The Case for Memory Safe Roadmaps

Why Both C-Suite Executives and Technical Experts Need to Take Memory Safe Coding Seriously

Publication: December 2023

United States Cybersecurity and Infrastructure Security Agency
United States National Security Agency
United States Federal Bureau of Investigation
Australian Signals Directorate's Australian Cyber Security Centre
Canadian Centre for Cyber Security
United Kingdom National Cyber Security Centre
New Zealand National Cyber Security Centre
Computer Emergency Response Team New Zealand

Executive Summary

Memory safety vulnerabilities are the most prevalent type of disclosed software vulnerability.[1,2,3] They are a class of well-known and common coding errors that malicious actors routinely exploit. These vulnerabilities represent a major problem for the software industry as they cause manufacturers to continually release security updates and their customers to continually patch. These vulnerabilities persist despite software manufacturers historically expending significant resources attempting to reduce their prevalence and impact through various methods, including analyzing, patching, publishing new code and investing in training programs for developers. Customer organizations expend significant resources responding to these vulnerabilities through onerous patch management programs and incident response activities.

Memory safe programming languages (MSLs) can eliminate memory safety vulnerabilities. Therefore, transitioning to MSLs would likely greatly lessen the need to invest in activities aimed at reducing these vulnerabilities or minimizing their impact. Additionally, investments to migrate unsafe codebases to MSLs would pay long-term dividends in the form of safer products—defraying some of the upfront cost of transitioning to MSLs.

The U.S. Cybersecurity and Infrastructure Security Agency (CISA), National Security Agency (NSA), Federal Bureau of Investigation (FBI), and the cybersecurity authorities of Australia, Canada, the United Kingdom, and New Zealand* (hereafter referred to as the authoring agencies) jointly developed this guidance as part of our collective Secure by Design campaign. With this guidance, the authoring agencies urge senior executives at every software manufacturer to reduce customer risk by prioritizing design and development practices that implement MSLs. Additionally, the agencies urge software manufacturers to create and publish memory safe roadmaps that detail how they will eliminate memory safety vulnerabilities in their products. By publishing memory safe roadmaps, manufacturers will signal to customers that they are taking ownership of security outcomes, embracing radical transparency, and taking a top-down approach to developing secure products—key Secure by Design tenets.

The Secure by Design campaign urges technology providers to take ownership of their customers' security outcomes by building cybersecurity into design and development. See Secure by Design, Secure Your Products, and Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Security-by-Design and -Default.

This guidance provides manufacturers with steps to create memory safe roadmaps and implement changes to eliminate memory safety vulnerabilities from their products. Eliminating this vulnerability class should be seen as a business imperative likely requiring participation from many departments. The authoring agencies urge executives to lead from the top by publicly identifying senior staff who will drive publication of their roadmap and assist with realigning resources as needed.

* The Australian Signals Directorate's Australian Cyber Security Centre (ASD's ACSC), Canadian Centre for Cyber Security (CCCS), United Kingdom's National Cyber Security Centre (NCSC-UK), and New Zealand's National Cyber Security Centre (NCSC-NZ) and Computer Emergency Response Team New Zealand (CERT NZ).

Table of Contents

Introduction 4
Mitigations 4
Mitigations to Reduce Prevalence 5
Developer Training 5
Code Coverage 5
Secure Coding Guidelines 5
Fuzzing 5
SAST/DAST 6
Safer Language Subsets 6
Mitigations to Reduce Impact 6
Non-Executable Memory 6
Control Flow Integrity 6
Address Space Layout Randomization (ASLR) 7
Other Compiler Mitigations 7
Sandboxing 7
Hardening Memory Allocators 7
Potential Future Mitigations: Using Hardware 8
The Case for Memory Safe Languages 9
Planning the Transition to Memory Safe Languages 10
Considerations 10
Prioritization Guidance 10
Picking a Memory Safe Language 11
Staff Capabilities and Resourcing 12
Implementation Challenges 12
Shifting Security Left 12
Performance Impact 13
Existing Memory Unsafe Libraries 13
Memory Protection Exceptions 13
Bringing Computer Science Education Up to Speed 14
OT, Low Power, and IoT Systems 14
Memory Safe Roadmaps 14
Conclusion 16
Resources 18
Appendix: Memory Safe Languages 19
Purpose 20
Acknowledgements 20
Disclaimer 20
Contact Information 20
References 21

Introduction

Memory safety vulnerabilities [CWE-1399: Comprehensive Categorization: Memory Safety] are a class of vulnerability affecting how memory can be accessed, written, allocated, or deallocated in unintended ways in programming languages.[4,5,6]

The concept underlying these errors can be understood by the metaphor of the software being able to ask for item number 11 or item number -1 from a list of only 10 items. Unless the developer or language prevents these types of requests, the system might return data from some other list of items.

Depending on the type of vulnerability, a malicious actor may be able to illicitly access data, corrupt data, or run arbitrary malicious code. For example, a malicious actor may send a carefully crafted payload to an application that corrupts the application's memory, then causing it to run malware. Alternatively, a malicious actor may send a malformed image file that includes malware to create an interactive shell on the victim system. If an actor can execute arbitrary code in this way, the actor may gain control of the account running the software.

Modern industry reporting indicates that defects first identified several decades ago remain common vulnerabilities exploited by malicious actors today to routinely compromise applications and systems.[7] Yet, according to modern industry reporting, these vulnerabilities remain common, and malicious actors routinely exploit them to compromise applications and systems:

- About 70 percent of Microsoft common vulnerabilities and exposures (CVEs) are memory safety vulnerabilities (based on 2006-2018 CVEs).[8]
- About 70 percent of vulnerabilities identified in Google's Chromium project are memory safety vulnerabilities.[9]
- In an analysis of Mozilla vulnerabilities, 32 of 34 critical/high bugs were memory safety vulnerabilities.[10]
- Based on analysis by Google's Project Zero team, 67 percent of zero-day vulnerabilities in 2021 were memory safety vulnerabilities.[11]

Mitigations

Over the past few decades, software developers have continually sought to address the prevalence and impact of memory safety vulnerabilities within their software development life cycle (SDLC) through the following mitigation methods. Despite these continued efforts, memory safety has remained a leading cause of disclosed vulnerabilities in software products. Nevertheless, these mitigations remain valuable, especially when used in combination, to protect code that has not yet, or cannot be, transitioned to MSLs.

