A brief and incomplete comparison of memory corruption detection tools

devblogs.microsoft.com/oldnewthing/20250124-00

January 24, 2025

I promised last time to do a comparison of memory diagnostic tools. We have runtime diagnostic tools Address Sanitizer (ASAN), Valgrind, and Application Verifier (AppVerifier, avrf), and we have recording tools rr, and Time Travel Debugging (TTD)

First, the runtime tools:

	ASAN	**Valgrind**	**AppVerifier**
Requires recompile	Yes	No	No
Supported languages	C,C++	All	All
Detects uninitialized read	No	Yes	No
Detects heap UAF	Yes	Yes	Yes
Detects stack UAF	Yes	Yes	No
Detects out-of-bound array access	Yes	No	No
Detects misuse of system APIs	No	No	Yes

ASAN detects a lot more types of memory errors, but it requires that you recompile everything. This can be limiting if you suspect that the problem is coming from a component you cannot recompile (say because you aren't set up to recompile it, or because you don't have the source code). Valgrind and AppVerifier have the advantage that you can turn them on for a process without requiring a recompilation. That means that you can ask a customer to turn it on at their site, without having to deliver a custom build to them. This is even more important on Windows because you have no chance of giving them an ASAN-enabled version of, say, `kernel32.dll`.

AppVerifier understands the semantics of many system APIs. For example, it will detect that you re-entered a non-reentrant lock, or you released a lock from the wrong thread.

ASAN works by inserting read and write barriers to memory accesses in order to catch semantically invalid accesses as they occur. Valgrind works by running the entire program under a CPU emulator so it can track memory accesses. AppVerifier works by adding additional tracing to memory allocations and checking whether unallocated bytes were modified. In lightweight mode, AppVerifier detects the corruption only after the fact, so you're not sure who did the corrupting. In Page heap mode, free memory is decommitted, and an attempt to access freed memory will trigger an access violation, stopping at the invalid access.

Bonus reading: Using AppVerifier to diagnose a crashing bug.

Next, the recording tools:

	rr	**TTD**
Requires recompile	No	No
Multithreading support	Yes	Yes
Multicore support	No	Yes
Supports shared memory	No	Yes
Supports async I/O	No	Yes

The rr tool records and replays the operations of every system call and assumes that the CPU behaves deterministically otherwise. Any memory reads are assumed to produce the same value that was last written by the program. This assumption reduces the size of the trace files and allows the program to run at nearly full speed.

Time Travel Tracing runs the program under a CPU emulator and records the actual results of every memory access, which might be different from the value last written if the memory was modified by something outside the program itself, such as the kernel, or another process modifying shared memory, or even another process doing a `WriteProcessMemory` into the process being traced. This allows it to observe that memory has changed due to asynchronous I/O (though it doesn't know *when* the change occurred), such as one we investigated some time ago, although that investigation didn't use Time Travel Tracing.

Combining a runtime tool with a recording tool gives a powerful one-two punch. You can use the runtime tool to detect the problem more reliably and more quickly, and you can use the recording tool to go back in time to watch the evolution of the memory block through the

lifetime of the program. For example, the runtime tool might tell you, “Erroneous double-free of memory block,” and you can use the recording tool to go back in time to the first time the memory block was freed to try to figure out why it was freed prematurely.