Application crash with no explanation

Asked 13 years, 9 months ago Modified 7 months ago Viewed 9k times

6

I'd like to apologize in advance, because this is not a very good question.

I have a server application that runs as a service on a dedicated Windows server. Very randomly, this application crashes and leaves no hint as to what caused the crash.

When it crashes, the event logs have an entry stating that the application failed, but gives no clue as to why. It also gives some information on the faulting module, but it doesn't seem very reliable, as the faulting module is usually different on each crash. For example, the latest said it was ntdll, the one before that said it was libmysql, the one before that said it was netsomething, and so on.

Every single thread in the application is wrapped in a `try/catch (...)` (anything thrown from an exception handler/not specifically caught), `__try/__except` (structured exceptions), and `try/catch` (specific C++ exceptions). The application is compiled with /EHa, so the catch all will also catch structured exceptions.

All of these exception handlers do the same thing. First, a crash dump is created. Second, an entry is logged to a new file on disk. Third, an entry is logged in the

application logs. In the case of these crashes, **none of this is happening**. The bottom most exception handler (the `try/catch (...)`) does nothing, it just terminates the thread. The main application thread is asleep and has no chance of throwing an exception.

The application log files just stop logging. Shortly after, the process that monitors the server notices that it's no longer responding, sends an alert, and starts it again. If the server monitor notices that the server is still running, but just not responding, it takes a dump of the process and reports this, but this isn't happening.

The only other reason for this behavior that I can come up with, aside from uncaught exceptions, is a call to `exit` or similar. Searching the code brings up no calls to any functions that could terminate the process. I've also made sure that the program isn't terminating normally (i.e. a stop request from the service manager).

We have tried running it with windbg attached (no chance to use Visual Studio, the overhead is too high), but it didn't report anything when the crash occurred.

What can cause an application to crash like this? We're beginning to run out of options and consider that it might be a hardware failure, but that seems a bit unlikely to me.

c++ **crash**

Improve this question

Follow

asked Mar 7, 2011 at 19:21

Collin Dauphinee

13.9k ● 1 ● 47 ● 73

IS the log file stream being flushed? – James Mar 7, 2011 at 19:27

Have you verified that your thread exception handlers actually work on the server? They may be attempting to generate the crash dump & additional logging, but lack the permissions to write to their target locations, etc... – Jon Mar 7, 2011 at 21:20

Yes, of course. They've found a few bugs.
– Collin Dauphinee Mar 8, 2011 at 9:16

5 Answers

Sorted by: Highest score (default)

9

If your app is evaporating and not generating a dump file, then it is likely that an exception is being generated which your app doesn't (or can't) handle. This could happen in two instances:

1. A top-level exception is generated and there is no matching `catch` block for that exception type.
2. You have a matching `catch` block (such as `catch(...)`), but you are generating an exception within that handler. When this happens, Windows will rip the bones from your program. Your app will simply cease to exist. No dump will be generated, and virtually nothing will be logged, This is Windows' last-

ditch effort to keep a rogue program from taking down the entire system.

A note about `catch(...)`. This is patently Evil. There should (almost) never be a `catch(...)` in production code. People who write `catch(...)` generally argue one of two things:

"My program should never crash. If anything happens, I want to recover from the exception and continue running. This is a server application! ZOMG!"

-or-

"My program might crash, but if it does I want to create a dump file on the way down."

The former is a naive and dangerous attitude because if you do try to handle and recover from every single exception, you are going to do something bad to your operating footprint. Maybe you'll munch the heap, keep resources open that should be closed, create deadlocks or race conditions, who knows. Your program will suffer from a fatal crash eventually. But by that time the call stack will bear no resemblance to what caused the actual problem, and no dump file will ever help you.

The latter is a noble & robust approach, but the implementation of it is much more difficult that it might seem, and it fraught with peril. The problem is you have to avoid generating any further exceptions in your exception handler, and your machine is already in a very

wobbly state. Operations which are normally perfectly safe are suddenly hand grenades. `new`, `delete`, any CRT functions, string formatting, even stack-based allocations as simple as `char buf[256]` could make your application go >POOF< and be gone. You have to assume the stack and the heap both lie in ruins. No allocation is safe.