Mitigations to Reduce Prevalence

Developer Training

Programming languages such as C and C++ are examples of memory unsafe programming languages that can lead to memory unsafe code and are still among the most widely used languages today. In attempts to mitigate the dangers of memory unsafe code in C and C++, many software manufacturers invest in training programs for their developers. Many of these training programs include tactics designed to reduce the prevalence of memory unsafe vulnerabilities produced by those languages. Additionally, there are numerous commercial and industry trade association training programs. Further, various organizations and universities offer trainings and a professional certificate for demonstrating knowledge of secure coding practices in C and C++.

While training can reduce the number of vulnerabilities a coder might introduce, given how pervasive memory safety defects are, it is almost inevitable that memory safety vulnerabilities will still occur. Even the most experienced developers write bugs that can introduce significant vulnerabilities. Training should be a bridge while an organization implements more robust technical controls, such as memory safe languages.

Code Coverage

Code coverage is the process of covering as much of the codebase as possible with unit and integration tests. Industry practices encourage development teams to strive for 80% coverage and greater, but this is not always achievable with time and resource constraints. Development teams aim to cover all critical and security-sensitive areas of an application with both positive and negative test cases. Teams can easily add these types of tests to an automation pipeline or script for repeatability and regression testing. The benefits lay in ensuring that no new vulnerabilities are added to functionality that has previously been tested but may have unintentionally been changed as part of an update and was therefore not in a release test plan.

Secure Coding Guidelines

Organizations and industry have developed many secure coding guidelines for most prevalent programming languages. These guidelines outline the areas where developers need to take more care due to language-specific traps, especially around memory handling. Organizations have attempted to ensure that development teams are not only using a secure coding guide for the programming language of choice but are actively updating the guide as the team identifies new issues or standardizes an approach to a common problem.

Fuzzing

Fuzzing tests software by sending it a wide variety of data, including invalid or random data, and detects when the test data causes the application to crash or fail code assertions.[12] Fuzzing is a common method for finding errors like buffer overflows. Fuzzing can aid in discovering vulnerabilities, but no tool can find every vulnerability. Since fuzzing is a non-deterministic tactic applied after the initial coding mistakes are made, there will be limits to how effective it can be. New fuzzing methods are continually created that find previously undiscovered vulnerabilities. Software manufacturers should ensure their fuzz testing strategies are continually updated.

SAST/DAST

Developers use Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) tools to find a variety of software vulnerabilities, including memory-related bugs.[13] SAST tools look at static resources, specifically source code or binaries, and DAST tools examine a running system (or the unit test suite, which can be similarly effective) to find problems that would be hard to detect by a SAST tool. Many organizations use both types of tools. Some larger organizations use more than one SAST or DAST tool from different vendors to provide additional coverage using a wider range of approaches.

Depending on the codebase, SAST tools and, to a lesser extent, DAST tools can generate a significant number of false positives, creating a burden for software developers. Furthermore, no SAST or DAST tool can catch every vulnerability.

Safer Language Subsets

The C++ community has been contemplating[14] the balance between backwards compatibility, memory-safety defaults, and other priorities for the base language.[15] There are multiple targeted efforts to make C and C++ less vulnerable for existing code bases and products. For example, Apple has modified the C compiler toolchain used in the iBoot system[16] to mitigate memory and type safety issues. External analysis[17] indicates that there may be non-trivial performance and memory usage costs. Microsoft has developed "Checked C" that "adds static and dynamic checking to C to detect or prevent common programming errors such as buffer overruns and out-of-bounds memory accesses."[18] There are more general efforts to improve C++ memory safety for existing code,[19] including efforts like Carbon.[20,21]

Mitigations to Reduce Impact

Non-Executable Memory

Most modern computer architectures do not contain separate memory for data and code, which allows malicious actors who exploit memory safety issues to introduce code as data that the processor could then be coerced into executing. An early attempted mitigation for memory safety issues was to mark some memory segments as non-executable. In such cases, a CPU would not execute instructions contained within such pages, as they were only intended for storing data, not code. Unfortunately, more sophisticated techniques have emerged, such as return oriented programming (ROP), which enables existing code segments within a program to be repurposed to execute on adversary-controlled data to subvert control of a program.[22]

Control Flow Integrity

Control Flow Integrity (CFI) technology identifies all indirect branches and adds a check on each branch.[23] At runtime, the program will detect invalid branches, causing the operating system to terminate the process. Despite some successes, numerous bypasses to CFI have been discovered,[24] including ones that have been exploited in the wild.[25]

Address Space Layout Randomization (ASLR)

Traditionally, malicious cyber actors who find a memory vulnerability will craft a payload to exploit that vulnerability and attempt to find a way to execute their code. Finding the exact memory layout to execute their code may require some experimentation. However, when they find it, the exploit will work on any instance of the application.

ASLR is a technique in which the runtime system moves various components, such as the stack and heap, to different virtual addresses every time the program runs. ASLR aims to ensure that malicious cyber actors do not have knowledge of how memory is organized, which makes it substantially harder to exploit the vulnerability. However, ASLR bypasses are common because programs can be coaxed into leaking memory addresses,[26,27,28,29] which means that ASLR does not entirely prevent exploitation of memory safety vulnerabilities.

Other Compiler Mitigations

Modern compilers include various mitigations against exploitation of memory safety issues. Techniques such as stack canaries and non-writable stacks leverage different approaches to mitigating some memory safety issues. However, actors have also identified techniques on the exploit side to bypass these mitigations, such as identifying data leaks and ROP.

Sandboxing

Developer teams can use sandboxing to isolate different parts of a system to limit the scope of any potential vulnerability. Developers will break the application into subsystems and restrict the resources they can use, including memory, network access, and process control. Sandboxing provides a layer of protection for many classes of vulnerability, even going back to chroot to prevent file system traversals.

A subsystem that handles untrustworthy data, such as network communications or user-generated content, may be a good candidate to isolate from other parts of the system using a sandbox. If malicious actors find a memory-related vulnerability in one subsystem, they are faced with the additional task of breaking out of the sandbox. Forcing adversaries to find multiple new defects raises the cost of attack.

