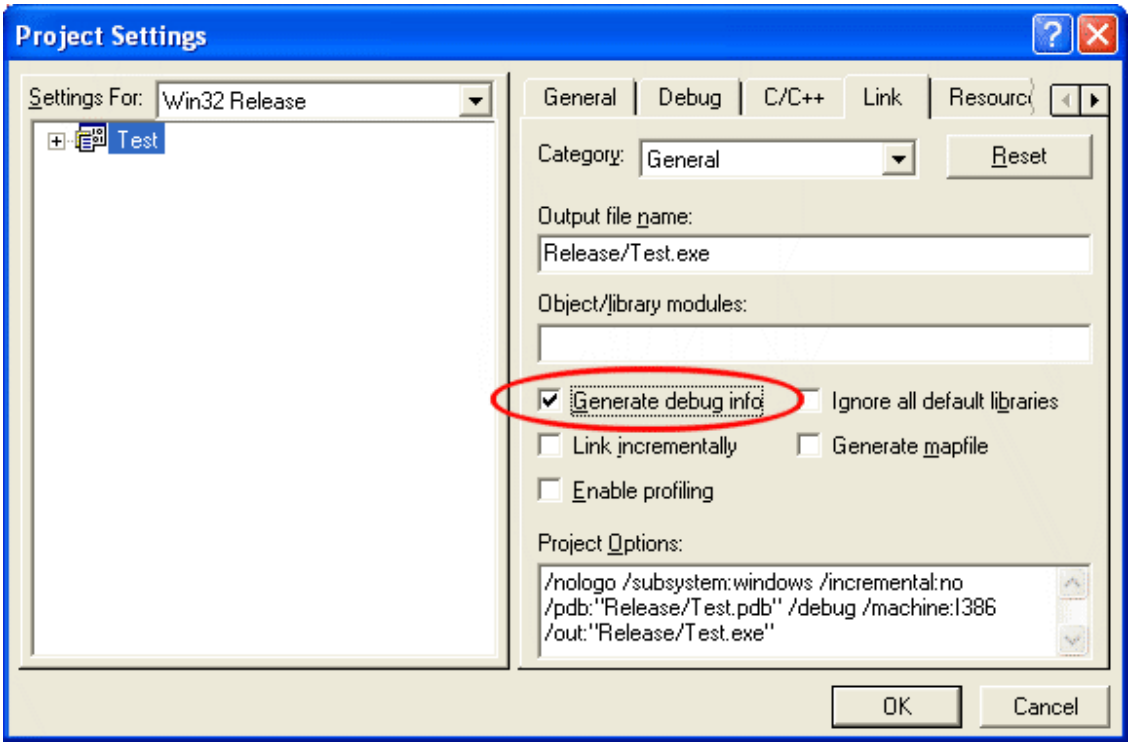
Visual Studio 7





Visual Studio 6





Note: you may not see MFC function names and line numbers in BugTrap stack trace window if your application is dynamically linked to MFC even if application's PDB file is accessible for BugTrap. Instead you may notice multiple entries in the form of `OrdinalXXX()`. This is because application's PDB file doesn't include symbolic information for MFC classes. You can solve this problem by copying MFC's PDB file from `System32` folder to the application's folder or by linking your project to MFC statically. But even if MFC's symbolic information is not available it still can be restored from minidump file or by running CrashExplorer.

3.3.2 Minidump files

BugTrap can produce user-mode minidump files with a useful subset of the information contained in a crash dump file. BugTrap creates minidump files very quickly and efficiently because minidump files are small, they can be sent over the Internet to technical support of the application. A minidump file does not contain as much information as a full crash dump file, but it contains enough information to perform basic debugging operations. To read a minidump file, you must have the binaries and symbol files available for the debugger.
Minidump files don't require PDB files on customer's computer, though you should keep PDB files on developer's computer for further error analysis in the debugger. Minidump files may be analyzed in WinDbg which is redistributed as part of Debugging Tools for Windows or in Visual Studio .NET. Minidump files provide the best option for reproducing customer-side errors on developer's computer. Minidump files have few disadvantages:

a) minidump files are relatively large comparing to default text output produced by BugTrap. BugTrap archives minidump files to reduce the size of generated report.
b) minidump files are stored in binary format, so you can't read them without special tool like WinDbg or Microsoft Visual Studio .NET.

c) minidump files can't be created on Windows 9x. Hopefully it is not a big issue for BugTrap because you can use CrashExplorer utility which can extract error location from hexadecimal error address.

BugTrap always generates log file in plain text or XML format. Minidump files are only generated in detailed report mode. You must specify `BTF_DETAILEDMODE` option in order to enable this mode:

```
BT_SetFlags(/* other options */ | BTF_DETAILEDMODE);
```

BugTrap stores log file and minidump file in one zip archive to reduce the size of error report. You can add custom log files to the same zip archive. Custom log files can be generated using built-in BugTrap functions:

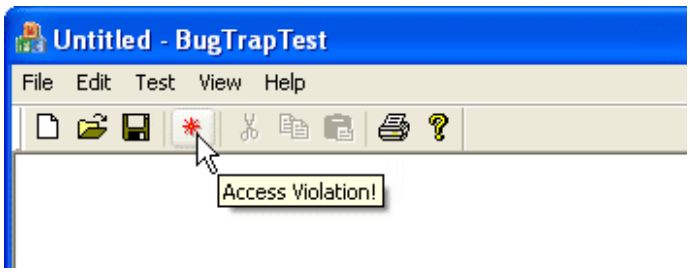
```
INT_PTR iLogHandle = BT_OpenLogFile(NULL, BTLF_TEXT);
BT_AddLogFile(BT_GetLogFileName(iLogHandle));

BT_InsLogEntry(iLogHandle, BTL_INFO, _T("custom log message"));
 - or -
BT_InsLogEntryF(iLogHandle, BTL_WARNING, _T("numeric output: %d"), 123);
```

See "Custom log files" topic for more information.

3.3.3 Running test application

BugTrap comes with several test applications. You can launch BugTrapTest example and hit "Access Violation!" button on the toolbar:



This button executes the following code:

```
void CBugTrapTestApp::OnTestAccessViolation()
{
	int* ptr = 0;
	*ptr = 0; // ACCESS VIOLATION!!!
}
```

After hitting the "Access Violation!" button you should see main BugTrap window. This window displays exception information, CPU registers, call stack and several buttons:

Button	Description
Close	Closes BugTrap window and quits the application.
Submit Bug	Sends error report to product support by e-mail or over the network. You should

Button	Description
	specify server address/e-mail address during application startup.
Mail To…	Depending on `BTF_EDITMAIL` flags, opens "Send Mail" dialog or launches system e-mail client where user can prepare custom e-mail message addressed to the support.
Preview…	Opens Preview dialog that displays the contents of error report files.
Save…	Saves error report to the file. File may include minidump and custom log files depending on `BTF_DETAILEDMODE` flag.
Information…	Displays generic information about installed operating system.
State…	Displays generic information about running processes and loaded modules.

It is not necessary to specify server address, support e-mail or URL of support site. Unspecified links will not be shown on the screen.
User may press `Preview` or `Save` buttons to examine report contents. By default report name includes date and time for the uniqueness.

Error report includes these sections:

1. application name and version;
2. computer and user names (used for identifying problems in local network);
3. date and time of the error;
4. error description;
5. user-defined message (if available);
6. COM error information (if available);
7. values of CPU registers;
8. generic CPU information;
9. operating system information;
10. memory usage statistics;
11. stack trace information for all running threads;
12. process command line and current directory;
13. process environment variables;
14. optional list of running processes and loaded modules;
15. optional screenshot taken during program crash.

Error information can be presented in plain text or in XML format:

Excerpt from log file in plain text format

```
BugTrapTest.exe caused ACCESS_VIOLATION in module "<Executable
Path>\BugTrapTest.exe" at 001B:00401333,
CBugTrapTestApp::OnTestAccessViolation()+19 byte(s) in "<Source
Path>\BugTrapTest.cpp", line 161+3 byte(s)
```

Excerpt from log file in XML format

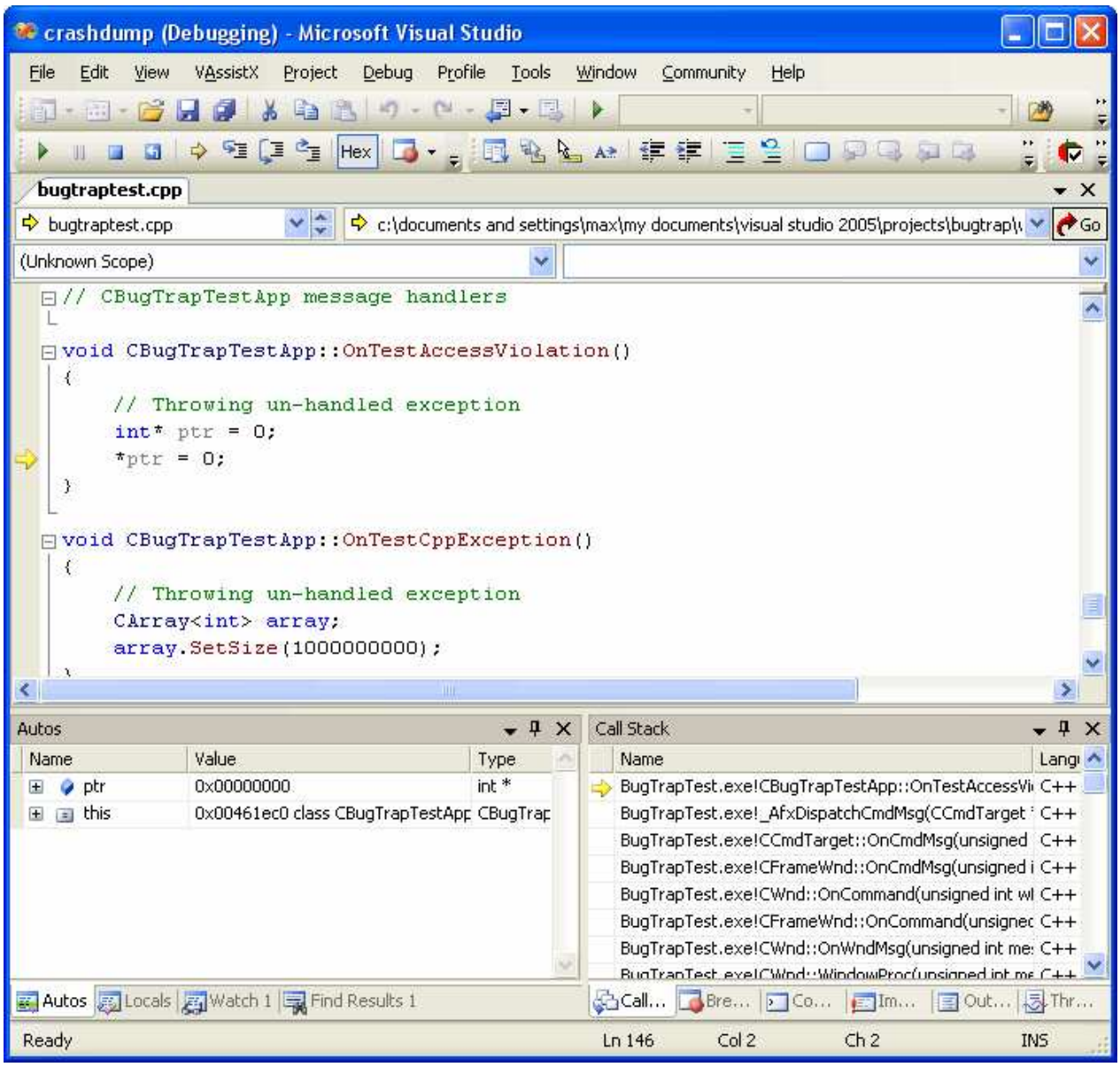
```
<error>
  <what>ACCESS_VIOLATION</what>
  <process>
```

```
    <name>BugTrapTestE.exe</name>
    <id>6936</id>
  </process>
  <module><Executable Path>\BugTrapTestE.exe</module>
  <address>001B:00401333</address>
  <function>
    <name>CBugTrapTestApp::OnTestAccessViolation</name>
    <offset>19</offset>
  </function>
  <file><Source Path>\BugTrapTest.cpp</file>
  <line>
    <number>161</number>
    <offset>3</offset>
  </line>
</error>
```

