Before the
Executive Office of the President at the White House
Office of the National Cyber Director
Washington, D.C. 20500

In the Matter of	)	
	)	
Open-Source Software Security:	)	ONCD-2023-0002
Areas of Long-Term Focus and Prioritization	)	
	)	
	)	

COMMENTS OF CONSUMER REPORTS

November 8, 2023

Stacey Higginbotham
Yael Grauer
Justin Brookman
Consumer Reports
1101 17th Street NW, Suite 500
Washington, DC 20036

Nov. 8, 2023

Acting Director Kemba Walden
Office of the National Cyber Director
The White House
1600 Pennsylvania Avenue, N.W.
Washington, DC 20500

Dear Acting Director Walden,

Consumer Reports appreciates the opportunity to comment on the Office of the National Cyber Director's request for information on areas of long term focus and prioritization related to open source software security. Last October, Consumer Reports hosted an online convening to discuss ways to encourage widespread adoption of code written in memory-safe languages. We issued two[1] reports[2] on the topic designed to help advance the use of memory safe code.

The use of memory unsafe languages has become readily apparent, threatening national security and the quality of software used in critical businesses. Roughly 60%-70% of browser and kernel vulnerabilities in C/C++ codebases are due to memory unsafety, which can be largely solved by using memory-safe languages.

While we believe it impossible to eliminate memory unsafe languages entirely, there are ways to mitigate the use of memory-unsafe language in use today. We recommend creating incentives that will encourage companies to implement memory-safe components in existing C/C++ projects, commit to using primarily memory-safe languages for new products and features, and address the most directly exposed attack surface in their most critical libraries and packages (often by rewriting that code).

There are also areas where we think the government can play a role in adoption of memory-safe languages. The government can create incentives for companies to share more information in their vulnerability disclosures to accurately track the problems associated with using memory-unsafe languages, and can create government-sponsored coding contests designed to promote memory-safe languages.

Thank you for considering our comments, and if you have questions on any of these areas please contact Stacey Higginbotham at stacey.higginbotham.consultant@consumer.org.

Sincerely,

Stacey Higginbotham, Policy Fellow
Justin Brookman, Director Privacy and Technology Policy

---

[1] Yael Grauer, "Future of Memory Safety Challenges and Recommendations" Consumer Reports. January 2023. https://advocacy.consumerreports.org/wp-content/uploads/2023/01/Memory-Safety-Convening-Report-1-1.pdf
[2] "How to Talk to Your Manager About Memory Safety" Consumer Reports and Internet Society. October 2023. https://innovation.consumerreports.org/wp-content/uploads/2023/10/Memory-Safety-One-Pager.pdf

Table of Contents

Questions for Respondents:
We are seeking insights and recommendations as to how the federal government can lead, assist, or encourage other key stakeholders to advance progress in the potential areas of focus described below.

Please consider providing input on these areas by addressing the questions below:

• Which of the potential areas and sub-areas of focus described below should be prioritized for any potential action? Please describe specific policy solutions and estimated budget and timeline required for implementation.

• What areas of focus are the most time-sensitive or should be developed first?

• What technical, policy or economic challenges must the Government consider when implementing these solutions?

• Which of the potential areas and sub-areas of focus described below should be applied to other domains? How might your policy solutions differ?

Potential Areas of Focus

- *Area: Secure Open-Source Software Foundations*
  - *Sub-area: Fostering the adoption of memory safe programming languages*
    - *Supporting rewrites of critical open-source software components in memory safe languages*
    - *Addressing software, hardware, and database interdependencies when refactoring open-source software to memory safe languages*
    - *Developing tools to automate and accelerate the refactoring of open-source software components to memory safe languages, including code verification techniques*
    - *Other solutions to support this sub-area*

Fix critical software elements first

It's clear that critical open source software components need to be rewritten in memory-safe languages. However, since it's impossible for all organizations to rewrite everything all at once, addressing dependencies and rewriting critical components in memory safe languages would improve the ecosystem.