Despite the value sandboxing brings, there are limits to how far developers can push this model. The more sandboxes they use, the more complex the code becomes. Further, there are practical limits to how many sandboxes a system can tolerate, especially on constrained devices, such as phones. Additionally, sandbox bypasses, also known as sandbox escapes, are often discovered, defeating security protections. Google's presentation on sandboxing[30] in the Android operating system demonstrates the limits associated with this mitigation tactic.

Hardening Memory Allocators

As is the case of ASLR and compiler mitigations, hardening allocators make creating a reliable exploit for a vulnerability more difficult, but it does not remove the memory safety vulnerability. For example, Apple reported their allocator, called “kalloc_type,” “…makes exploiting most memory corruption vulnerabilities inherently unreliable.” There are also commercial memory safe allocators that target specific domains, like OT devices.

Potential Future Mitigations: Using Hardware

A promising area under active development involves using hardware to support memory protections. The Capability Hardware Enhanced RISC Instructions (CHERI)[31] project is a joint research project of SRI International and the University of Cambridge that adds new features to existing chip architectures. The UK government’s Digital Security by Design (DSBD) program brought together £70m of government funding with £117m of industry co-investment to develop the technology further.[32] In addition to a range of academic and industry-supported research and development activities, the program enabled Arm to ship its CHERI-enabled Morello prototype processor, system-on-chip (SoC), and board in January 2022. Both Arm and Microsoft have documented their CHERI efforts and a range of other activities supported by DSBD. There is now a community of developers building tools and libraries to enable widespread adoption of the technology.

CHERI can be deployed on architectures other than Arm; DSBD also recently announced £1.2m of investment in a demonstrator project using the RISC-V architecture.[33] The aim is to show how the technology can beneficially be deployed in automotive systems, in which safety is critical.

Arm introduced another technology called the Memory Tagging Extension (MTE) to some of its CPU product lines to detect use after free and out-of-bounds (also called buffer overflow) type bugs.[34] When memory is allocated, the system assigns it a tag. All further access to that memory must be made with that tag. The CPU will raise an error if the tags do not match. Arm estimates that the overhead for MTE is between 1–2 percent. Mobile devices may see the first widespread deployments.[35] Intel also announced memory tagging capabilities in future chipsets. [36]

Other hardware-based research includes FineIBT, which includes Control Flow Integrity (CFI) support on top of Intel's hardware-based Control-flow Enforcement Technology (CET).[37]

Some hardware features like MTE are going to be needed even in systems written in MSLs. For example, Rust programmers can mark some code as “unsafe,” benefitting from hardware controls.

Although memory protections in hardware are not yet widely available, some industry observers believe they will be helpful in many deployment scenarios where migration to MSLs will take an extended amount of time. In such scenarios, hardware refresh cycles may be short enough to provide important memory protections for customers until other protections are available. Experiments with these hardware protections are underway, including work to measure the real-world performance impact and memory consumption characteristics of these new designs. In some cases, it is possible that hardware protections will enable increased performance if used optimally by the software.

The Case for Memory Safe Languages

Despite software manufacturers investing vast resources attempting to mitigate memory safety vulnerabilities, they remain pervasive. Customers must then expend significant resources responding to these vulnerabilities through both onerous patch management programs and incident response activities.

As previously noted by NSA in the Software Memory Safety Cybersecurity Information Sheet and other publications,[38] the most promising mitigation is for software manufacturers to use a memory safe programming language because it is a coding language not susceptible to memory safety vulnerabilities. However, memory unsafe programming languages, such as C and C++, are among the most common programming languages.[39] Internet applications and devices throughout the technology landscape use memory unsafe programming languages. These languages run operating systems, resource-constrained systems, and applications that require high-performance. The pervasiveness of memory unsafe languages means that there is currently significant risk in the most critical computing functions.

At the same time, the authoring agencies acknowledge the commercial reality that transitioning to MSLs will involve significant investments and executive attention. Further, any such transition will take careful planning over a period of years. Although there is an upfront cost in migrating codebases to MSLs, these investments will improve product reliability, quality, and—critically—customer security.

Software manufacturers will benefit from using MSLs and their customers will benefit from more secure products. Benefits for developers and customers may include:

- **Increased reliability**. MSLs create more reliable code than memory unsafe programming languages. For example, an operating system that uses a memory unsafe programming language can only crash the application if it detects a memory violation. If the application is a client server process, it could drop the connections of all connected users. Additionally, the operating system itself may crash if the memory corruption happens in the kernel. MSLs, on the other hand, prohibit memory violations from occurring.
- **Fewer interruptions for developers**. When someone reports a memory safety bug, developers go into reactive mode and must stop other work to diagnose and mitigate the problem. Fewer memory safety defects may mean fewer unplanned and often urgent responses to vulnerability discoveries. Development teams have reported that memory safety bugs are some of the most challenging to diagnose and correctly address. Consequently, they often must pull their most senior developers from other important work to find and correctly fix these defects. Transitioning to MSLs would free developers to focus on current work priorities instead of reacting to newly discovered vulnerabilities.
- **Fewer emergencies for supporting staff**. Fewer memory safety vulnerabilities may result in fewer emergency releases, saving time of teams like Build, Quality Assurance, Product Management, and Support.
- **Fewer emergencies (and breaches) for customers**. Removing the memory safety class of vulnerability from a product by transitioning to an MSL eliminates the need

for memory issue security releases. This will reduce the number of urgent product releases that a customer will need to accommodate, saving time and averting breaches.

In addition to bringing benefits to software manufacturers and their customers, MSLs reduce a product's attack surface. That reduction in attack surface will increase the cost to malicious actors who then need to invest more resources discovering other exploitable vulnerabilities. As is the case with any mitigation, transitioning to MSLs will not, by itself, halt or deter cybercrime or espionage. Malicious cyber actor economics will dictate where the actor looks for the next intrusion vector to accomplish their mission. Yet, given the current pervasiveness of memory safety vulnerabilities and exploitations, reducing and eventually eliminating that attack path will significantly raise the cost of an attack.

