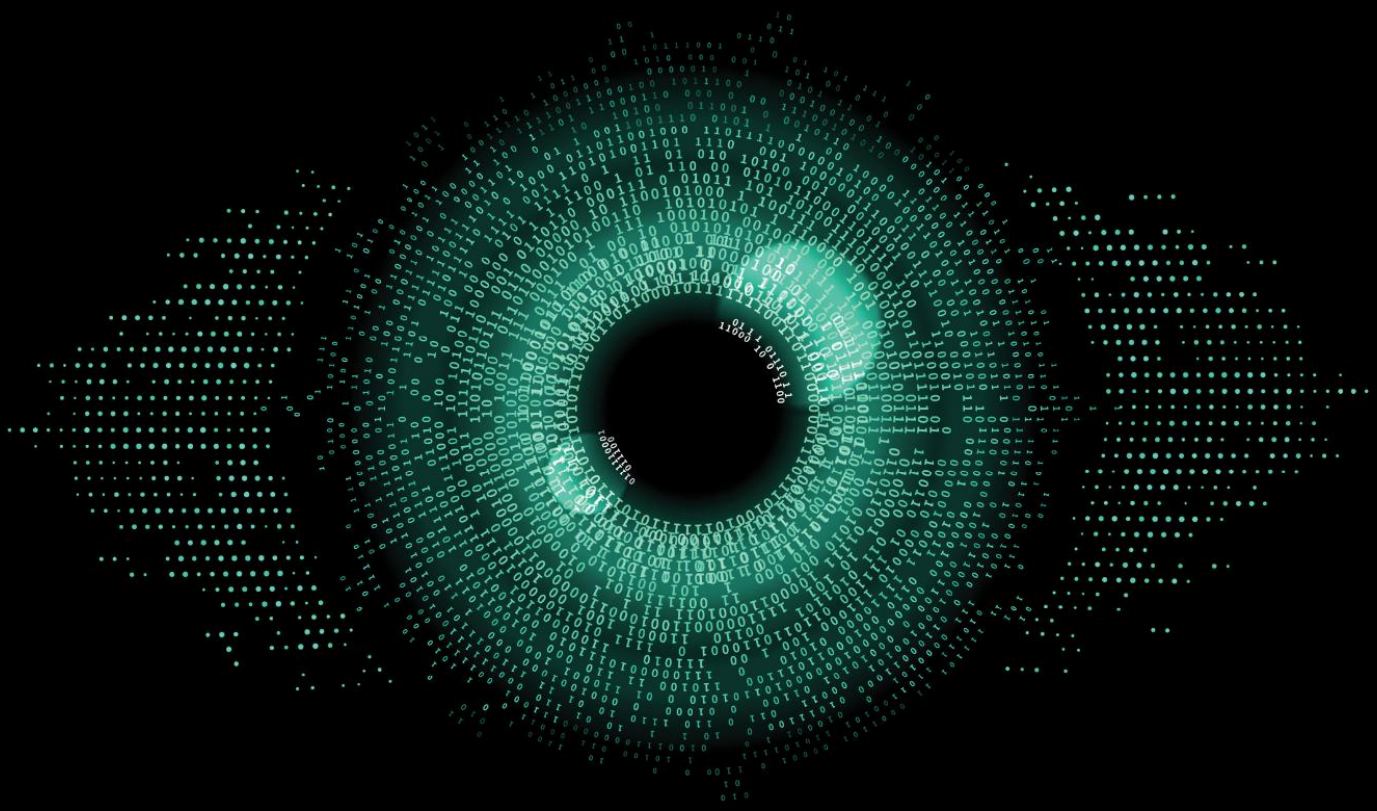

Future of Memory Safety

Challenges and Recommendations

YAEL GRAUER
JANUARY 2023

CR Consumer Reports | Security Planner

On October 27th, 2022, Consumer Reports hosted an online convening to discuss ways to encourage widespread adoption of code written in memory-safe languages. The event was hosted by Amira Dhalla and Yael Grauer from Consumer Reports and facilitated by Georgia Bullen from Superbloom. Attendees included approximately 25 individuals across civil society, education, government, industry, and the technical community, including Josh Aas from Internet Security Research Group and Prossimo; Jack Cable, Alex Gaynor, Joseph Lorenzo Hall from the Internet Society; Jacob Hoffman-Andrews from Electronic Frontier Foundation and Internet Security Research Group; Per Larsen from Immunant, Inc.; Bob Lord from CISA; Art Manion, Eric Mill, and Conrad Stosz from Office of Management and Budget; Harry Mourtos from Office of the National Cyber Director; Shravan Narayan from the University of Texas at Austin; Maggie Oates from Consumer Reports; Miguel Ojeda, Matthew Riley from Google; Christine Runnegar from the Internet Society; Deian Stefan from the University of California, San Diego: Ben L. Titzer from Carnegie Mellon University; and Zachary Weinberg from CMU.