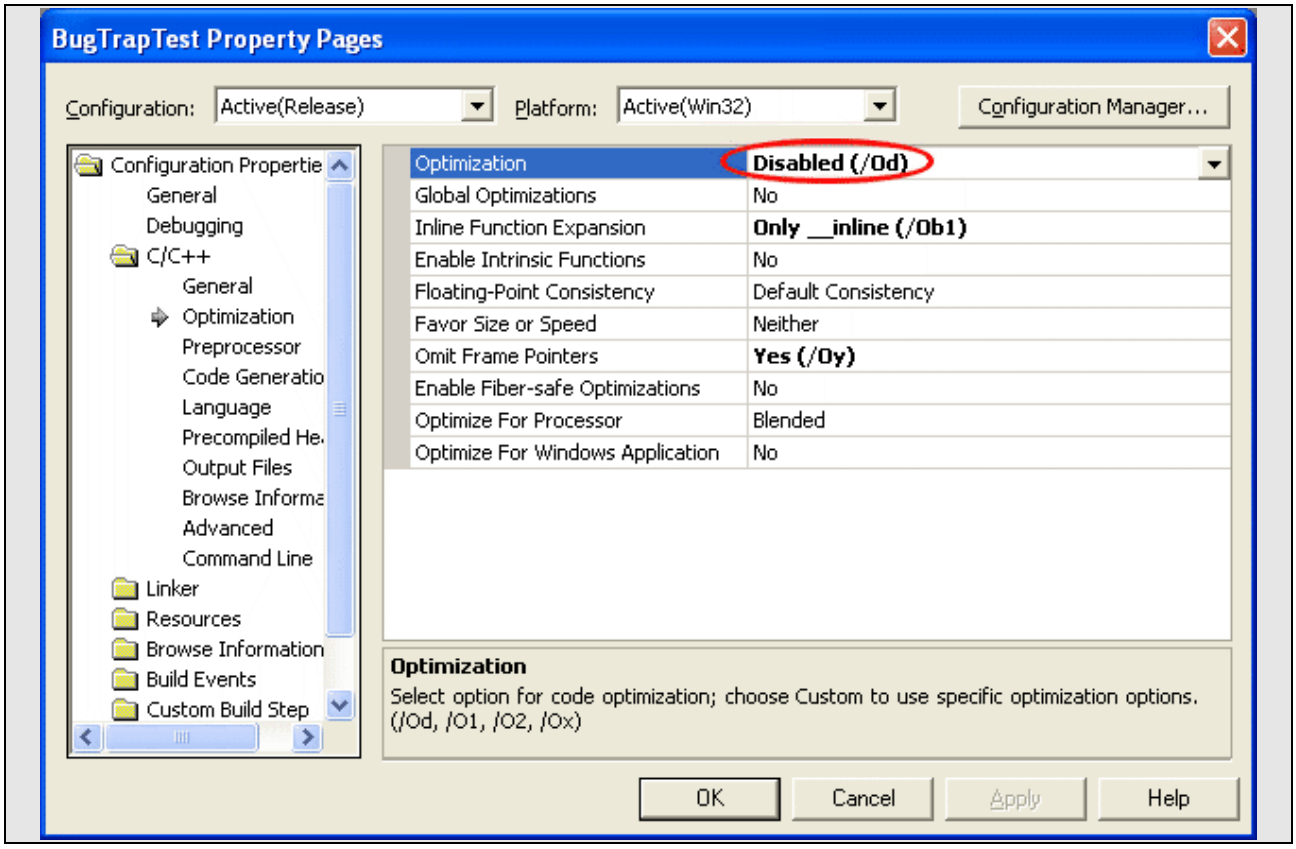
You can open "`BugTrapTest.cpp`" and check line 161: `*ptr = 0;`

3.3.3.1Minidump files in Visual Studio .NET

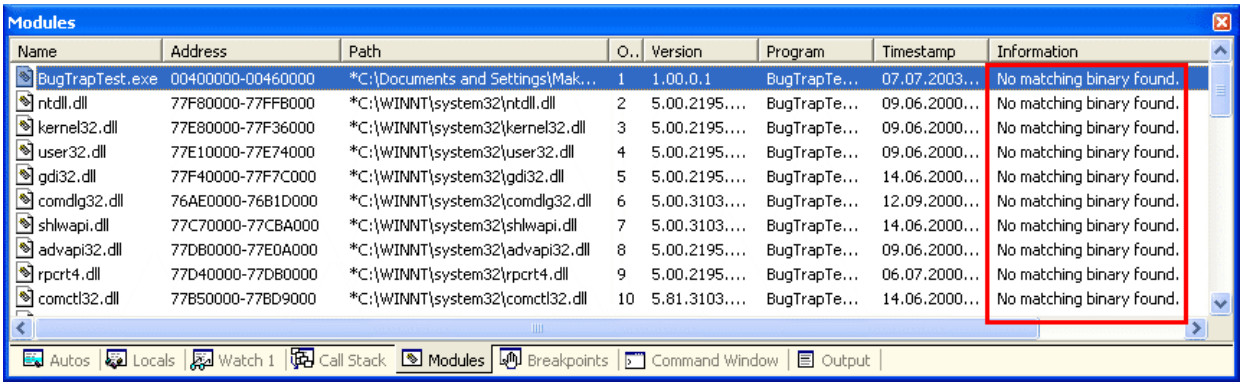
Minidump files can be opened in Visual Studio IDE. Open such file in Visual Studio and start the debugger – the IDE will ask you to create a new solution and the debugger will create a fake process. Now you can examine the problem using well-known environment:



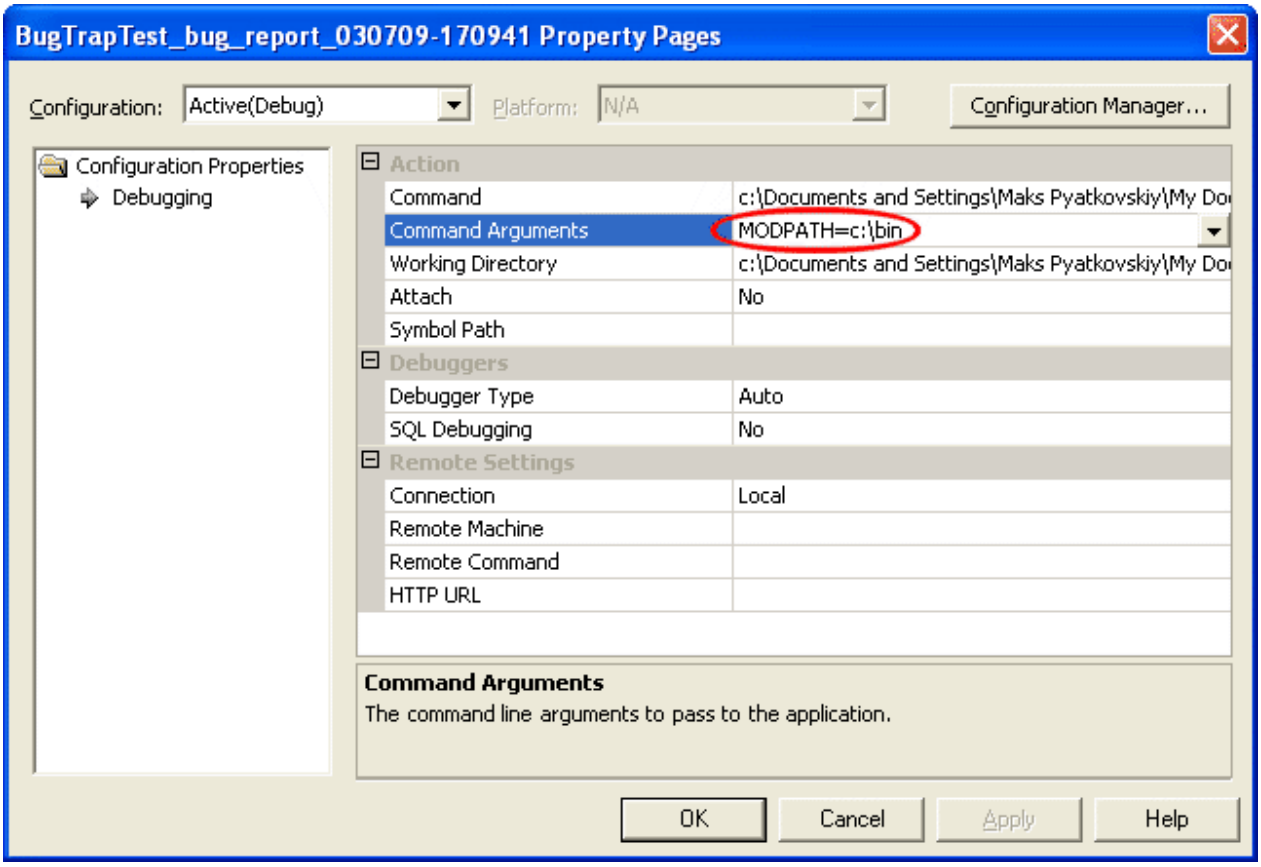
Note: often it's difficult to find a problem in Release version because optimizing compiler may remove some variables and even reposition pieces of code. Optimization affects information stored in minidump files and BugTrap reports. In this particular case it is not possible to see `ptr` value in Visual Studio debugger with enabled optimization.
Usually it's better to disable compiler optimization during project development and testing:



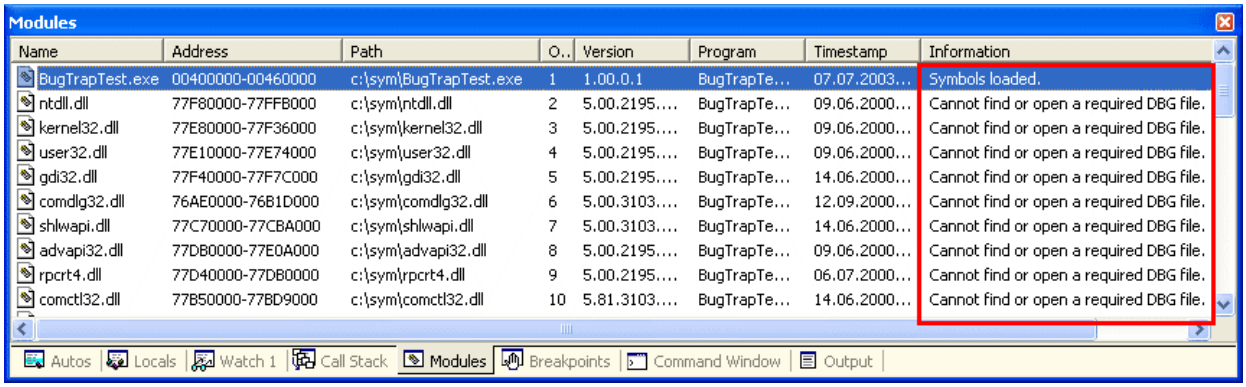
In most cases environment on client's computer differs from the environment on developer's computer: application binaries may be located in different folders, versions of system DLLs may not match. In this case "Call Stack" window won't display much useful information and Modules window will display warnings "No matching binary found":