Planning the Transition to Memory Safe Languages

Considerations

Software manufacturers should consider the following technical and non-technical factors when developing their roadmap:

1. **Prioritization guidance.** Manufacturers should consider how to prioritize migration to MSLs through the development of roadmaps and specific guidance for development and technical teams.
2. **Picking use-case appropriate MSLs.** There are numerous MSLs, and each one has its own set of tradeoffs in terms of architecture, tooling, performance, popularity, cost, and other factors. No one MSL is right for all programming needs. Manufacturers should look at use cases that use memory unsafe languages and pick the most appropriate MSL for each. When selecting an MSL, software manufactures should follow standard risk management processes, as MSLs are not free from other potential vulnerabilities of critical severity. As part of their risk management program, manufacturers should closely follow supply chain and secure development lifecycle practices as defined in the National Institute of Standards and Technology (NIST) Risk Management Framework and Secure Software Development Framework (SSDF).
3. **Staff capabilities and resourcing.** Manufacturers should consider how they will train developers in a selected MSL, how they can prioritize hiring developers with the relevant skills, and what resources they may need to support the selected language.

Prioritization Guidance

Although there is no one way to prioritize a transition to MSLs, the following list of options will assist development teams in picking appropriately sized migration projects that will provide them experience, a tight feedback loop, and a manageable amount of risk.

- **Start with new and smaller projects**. Re-writing existing code in an MSL can be a significant challenge, especially if the code is already performing well and the organization does not already have expertise in the chosen MSL. Consider starting with new and smaller projects that carry lower risk to give teams time to experiment with new tools and processes.

- **Replace memory unsafe components**. Consider taking a self-contained component written in a memory unsafe programming language and rewriting it in your chosen MSL. Consider ways to run the existing component in parallel with the newly updated MSL component and compare the results. Once the updated MSL component consistently produces the same output as the older component, it should be possible to retire the older component.
- **Prioritize security critical code**. Parts of your codebase may be sensitive from a security perspective or along a critical attack path.[40] Examples include code that performs operations on user-generated content, which is a notorious vector for abuse. Other examples include code that handles secret keys, opens network connections, performs authentication and authorization, or operates at low levels, such as firmware.[41] Prioritize security-sensitive code when the team has the required MSL expertise. Review the use of cryptographic components and other "roots of trust" on which the system is built during the prioritization process.
- **Use instrumentation**. For example, the GWP-ASan allocation tool in the Android operating system can detect heap memory errors that fuzzing does not detect.
- **Evaluate performance and complexity**. There are often reasons to re-write components or larger systems to incorporate new requirements not anticipated by the original design. The existing implementation of a system may have become brittle, and the development team may therefore find it hard to support the evolution needed to meet new requirements. If the team is going to rewrite the system for any reason, they should consider breaking the requirements into smaller pieces and writing parts or all of it in the organization's chosen MSL.
- **Determine which modules are CPU-bound**. Some applications are CPU-bound, which means CPU speed limits the performance of the overall system. Specifically, in the case of garbage collecting MSLs, a CPU-bound application may experience more performance fluctuations than one that is limited by things like human response time, network latency, or disk I/O.
- **Ramp up parallel systems**. If a development team ports a highly parallel application to an MSL, they should direct a small portion of the workflow to the new codebase and monitor the results. Once there is confidence that the system is performing correctly, the team should increase load-in increments until they completely phase out the old system.
- **Wrap applications**. Where a team cannot update an existing memory unsafe application to an MSL, they should write an intermediary application for all public interfaces in the MSL. The wrapping application will need to ensure all inputs cannot exceed memory bounds within the child application.

Picking a Memory Safe Language

The authoring agencies recommend software manufacturers evaluate multiple MSLs before integrating them into their programs of work. See the appendix for an overview of some memory safe languages.

Staff Capabilities and Resourcing

Software manufacturers should consider:

- **Planning time for learning.** Ensure that teams have access to training and learning material they may need. Provide teams dedicated learning time for both self-study and to learn from senior team members. Give new hires example problems previously solved; include both stand-alone coding exercises and debugging challenges in the MSL.
- **Planning time for integration**. Create a strategy for integrating new staff into existing teams and resolving potential conflicts, e.g., between new team members who are familiar with the MSL and senior members more familiar with the existing languages.
- **Establishing both internal and external communities**. Establish a group of internal champions of MSLs. In addition to assigning them migration and development on their normal projects, engineering leadership should give them time to work with and understand the efforts, challenges, and successes of other teams. These champions can build an internal community and cross-pollinate information across projects by hosting internal meetups, training sessions, and chat rooms. In addition to building an internal community, they can help connect their organization to external experts by setting up both virtual and in-person meetups.
- **Hiring and onboarding**. It will not always be practical for organizations to hire developers who are already MSL experts. Rather than waiting to find a developer who has the perfect set of skills, experience, and knowledge, some organizations report success in hiring developers who have proficiency in multiple programming languages, often including C and C++. To compensate for the new hire's lack of experience in the MSL of choice, organizations can modify their existing developer onboarding process to include a buddy/mentor system and a boot camp in the MSL.
- **Creating a staffing pipeline**. Organizations should signal their demand for developers trained in security and memory safety to colleges, universities, and educational institutions. Many institutions base their computer science and software engineering curricula on the demand they receive from students expecting to receive offers from employers. If there is no expectation that students will need a specific skill, institutions will not expend resources to provide it.

Implementation Challenges

There are several challenges that software manufacturers need to be aware of as they develop and begin to implement their memory safe roadmap, including the following examples.

Shifting Security Left

The authoring agencies encourage software manufacturers to move security considerations earlier in the SDLC, which can improve product security and reliability upon deployment. Depending on the programming language in question, there may be an increase in the amount of up-front work. Some MSLs require that the development team access memory in particular ways, which may require more time for the team to become proficient in creating idiomatically correct code. Although this extra effort may seem burdensome, mitigating the memory safety vulnerabilities benefits the development team and customer.

It is worth considering the benefits of migrating from an environment where the code manages memory properly most of the time to one where the programming language ensures that the code will manage memory properly all the time.

