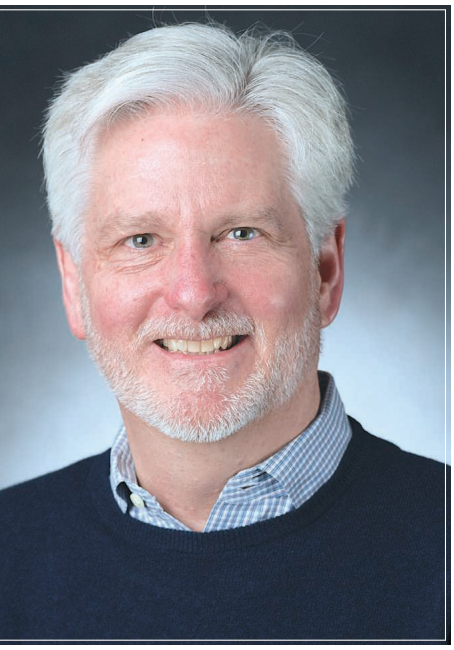

Trent Jaeger
Associate Editor in Chief

On Bridges and Software

Lately, I have been thinking about how the lessons of traditional engineering disciplines may be leveraged to develop more secure software. In particular, I have been thinking about how designing bridges to avoid failure may help us to develop software in a manner that avoids introducing exploitable flaws.

I was inspired to consider bridges by my son's visit to the city of Mostar in Bosnia and Herzegovina. Mostar is the site of a 16th-century bridge (Stari Most, the "Old Bridge") that spans high above the Neretva River. When my son visited, the most notable aspect of the bridge was how it has become a popular spot for high diving (not by my son). Historically, this bridge is famous for being the widest man-made arch of its time, designed by an apprentice under penalty of death if the bridge failed.

Two thoughts occurred to me. My first thought was the obvious notion that liability penalties for 16th-century bridges were quite severe. In this case, the bridge designer arranged his own funeral, in the event that the bridge failed. The second thought was that perhaps bridge construction in the 16th century may have more similarities to our current software development practices than we realize. I will start with the second thought.

While safety in bridge construction is not the same as safe software development, there are some interesting parallels. For example, modern bridge designers group the forces on a bridge into two categories (e.g., per *Bridges: The Science and Art of the World's Most Inspiring Structures* by David Blockley[1]): permanent and variable loads. While permanent forces are determined by the bridge components themselves, variable loads try to capture the less predictable loads deriving from environmental factors and exceptional use cases, such as earthquakes and collisions with bridge piers. With respect to software, permanent loads are analogous to the expected functionality, and variable loads are analogous to unexpected (i.e., possibly malicious) uses that may lead to failures.

Bridge design now requires compliance with a range of regulations, which prescribe tests to perform, including ones to assess variable loads. Although I greatly oversimplify the bridge design process, the basic idea, as I understand it, is that the bridge designers aim to construct a bridge whose strength is greater than the combination of loads (i.e., variable and permanent loads) with a high probability, as both the bridge's strength and its loads are not known precisely. As a result, the strength required of a bridge is often significantly greater than its expected loads.

How does the current approach to bridge design relate to safe software design? Consider software failures due to memory errors. In many cases, these types of errors are analogous to variable loads in bridges, in that inputs that trigger memory errors are typically not expected uses. However, unlike modern bridge design, we lack techniques (i.e., corresponding to tests for bridges) to estimate the impact of memory errors in a comprehensive or at least systematic manner. As a result, exploits of these software flaws continue.

What does the current approach to bridge design tell us about how to proceed? Ideally, we want to strengthen our programs to withstand the impact of the variable loads introduced by unsafe memory operations that may lead to memory errors. In addition, for both bridges and software, the costs of strengthening must be managed. However, it seems apparent that the tradeoff between failure and cost differs greatly between bridges and software, where bridge failure is currently deemed to be much more catastrophic (i.e., deadly, expensive to fix, visible to end users, etc.) than software failures. As a result, various proposals for systematic defenses to prevent memory errors (and/or their exploitation) and memory-safe programming languages free from such errors have not been adopted in software development for high-performance applications, leaving only incomplete, low-cost defenses in current systems.

Digital Object Identifier 10.1109/MSEC.2023.3258207
Date of current version: 12 May 2023

What are we missing to move forward? Software development still lacks techniques to assess the impact of memory errors or the costs of preventing such errors. For example, at present, we do not compute how many memory access operations a program may contain that may violate memory safety. If we do not even know the variable loads on our programs, how can we determine how much we need to strengthen our defenses against them? In addition, we lack systematic techniques to assess the cost of defenses, particularly for a combination of defenses. Without such techniques, how can we make decisions to utilize the proposed defenses wisely? That is, how can we start to assess the tradeoffs between strength and cost effectively?

Getting back to the issue of liability (the first thought), it seems apparent that without systematic techniques for strengthening software, we lack a foundation for establishing enforceable liability protections. Security researchers have long argued for "secure-by-design" software, but it can be argued that we have not yet provided sufficiently practical development techniques to achieve this goal. However, it can also be argued that the software industry has been content not to utilize systematic techniques to strengthen software. Pressure is mounting for change. For example, America's Cybersecurity and Infrastructure Agency Director Jen Easterly recently said that "the fact that we've accepted a monthly 'Patch Tuesday' as normal is further evidence of our willingness to operate dangerously at the accident boundary."[2] Systematic techniques to strengthen software automatically while accounting for cost may motivate a new mindset. Hopefully, we can develop and adopt such techniques before a really major disaster occurs.

Any discussion of memory safety in a security forum will naturally need to examine the question of why we do not simply switch to memory-safe languages. The options are certainly improving, so this question is becoming more pertinent for software development at large. However, even (mostly) memory-safe languages like Rust can introduce strengthening costs (e.g., runtime bounds checking), so they also inherently create a strength versus cost tradeoff.

Ideally, we would develop techniques to measure these tradeoffs quantitatively, so that we can make educated decisions about how to strengthen our software and provide more effective hardware support for strengthening software, not just for memory errors but for all classes of unsafe operations in programs.

For more on the topic of memory safety, please see Paul van Oorschot's two-part column on memory safety.[A1,A2] In addition, readers are encouraged to submit pieces on their efforts to the upcoming special issue on "Memory Safety" announced in this issue. ■

References

1. D. Blockley, *Bridges: The Science and Art of the World's Most Inspiring Structures*. Oxford, U.K.: Oxford Univ. Press, 2012.
2. J. L. Hardcastle, "US cybersecurity chief: Software makers shouldn't lawyer their way out of security responsibilities," *The Register*, Feb. 2023. [Online]. Available: https://www.theregister.com/2023/02/28/cisa_easterly_secure_software/

Appendix: Related Articles

A1. P. C. van Oorschot, "Memory errors and memory safety: C as a case study," *IEEE Security Privacy*, vol. 21, no. 2, pp. 70–76, Mar./Apr. 2023, doi: 10.1109/MSEC.2023.3236542.

A2. P. C. van Oorschot, "Memory errors and memory safety: A look at Java and Rust," *IEEE Security Privacy*, vol. 21, no. 3, pp. 62–68, May/Jun. 2023, doi: 10.1109/MSEC.2023.3249719.

Digital Object Identifier 10.1109/MSEC.2023.3264185