The event gave participants the opportunity to share resources related to memory safety, discuss opportunities and barriers in the security ecosystem, and to brainstorm potential solutions to memory access vulnerabilities that exist in products across the marketplace.

Background (Why Consumer Reports)

This event stemmed from internal conversations about limitations of a Consumer Reports project, Security Planner, a free, interactive guide to staying safer online. Security Planner provides step-by-step, consumer-focused digital consumer tips that are easy to understand and implement, even for users without a technical background. In some instances, Security Planner provides recommendations of available consumer products that have been evaluated for privacy and security.

However, there are industry-wide threats that cannot be solved through user behavior or even consumer choice, and memory unsafety is one such issue. So we decided to reach out to experts to learn more about what they are doing to help reduce the vast number of security vulnerabilities related to memory access.

We also wanted to determine how Consumer Reports could complement ongoing efforts to improve the ecosystem for all end users. Consumer Reports is an independent, nonprofit member organization that works with consumers to create more fairness, safety, and transparency in the marketplace.

Why Memory Safety

Roughly 60 to 70 percent of browser and kernel vulnerabilities—and security bugs found in C/C++ code bases—are due to memory unsafety, many of which can be solved by using memory-safe languages. 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.

Challenges

Education

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. (However, most other memory-safe languages, like Python, Go, and JavaScript, achieve temporal memory safety through garbage collection, which substantially simplifies many aspects of programming, making the languages famously easier to learn.)

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. We also need more 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. Some of this information can be gleaned through an SBOM (software bill of materials), which could be useful in ascertaining which of the constituent parts of a final good are memory-safe, which old, unsupported libraries are being used, and so forth. However, hardware vendors self-assessing and making a legal attestation of the parts in a final good may have more information.

Distrust or Dislike of Memory-Safe Languages

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.

In some cases, memory safety isn't yet possible. For example, IoT/embedded devices continue to be built using C/C++ for platform compatibility.

And some resistance to moving off C/C++ is due to the sunk cost fallacy. Through joint partnerships, it may be helpful to explain that changing languages now, rather than avoiding

doing so, will yield less cost than it would in the future. Additionally, using Rust for new projects will ultimately result in higher productivity.

Approaches

There is general agreement among participants on an incremental approach targeting components of larger software packages rather than aiming for complete rewrites of large and complex software packages. However, opinions differ heavily about whether and when it makes 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.

Another area of agreement is that companies, government organizations, and other entities can at least identify and prioritize the most critical libraries and packages to shift to memory-safe languages. This would include those 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.

But making informed decisions would require additional data. CVE, the database of Common Vulnerabilities and Exposures, which classifies vulnerabilities, uses a Common Vulnerability Scoring System (CVSS), but the 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,

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.

Improving Incentives

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, etc. 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.

It's not yet possible for government procurement to only buy memory-safe software. For example, you 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.

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. Competitions and awards may also be some areas where we 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.

Public Accountability

Targeting the general public is a tactic that's worked well for other industries facing large problems in the past. In 1965, the book *Unsafe At Any Speed: The Designed-In Dangers of the American Automobile* by Ralph Nader was instrumental in making it widely known to the public that car manufacturers were resisting the introduction of safety features. It also showed that while manufacturers had patented safety interventions already, they told Congress it would take ages.

Similarly, a 1906 novel called *The Jungle* by Upton Sinclair portrayed harsh conditions and exploitations of immigrants in Chicago and similar industrialized U.S. cities. The book exposed unsanitary practices and health violations in the meatpacking industry, which led to public outcry and reforms such as the Federal Meat Inspection Act. Consumer Reports also has a long history of working with consumers to secure victories that have made the marketplace safer, healthier, and fairer. We have advocated for seat belt mandates, played a key role in encouraging Congress to pass the Safe Drinking Water Act, spurred fast food chains to serve chicken and beef raised without medically important antibiotics, and helped prod the U.S. Food and Drug Administration to issue guidance limiting the amount of arsenic in infant rice cereals.

How can we learn from this, and from medical, aviation, and automotive safety to make improvements? 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, for example, memory safety is incorporated by default.

The PR incentive would work best if memory safety became easier to implement. What if an existing group took on tooling that would make this all 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.

For 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 Worker.