Performance Impact

Some MSLs use a garbage collector as part of their memory management. The process of garbage collecting can introduce unpredictable latency that affects the application's overall performance characteristics—although, some languages can use additional threads to clean up and free memory. The garbage collector will also introduce some additional overhead in terms of CPU and memory. While the impact of garbage collection on 5th generation languages and modern hardware is negligible, these performance characteristics can still affect constrained devices, such as those in embedded systems. Developers will need to pay attention to these performance characteristics, especially in environments that demand real-time performance and high scalability.

Existing Memory Unsafe Libraries

MSLs, whether they use a garbage collection model or not, will almost certainly need to rely on libraries written in languages such as C and C++. Although there are efforts to re-write widely used libraries in MSLs, no such effort will be able to re-write them all anytime soon.

For the foreseeable future, most developers will need to work in a hybrid model of safe and unsafe programming languages. Developers who start writing in an MSL will need to call C and C++ libraries that are not memory safe. Likewise, there are going to be situations where a memory unsafe application needs to call into a memory safe library. When calling a memory unsafe component or application, the calling application needs to be explicitly aware of—and limit any input passed to—the defined memory bounds.

The memory safety guarantees offered by MSLs are going to be qualified when data flows across these boundaries. Other potential challenges include differences in data marshalling, error handling, concurrency, debugging, and versioning.

This class of challenge is common and well-studied. Tools and techniques to manage the interaction between languages are available and it is likely to be an area for future innovation and development.

Memory Protection Exceptions

It is also worth noting that it is possible to defeat the memory protection guarantees in some MSLs. For example, software written in Rust can contain the "unsafe" keyword.[42] There are several important reasons to use this keyword, such as directly interacting with the operating system. However, developers should not use this keyword to avoid introducing memory safety vulnerabilities.

Bringing Computer Science Education Up to Speed

Executive-level leadership should drive the transition to memory safe programming languages because memory unsafety is fundamentally a business strategy problem. As such, the CEO or other business executive should sign the roadmap.

Many, perhaps most, computer science degree programs do not teach students in depth about the dangers of memory unsafe programming languages and the real-world harms that come as a result. The reasons for that fact are numerous. For example, hiring managers at many software manufacturers need developers who can work in existing environments, many of which include large C and C++ codebases. They may have a short-term interest in hiring students who are proficient in the languages they predominantly use. Yet, by addressing the short-term needs of software manufacturers, it has become hard to create momentum around memory safe code.

Another reason for a lack of emphasis on MSLs in universities may stem from the incentive structures that shape how professors spend their time. The cost to master new programming languages and update coursework is considerable, and few professors have spare time to make that investment in addition to fulfilling their other obligations. This area is ripe for additional research and exploration of possible options to provide professors with the right tools to begin teaching MSLs.

It is worth noting that languages like C and C++ can help students understand how the computer works at lower levels, a useful skill when these students need to think deeply about performance and scalability of complex systems. A thoughtful balance of coursework on language choice and exercises on memory safety can produce a well-rounded graduate.

There are many ways that people learn how to write software outside of a university setting. Many people teach themselves to program by reading a book, taking an online course, reading blogs, or watching videos. Some start by modifying someone else's extensions to their favorite video game or browser, and then learning to write their own. MSL advocates in the industry should think about how to implement a bias toward MSLs in these common ways people start to program.

OT, Low Power, and IoT Systems

Although many MSLs are widely available for desktop, server, and mobile platforms, they are less available on constrained systems, where memory, CPU, and network connections are severely limited. OT systems often prioritize availability and reliability over many other considerations and lack the wide spectrum of programming languages found on more powerful systems. Any transition to an MSL in OT systems will require a demonstration of performance, reliability, and real-time guarantees. Further, that transition will require the same type of tooling available to less constrained platforms so developers can build, test, and debug their systems.

Memory Safe Roadmaps

The authoring agencies strongly encourage software manufacturers to write and publish memory safe roadmaps. By doing so, manufacturers will signal to customers that they are embracing key secure by design principles of (1) taking ownership of their security outcomes, (2) adopting radical transparency, and (3) taking a top-down approach to developing secure products.

Software developers and support staff should develop the roadmap, which should detail how the manufacturer will modify their SDLC to dramatically reduce and eventually eliminate memory unsafe code in their products. To ensure adequate resourcing and signal executive-level support to customers, the authoring agencies strongly urge executives to publicly identify senior staff to drive publication of the roadmap and assist with realigning resources as needed.

The roadmap should include the following elements:

1. **Defined phases with dates and outcomes**. As with all software development efforts, development teams can break the larger effort into smaller projects with clear outcomes to measure their progress. Phases might include:
    a. Evaluation of MSLs.
    b. A pilot to test writing a new component in an MSL or incorporating an MSL into an existing component.
    c. Threat modeling to find the most dangerous memory unsafe code.
    d. Refactoring memory unsafe code.
2. **Date for MSLs in new systems**. Publish the date after which the company will write new code solely in an MSL. Organizations can put a cap on the number of potential memory safety vulnerabilities by writing new projects in an MSL. Publicly setting a date for that change will demonstrate a commitment to customer security.
3. **Internal developer training and integration plan**. No MSL transition will be free, and the manufacturer will need to set aside time for developers to become proficient at:
    a. Writing software in the selected language.
    b. Debugging.
    c. Tooling.
    d. Integrating the MSL into the builds.
    e. Overall quality control processes.
4. **External dependency plan**. The roadmap should document the plan to handle dependencies on libraries written in C and C++. Most software products are based on numerous open source software (OSS) libraries, and many of those are written in C and C++. A memory safe roadmap will not be complete without including OSS, especially since most existing products use OSS.
5. **Transparency plan**. Keeping the above information current with regular, e.g., perhaps quarterly or semi-annual, updates will further build confidence that the organization is taking memory safety vulnerabilities seriously. Additionally, publishing a detailed analysis of wins and challenges—especially SDLC improvements—can inspire others to begin their memory safety journey.
6. **CVE support program plan**. The industry needs detailed and correct public data on the classes of vulnerability that create risks for customers. It needs vulnerability descriptions that provide enough details about the coding errors to distinguish between C and C++ memory safety defects and other classes of defect. To that end,

organizations should publicly commit to supplying CWEs for 100 percent of CVEs in a timely manner as well as any additional context to help the industry understand the defect. While some vendors do this well today, there are notable players who do not, limiting the insights the industry can learn from vulnerability data.