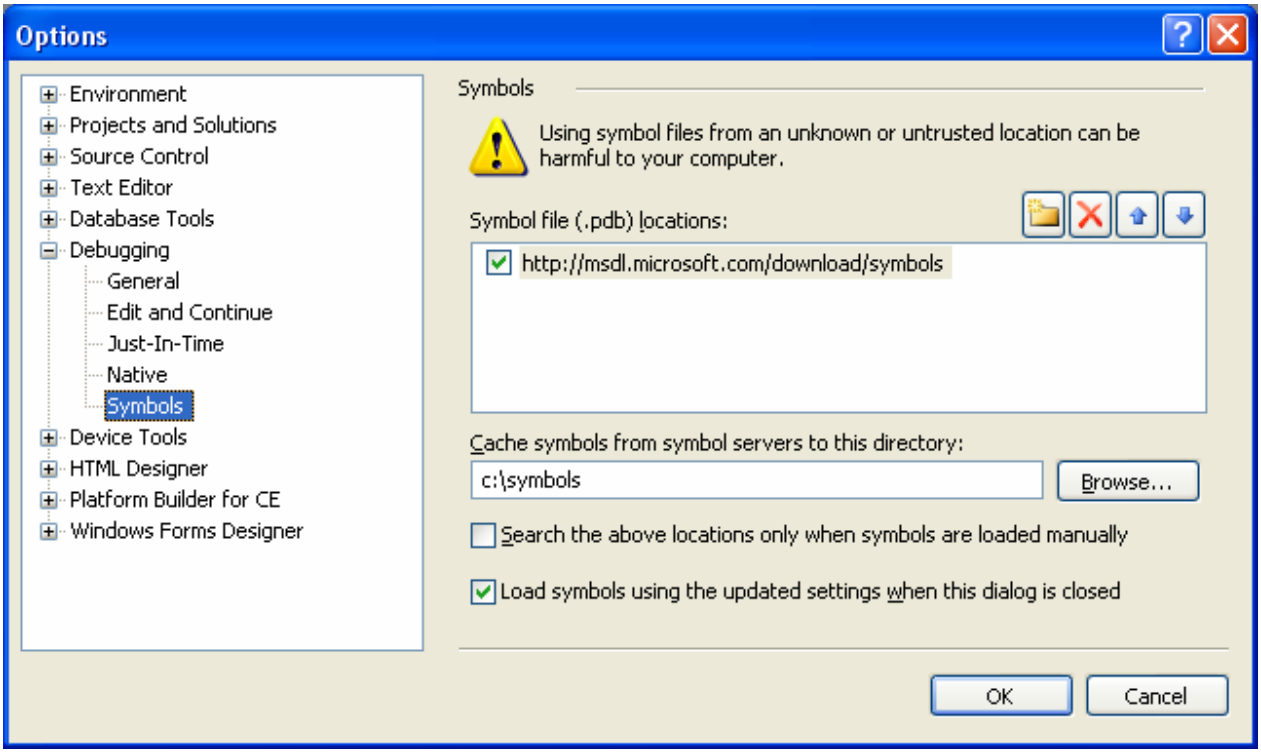
In this case you should create a folder, copy valid binaries with appropriate versions to that folder and specify path to that folder in `MODPATH` command argument:



The debugger should work as expected after the restart, and you should be able to discover the problem. Though you may notice that Modules window still displays warnings "*Cannot find or open a required DBG file*" or "*No symbols loaded*" for system DLLs:



Such warnings can be fixed after copying remaining symbol files (PDB and DBG files) to binaries folder or after specifying up a path to Microsoft Symbol Server:



3.3.3.2Minidump files in WinDbg

WinDbg is a powerful debugger with a graphical interface that can debug both user-mode and kernel-mode code. WinDbg can view source code, set breakpoints, view variables (including C++ objects), stack traces, and memory. WinDbg includes a Command window to issue a wide variety of commands, and supports kernel-mode remote debugging using two computers (host and target machine). It also allows remote debugging of user-mode code, and 64-bit debugging. WinDbg can be downloaded for free from Microsoft Windows Debugging Tools web site.
Open crash dump file in WinDbg by selecting "File\Open Crash Dump" menu command. Switch to command view by selecting "View\Command" menu command. Enter the following commands:

a) enter `.sympath` command followed by semi-colon delimited list of directories with PDB and DBG files;
b) enter `.srcpath` command followed by semi-colon delimited list of directories with source files;
c) enter `.exepath` command followed by semi-colon delimited list of directories with executable files;
d) enter `.ecxr` to display the context record associated with current exception.

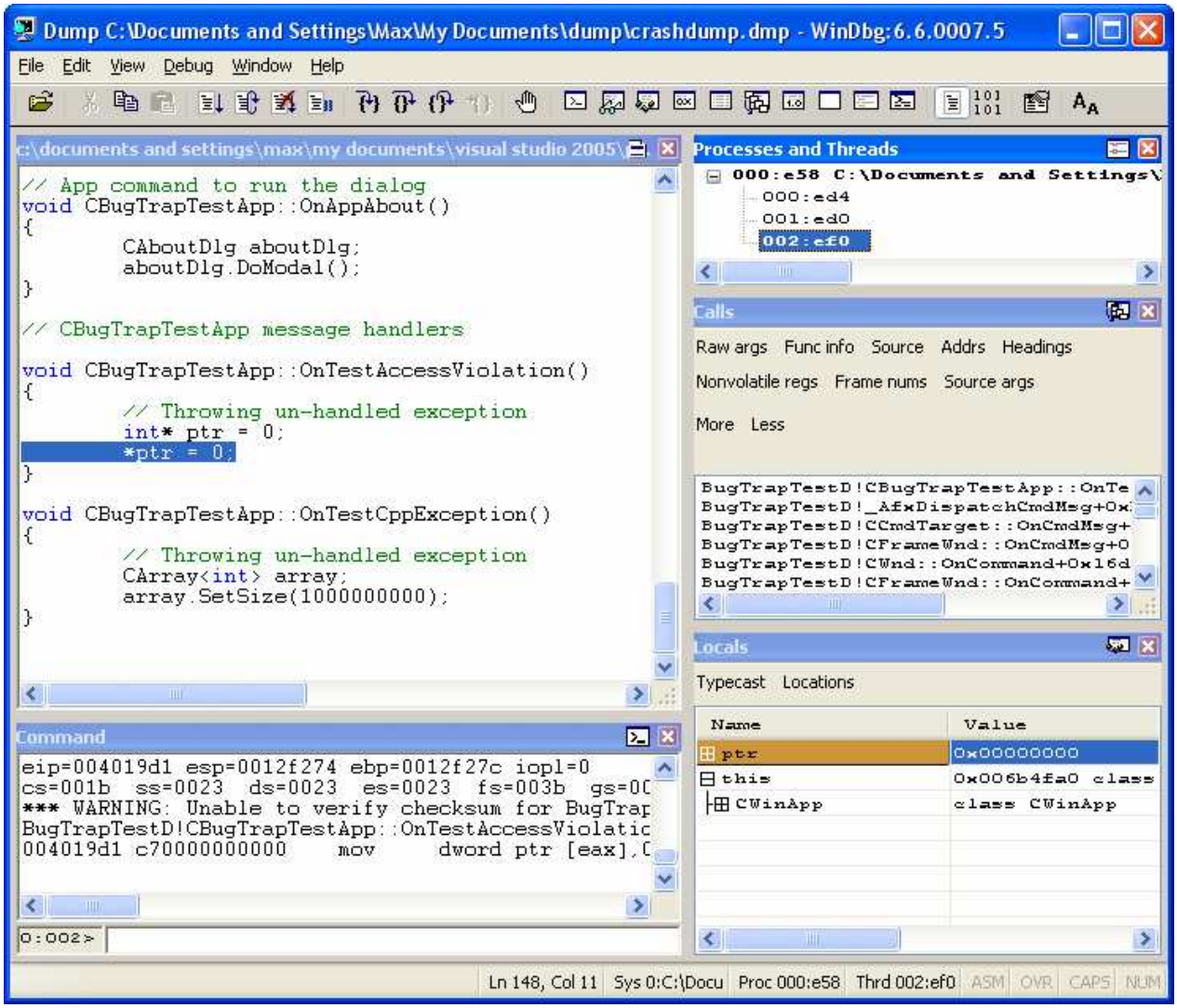
For example:

```
.sympath c:\test\sym
.srcpath c:\test\src
.exepath c:\test\bin
.ecxr
```

e) You may wish to append a path to Microsoft Symbol Server to `.sympath` command in order to download symbols for system DLLs:

```
.sympath c:\test\sym;SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
```

If everything is configured well, WinDdg may display this information:



3.3.4 Automatic MAP files analysis

A MAP file is a text file that contains the following information about the program being linked:

- the module name, which is the base name of the file;
- the timestamp from the program file header (not from the file system);
- a list of groups in the program, with each group's start address (as *section*:*offset*), length, group name, and class;
- a list of public symbols, with each address (as *section*:*offset*), symbol name, flat address, and OBJ file where the symbol is defined;
- the entry point (as *section*:*offset*).

Note: Microsoft has discontinued proper support of MAP files starting from Microsoft Visual Studio 2005. New linker can't generate line numbers in a MAP file and `/MAPINFO:LINES` option is no longer supported. CrashExplorer won't extract source file names and line numbers from such MAP files which makes MAP files useless on the new platform.

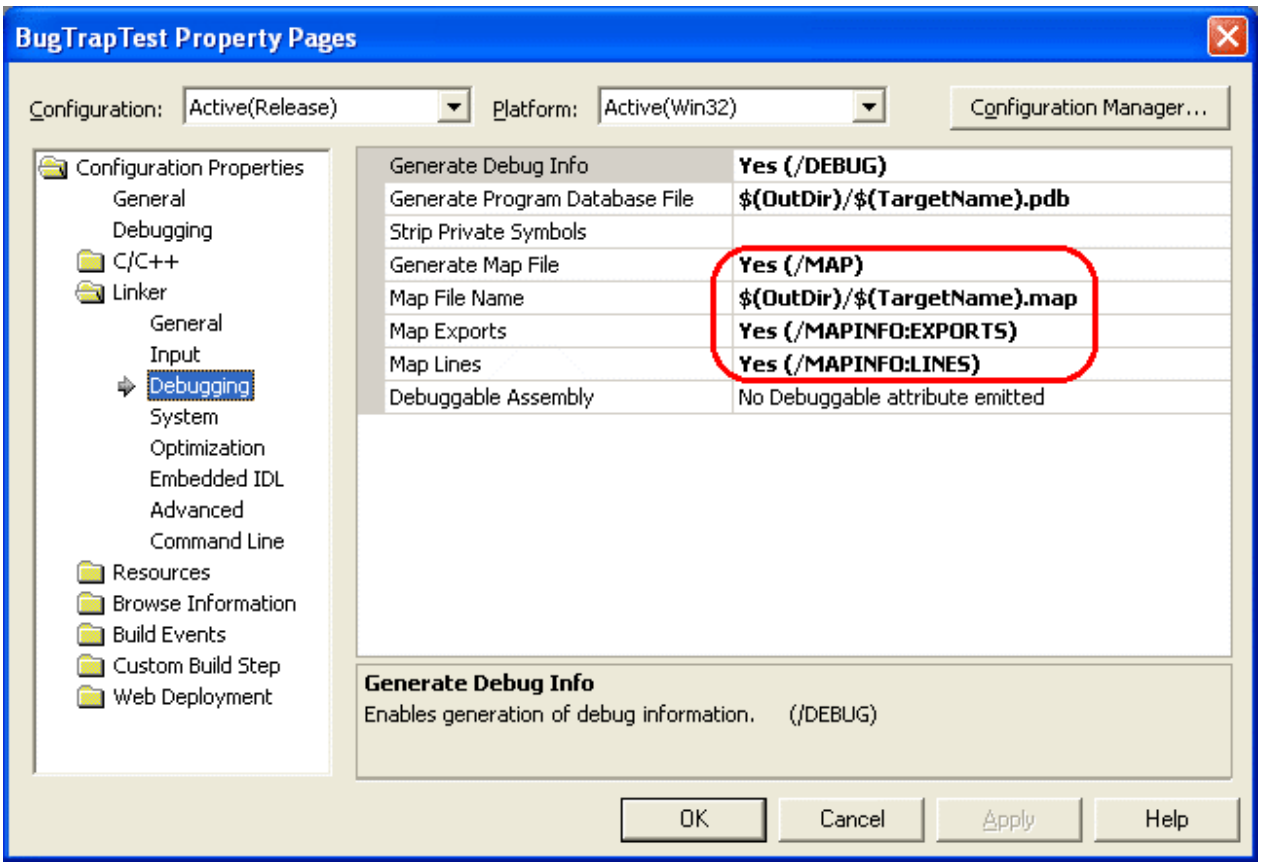
However this doesn't mean that there is no reason to use CrashExplorer with Microsoft Visual Studio 2005 because CrashExplorer can extract symbolic information from PDB files. Most developers won't release products to their customers with accompanying PDB files because PDB files can simplify reverse engineering. Therefore it is better to keep PDB files for every public release on developer's machine and use CrashExplorer to convert raw addresses from a log file to human readable report with symbolic information.