Incremental Changes

One popular potential solution: advocacy and education for the general public that would rank companies by memory safety bugs and their severity that's accessible for a non-technical audience. But such a ranking would be impossible to do without rating companies on partial success.

Because potential solutions depend on the size of the system at issue and the sophistication/funding of the owning organization, it would be important to keep this in mind in rankings (while also caring for the safety of the end user).

Giving organizations incentives to work incrementally is an important way to encourage incremental improvements. However, any kind of ranking system could increase the risk that companies would aim to game the rankings by meeting specific criteria without actually improving safety for end users.

One possible solution is to include transparency in ranking criteria, by prescribing a set of information that should be in an annual report. Looking at both partial progress and transparency in an assessment might lead to greater change than an all-or-nothing approach.

Because large changes take time, several participants floated the idea of asking application developers or organizations to list the memory safety mitigations used by a piece of software as part of its feature set. That said, projects written entirely in memory-safe languages should not be penalized for not using exploit mitigations intended for C/C++, and developers should not be incentivized to apply exploit mitigations that are redundant, are inefficient, or don't work. A "nutrition label" approach could theoretically indicate what percentage of code is covered by, for example, safe languages, audits, fuzzing, sandboxing, low privilege, and so forth. This type of research could also lead to a "State of Memory Safety" report compiling information in various annual reports to see what we've learned industry-wide year over year. However, it's important to not get caught up in engineering details that are heavily context-dependent and not comparable across projects. Any nutrition-label approach should lead to good engineering results and emphasize real-world outcomes.

Large organizations like Apple or Microsoft could be pressured to put a plan in place and make it public, but we'd need to consider software dependencies as well as the amount of time it could take. With the proliferation of unsafe libraries to depend on, as well as compatibility concerns, it may be difficult to quickly evolve libraries and other parts of the ecosystem that grew organically over a long time. (That said, using memory-safe languages that are compatible with older C/C++ libraries is still a step forward.)

For high-risk targets, companies may get higher rankings for creating a "safe mode" limited to features that we know are memory-safe. Some companies already have some type of safe mode that, for example, hides unsafe attack surface from some attackers. Could companies create safe modes that incorporate memory safety to be used for security-sensitive cases, such as journalists, human rights activists, and so forth? The eventual ideal, though, is not to need a safe mode, because the normal mode would be safe.

Awareness and Advocacy

Often engineers put a lot of attention and urgency on memory safety, but it is not taken seriously as a top cybersecurity issue by business executives. Others in the organizations may be memory-safety enthusiasts getting pushback from product managers/designers. There are even language experts who might not quite grasp the full complexity of the issue. So far browser and operating systems have been most responsible for this type of work, and expanding outside of that may be important to create an industry-wide change that leads to safe adoption across all platforms and products.

As an industry, we need to draw more attention to memory safety.

We could learn from the lessons of MANRS (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. 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 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?

There could be multiple campaigns to raise awareness, one targeting the public and one targeting enterprise management.

To have the greatest impact, it's important to break out of the technical community as an issue, similar to encrypting the web or the importance of MFA, where awareness has moved beyond technical circles.

Consumer advocacy organizations and journalists can help educate consumers. Both groups can use their position in industry to push baseline expectations to encompass perceived risk due to technology choices. Journalists could make the quality of the software embedded in consumer products—including but not limited to memory safety—part of their reviews. (Unfortunately, analyzing exposed attack surface requires significant expertise and is extremely time-consuming, making this task more difficult.)

Consumer Reports uses the Digital Standard, which already has two areas that overlap with memory safety: checking to see whether products are built with effectively implemented safety features, and checking to see that the software does not make use of unsafe functions or

libraries. These standards allow the use of memory-unsafe languages with mitigations, but the mitigations do not prevent all vulnerabilities.

An advocacy approach might include encouraging or training tech journalists to call out memory safety bugs in more clear terms in phone reviews, though putting particular security bugs in the right context may require engineering expertise that most journalists don't have—and even the experts argue about the details.

Awareness-raising might also include op-eds in tech, government, or policy-focused publications, as well as storytelling narratives around efforts in other initiatives.