Finally, the authoring agencies strongly encourage organizations that use a maturity model to evolve and improve their SDLC over time to integrate their memory safety efforts to demonstrate a higher level of maturity.

Conclusion

Memory unsafe code is a major problem for software manufacturers and their customers. Previous attempts at solving the problem have made only partial gains, and today, two-thirds of reported vulnerabilities in memory unsafe programming languages still relate to memory issues. The most promising path towards eliminating memory safety vulnerabilities is for software manufacturers to find ways to standardize on memory safe programming languages, and to migrate security critical software components to a memory safe programming language for existing codebases.

The authoring agencies urge executives of software manufacturers to prioritize using MSLs in their products and to demonstrate that commitment by writing and publishing memory safe roadmaps. The authoring agencies encourage software manufacturers to lead from the top by publicly naming a business executive who will personally drive the elimination of memory safety vulnerabilities from the product line.

By publishing memory safe roadmaps, software manufacturers will signal to customers and industry that they are aligned to the secure by design principles of:

- **Taking ownership of the security outcomes of their customers**. Given the prevalence of memory safety vulnerabilities in the software market, eliminating this class of vulnerability will improve the customers' security postures.
- **Radical transparency**. The roadmap will be a public plan detailing the approach the manufacturer plans to adopt to eliminate this class of vulnerability from their product lines. The balance between short-term mitigations to reduce the dangers of memory unsafe programming languages, MSL transitions, and hardware research will vary widely among manufacturers. However, regardless of approach, the goal should be the same: To set a public timeline with clear milestones to demonstrate to customers that they are making urgent investments to solve this problem.
- **Lead from the top**. Software manufacturers who publish their roadmap and publicly name a business leader to support the efforts are demonstrating they are leading from the top and that these important initiatives are not afterthoughts merely delegated to lower-level staff.

When software manufacturers run into stumbling blocks, they should articulate those challenges and suggest potential solutions as part of the community of interest that seeks to solve this class of coding errors. By working on these challenges together, the software industry can identify and promote solutions that no one organization can accomplish on its own. This is an industry-wide problem and solving it will require a whole-of-industry response.

Regardless of approach, the authoring agencies urge organizations to act immediately to reduce, and eventually eliminate, memory safety vulnerabilities from their products.

Resources

The following is a non-exhaustive list of publicly available resources on memory safety and memory safe programming languages:

- CISA Blog: The Urgent Need for Memory Safety in Software Products
- Open Source Security Foundation (Linux Foundation) https://openssf.org/
- Internet Security Research Group (abetterinternet.org) https://www.abetterinternet.org/
- Chris Palmer presents Google's efforts to prevent memory safety vulnerabilities in the Android Operating System. https://www.usenix.org/conference/enigma2021/presentation/palmer
- Alex Gaynor presents common reactions to memory safe programming languages https://www.usenix.org/conference/enigma2021/presentation/gaynor
- Introduction to Memory Unsafety for VPs of Engineering https://alexgaynor.net/2019/aug/12/introduction-to-memory-unsafety-for-vps-of-engineering/
- Prossimo, an Internet Security Research Group (ISRG) project: https://www.memorysafety.org/docs/memory-safety/
- Atlantic Council, Buying down risk: Memory safety: https://www.atlanticcouncil.org/content-series/buying-down-risk/memory-safety/
- "Retain Cycles and Memory Management in Swift" by İsmail GÖK which includes a good explanation of Swift's Automatic Reference Counting (ARC) system. https://betterprogramming.pub/retain-cycles-and-memory-management-in-swift-fb6226165b17
- Visualizing memory management in Rust (part of a series) by Deepu K Sasidharan: https://deepu.tech/memory-management-in-rust/
- Ada SPARK is a programming language, a verification toolset, and a design method intended for environment where high reliability is a requirement. https://www.adacore.com/about-spark
- Information about memory safe programming languages designed around the same time as C: https://noncombatant.org/2023/05/21/protel-sos-dsm-100/
- Adam Zabrocki, Alex Tereshkin: Exploitation in the era of Formal Verification https://www.youtube.com/watch?v=TclaZ9LW1WE SPARK mitigation at DEF CON 30.
- C versus Rust performance: https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust.html
- NSA Cybersecurity Information Sheet: Software Memory Safety: https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

Appendix: Memory Safe Languages

Language	Description
C#	Microsoft introduced C# in 2000 and designed it to be a simple, efficient, and type-safe language suitable for a wide range of applications, including mobile applications, web applications, and games. C# source code is compiled to an intermediate language, called Common Intermediate Language (CIL), before being executed by the .NET runtime environment. C# is widely used for building Windows desktop and server applications, and is also available on Linux, and MacOS for x86, x64, and ARM architectures.
Go	Go is a cross-platform, compiled programming language developed by Google and released in 2007. Google designed it to be simple and efficient to use and it is well-suited for developing concurrent and networked applications. It is syntactically like C, but with memory safety, garbage collection, and structural typing. Several high-profile companies have migrated some systems to Go from other languages like Python. Apps like Terraform, Docker, and Kubernetes are written in Go.
Java	Java is a garbage collecting MSL owned by Oracle and released in the mid-1990s. It is one of the most popular languages[43] and is used in web applications, enterprise software, and mobile applications. Java source code is compiled to Java Virtual Machine (JVM) bytecodes that can run on any JVM machine and is platform independent.
Python	Python was first released in 1991. It is generally an interpreted language, though it can be compiled into bytecode. It is dynamically typed, and garbage collected. It runs on Windows, Mac, and Linux, and is popular for writing web applications and some embedded systems like Raspberry Pi. It is frequently cited as the most popular programming language.[44]
Rust	Mozilla released Rust in 2015. It is a compiled language and focuses on performance, type safety, and concurrency. It has an ownership model designed to ensure that there is only one owner of a piece of data. It has a feature called a "borrow checker" designed to ensure memory safety and prevent concurrent data races. While not perfect, the borrow checker system goes a long way to addressing memory safety issues at compile time. Rust enforces correctness at compile time to prevent memory safety and concurrency problems at runtime. As an example, a data race is a class of software bug that is notoriously hard to track down. "With Rust, you can statically verify that you don't have data races. This means you avoid tricky-to-debug bugs by just not letting them into your code in the first place. The compiler won't let you do it."[45] Rust has been getting a great deal of attention from several high-profile technologies, including the Linux kernel, Android, and Windows. It is also used in apps like those from Mozilla, and other online services, such as Dropbox, Amazon, and Facebook.[46]
Swift	Apple released the Swift programming language in 2014 and designed it to be easy to read and write. It is intended to be a replacement for C, C++, and Objective-C. It is possible to incorporate Swift code into existing Objective-C code to make migration to Swift simpler. Swift is primarily used for developing iOS, Watch OS, and Mac OS X applications. Apple claims that Swift is up to 2.6 times faster than Objective-C.