When companies or organizations are trying to figure out which components are critical and should be rewritten first, we recommend the following steps:

- First, companies should evaluate their code to understand what software the organization is using, as well as understand the whole supply chain for that software. Modern software often pulls in third-party code, such as software libraries from multiple sources.
- Second, organizations should figure out where their biggest source of risk is. The source of risk might be the component with the most vulnerabilities reported the prior year, or components with network and privilege boundaries. Managers might need to reach out to external security experts and check the results of public code reviews.
- Finally, companies need to assess the blocks to memory-safe solutions such as a lack of capital, software dependencies within a service or product, available staff who can code in

memory-safe languages and a willingness to change. Once identified, start knocking those barriers down.

This means that getting to memory-safe code is both a code problem and a business process problem that companies need to tackle. It's also worth noting that in some cases, memory safety isn't yet possible. For example, IoT/embedded devices continue to be built using C/C++ for platform compatibility.

There are also gaps in memory-safe software across products today, but there are ways the government could continue to incentivize industry to address those gaps. Right now, it's not yet possible for governments to only buy memory-safe software.

For example, one can't say routers must be memory-safe top to bottom because no such products currently exist. But it may be possible for the government to say that newly developed custom components have to be memory-safe to slowly shift the industry forward. This would require some type of central coordination and trust in that system. The government could ask for a memory safety road map as part of procurement. The map would explain how the companies plan to eliminate memory-unsafe code in their products over time.

- *Sub-Area: Reducing entire classes of vulnerabilities at scale*
  - *Increasing secure by default configurations for open-source software development*
  - *Fostering open-source software development best practices, including but not limited to input validation practices*
  - *Identifying methods to incentivize scalable monitoring and verification efforts of open-source software by voluntary communities and/or public-private partnerships*
  - *Other solutions to support this sub-area*

Mitigating the harms of unsafe code

The way to reduce entire classes of vulnerabilities at scale is to use memory safe languages. This can be expensive, time consuming, and in some industries, such as embedded and IoT, will still leave behind legacy code in C/C++. It's important to note that while there are definitely things that can be done to reduce issues in C/C++ code (such as code review, fuzzers and sanitizers, exploit mitigations, privilege separation, etc.), these steps do not result in memory safety or memory safe code.

Sticking with memory-unsafe code is not an option going forward. The only scalable way to accomplish memory-safe code going forward is with languages that are considered memory-safe such as Rust. While developers using memory-unsafe languages can attempt to avoid all the pitfalls of these languages, this is a losing battle, as experience has shown that individual expertise is no match for a systemic problem. Even when organizations put significant effort and resources into detecting, fixing, and mitigating this class of bugs, memory unsafety continues to represent the majority of high-severity security vulnerabilities and stability issues. It is important to work not only on improving detection of memory bugs but to ramp up efforts to prevent them in the first place.

Options for incremental rewrites of code packages

However, we cannot replace all existing memory unsafe code, so organizations must include strategies to mitigate the harms of memory-unsafe code. An incremental approach targeting components of larger software packages is more feasible than aiming for complete rewrites of large and complex software packages. It may make sense to isolate components (i.e. sandbox them), compile them to something like WebAssembly, or rewrite them in a memory-safe language like Rust. These options present a number of trade-offs, including implementation effort, execution performance, and safety. For example, code that is otherwise security critical in a logic sense (like JITs, cryptographic primitives, etc.) may be comparatively worse for rewriting.

Sandboxing doesn't prevent memory safety bugs from causing problems within a sandbox. In particular, process-based sandboxing (unlike solutions like WebAssembly and Rust) hasn't historically prevented attackers from exploiting memory safety bugs to run arbitrary code within the sandbox. And, this gives them more freedom to exploit bugs in the sandbox itself or the trusted code interfacing with the sandbox code.

Compiling to something like WebAssembly addresses the limitations of process-based sandboxing, but currently comes at a runtime performance cost and can add significant toolchain complexity.

Rewriting in a memory-safe language like Rust has a high up-front cost for implementation, and may temporarily increase the number of logic bugs as a result of the rewrite (the bug count will go down over time, like it did for the original software), but the result is likely to be both fast and safe.

Which approach makes sense depends heavily on the desired outcome as well as available resources. Stakeholders may not agree on what makes sense in any given situation. In many cases, it is even desirable to use multiple techniques—Rust can be used to eliminate memory safety bugs, and running components in isolation with least privilege (for example, by compiling them to WebAssembly) can be used to deal with supply chain attacks.