Moreover, there are exceptions that can occur that a `catch` block simply can't catch, such as SEH exceptions. For that reason, I always write an unhandled-exception handler, and register it with Windows, via SetUnhandledExceptionFilter. Within my exception handler, I allocate every single byte I need via static allocation, before the program even starts up. The best (most robust) thing to do within this handler is to trigger a separate application to start up, which will generate a MiniDump file from outside of your application. However, you can generate the MiniDump from within the handler itself if you are extremely careful no not call any CRT function directly or indirectly. Basically, if it isn't an API function you're calling, it probably isn't safe.

edited Apr 27 at 21:53

halfer

20.4k ● 19 ● 108 ● 200

answered Mar 7, 2011 at 19:52

John Dibling

101k ● 32 ● 191 ● 331

What bothers me most is that the bottom level catch-all has no body and exists only to prevent the whole application from going down, so we can at least see the absence of the thread. It shouldn't be possible for an uncaught exception to bring the application down. Even if a higher level catch/except throws an exception, it should be caught.
– Collin Dauphinee Mar 7, 2011 at 20:27

The empty catch-all is probably only masking the true problem. If, for instance, something caused a heap corruption your catch-call will let the program continue, but it is running on only 2 wheels, and will fall over eventually. You would be much better off getting rid of all the empty catch-alls and replacing them with the unhandled-exception handler I mention above, generating a minidump within the handler.
– John Dibling Mar 7, 2011 at 20:35

I'm pretty sure the problem is heap corruption. Do you have any hints for finding where it occurs? Memory profilers and similar aren't useful, since we can't use them with the production server and the issue doesn't occur on our test servers. – Collin Dauphinee Mar 8, 2011 at 9:55

You may find pageheap.exe useful. See the following two links: support.microsoft.com/default.aspx?scid=kb;en-us;286470
blogs.msdn.com/b/akshayns/archive/2007/11/24/…
– John Dibling Mar 8, 2011 at 15:10

1 Thanks for the help. For anyone reading this in the future, pageheap was replaced by gflags. As far as I can tell, Application Verifier also has equivalent functionality.
– Collin Dauphinee Mar 9, 2011 at 17:39

I've seen crashes like these happen as a result of memory corruption. Have you run your app under a

1

memory debugger like Purify to see if that sheds some light on potential problem areas?

Share Improve this answer

Follow

answered Mar 7, 2011 at 19:27

Timo Geusch

24.3k ● 5 ● 54 ● 71

We ran it under Purify, but we can't use it in production. It found no problems on our test server, though.
– Collin Dauphinee Mar 7, 2011 at 19:32

Side note, this is a 64-bit application. I think we had to make a 32-bit build, because Purify can't instrument 64-bit processes. – Collin Dauphinee Mar 7, 2011 at 19:38

@dauphic, I assume you purify'd the release build? The 32-bit build should not make any difference unless someone's being 'clever' with pointers (like casting a pointer to an integer type that's too small to hold the 64-bit pointer, then casting it back to a pointer). – Timo Geusch Mar 7, 2011 at 19:41

Yes, it was a release build. The code is all fairly well written C++ and has been through more static analysis tools than I can count, there should be no unsafe operations like that.
– Collin Dauphinee Mar 7, 2011 at 19:44

1

Analyze memory in a signal handler

http://msdn.microsoft.com/en-us/library/xdkz3x12%28v=VS.100%29.aspx

Share Improve this answer

Follow

edited Mar 7, 2011 at 19:38

answered Mar 7, 2011 at 19:28

Murali VP

6,397 ● 4 ● 31 ● 36

I'll try adding a signal handler for SIGABRT tomorrow, thanks for the suggestion. – Collin Dauphinee Mar 7, 2011 at 19:43

1

This isn't a very good answer, but hopefully it might help you.

I ran into those symptoms once, and after spending some painful hours chasing the cause, I found out a funny thing about Windows (from MSDN):

> Dereferencing potentially invalid pointers can disable stack expansion in other threads. A thread exhausting its stack, when stack expansion has been disabled, results in the immediate termination of the parent process, with no pop-up error window or diagnostic information.

As it turns out, due to some mis-designed data sharing between threads, one of my threads would end up dereferencing more or less random pointers - and of course it hit the area just around the stack top sometimes. Tracking down those pointers was heaps of fun.

There's some technincal background in Raymond Chen's [IsBadXxxPtr should really be called CrashProgramRandomly]

Share Improve this answer

Follow

answered Mar 7, 2011 at 21:15

molbdnilo

66.3k ● 3 ● 44 ● 85

0

Late response, but maybe it helps someone: every Windows app has a limit on how many handles can have open at any time. We had a service not releasing a handle in some situation, the service would just disappear, after a few days, or at times weeks (depending on the usage of the service). Finding the leak was great fun :D (use Task Manager to see thread count, handles count, GDI objects, etc)

Share Improve this answer

Follow

answered Oct 15, 2021 at 18:14

mBardos

3,007 ● 1 ● 21 ● 17