Purpose

This guidance was developed by U.S., Australian, Canadian, UK, and New Zealand cybersecurity authorities to further their respective cybersecurity missions, including their responsibilities to develop and issue cybersecurity specifications and mitigations.

Acknowledgements

The U.S., Australian, Canadian, UK, and New Zealand cybersecurity authorities would like to acknowledge Microsoft for their contributions to this guidance.

Disclaimer

Contact Information

U.S. organizations: report incidents and anomalous activity to CISA 24/7 Operations Center at report@cisa.gov or (888) 282-0870 and/or to the FBI via your local FBI field office, the FBI’s 24/7 CyWatch at (855) 292-3937, or CyWatch@fbi.gov. When available, please include the following information regarding the incident: date, time, and location of the incident; type of activity; number of people affected; type of equipment used for the activity; the name of the submitting company or organization; and a designated point of contact. For feedback on this document, please contact SecureByDesign@cisa.dhs.gov. **Australian organizations:** visit cyber.gov.au or call 1300 292 371 (1300 CYBER 1) to report cybersecurity incidents and to access alerts and advisories. **Canadian organizations:** report incidents by emailing CCCS at contact@cyber.gc.ca. **United Kingdom organizations:** report a significant cyber security incident at ncsc.gov.uk/report-an-incident (monitored 24 hours) or, for urgent assistance, call 03000 200 973. **New Zealand organizations:** report cyber security incidents to incidents@ncsc.govt.nz or call 04 498 7654.

References

[1] Microsoft. "A Proactive Approach to More Secure Code." Microsoft Security Response Center (MSRC) Blog. July 16, 2019. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/.

[2] Chromium. "Memory Safety." Chromium Security. n.d. https://www.chromium.org/Home/chromium-security/memory-safety/.

[3] Hosfelt, Diane. "Implications of Rewriting a Browser Component in Rust." Mozilla Hacks - the Web Developer Blog. March 5, 2019. https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/.

[4] There are several types of memory-related coding errors including, but not limited to:

1. **Buffer overflow** [CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')], where a program intends to write data to one buffer but exceeds the buffer's boundary and overwrites other memory in the address space.
2. **Use after free** [CWE-416: Use After Free], where a program dereferences a dangling pointer of an object that has already been deleted.
3. **Use of uninitialized memory** [CWE-908: Use of Uninitialized Resource], where the application accesses memory that has not been initialized.
4. **Double free** [CWE-415: Double Free], in which a program tries to release memory it no longer needs twice, possibly corrupting memory management data structures.

[5] MITRE. "CWE CATEGORY: Comprehensive Categorization: Memory Safety." Common Weakness Enumeration. n.d. https://cwe.mitre.org/data/definitions/1399.html#:~:text=CWE%20-%20CWE-1399%3A%20Comprehensive%20Categorization%3A%20Memory%20Safety%20%284.12%29,Community-Developed%20List%20of%20Software%20%26%20Hardware%20Weakness%20Types

[6] Hicks, Michael. "What Is Memory Safety? - The PL Enthusiast." The Programming Languages Enthusiast. January 22, 2018. http://www.pl-enthusiast.net/2014/07/21/memory-safety/.

[7] One, Aleph. "Smashing The Stack For Fun And Profit." University of California, Berkeley. 2008. https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf.

[8] Microsoft. "A Proactive Approach to More Secure Code." Microsoft Security Response Center (MSRC) Blog. July 16, 2019. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/.

[9] Chromium. "Memory Safety." Chromium Security. n.d. https://www.chromium.org/Home/chromium-security/memory-safety/.

[10] Hosfelt, Diane. "Implications of Rewriting a Browser Component in Rust." February 28, 2019. https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/.

[11] Stone, Maddie. "The More You Know, The More You Know You Don't Know." April 2022. https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html.

[12] Manès VJ, Han H, Han C, Cha SK, Egele M, Schwartz EJ, Woo M. "The Art, Science, and Engineering of Fuzzing: A Survey . IEEE Transactions on Software Engineering." 2019 Oct 11;47(11):2312-31 [LINK]

[13] Open Web Application Security Project (OWASP) Foundation. "OWASP Top Ten." OWASP. n.d. https://owasp.org/www-project-top-ten/.

[14] Bastien, JF. "Keynote: Safety and Security: The Future of C++." CppNow (YouTube Channel). 2023. https://www.youtube.com/watch?v=Gh79wcGJdTg.

[15] Hinnant, H, Orr, B, Stroustrup, B, Vandevoorde, D, Wong, M. "Opinion on Safety for ISO C++" https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2759r0.pdf.

[16] Apple. “Memory Safe iBoot Implementation.” Apple Support. February 18, 2021. https://support.apple.com/en-il/guide/security/sec30d8d9ec1/web.

[17] Amar, Saar. “Introduction to Firebloom (iBoot).” iBoot_Firebloom. n.d. https://saaramar.github.io/iBoot_firebloom/.

[18] Microsoft. “GitHub - Microsoft/Checked” n.d. https://github.com/microsoft/checkedc.

[19] Neumann, Thomas “P2771R0: Towards Memory Safety in C++.” Open STD. January 17, 2023. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2771r0.html.

[20] Carbon-Language. “GitHub - Carbon-Language.” n.d. https://github.com/carbon-language/carbon-lang#memory-safety.

[21] CppNow23. “YouTube: Carbon Language Successor Strategy: From C++ Interop to Memory Safety - Chandler Carruth - CppNow 23.” September 2023. https://youtu.be/1ZTJ9omXOQ0?si=t8KsLWBv1DDFOUXs&t=3455.