Incentivize data-sharing about memory-unsafe code

There is a key role the government or industry can play when it comes to reducing memory safety issues. Since a primary way to mitigate memory-unsafe code is to identify that code and rewrite, recompile, or otherwise take action, it would be helpful to incentivize participants to share their findings when they run across memory-unsafe libraries and software packages.

This would be especially valuable for critical libraries and packages that handle sensitive data and support vulnerable users—medical data, government data, and tools used by journalists and human rights activists. It should also start with the softest and most directly-exposed attack surface[3]. But making informed decisions would require additional data that may be hard to get.

It’s a problem currently faced by the CVE, the database of Common Vulnerabilities and Exposures, which classifies vulnerabilities, and uses a Common Vulnerability Scoring System (CVSS). However the

[3] Chris Palmer, “Prioritizing Memory Safety Migrations,” Noncombatant.org, April 11, 2021, https://noncombatant.org/2021/04/09/prioritizing-memory-safety-migrations/

root causes of bugs are often vague, meaning that we lack meaningful visibility into the scale of the problem for many types of vendors.

For example, 4 Apple's security bulletins currently don't provide enough details to distinguish C/C++-induced memory vulnerabilities from logic bugs. And the metrics we have on the percentage of vulnerabilities that are due to memory unsafety are for only some cross-sections of the industry, such as Mozilla, Android, and Microsoft, which makes our understanding of the current state incomplete. It would be great to get broad, updated statistics on the percentage of vulnerabilities due to memory unsafety.

Support Memory-Safe Libraries and Coding Programs

The government can play a role by investing in and providing ongoing support for memory-safe FIPS certified cryptographic libraries. This would allow developers to use memory safe code instead of, e.g., openSSL, which people often revert to simply because those libraries have already been certified as being safe for government use.

The government can also create incentives for better code by creating and promoting competitions that can help create the conditions for a "space race" for memory safety. This would include reasons to upgrade to memory-safe libraries that are not just security-driven: replacements that are faster, better, have additional features, etc. The carrot approach for memory safety may include not just decreased future costs in cybersecurity, but also reliability and efficiency. Ideally, memory safety will be viewed as a proxy for funded, competent risk management strategy and for software that's currently evolving and malleable.

Additionally, creating an industry-wide focus on safety-by-design means that developers begin to write memory-safe code by default The goal is to speed along the transformation of the software industry so that we no longer tolerate and normalize companies placing the burden of staying cyber-safe on the enterprises and individuals who are least capable of doing so. Instead, products should be safe by design, and we should be increasingly intolerant of manufacturers who decide to choose unsafe practices for making products.

In some cases, even developers unaware of the large percentage of security bugs stemming from memory-unsafe code would be able to minimize them, if the industry norm becomes a state where memory safety is incorporated by default.

The PR incentive would work best if memory safety became easier to implement. The government could incentive tooling that would make this easier. Options that may help move the needle might be funding projects to sponsor new OSes or support existing Linux distros, or to team up with companies and other organizations to have a cash prize for the top 20 libraries that can securely use memory-safe alternatives.

As a peripheral example, Let's Encrypt had the goal of reducing the friction of deploying HTTPS by automating deployment and offering certificates free of charge, so they built the systems to do that. It was free and automated, and once HTTPS was ubiquitous enough, it became mandatory for crucial web platform features like service workers.

- *Sub-Area: Developer education*
  - *Integrating security and open-source software education into computer science and software development curricula*
  - *Training software developers on security best practices*
  - *Training software developers on memory safe programming languages*
  - *Other solutions to support this sub-area*

Change the developer pipeline

Before we even look at industry-wide issues with memory safety, it's important to address the pipeline.

Currently, some computer science courses expect students to do much of their systems-level work in C, which is notoriously memory-unsafe. Professors have a golden opportunity here to explain the dangers of C and similar languages, and possibly increase the weight of memory safety mistakes on exercise grading, which proliferate in student-written code just as they do outside of the classroom. Another opportunity is to switch languages for part of those courses. However, teaching parts of some courses in, for example, Rust could add inessential complexity, and may be impractical in some cases. There's a hard upper limit on how many new ideas and the number of programming languages you can throw at someone in a class before their brain shuts off, and many computer science classes are already at capacity.

There's also a perception that memory-safe languages, namely Rust, are harder to learn and will be difficult to use with hardware, which may dissuade people from learning it in the first place.