MAP file may be generated during project linking. You should not distribute this file to your customers. It should be saved on developer's computer for every public release. MAP file is your latest opportunity to find symbolic information about the error if you don't have PDB file or crash dump was not successfully generated (DbgHelp has certain problems on Windows 9x).
In order to generate MAP file for your project, follow these steps:

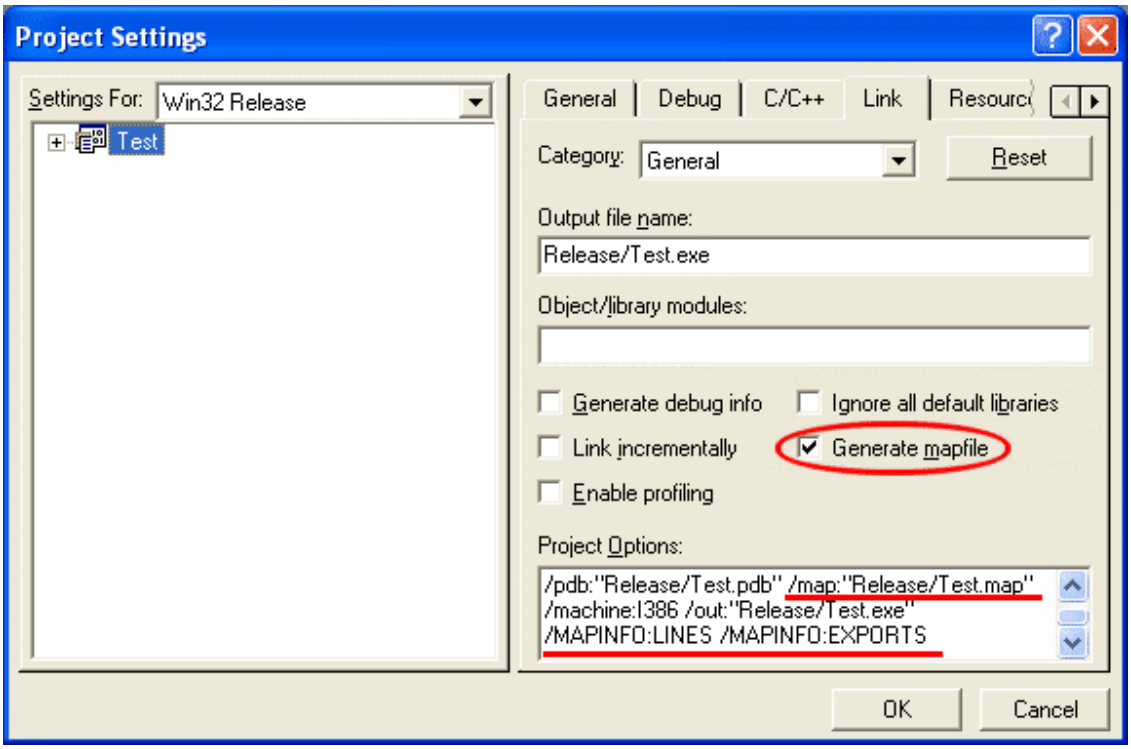
1. select Release configuration in "Project Settings" dialog.
2. you should already have selected "Program Database" format of debug information on "C/C++\General" tab if you make PDB files for Release configuration. This option can be used with MAP files. If you don't want to make PDB files you should at least select "Line Numbers Only" format (it enables `/Zd` compiler option).
3. enable "Generate Map File" option on "Linker\Debugging" tab.
4. enable "Max Exports" and "Map Lines" options in Visual Studio 2003 or add custom linker switches "`/MAPINFO:EXPORTS`" and "`/MAPINFO:LINES`" in Visual Studio 6.
5. optionally configure "Map File Name" option in Visual Studio 2003 or adjust custom linker switch "`/MAP:<Map File Name>`" in Visual Studio 6.

Use these pictures for the reference:

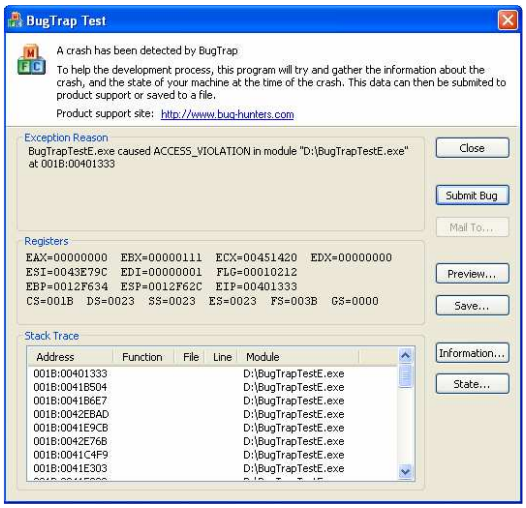
Visual Studio 7



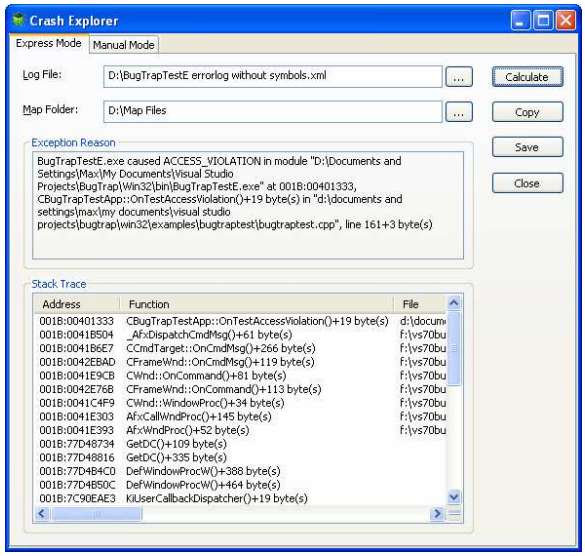
Visual Studio 6



Log file can be generated either in plain text (`BTRF_TEXT`) or in XML (`BTRF_XML`) format. Plain text is more suitable for human but it is almost useless if you want to parse and automatically handle log information. By default BugTrap generates log files in XML format. Output format can be changed by calling `BT_SetReportFormat()`.
XML format has one great advantage: it can be parsed by CrashExplorer. CrashExplorer merges raw addresses retrieved from XML file with symbolic information found in MAP or PDB files and restores complete stack trace even when PDB file with symbolic information was not available on customer's computer:



BugTrap running without PDB file



CrashExplorer restores stack trace

Before using CrashExplorer you should prepare a folder with MAP or PDB files for all modules (EXE or DLL files) in your project. If you have not used any libraries except of MFC or standard Windows DLLs, simply copy one MAP or PDB file for the main executable. If your project includes main executable and two additional DLLs, you should copy three MAP/PDB files to this folder. If you don't have MAP and PDB files for certain modules and XML log file doesn't include symbolic information for these modules, CrashExplorer won't be able to display stack trace with function names and line numbers for such modules. Every MAP/PDB file should have the same base name as corresponding module:

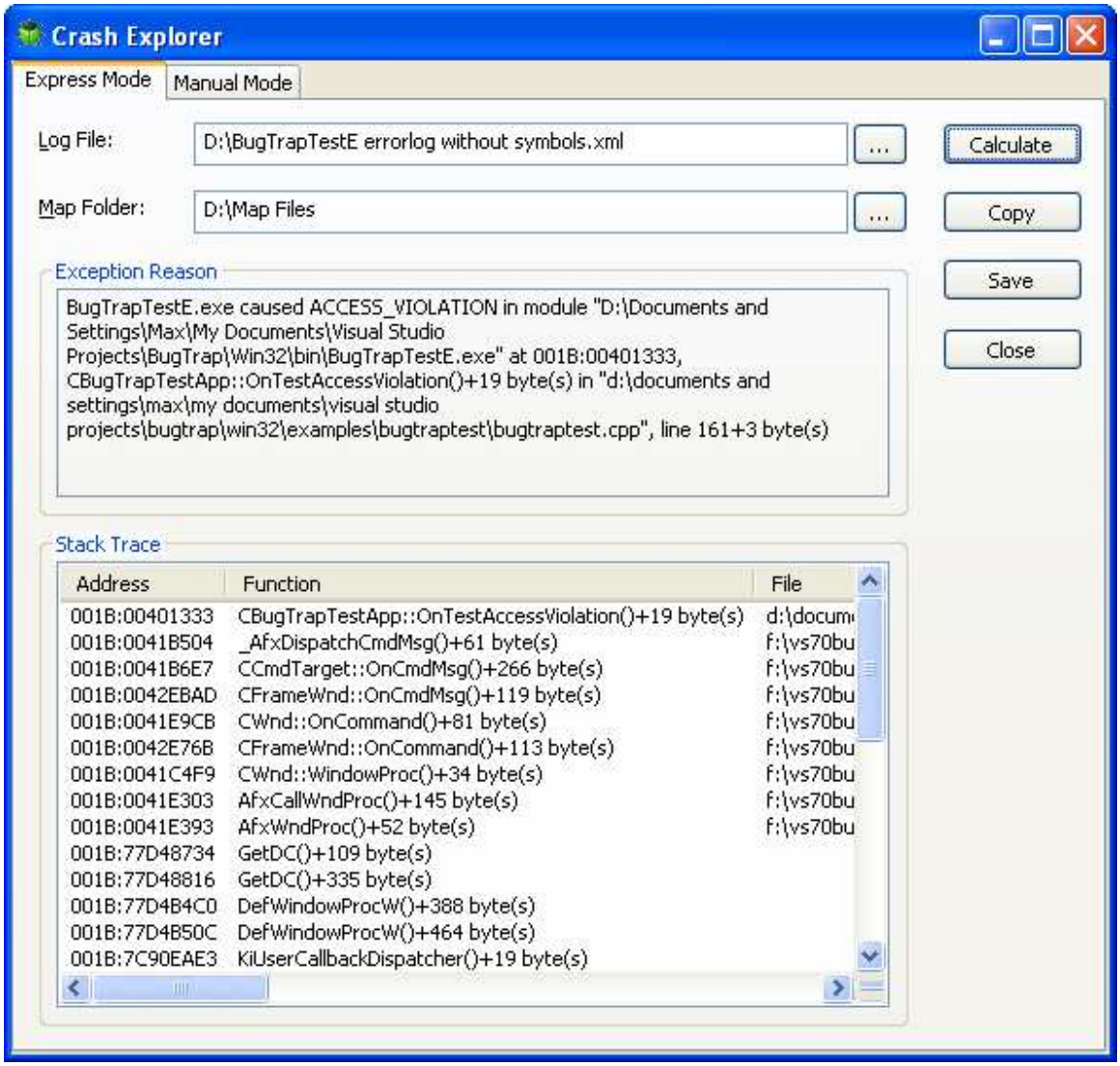
MAP file mapping

MyApp.exe => **MyApp**.map
- or -
MyLib.dll => **MyLib**.map

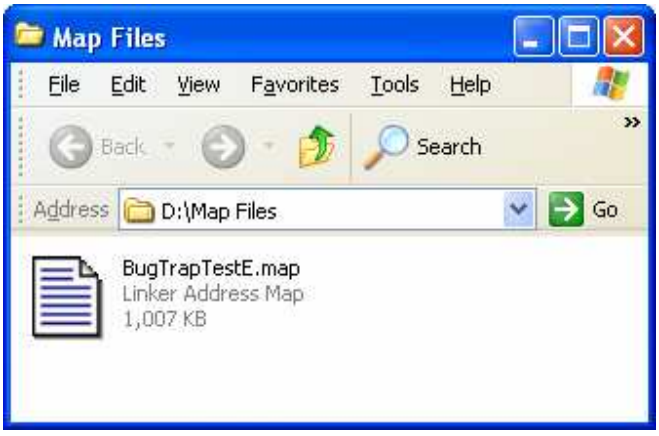
PDB file mapping

MyApp.exe => **MyApp**.pdb
- or -
MyLib.dll => **MyLib**.pdb

Once you have prepared a folder with MAP and PDB files, specify a path to the XML log file and press Calculate button. Program output can be copied to the clipboard or saved to a text file:



CrashExplorer running in express mode



We have been working on one module for the main executable in this project, so we have only one MAP file

You may also restore symbolic information manually (on Manual Mode tab). This might be useful if you have chosen plain text format of log files. BugTrap always stores hexadecimal

crash address in text log file, even if you don't distribute PDB file to customers. The following example shows error message without symbolic information (without source file name and line number):

```
BugTrapTest.exe caused ACCESS_VIOLATION in module "<Executable
Path>\BugTrapTest.exe" at 001B:00401333
```

`001B:00401333` is a crash address, but you only need to know the address offset (address part after the colon). It's `00401333`.
Text log also includes physical load addresses for every module (DLL or EXE file) loaded in process address space. Physical load address is vital for mapping crash address to source file name and line number in your project. Most developers rarely rebase project modules (adjust load addresses of every loaded module to avoid module overlapping in the memory). Operating system changes load addresses for overlapped modules. Therefore physical load addresses may not match to preferred load addresses specified during module linking. You may find module load addresses in module information block, for example:

```
Process: BugTrapTest.exe, PID: 2312, Modules:
-------------------------------------
<Executable Path>\BugTrapTest.exe (1.0.0.1), Base: 00400000, Size: 002C8000
C:\WINDOWS\system32\ntdll.dll (5.1.2600.2180), Base: 7C900000, Size: 000B0000
C:\WINDOWS\system32\kernel32.dll (5.1.2600.2180), Base: 7C800000, Size: 000F4000
C:\WINDOWS\system32\USER32.dll (5.1.2600.2622), Base: 77D40000, Size: 00090000
C:\WINDOWS\system32\GDI32.dll (5.1.2600.2818), Base: 77F10000, Size: 00047000
```

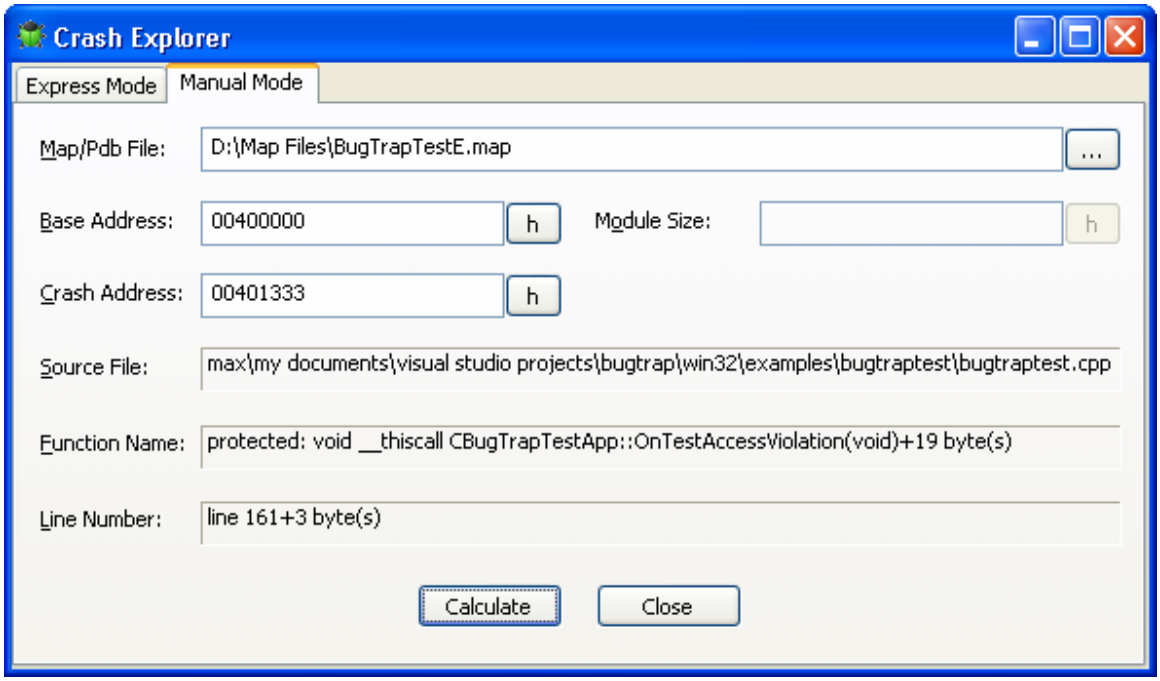
There are a lot of modules loaded in address space of the process. We need to know physical load address of one module. We are looking for a module that caused an exception. According to the log file this module is `BugTrapTest.exe`:

```
BugTrapTest.exe caused ACCESS_VIOLATION in module "<Executable
Path>\BugTrapTest.exe" at 001B:00401333
```

So, now you have physical load address of `BugTrapTest.exe` module - according to the log file, it's `00400000`. By the way most EXE files are loaded at this address, though it may vary for different DLLs. Let's sum up all available information:

Crash Address	00401333
Module Name	BugTrapTest.exe
Physical Load Address	00400000

It's quite enough to find source file name, function name and line number if you have MAP file for `BugTrapTest.exe` module. Note, CrashExplorer automatically extracts preferred load address (`00400000` in our example) from `BugTrapTest.map` file. Preferred load address is the same as physical load address for this module, so you can leave it as-is. CrashExplorer doesn't know crash address, so you should put `00401333` to corresponding field.
Press Calculate button and check the result:



CrashExplorer running in manual mode

It's also possible to walk through call stack entries by specifying different call addresses in "Crash Address" field.

3.3.4.1Summary

There are three different approaches to track information: PDB files, minidumps and MAP files. Which approach is better? There is no absolute answer, but I would recommend the following:

a) redistribute PDB files during beta testing to internal testers and SQA groups;
b) don't redistribute PDB files to end users;
c) instead, store PDB files for all public releases in local repository;
d) always use minidump files;
e) use CrashExplorer along with PDB or MAP files for quick error analysis;
f) use minidumps along with PDB files for deep error analysis in the debugger.

3.4 Native C++ exceptions

It is also possible to intercept unhandled C++ exceptions using BugTrap. The following line of code installs BugTrap *terminate*-handler called by the runtime on unhandled C++ exception:

```
BT_SetTerminate();
```

`BT_SetTerminate()` is just a macro that calls `set_terminate()` defined in C runtime. Since `set_terminate()` is not defined in BugTrap headers, you should include `<eh.h>` in program source. Note that `set_terminate()` installs termination routine only in the active thread. In multithreaded environment you should call `BT_SetTerminate()` in every thread. After proper

installation of termination routine, BugTrap automatically unwinds stack trace to the location of `throw` statement that caused unhandled C++ exception and you may normally see your function on top of the stack.

3.4.1 Integration with MFC

By default, MFC intercepts all un-caught exceptions derived from `CException` class, but you may use BugTrap instead of MFC error handler. This, however, requires few additional changes in your code, because BugTrap can't intercept errors caught by MFC before BugTrap handler. In order to let BugTrap be notified about exception first, you should derive frame, view, dialog and window classes from `BTWindow` class. `BTWindow` class is a template and it takes a name of base window class as a parameter. For example, if you want to handle all uncaught exceptions in main frame class, you should derive your frame class from `BTWindow<CFrameWnd>` rather than `CFrameWnd`:

```
class CMainFrame : public BTWindow<CFrameWnd>
```

Similarly, if you want to handle all exceptions in your view class, you should use this code:

```
class CMyView : public BTWindow<CView>
 - or -
class CMyView : public BTWindow<CScrollView>
```

Other than that, no additional changes are required, it is not necessary to change base class name in RTTI macros (`IMPLEMENT_XXX`) or in message map macros (`BEGIN_MESSAGE_MAP`). As a bonus, BugTrap also recognizes exceptions derived from STL `exception` class. For both types of exceptions, BugTrap can extract error description and put it to the report.

If you have noticed, that after these changes Visual Studio doesn't treat your frame or view class appropriately and doesn't display command handlers for your class in a wizard, you can add these lines to the header of your class:

```
// this is required to bypass VS.NET parser issues
#ifdef __NEVER_DEFINED__
 #define BTWindow<CView> CView
#endif

class CMyView : public BTWindow<CView>
```

3.4.2 Integration with ATL/WTL

ATL/WTL projects may also take advantage of BugTrap window classes. Even though ATL doesn't provide default exception handler, it is still better to let BugTrap handle all uncaught errors before exception is passed to Windows. There is special version of `BTWindow` class defined for ATL. From user perspective, this class takes the same template arguments and provides the same syntax:

```
class CMainFrame : public BTWindow< CFrameWindowImpl<CMainFrame> >
```

Unlike MFC which internally handles dialogs as normal windows, ATL uses specific classes for dialogs. That's why all ATL dialog-derived classes should use `BTDialog` rather than `BTWindow`:

```
class CMyDialog : public BTDialog< CDialogImpl<CMyDialog> >
```

More complex projects may use ATL and MFC simultaneously. For such projects you should prefix `BTWindow` class name with `ATL` or `MFC` namespace:

```
class CMyAtlView : public ATL::BTWindow< CWindowImpl<CMyAtlView> >
 - and -
class CMyMfcView : public MFC::BTWindow<CView>
```

3.5 Custom log files

Often normal code causes an exception as a side effect of another logical mistake. In this case standard error report may not help. Unfortunately there is no general approach which could solve all logical errors. There are only some techniques which can simplify error detection. Custom logging is probably the most efficient error-preventive mechanism. Developers track important program activity in log files. These files can be viewed and discovered after the crash.
BugTrap has built-in functions that can attach an arbitrary number of additional files to the report in detailed mode (with flag `BTF_DETAILEDMODE` was specified).
You can attach custom log files to the report as shown below:

```
BT_AddLogFile(_T("LogFile1.txt"));
BT_AddLogFile(_T("LogFile2.txt"));
```

If you want to export some keys from Windows registry and attach them to the report, you can take advantage of built-in BugTrap function:

```
BT_AddRegFile(_T("Settings.reg"),
      _T("HKEY_CURRENT_USER\\Software\\My Company\\My Application\\Settings"));
```

Custom log files can be added when application is being started or you can set custom error handler and perform additional initialization in this handler. Custom error handler is called by BugTrap in response to the unhandled exception:

```
void CALLBACK MyErrHandler(INT_PTR nErrHandlerParam)
{
      ... // last-minute customization
}
...
BT_SetPreErrHandler(MyErrHandler, 0);
```

C/C++ developers don't have standard logging functions. Many developers write their own code. Typically, developers open a file, append a string and close the file. There is nothing bad in this approach; however BugTrap includes built-in functions that can further simplify this task. These functions have several advantages over the regular method:

- BugTrap keeps all messages in a list during normal application processing so that new messages can be quickly added and old messages can be quickly removed from the log. BugTrap automatically flushes all messages to the disk before quitting the application. Such approach provides the best tradeoff between the speed of log updates and crash resistance, but you should not use these functions on very large log files (usually more than 10,000 records).
- Log files can be stored in XML or in plain text format. XML data can be exported to any other format, for example you can export XML records to nice HTML table with few lines of XSL code. However, plain text can be loaded into memory 5-10 times faster than XML dataset. BugTrap uses custom XML parser, optimized for streaming large XML datasets and you will not notice much difference in speed on files with few thousand entries. Larger files might require more time to be parsed. So, you should consider the size of the file before making decision about log format.
- All logging functions are thread safe and you can safely add log entries to the same log file from different threads without explicit synchronization.
- Logging functions correctly deal with national characters. Log information is encoded in UTF-8 format and you can correctly interpret any locale-dependent information such as file paths.
- Log messages can be echoed to the `STDOUT`, `STDERR` and to debugger console.
- Log messages can be automatically filtered according to their severity.