Recommendations

Industry

- As much as possible, companies, government organizations, and other entities should commit to using memory-safe languages for new products and tools and newly developed custom components. They should also support the development of open source memory-safe code.
- They should also identify and prioritize the most critical libraries and packages to shift to memory-safe languages, starting with the softest and most directly exposed attack surface, and to sandbox components that can be easily isolated. Special care should be taken for those that handle sensitive data and support vulnerable users—medical data, government data, and tools used by journalists, human rights activists, and other targeted or vulnerable groups. While it's important for everyone, everywhere to have memory-safe browsers and operating systems, special care should be taken to ensure that special-use tools intended for narrow and at-risk populations are memory-safe.
- Companies should be transparent about the causes of bugs, providing detailed information on security vulnerabilities to help researchers and industry experts ascertain which percentage of vulnerabilities are due to memory safety. This is particularly true for large breaches.
- Companies should voluntarily provide memory safety roadmaps to explain how they plan to eliminate memory-unsafe code in their products over time. Specifically, we have identified Microsoft and Apple as companies we hope will get on board due to their impact (and limited public statements).
- Companies should provide data on what percentage of jobs or projects use memory-safe languages, and which memory-safe languages are being taught to engineers internally.

Government & Advocacy

- There need to be regulatory or monetary incentives to transition legacy code to memory-safe languages, as well as policies to promote memory-safe code. Organizations with resources may consider providing funding to help create a "space race" for memory safety. Advocacy organizations and existing technical groups can also work on making memory safety easier by taking on tooling to make this easier, committing direct resources to ecosystem infrastructure and development, funding projects to sponsor new operating systems for existing Linux distros, or teaming up with companies and other organizations to have a cash prize for the top 20 libraries that can use memory-safe alternatives. There also needs to be an IOT sector-specific effort around memory safety. In some cases, this effort could eventually even extend to developers that are unaware of security bugs stemming from memory unsafe code, if the industry norm becomes a state where memory safety is incorporated by default.
- Advocates must work on raising awareness on memory safety outside of the tech community through a public pressure campaign on companies that place the burden of staying safe on enterprises and individuals who are least capable of doing so, and is less tolerant of manufacturers who choose unsafe build practices and defaults.

- Awareness-raising should target multiple groups, including enterprise management, journalists, and the public. Tactics could include op-eds in tech, government or policy-focused publications, and storytelling narratives around these efforts. Regardless of the audience, this awareness-raising should focus on all of the benefits of memory safety, not just in decreased future cybersecurity costs but also efficiency, speed, and so forth.
- Consumer advocacy organizations and journalists should push baseline expectations to encompass perceived risk due to technology choices. Advocacy organizations should train tech journalists on the dangers of memory unsafety so that this information is incorporated into phone and product reviews. Because memory safety is a long game, journalists or advocacy organizations might consider some kind of rating or ranking that would reward incremental improvements and transparency.
- Government, advocacy groups, and technical organizations can join forces to create industry development standards for safe code.
- Since many of the software projects that need to transition to memory-safe programming are open-source, advocates should explore possibilities for collaboration with organizations working to improve the sustainability of open source software development.

Case Studies

1. The Python cryptographic authority is one of the most widely used cryptography libraries in the Python ecosystem. Many of the tools are largely built on OpenSSL. The popular cryptography library is written in C. About two years ago, the maintainers started the process of migrating some of their dependence on OpenSSL away from that to their own Rust code, particularly starting with areas around certificate parsing and parsing of other structures. These are some of the most classical places to find memory safety vulnerabilities in C libraries, and they wanted to mitigate the risk that they were having by relying on OpenSSL.

   Another benefit was getting huge performance improvements, because the greater safety guarantees they were getting from the language allowed them to be more aggressive in doing things like not copying memory. Specifically, the safety guarantees of Rust mean that one can easily represent structures like X.509 certificates as an array of bytes, and then a parsed structure containing pointers into the original array. This is in contrast to OpenSSL's representation, where the parsed structure contains separate allocations for any references into the original bytes. Using Rust led to a 10x performance improvement over C, showing that using memory-safe languages can lead to valuable functionality. However, the launch was bumpy because people were initially upset that the new release had a dependency on a new programming language. They still have to do iterations to improve the build tool and error messages, and to offer pre-built versions of the library for more platforms to help people who need to update their own tool chains. But after the initial two weeks of frustration, things went much more smoothly. They incorporated Rust into a second library, moving the vast majority of their users to a new programming language, and did not have the same types of

complaints. This showed the maintainers that the ecosystem is now ready for new Python libraries that incorporate Rust.

2. Moving code from C to Rust (or any other language pair) is fundamentally hard because a machine does not understand the code the way a skilled programmer does. Relying on a skilled programmer to migrate code to a memory-safe language is too costly. Relying entirely on machine translation does not produce the desired result with current technology. Specifically, the output of a tool such as C2Rust can be compiled by the Rust compiler but it is not written the way a human writes Rust code. That means that the code is not sufficiently readable and does not make use of the language features that make Rust preferable to C with regard to security, maintainability, and productivity. There are ongoing efforts to shrink the gap between the output produced by a machine and a human while preserving the benefits of automation, but it appears likely that a hybrid model where the machine and programmer work together is the most practical model.