[22] Fei, Shufan, Zheng Yan, Wenxiu Ding, and Haomeng Xie. “Security Vulnerabilities of SGX and Countermeasures: A Survey.” July 13, 2021. *ACM Computing Surveys* 54 (6): 1–36. https://dl.acm.org/doi/abs/10.1145/3456631.

[23] Microsoft. “Control Flow Guard for Clang/LLVM and Rust.” Microsoft Security Response Center. August 2020. https://msrc-blog.microsoft.com/2020/08/17/control-flow-guard-for-clang-llvm-and-rust/.

[24] Yunhai, Zhang. “Bypass Control Flow Guard Comprehensively.” BlackHat. July 19, 2015. https://www.blackhat.com/docs/us-15/materials/us-15-Zhang-Bypass-Control-Flow-Guard-Comprehensively-wp.pdf.

[25] iamelli0t. “Exploiting Windows RPC to Bypass CFG Mitigation: Analysis of CVE-2021-26411 In-the-Wild Sample.” iamelli0t’s Blog (blog). April 10, 2021. https://iamelli0t.github.io/2021/04/10/RPC-Bypass-CFG.html.

[26] Groß, Samuel. “Remote iPhone Exploitation Part 2: Bringing Light into the Darkness -- a Remote ASLR Bypass.” Google Project Zero blog. January 2020. https://googleprojectzero.blogspot.com/2020/01/remote-iphone-exploitation-part-2.html.

[27] Jurczyk, Mateusz. “MMS Exploit Part 5: Defeating Android ASLR, Getting RCE.” Google Project Zero blog. August 2020. https://googleprojectzero.blogspot.com/2020/08/mms-exploit-part-5-defeating-aslr-getting-rce.html.

[28] Microsoft. “MS15-124. Vulnerability in Internet Explorer Could Lead to ASLR Bypass: December 16, 2015.” Microsoft Support. n.d. https://support.microsoft.com/en-us/topic/ms15-124-vulnerability-in-internet-explorer-could-lead-to-aslr-bypass-december-16-2015-7e012708-4af6-487c-12e2-4ffa0f9d7b66.

[29] Boneh, Dan. “On the effectiveness of address-space randomization.” ACM Digital Library. October 25, 2004. https://dl.acm.org/doi/10.1145/1030083.1030124.

[30] Palmer, Chris (Google Chrome Security). “The Limits of Sandboxing and Next Steps.” February 03, 2021. https://www.usenix.org/conference/enigma2021/presentation/palmer.

[31] Watson, Robert N. M. (University of Cambridge)., Moore, Simon W. (University of Cambridge), Sewell, Peter (University of Cambridge), Davis, Brooks (SRI International), Neumann, Peter (SRI International). “Capability Hardware Enhanced RISC Instructions (CHERI).” September 2023. https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/.

[32] Innovate UK, EPSRC and ESRC. “Digital Security by Design.” Digital Catapult. n.d. https://www.dsbd.tech/.

[33] Cowie, Alan. “Digital Security by Design Driving Investment in the Automotive Sector and Embedded Systems - Innovate UK KTN.” Innovate UK KTN (blog). September 12, 2023. https://iuk.ktn-uk.org/news/digital-security-by-design-driving-investment-in-the-automotive-sector-and-embedded-systems/.

[34] Mitsunami, Koki. "Enhanced Security through Memory Tagging Extension." June 24, 2021. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhanced-security-through-mte.

[35] Rahman, Mishaal. "Android 14 May Add an Advanced Memory Protection Feature to Protect Your Device from Memory Safety Bugs." XDA Developers. February 8, 2023. https://www.xda-developers.com/android-14-advanced-memory-protection/.

[36] Koduri, Raja. (Intel). Intel Architecture Day 2020. Intel. https://d1io3yog0oux5.cloudfront.net/_6f1902c731ed10bd8538c1c8c9ca7ca1/intel/db/861/8422/pdf/Intel-Architecture-Day-2020-Presentation-Slides.pdf.

[37] Larabel, Michael. "Intel Working To Combine The Best Of CET + CFI Into 'FineIBT'" phoronix. August 6, 2021. https://www.phoronix.com/news/Intel-FineIBT-Security.

[38] Grauer, Yael, Dhalla, Amira (Consumer Reports). "Report: Future of Memory Safety." CR Advocacy. January 23, 2023. https://advocacy.consumerreports.org/research/report-future-of-memory-safety/.

[39] TIOBE. "TIOBE Index for November 2023, November Headline: Kotlin still on the rise in the TIOBE index." TIOBE Index. November 2023. https://www.tiobe.com/tiobe-index/.

[40] Palmer, Chris (Noncombatant.org). "Prioritizing Memory Safety Migrations." April 9, 2021. https://noncombatant.org/2021/04/09/prioritizing-memory-safety-migrations/.

[41] Walbran, Andrew (Android Rust Team, Google). "Bare-Metal Rust in Android." Google Online Security Blog. October 9, 2023. https://security.googleblog.com/2023/10/bare-metal-rust-in-android.html.

[42] Klabnik, Steve, Nichols, Carol. "Unsafe Rust - The Rust Programming Language." The Rust Programming Language. February 9, 2023. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.

[43] TIOBE. "The Java Programming Language." TIOBE Index. November 2023. https://www.tiobe.com/tiobe-index/java/.

[44] Cass, Stephen. "The Top Programming Languages 2023." IEEE Spectrum, November 14, 2023. https://spectrum.ieee.org/the-top-programming-languages-2023.

[45] Clark, Lin. "Inside a Super Fast CSS Engine: Quantum CSS (Aka Stylo) - Mozilla Hacks - the Web Developer Blog." Mozilla Hacks – the Web Developer Blog. October 9, 2017. https://hacks.mozilla.org/2017/08/inside-a-super-fast-css-engine-quantum-css-aka-stylo/.

[46] Newman, Lily Hay. "The Rise of Rust, the 'Viral' Secure Programming Language That's Taking over Tech." *WIRED*, November 2, 2022. https://www.wired.com/story/rust-secure-programming-language-memory-safe/.