1. Opening & closing

You can open an arbitrary number of log files. Use `BT_OpenLogFile()` to get the handle of newly opened log file. This handle must be passed to one of the functions: `BT_AppLogEntry()` or `BT_InsLogEntry()`. You can use an arbitrary log file name in `BT_OpenLogFile()` or you can pass `NULL` pointer to assign default name to the log. Default log file name is `"%APPDATA%\<Application Name>\<Main Module Name>.log"`. Log file should be closed using function `BT_CloseLogFile()`. This function releases internally allocated resources.

2. Controlling log size

You can limit maximum number of bytes (`BT_SetLogSizeInBytes()`) or records in a log file (`BT_SetLogSizeInEntries()`). Older records are automatically pulled out from the file. By default log files are unlimited (log size = `MAXDWORD`).

3. Adding new records

`BT_InsLogEntry()` inserts new records at the beginning of the file. `BT_AppLogEntry()` appends new records to the end of the file. There are special versions of these functions with `printf`-like syntax. It's also possible to enable time statistics for all log entries by specifying `BTLF_SHOWTIMESTAMP` flag in `BT_SetLogFlags()` function. Timestamps are stored in locale-independent format `YYYY/MM/DD HH:MM:SS`.

4. Filtering output

Log output can be filtered using various log levels. Log level is assigned to every log entry. The following log levels are available:

- `BTL_ERROR` – error messages (the highest priority);
- `BTL_WARNING` – warning messages;
- `BTL_INFO` – information messages (the lowest priority).

You can pass minimum desirable log level to `BT_SetLogLevel()` and subsequent `BT_InsLogEntry()` or `BT_AppLogEntry()` operations will add all messages with specified or higher priority to the log. For example you can set log level to `BTL_WARNING` and only warning and error messages will be added to the log. You may even disable the output for all log messages by passing `BTL_NONE` to `BT_SetLogLevel()`.
Every `BT_InsLogEntry()` or `BT_AppLogEntry()` operation takes message level as an argument. Different parts of your program may add information messages, warning messages and error messages. You don't have to add any extra code to specify which of these messages must be added to the log. Single `BT_SetLogLevel()` call controls all output. Usually this setting is stored somewhere in program configuration and user may dynamically control the amount of produced log output. By default all messages are added to the log file.

5. Echo mode

It is possible to duplicate log messages on the screen in console applications or dump messages to the debugger console. By default log messages are only stored in a file, but echo mode can be enabled using `BT_SetLogEchoMode()` function. Please note that enabled echo mode decreases the speed of log updates because the output of log files sharing the same echo mode is mutually synchronized. Normal log updates that don't use any echoing are performed much faster – literally instantaneously.

6. Code example

The following snippet of code calls different logging functions:

```
// open new log file, use the default log file name
INT_PTR iLogHandle = BT_OpenLogFile(NULL, BTLF_TEXT);
// set log size = 100 records
BT_SetLogSizeInEntries(iLogHandle, 100);
// automatically add time statistics to log output
BT_SetLogFlags(iLogHandle, BTLF_SHOWLOGLEVEL | BTLF_SHOWTIMESTAMP);
// apply filter to log output
BT_SetLogLevel(iLogHandle, BTL_WARNING);

// get default log file name
PCTSTR pszLogFileName = BT_GetLogFileName(iLogHandle);
// add custom log file to the report
BT_AddLogFile(pszLogFileName);

// insert log entries at the begging of the file
BT_InsLogEntry(iLogHandle, BTL_INFO, _T("custom log message"));
BT_InsLogEntryF(iLogHandle, BTL_WARNING, _T("numeric output: %d"), 123);
// - or -
// append log entries to the end of file
BT_AppLogEntry(iLogHandle, BTL_ERROR, _T("another message"));
BT_AppLogEntryF(iLogHandle, BTL_INFO, _T("printf-like syntax: %s"), pszMessage);

// Close log file
BT_CloseLogFile(iLogHandle);
```

C++ developers may prefer simplified interface of built-in `BTTrace` class that wraps these functions:

```
BTTrace trace(NULL, BTLF_TEXT);
```

```
trace.InsertF(_T("printf-like syntax: %s"), pszMessage);
// - or -
trace.Append(BTL_WARNING, _T("something is wrong"));
```

3.6 Configuring reports delivery

Report may be delivered to product support by e-mail, over HTTP or TCP-based network protocol. Every approach has its own advantages, see BugTrap server topic for details. Destination e-mail address for error reports may be specified using such code:

```
BT_SetSupportEMail(_T("your@email.com"));
```

You may configure BugTrap to send error reports to BugTrap server by low-level TCP-based network protocol, just specify desirable server host and port number:

```
BT_SetSupportServer(_T("localhost"), 9999);
```

If you want to send error reports over HTTP rather than native BugTrap protocol, simply specify server URL as host name, and `BUGTRAP_HTTP_PORT` or `80` as a port number:

```
BT_SetSupportServer(_T("http://localhost/BugTrapWebServer/RequestHandler.aspx"),
BUGTRAP_HTTP_PORT);
```

You may specify e-mail address where you want to receive notification messages about incoming error reports from BugTrap server:

```
BT_SetNotificationEMail(_T("another@email.com"));
```

Note: notification e-mails may be disabled in BugTrap server configuration file.

3.7 Using BugTrap for Win32 in server applications

Server applications and various services must not show GUI. Default action can be pre-configured for such applications and BugTrap won't display any dialogs for such applications. For example:

```
// Force BugTrap to submit reports to support server w/o GUI
BT_SetActivityType(BTA_SENDREPORT);
```

It's also possible to restart your server after the problem from the custom error handler. You can set custom error handler using `BT_SetPostErrHandler()`.

4 BugTrap for .NET

4.1 Adding BugTrap to .NET application

.NET version of BugTrap is redistributed as managed library: BugTrapN.dll. This DLL consists of managed and unmanaged code. Such design lets BugTrap support pure managed .NET assemblies as well as mixed C++ assemblies that could throw managed .NET exceptions and native Win32 exceptions.
BugTrap for .NET exposes both managed and unmanaged (native) interfaces. Managed interface is accessible from C# or VB.NET code:

```
ExceptionHandler.AppName = "Your application name";
ExceptionHandler.Flags = FlagsType.DetailedMode | FlagsType.EditMail;
ExceptionHandler.DumpType = MinidumpType.NoDump;
ExceptionHandler.SupportEMail = "your@email.com";
ExceptionHandler.SupportURL = "http://www.your-web-site.com";
ExceptionHandler.SupportHost = "localhost";
ExceptionHandler.SupportPort = 9999;
```

Unmanaged interface is accessible from native Win32 code and was discussed earlier. It is possible to use any interface or even both interfaces in the same application.

4.2 Redistributing BugTrap for .NET

For performance reasons, .NET version of BugTrap uses wide character strings. This code is compatible with Windows NT/2000/XP. Windows 95/98/Me support is abandoned .NET version also depends on DbgHelp - you may find more information above. More importantly, it requires Microsoft .NET Framework 2.0 and Visual C++ runtime. In particular, the following DLLs are required:

- msvcm80.dll
- msvcp80.dll
- msvcr80.dll

You may ship these DLLs with your application or you may wish to install Microsoft Visual C++ 2005 Redistributable Package.

4.3 Error analysis for .NET applications

As well as Win32 version, BugTrap for .NET also generates error information in few formats:

a) log files enriched with PDB files when possible;
b) minidump files.

4.3.1 Exception log

BugTrap for .NET generates detailed error log in XML or text format. Log file includes managed stack trace for the thread that causes an exception. Due to certain limitations of .NET framework, current version of BugTrap cannot produce managed stack trace for any

other threads (i.e. only one thread is logged). We are working on this issue, trying to extend BugTrap functionality.
Stack trace for the current thread, though, includes all necessary details:

```
<stack>
  <frame>
    <assembly>BugTrapNetTest</assembly>
    <native-offset>200</native-offset>
    <il-offset>49</il-offset>
    <type>BugTrapNetTest.MainForm</type>
    <method>System.Void exceptionButton_Click(System.Object sender,
System.EventArgs e)</method>
    <method>System.Void exceptionButton_Click(System.Object sender,
System.EventArgs e)</method>
    <file><Source Path>\MainForm.cs</file>
    <line>2d</line>
    <column>4</column>
  </frame>
  ...
<stack>
```

If you redistribute your application along with PDB file as it was discussed earlier, log file includes line number and source file names. If PDB file is not accessible, log file includes only assembly, type and method names.

4.3.2 Minidump files

As discussed, it is possible to generate redistribute applications without accompanying PDB files, produce a minidump file and analyze it in WinDbg or in Visual Studio. It is handled with the help of SOS Debugging Extension (sos.dll). This extension is redistributed as a part of Microsoft .NET Framework. SOS adds several commands to Visual Studio Debugger and WinDbg. By this time, the most recent version of SOS has one limitation: it reveals readable call stack with method names and types, but it doesn't show line numbers and source file names. Supposedly, this issue should be resolved in the next revision of Windows Debugging Tools. Moreover, there is yet another issue - SOS doesn't seem to work with small minidump files (`MiniDumpNormal`). Full minidump files (`MiniDumpWithFullMemory`) definitely work, but they are significantly larger. Unpacked crash dump may consume 100 – 150MB. Archived dump file is smaller – 40MB which is better, but still not usable in most cases. It also takes more time to prepare large dump and to make an archive. Frankly speaking, current version of SOS doesn't look like finished product. It might be useful during product testing, but it is not recommended in production environment. This line of code disables minidump output:

```
ExceptionHandler.DumpType = MinidumpType.NoDump;
```

To use minidump files in managed application, set dump type to full:

```
ExceptionHandler.DumpType = MinidumpType.WithFullMemory;
```

The following example demonstrates sample session in WinDbg:

1. load SOS extension from Microsoft .NET Runtime folder:

```
.load C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\SOS.dll
```

2. check if the extension was loaded:

```
.check
```

Output:

```
Extension DLL chain:
    C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\sos.dll: image 2.0.50727.42,
API 1.0.0, built Fri Sep 23 00:27:26 2005
        [path: C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\sos.dll]
```

3. set symbol/source search paths:

```
.sympath c:\test\sym
.srcpath c:\test\src
.exepath c:\test\bin
```

4. list managed threads:

```
!threads
```

Output:

```
ThreadCount: 3
UnstartedThread: 0
BackgroundThread: 2
PendingThread: 0
DeadThread: 0
Hosted Runtime: no
                                  PreEmptive   GC Alloc           Lock
       ID OSID ThreadOBJ    State     GC       Context       Domain   Count APT
Exception
   0    1  970 00164368      6020 Enabled  00000000:00000000 001584f8     1 STA
System.Exception (01293190)
   2    2  4b4 00170550      b220 Enabled  00000000:00000000 001584f8     0 MTA
(Finalizer)
   7    3  f40 0015bf48       220 Enabled  00000000:00000000 001584f8     0 Ukn
```

Note: there is an exception object in the Exception column for the first thread:
System.Exception (01293190)

5. display exception information of this object:

```
!pe 01293190
```

Output:

```
Exception object: 01293190
Exception type: System.Exception
```

```
Message: <none>
InnerException: <none>
StackTrace (generated):
    SP       IP       Function
    0012F02C 00C8A759
BugTrapNetTest.MainForm.exceptionButton_Click(System.Object, System.EventArgs)
    0012F044 7B060A6B System.Windows.Forms.Control.OnClick(System.EventArgs)
    0012F054 7B105379 System.Windows.Forms.Button.OnClick(System.EventArgs)
    0012F060 7B10547F
System.Windows.Forms.Button.OnMouseUp(System.Windows.Forms.MouseEventArgs)
    0012F084 7B0D02D2
System.Windows.Forms.Control.WmMouseUp(System.Windows.Forms.Message ByRef,
System.Windows.Forms.MouseButtons, Int32)
    0012F0D0 7B072C74
System.Windows.Forms.Control.WndProc(System.Windows.Forms.Message ByRef)
    0012F134 7B0815A6
System.Windows.Forms.ButtonBase.WndProc(System.Windows.Forms.Message ByRef)
    0012F170 7B0814C3
System.Windows.Forms.Button.WndProc(System.Windows.Forms.Message ByRef)
    0012F178 7B07A72D
System.Windows.Forms.Control+ControlNativeWindow.OnMessage(System.Windows.Forms.
Message ByRef)
    0012F17C 7B07A706
System.Windows.Forms.Control+ControlNativeWindow.WndProc(System.Windows.Forms.Me
ssage ByRef)
    0012F190 7B07A515 System.Windows.Forms.NativeWindow.Callback(IntPtr, Int32,
IntPtr, IntPtr)

StackTraceString: <none>
HResult: 80131500
```

Now we have long awaited stack trace.