Professors also want to do their best to make sure their students graduate with the skills that will help them find the type of job they want, which becomes a chicken and egg problem—students often learn to program in C assuming it is the universal language that will allow them to be easily employable in the future, which results in companies wanting to hire students who can code in memory-safe languages such as Rust to have a smaller hiring pool.

To change this pattern, the industry itself must shift.

To help push the industry, the government could provide data on which companies are hiring people who know memory-safe languages, and which require C/C++ (which will also change with time). It might also be useful to get information on companies providing training in memory-safe languages to their engineers or writing specific projects in them.

The lowest-hanging fruit for memory safety is brand new code, but to be successful, we must recognize that some programmers may find memory-safe languages more difficult or be resistant to shifting to them. This can be mitigated by explaining that memory-safe languages force programmers to think through important concepts that ultimately improve the safety and performance of their code. In some cases, the concerns exist at executive levels of an organization. Management may distrust new languages, as well as have concerns that tools may not work properly. Perhaps the tools are workable but there is the sense that C/C++ equivalents are more reliable or easier to use. People realizing they

need new toolchains on platforms they support—and need to be able to debug them—leads to significant ecosystem drag. It requires significant activation energy to bootstrap an ecosystem into something government, organizations, and individuals can buy into without having to build expertise in the tool chain.

Shine a light on memory-safe code adoption

Outside of providing guidance to computer science professors and coding programs, or providing data about job prospects for coders programming in memory-safe languages, there is also an opportunity for the government to fund research.

For example, we could learn from the lessons of MANRS[4] (Mutually Agreed Norms for Routing Security), a global initiative that helps reduce the most common threat to the security of global Internet routing infrastructure. Routing security is a similar collective action problem to memory safety, and the MANRS "actions"—a set of practices that improve routing security—serve as a voluntary set of norms that stakeholders in internet routing can abide by to secure their smaller piece of the bigger picture, adding to the aggregate state of routing security.

When applied to memory safety this could involve two tactics: ensuring that memory safety is a clear basis for competition between similar offerings (for example, who has pledged to adopt memory-safe systems and software?) or worked-through case studies (or horror stories) that show the serious risks and costs of memory unsafety. Relating back to the MANRS initiative in routing security, a useful case study was the early 2022 incident[5] where Twitter was able to recover quickly in the face of Russian network hijacking, having learned a lesson from a few years before when Myanmar did the same thing and gravely impacted Twitter operations worldwide. Can we show that companies both gain in the market and avoid costly risks by making their products more memory-safe?

- *Area: Behavioral and Economic Incentives to Secure the Open-Source Software Ecosystem*
    - *Frameworks and models for software developer compensation that incentivize secure software development practices*
    - *Other solutions to support this sub-area*

Convince executives to invest in memory safety

One challenge at the private company level is convincing managers and executives to invest in memory-safety. Obviously, some of the prior actions, such as using data and government influence to equate memory-safe code with securely and competently designed software will help, but executives may need greater incentives, more focused on their ROI.

Developers can tell managers that memory safety is an up-front investment that will reduce a company's long-term support costs. Having fewer vulnerabilities will reduce an expensive triage

[4] "Mutually Agreed Norms for Routing Security," Internet Society, accessed Nov. 8, 2023, https://www.internetsociety.org/learning/manrs/
[5] Aftab Siddiqui, "Lesson Learned: Twitter Shored Up Its Routing Security," manrs.org, March 29, 2022, https://www.manrs.org/2022/03/lesson-learned-twitter-shored-up-its-routing-security/

process, and they will have fewer stability problems and nonsecurity crashes as well as performance improvements due to concurrency.

The biggest memory-safety challenge appears to be both technical and commercial. How do we deal with very large legacy codebases written in unsafe programming languages? Those working in industry pointed out that the social and commercial incentives to encourage fully addressing a problem of this scale do not exist.

To get to a place where everything is memory-safe, organizations need some regulatory or market incentive. There are many barriers to adoption, even as CISA, FTC, and the NSA push companies toward addressing memory-safe code. Getting over the perceived barriers to adoption and the time and monetary costs requires effective advocacy. Engineers at companies need support for this type of work because they often don’t have sufficient internal resources, and the options for hiring external contractors to deal with the security bugs is limited.