Mandating Memory Safety: The Time is Now

(Whitepaper area 4)

Michael Hicks

mwh@cs.umd.edu mwh@correctcomputation.com

Professor of Computer Science, the University of Maryland, and CTO, Correct Computation, Inc

Many of the vulnerabilities discovered and fixed in production code arise from highly dangerous **violations of memory safety**; these include *use-after-frees*, *buffer overflows*, and *out-of-bounds reads/writes*. Despite their more than 33-year history and the many attempts aimed at mitigating or blocking their exploitation, such vulnerabilities have remained a consistent, and sometimes worsening, threat.

- Common Weakness Enumerations (CWEs) 787, 125, 125, and 416 cover memory safety-based vulnerabilities, and these constitute four of MITRE's top eight weaknesses of 2020, considering prevalence and severity [15].
- For the last 12 years, nearly 70% of the bugs addressed in a yearly security update in Microsoft products were memory safety bugs [5].
- In May 2020, Google reported that 70% of the serious vulnerabilities in the Chrome browser codebase catalogued since 2015 are memory-safety violations [11], and the trend has persisted since (see `https://twitter.com/LazyFishBarrel`).
- A similar vulnerability percentage has been observed in MacOS & iOS, Firefox, Ubuntu, and in other large, critical systems [10].

Internet of Things (IOT) devices are also at risk. For example, recently discovered vulnerabilities in TCP/IP stacks used by millions of IOT devices owe to memory safety violations [1]. In general, IOT designs are often "close to the metal," and so developers turn to C or C++. But such developers tend not to have the experience or training of those working on browsers and operating systems, which are themselves routinely patched for memory safety problems.

> I believe that now is the time to **mandate** that a system with even modest security requirements provide increasingly **strong evidence of its memory safety**.

What forms might this evidence take? In the best case, the evidence results from some form of **automated reasoning** – *type checking*, *static analysis*, *formal verification* – and at the least, employs automated testing.

For new development, memory safety is not hard to attain: **memory safety vulnerabilities essentially occur in *only* C and C++ code** [19], so the requirement can be met by programming in *any other programming language*. In the not-so-distant past, one might have reasonably argued that for certain kinds of development, popular languages such as Java and Python are not efficient enough, and C and C++ are the only reasonable choices. But within the last ten years Google has developed Go [12, 18] and Mozilla developed Rust [16] to be practical but secure alternatives to C and C++; these languages aim to be fast, low-level, *and* memory-safe. Rust and Go have been rising in popularity—IEEE's 2019 Top Programming languages list ranks them 17 and 10, respectively, and both can be used for IOT development [13, 17].

What about active codebases already written in C and C++? Since Rust and Go are very different languages from C and C++, legacy code could be ported, perhaps incrementally, to **use language extensions such as Microsoft's Checked C**, which comprises annotations that when used pervasively will ensure spatial memory safety, ruling out out-of-bounds reads and buffer overflows [8]. Microsoft is developing Checked C to be part of their Azure Sphere IOT cloud service. Checked C is open source (built on Clang/LLVM) [6].

Another approach is to use **static program analysis** to provide assurance that developed C code is memory safe. This is the approach being taken in FreeRTOS, a popular IOT operating system, to ensure the memory safety of its TCP

library [4], and has been used in Amazon's s2n TLS library [14]. Static analysis differs from testing in an important way: While testing is able to uncover bugs, it is not able to prove their absence. Certain kinds of static analysis tools can do this because they are able to *consider all possible program executions* as part of their analysis. Even static analyzers that do not consider all executions, such as Facebook's open-source Infer tool [9], are precise enough that they would provide helpful evidence of memory safety. Running any such tools as part of a DevOps/continuous integration (CI) process is likely to increase code security [7].

Barring all of the above, some useful evidence of memory safety can be provided by **automated reasoning-enhanced testing**, such as *fuzz testing* and *symbolic execution* [2, 3]. These strategies use automated intelligence to cover more execution paths than would be covered with typical testing, thereby potentially revealing dangerous, memory safety-violating vulnerabilities. As mentioned above, testing cannot guarantee the absence of bugs, so it constitutes weaker evidence than we would like. Indeed, Chrome is regularly fuzz tested, but still suffers from memory safety bugs, as mentioned at the start.

In sum: Vulnerabilities owing to a lack of memory safety have been around far too long; they constitute a significant but addressable risk. The standard we develop should mandate evidence of memory safety. We should be flexible, at least at first, about the evidence required to prove it, but we can and should **mandate that that evidence employs automated reasoning** of increasing degrees.

References

[1] Keumars Afifi-Sabet. Weekly threat roundup: Chrome, Exchange Server, IoT devices. `https://www.itpro.com/security/vulnerability/359225/weekly-threat-roundup-chrome-exchange-server-iot-devices`, 2021.

[2] American fuzzing lop (afl). `http://lcamtuf.coredump.cx/afl/`, 2021.

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Conference on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008.

[4] Nathan Chong. Ensuring the memory safety of FreeRTOS. `https://www.freertos.org/2020/02/ensuring-the-memory-safety-of-freertos-part-1.html`, 2020.

[5] Catalin Cimpanu. Microsoft: 70 percent of all security bugs are memory safety issues. `https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/`, 2019.

[6] Microsoft Corporation. The Checked C clang repo. `https://github.com/microsoft/checkedc-clang`, 2021.

[7] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at facebook. *Commun. ACM*, 62(8):62–70, July 2019.

[8] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. Checked C: Making C safe by extension. In *Proceedings of the IEEE Conference on Secure Development (SecDev)*, September 2018.

[9] Facebook. Infer: A static analysis tool. `https://fbinfer.com/`, 2021.

[10] Alex Gaynor. What science can tell us about c and c++'s security. `https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/`, 2020. Presentation at Enigma.

[11] Google. Chrome: 70% of all security bugs are memory safety issues. `https://www.chromium.org/Home/chromium-security/memory-safety`, 2020.

[12] Google. Go programming language. `https://golang.org/`, 2020.

[13] Fredrik Lundström. Manage security vulnerabilities in embedded IoT devices with Rust. `https://medium.com/@flundstrom2/manage-security-vulnerabilities-in-embedded-iot-devices-with-rust-14aeabada68b`, 2019.

[14] Colm MacCarthaigh. Automated reasoning and Amazon s2n. `https://aws.amazon.com/blogs/security/automated-reasoning-and-amazon-s2n/`, 2016.

[15] MITRE. 2020 CWE top 25 most dangerous software weaknesses. `https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html`, 2020.

[16] Mozilla. The Rust programming language. `https://developer.mozilla.org/en-US/docs/Mozilla/Rust`, 2020.

[17] Poornima Narasimhan. Golang for Internet of Things — a perspective. `https://talktopoorni.medium.com/my-2-cents-golang-for-internet-of-things-596de9b1f8df`, 2019.

[18] Rob Pike. Go at Google: Language design in the service of software engineering. `https://talks.golang.org/2012/splash.article`, 2020.

[19] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 48–62, 2013.