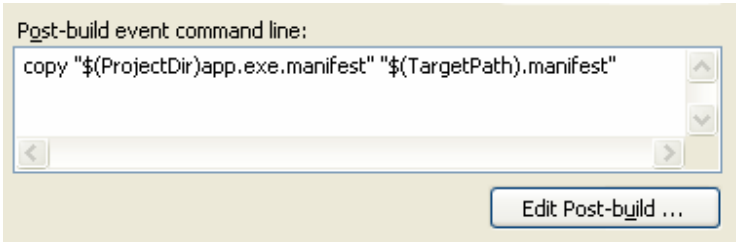
4.4 Notes for GUI .NET applications

Microsoft .NET Framework 2.0 lets you enable Windows XP themes support by calling `Application.EnableVisualStyles()`. By default, this code is added to every new project by AppWizard. BugTrap may initialize Windows Common Controls before XP themes are enabled in your code. Windows Common Controls 5.x handles may be impurely interpreted by ComCtl32 version 6.x. You may notice this problem when you do not see any icons in BugTrap GUI. The issue may be fixed by adding a manifest file to your application:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity
    version="1.0.0.0"
    processorArchitecture="X86"
    name="Microsoft.Windows.Common-Controls"
    type="win32"
/>
  <description>Windows forms common controls manifest</description>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity
        type="win32"
        name="Microsoft.Windows.Common-Controls"
        version="6.0.0.0"
```

```
        processorArchitecture="X86"
        publicKeyToken="6595b64144ccf1df"
        language="*"
        />
    </dependentAssembly>
  </dependency>
</assembly>
```

If your executable is called `MyApp.exe`, then this file must be named `MyApp.exe.manifest`. It must be copied to the same folder as your executable. To automate this task, save this file as `app.exe.manifest`, put it to your project and add a command to post build events:



This command will automatically rename and copy manifest file to the folder with the executable.

5 BugTrap server

Error reports may be delivered to product support in e-mail attachments, over HTTP or low-level TCP-based network protocol.

1. E-mail attachments

E-mail attachments are ideal for distributing small files across the Internet.

Advantages:

a) this approach doesn't require installing any additional servers. You may simply use your Internet Service Provider;
b) error reports can be transparently delivered through firewalls;
c) product support may handle error reports in standard e-mail clients and provide customers with feedback via e-mail.

Disadvantages:

a) it's very difficult or even impossible to handle e-mail messages with attached error reports automatically. For example, you cannot effectively filter error reports for multiple products;
b) most SMTP servers reject e-mail messages with large attachments – it may cause problems with large minidump files stored in the attachment.

2. Low-level TCP-based network protocol

BugTrap may deliver error reports to BugTrap server over light-weight network protocol optimized for transferring large amounts of data. BugTrap server provides better opportunity for products with complex support on the local network. Usually this is the most suitable option for software developers, SQA groups and local testers during product development and beta-testing.

Advantages:

a) error reports may be automatically stored in the repository, arranged by product name and filtered according to configuration file;
b) there are no limitations regarding the size of the report, it may include large minidump file as well as you can attach arbitrary number of custom log files to the report;
c) BugTrap server is extremely fast and lightweight server. It can be effectively used on the local network. It can be installed on any computer and it doesn't require Web-server.

Disadvantages:

a) native BugTrap protocol may be blocked by firewalls, that's why this option is primarily intended for local area network.

3. HTTP protocol

Since most computers are protected by firewalls, BugTrap server may not successfully receive error reports from the Internet. This can be solved using BugTrap Web server. BugTrap Web server may handle error reports received over HTTP protocol - a standard transport protocol transparently passed through firewalls.

HTTP protocol is not as efficient for data transfers as native BugTrap server protocol and BugTrap Web server requires a computer with Web-server (Microsoft IIS 5 – 6), but

BugTrap Web server may handle all reports automatically without any user interaction. This is the best option for products with complex support on the Internet.

Advantages:

a) error reports may automatically be stored in the repository, arranged by product name and filtered according to configuration file;
b) there are no limitations for report size, it may include large minidump file as well as you can attach arbitrary number of custom log files;
c) error reports can be transparently transferred through firewalls.

Disadvantages:

a) you have to maintain a computer with Web-server (Microsoft IIS 5 – 6) on the network;
b) HTTP protocol is not as efficient as native BugTrap protocol and BugTrap server is much faster than BugTrap Web server running on Web-server.

The rest of this chapter describes BugTrap server and BugTrap Web server.

5.1 BugTrap server

BugTrap server is a standalone server that handles all requests using lightweight TCP-based protocol. It doesn't require Web-server and it can be started on any computer.
The same server may receive reports from multiple products. You may restrict the list of products and the amount of the received information in one config-file. Win32 and .NET versions reuse the same network protocol, so the same server may be used for both types of applications. It's possible to receive e-mail notifications about incoming reports from BugTrap server.
There are two versions of BugTrap server available:

a) .NET version of BugTrap server optimized for Windows 2000/XP;
b) platform-independent Java version of BugTrap server.

Both versions reuse the same format of XML configuration file and maintain the same repository. Both versions use asynchronous network operations and thread pooling for optimal performance and scalability.
BugTrap server is designed to be stable and to continue normal reports processing even in case of unexpected internal errors: it just writes error information to system event log *(.NET version)* or standard error stream *(Java version)*, closes broken connection and terminates broken task. At the same time other threads remain stable and unaffected.

5.1.1 Installing .NET version of BugTrap server

.NET version of BugTrap server requires Microsoft .NET Framework 2.0.
BugTrap server can be installed on Windows NT platform as Windows service or it can be launched as typical Win32 application with `/run` command line option.
Though BugTrap server may be used as normal Windows application, it's strongly recommended to run BugTrap server as Windows service on Windows 2000 or Windows XP operating systems. You may install and uninstall BugTrap service in the command prompt:

Installation command line: `BugTrapServer.exe /install`

De-installation command line: `BugTrapServer.exe /uninstall`

5.1.2 Installing Java version of BugTrap server

Java version of BugTrap server requires the following components:

a) Java 2 Platform, Standard Edition 5.0 (J2SE)
   http://java.sun.com/j2se/1.5.0/download.jsp
b) JavaMail 1.4
   http://java.sun.com/products/javamail/downloads/index.html
c) JavaBeans Activation Framework 1.1
   http://java.sun.com/products/javabeans/jaf/downloads/index.html

Server code is located in `BugTrap\Server\JBugTrapServer\JBugTrapServer.jar` file. BugTrap server can be executed using the following batch file (the example assumes J2SE, JavaMail and JAF are installed on your computer):

```
@echo off

set JAVAHOME=%ProgramFiles%\Java
set JDK=jdk1.5.0_07
set JMAIL=javamail-1.4
set JAF=jaf-1.1
set JAVABIN=%JAVAHOME%\%JDK%\bin
set
CLSPATH=".;%JAVAHOME%\%JDK%\jre\lib\rt.jar;%JAVAHOME%\%JMAIL%\mail.jar;%JAVAHOME
%\%JAF%\activation.jar"

"%JAVABIN%\java.exe" -classpath JBugTrapServer.jar;%CLSPATH%
BugTrapServer.ServerApp 2>>BugTrapServerError.log
```

Note: the example above redirects standard error output stream to `BugTrapServerError.log` file. This is useful for tracking internal server errors.

5.2 BugTrap Web server

BugTrap Web server is a typical ASP.NET Web-application. It should be registered as typical Web-application in Internet Information Server 5 or 6.
The same server may receive reports from multiple products. You may restrict the list of products and the amount of received information in one config-file. Win32 and .NET versions reuse the same network protocol, so the same server may be used for both types of applications. It's possible to receive e-mail notifications about incoming reports from BugTrap Web server.

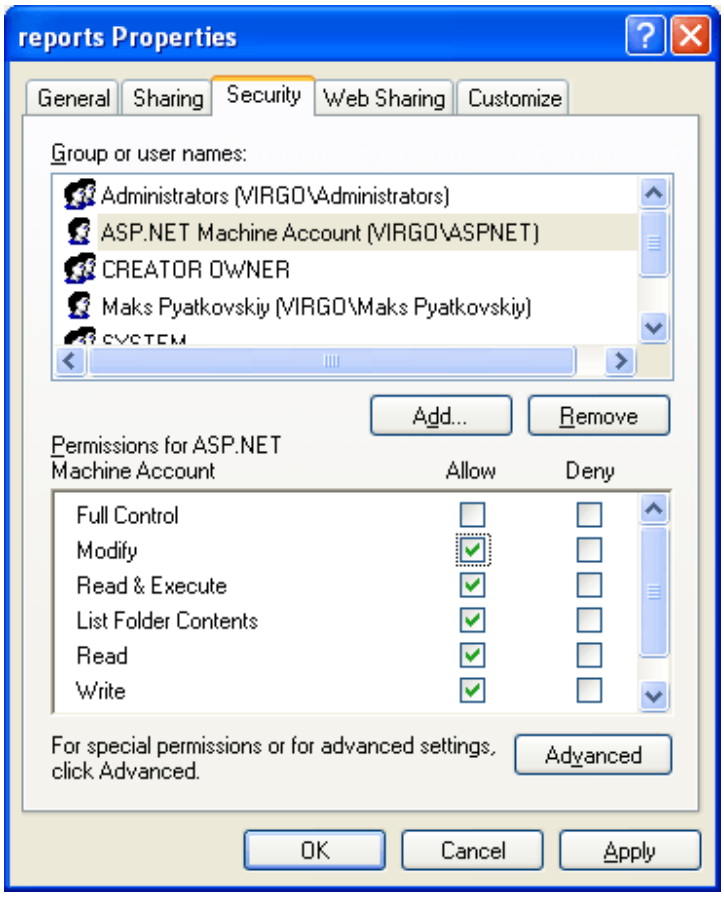
5.2.1 Installing BugTrap Web server

BugTrap Web server requires ASP.NET 2.0 and Microsoft .NET Framework 2.0.
BugTrap Web server directory contains several files such as `RequestHandler.aspx` and `Web.config`. `RequestHandler.aspx` intercepts client requests and redirects them to corresponding DLL located in `bin` directory. `RequestHandler.aspx` should be explicitly specified in HTTP address passed to `BT_SetSupportServer()`:

```
http://<domain name>/BugTrapWebServer/RequestHandler.aspx
```

By default BugTrap Web server doesn't have permission for creating new folders on your computer. You should create the repository and manually grant `Modify` privilege for this folder to `ASPNET` user.
Let's suppose you want to store error reports in `reports` sub-folder of your Web application. In this case you should complete the following steps:

1. create new `reports` folder (typically in `C:\Inetpub\wwwroot\BugTrapWebServer`);
2. open properties window for newly created folder;
3. switch to `Security` tab in folder properties;
4. click `Add` button;
5. type `ASPNET` in object names field and press `OK` button;
6. select `Modify` checkbox;
7. press `OK` button to save changes.



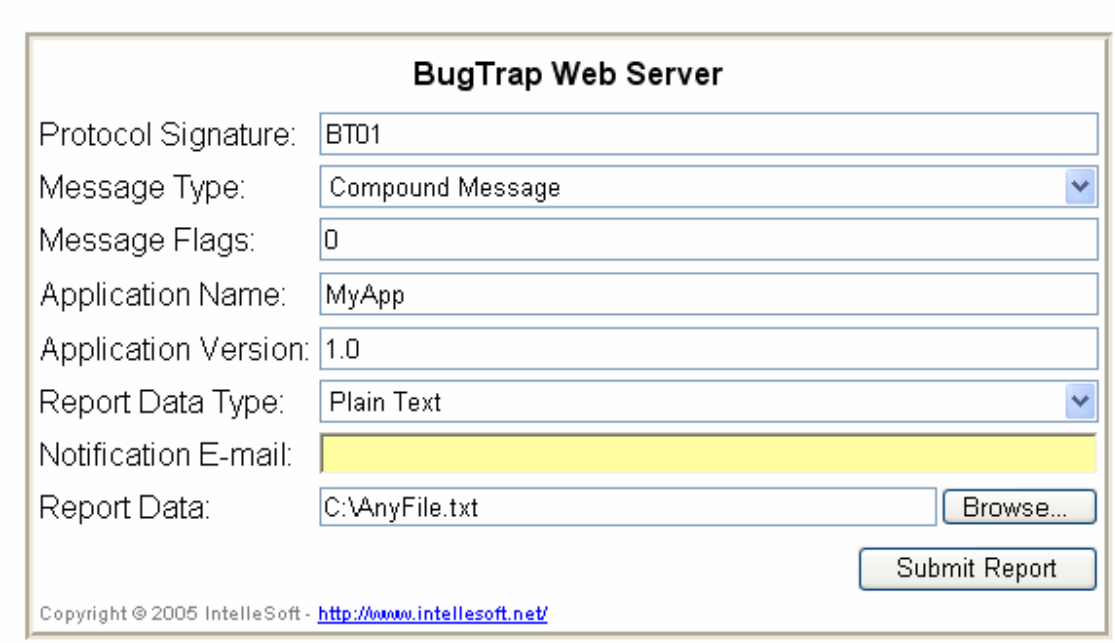
5.2.2 Testing BugTrap Web server

Usually it's easier to test BugTrap Web server using "BugTrap Request Simulator" web page and only then use real BugTrap client.

1. custom errors mode is enabled in `Web.config` for easier diagnostics. By default detailed error information will be shown to users locally connected to Web-server.

You can make this information available for remote users, just set `<customErrors mode="On"/>` in `Web.config` and save your changes;

2. open your browser and go to `http://<server name>/BugTrapWebServer/RequestSimulator.htm`;
3. fill out test form with arbitrary product information:



Note: you should leave notification e-mail empty unless you have properly configured SMTP-server in `Web.config` as described below.

4. press "Submit Report" button;
5. you should see the following message if everything is properly configured and the repository is accessible for `ASPNET` user:

```
- <result>
    <code>OK</code>
    <description>The operation completed successfully</description>
  </result>
```

6. after completing the test you may want to disable custom errors mode in `Web.config` and delete `RequestSimulator.htm` file;
7. now it is good time to update server settings in BugTrapTest and make sure BugTrap client can establish a connection.

5.3 Configuring BugTrap server

Both BugTrap server and BugTrap Web read configuration from XML files with almost the same structure.

5.3.1 BugTrap server configuration file

BugTrap server configuration is stored in `BugTrapServer.exe.config` XML file that must be located in the same folder with `BugTrapServer.exe` *(.NET version)* or `JBugTrapServer.jar` *(Java version)* file.

BugTrap server configuration file has the following structure:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="applicationSettings"
type="BugTrapServer.ApplicationSettingsHandler, BugTrapServer"/>
  </configSections>
  <applicationSettings>
    <logEvents>true</logEvents>
    <serverPort>9999</serverPort>
    <reportPath>c:\reports</reportPath>
    <reportsLimit>-1</reportsLimit>
    <maxReportSize>-1</maxReportSize>
    <smtpHost>smtp.server.address</smtpHost>
    <!--<smtpPort>25</smtpPort>-->
    <smtpUser>username</smtpUser>
    <smtpPassword>password</smtpPassword>
    <senderAddress>sender@email.com</senderAddress>
    <reportFileExtensions>log,xml,zip</reportFileExtensions>
    <applicationList>
    <!--
      <application>FirstApp</application>
      <application version="1.2">SecondApp</application>
    -->
    </applicationList>
  </applicationSettings>
</configuration>
```

These parameters are described below.

5.3.2 BugTrap Web server configuration file

BugTrap Web server may be configured using almost the same XML file but this file should be located in BugTrap Web server application folder and it should be called `Web.config`. Usually this file contains standard information about .NET Web-application, but BugTrap Web server configuration file has several custom fields:

```
<?xml version="1.0"?>
<configuration>
  <configSections>
    <section name="applicationSettings"
type="BugTrapServer.ApplicationSettingsHandler"/>
  </configSections>
  <system.web>
    <compilation debug="false" urlLinePragmas="true"/>
    <authentication mode="None"/>
    <customErrors mode="RemoteOnly"/>
    <sessionState mode="Off" cookieless="true"/>
    <!--<trust level="BugTrapWebTrust"/>-->
```

```
  </system.web>
  <applicationSettings>
    <logEvents>false</logEvents>
    <serverPort>9999</serverPort>
    <reportPath>reports</reportPath>
    <reportsLimit>-1</reportsLimit>
    <maxReportSize>-1</maxReportSize>
    <smtpHost>smtp.server.address</smtpHost>
    <!--<smtpPort>25</smtpPort>-->
    <smtpUser>username</smtpUser>
    <smtpPassword>password</smtpPassword>
    <senderAddress>sender@email.com</senderAddress>
    <reportFileExtensions>log,xml,zip</reportFileExtensions>
    <applicationList>
    <!--
      <application>FirstApp</application>
      <application version="1.2">SecondApp</application>
    -->
    </applicationList>
  </applicationSettings>
</configuration>
```

5.3.3 Configuration settings

Both files have the same set of parameters specified in `appSettings` section:

Parameter	Description
`logEvents`	Indicates whether to report Start, Stop commands and error messages in system event log.
`serverPort`	Port number where server listens incoming error reports. This number must be the same as a value specified in `BT_SetSupportServer()`.
`reportPath`	Path to the repository where reports are stored.
`reportsLimit`	Maximum number of reports accepted by the server for the same product. -1 means unlimited number of reports.
`maxReportSize`	Maximum accepted size of report file. -1 means unlimited report size.
`smtpHost`	Address of SMTP server used for sending notification e-mails to product support about incoming reports. E-mail notifications will be disabled if this field is empty.
`smtpPort`	This field can be used to override default port number of SMPT server.
`smtpUser`	User name to establish a connection to SMTP server.
`smtpPassword`	Password to establish a connection to SMTP server.
`senderAddress`	Sender address (“from” field) of notification e-mails sent about new errors.
`reportFileExtensions`	Colon delimited list of report file extensions accepted by the server. Report files will not be filtered if this list is empty. You should not change this setting in most cases.

You can configure the list of accepted products in `applicationList` section. BugTrap server rejects reports for products not listed in `applicationList` section. You may leave this section empty if you want to accept reports from any application.

5.3.4 System event log and BugTrap Web server

By default BugTrap Web server won't write error information to system event log. If you want to take advantage of error logging, you may set `logEvents=true` in `Web.config` file. This change requires few additional steps because access to system event log is disabled for ASP.NET applications. BugTrap installs custom security policy file that overrides default security settings for BugTrap Web server. This file is called `bugtrap_web_trust.config` and it is typically located in `C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\CONFIG`. This folder also includes machine-level `Web.config` file. Go to this folder and open main `Web.config` file in any text editor. At the top of this file you will notice the list of available security policies:

```
<location allowOverride="true">
    <system.web>
        <securityPolicy>
            <trustLevel name="Full" policyFile="internal" />
            <trustLevel name="High" policyFile="web_hightrust.config" />
            ...
        </securityPolicy>
        <trust level="Full" originUrl="" />
    </system.web>
</location>
```

Add to this list a new policy as shown below:

```
<location allowOverride="true">
    <system.web>
        <securityPolicy>
            <trustLevel name="Full" policyFile="internal" />
            <trustLevel name="High" policyFile="web_hightrust.config" />
            <trustLevel name="Medium" policyFile="web_mediumtrust.config" />
            <trustLevel name="Low"  policyFile="web_lowtrust.config" />
            <trustLevel name="Minimal" policyFile="web_minimaltrust.config" />
            <trustLevel name="BugTrapWebTrust"
                        policyFile="bugtrap_web_trust.config" />
        </securityPolicy>
        <trust level="Full" originUrl="" />
    </system.web>
</location>
```

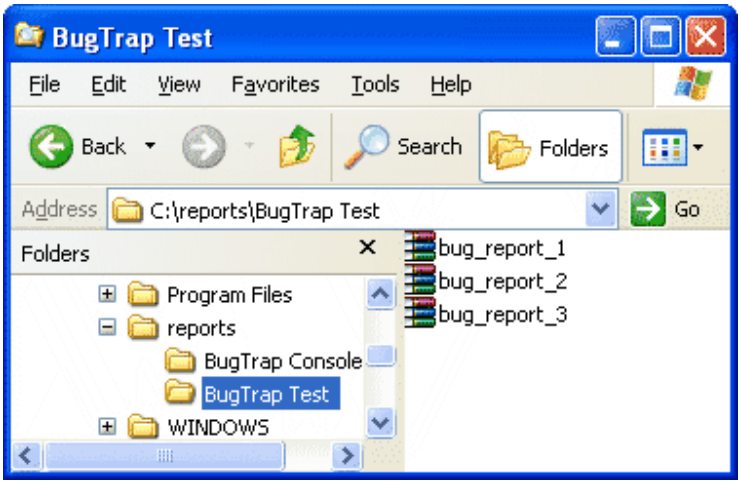
Save your changes and open local BugTrap `Web.config` file (typically stored in `C:\Inetpub\wwwroot\BugTrapWebServer`).

Locate the following line: `<!-- <trust level="BugTrapWebTrust"/> -->`

Remove comments from XML tag and save your changes. Now BugTrap Web server should be running in custom security policy.

5.3.5 BugTrap server repository

BugTrap server creates folders tree for different projects and arranges reports among those folders. Every report gets its own unique name to avoid collisions:



5.4 A conclusion

BugTrap package provides you with comprehensive set of tools that can track and manage unexpected errors in most kinds of applications on most environments. It simplifies error analysis and problem fixing, it makes product more stable and improves the quality of product support. BugTrap decreases maintenance costs during whole product life cycle.

APPENDIX A – Folders list

BugTrap folders tree:

Folder	Description
BugTrap for Win32 & .NET\	Root product folder
doc\	Product documentation
Net\	.NET client, supplementary applications and examples
BugTrapNet\	Project folder of BugTrap DLL for .NET
Examples\	Folder with .NET examples
BugTrapConsoleNetTest\	Project folder of sample console .NET application
BugTrapNetTest\	Project folder of sample GUI .NET application
ThreadsNetTest\	Project folder of sample multithreaded .NET application
Win32\	Win32 client, supplementary applications and examples
bin\	Executables of BugTrap for Win32 and sample applications
BugTrap\	Project folder of BugTrap DLL for Win32
Examples\	Folder with Win32 examples
BugTrapConsoleTest\	Project folder of sample console Win32 application
BugTrapTest\	Project folder of sample GUI Win32 application
Server\	Root folder for server applications
BugTrap Server\	.NET service files
JBugTrapServer	Java server files

BugTrap Web server folders tree (typically installed in C:\Inetpub\wwwroot folder):

Folder	Description
BugTrapWebServer\	Root folder for Web server files
bin\	BugTrap Web server executable files