   There is some funding for moving code from C to Rust, but other problems remain. Creating C2Rust, an open source translator that moves code from C to Rust, led to some code migration challenges. It also led to the question of who maintains the code after it is removed, since some maintainers can't promise to be around as long as people in the open source community might use the work product, which is an obstacle to adoption. Some code may need to be maintained by organizations that don't know Rust and have little time to maintain C.

3. The Linux kernel now includes initial support for Rust. This will enable writing kernel modules such as drivers in a memory-safe language, which is important because drivers in the Linux kernel represent the majority of the code and run at a privileged level. Getting there required supporting the open source project with a few full- or part-time engineers in order to give the project the necessary initial momentum. Some companies released public statements of support too, which proved very useful. Communication with the Rust community and teams was also important to get some needed features into the toolchain and language. The next milestone for Rust in the kernel is getting the first drivers upstreamed, which several major companies are looking forward to.

4. Two years ago, Mozilla began the process of migrating third-party libraries bundled with the Firefox browser to WebAssembly. This effort complements their efforts to rewrite parts of Firefox in Rust by making it possible to retrofit coarse-grain memory safety for existing memory-unsafe C code, using WebAssembly as an intermediate compilation step. In roughly two years, the team ported five libraries to WebAssembly. Getting there required developing a new framework called RLBox that makes it possible to incrementally and securely change Firefox to use the WebAssembly libraries without imposing noticeable overheads.

   Over the last year alone the benefits of WebAssembly became even clearer: the Expat XML parsing library had over fourteen CVEs disclosed since Firefox shipped a WebAssembly-safe version of it, and these CVEs did not impact the security of Firefox nor the huge engineering burden of managing and applying emergency patches to

hundreds of millions of users. The next milestone for this project is to improve the performance and usability of WebAssembly toolchains and RLBox framework itself, and work with other companies and teams to deploy WebAssembly.

Josh Aas, Georgia Bullen, Amira Dhalla, Alex Gaynor, Joseph Lorenzo Hall, Per Larsen, Miguel Ojeda, Christine Runnegar, Deian Stefan, and Zack Weinberg contributed to this report.

Resources, Articles, and Research Related to Memory Safety

General

"Buying Down Risk: Memory Safety" (Atlantic Council)

"Demystifying Magic: High-Level Low-Level Programming" (PDF) (Eliot Moss)

"Fantastic Memory Issues and How to Fix Them" (Educated Guesswork)

"Introduction to Memory Unsafety for VPs of Engineering" (Alex Gaynor)

"The ISRG Wants to Make the Linux Kernel Memory-Safe With Rust" (Ars Technica)

"Linus Torvalds: Rust Will Go Into Linux 6.1" (ZDNet)

"Memory Safe Languages in Android 13" (Google Online Security Blog)

"Memory Safety for the Internet's Most Critical Infrastructure" (Prossimo)

"MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code" (Arxiv)

"Practical Third-Party Library Sandboxing With RLBox" (RLBox)

"Previewing Rust on Azure Sphere" (Microsoft Community Hub)

"Prioritizing Memory Safety Migrations" (Chris Palmer)

"Rust in the Linux Kernel: Just the Beginning" (Prossimo)

"Safety Features" (Cyber ITL). Metrics including memory hardening techniques.

"Taxonomy Of In-The-Wild Exploitation" (Chris Palmer)

"Towards the Next Generation of XNU Memory Safety: kalloc_type" (Apple Security Research blog)

"An Update on Memory Safety in Chrome" (Google Security blog)

"WebAssembly and Back Again: Fine-Grained Sandboxing in Firefox 95" (Mozilla Hacks)

Reports/Convenings

"National Security Agency | Cybersecurity Information Sheet: Software Memory Safety" (PDF, National Security Agency, November 2022)

"NSA Releases Guidance on How to Protect Against Software Memory Safety Issues" (National Security Agency, November 2022)

"Position Paper: Progressive Memory Safety for WebAssembly" (2019)

"Recommendations From the Workshop on Open-Source Software Security Initiative" (September 2022)

"Retrofitting Fine Grain Isolation in the Firefox Renderer" (Usenix, 2020) - Presentation

The Road to Less Trusted Code: Lowering the Barrier to In-Process Sandboxing (Usenix, Winter 2020)

U.S. Open-Source Software Security Initiative Workshop (August 2022)

Related Reading

"Crashworthy Code" (Bryan H. Choi)

"REPORT: Strict Product Liability and the Internet of Things" (Center for Democracy and Technology)