Memory Safe Languages: Reducing Vulnerabilities in Modern Software Development

Executive summary

Memory safe languages (MSLs) are gaining momentum. In 2022, the National Security Agency (NSA) released a cybersecurity information sheet (CSI), "Software Memory Safety." [1] In 2023, the Cybersecurity and Infrastructure Security Agency (CISA) published the joint guide, "The Case for Memory Safe Roadmaps," [2] and in 2024, the White House issued "Back to the Building Blocks: A Path Toward Secure and Measurable Software." [3] Though these each address the problem of memory-unsafe code from a different perspective, they all agree that adopting MSLs is a key part to decreasing vulnerabilities and reducing the risk of security incidents.

The goal of these documents is to strengthen national cybersecurity by reducing memory-related vulnerabilities, which requires more than developer discipline and best practices. Achieving better memory safety demands language-level protections, library support, robust tooling, and developer training. While decades of experience with non-MSLs have shown that secure coding standards and analysis tools can mitigate many risks, they cannot fully eliminate memory safety vulnerabilities inherent to these languages as effectively as the safeguards used in MSL.

MSLs offer built-in safeguards that shift safety burdens from developers to the language and the development environment. By integrating safety mechanisms directly at the language level, MSLs enhance security outcomes and reduce reliance on after-the-fact analysis tools. However, adoption comes with challenges. Selecting the appropriate MSL depends on factors such as concurrency and performance, which may increase in difficulty with large or complex existing codebases. Starting MSL adoption is not currently practical in all circumstances or solution areas; additional investments may be necessary to reduce memory safety bugs.

This report, released by NSA and CISA, acknowledges the challenges and aims to provide a balanced view of the state of MSLs. Reducing memory safety vulnerabilities requires understanding when MSLs are appropriate, knowing how to adopt them effectively, and recognizing where non-MSLs remain practical necessities.

Introduction

Memory safety vulnerabilities, such as buffer overflows, have long plagued software systems. The Heartbleed and BadAlloc vulnerabilities exemplify the dangers posed by poor memory management. Heartbleed affected over 800,000 of the most visited websites and resulted in the theft of sensitive personal data, including millions of hospital patient records. [4],[5] BadAlloc impacted embedded devices, industrial control systems, and over 195 million vehicles, demonstrating how memory vulnerabilities threaten national security and critical infrastructure. [6] These examples underscore the urgency of finding better solutions. MSLs such as Ada, C#, Delphi/Object Pascal, Go, Java, Python, Ruby, Rust, and Swift offer built-in protections against memory safety issues, making them a strategic choice for developing more secure software. MSLs can prevent entire classes of vulnerabilities, such as buffer overflows, dangling pointers, and numerous other Common Weakness Enumeration (CWE) vulnerabilities. [7] Unlike non-MSLs, which rely heavily on developer discipline to ensure safe memory handling, MSLs embed memory safety mechanisms directly into the language itself, making them more secure by design. [1]

The importance of memory safety cannot be overstated.

MSLs represent a significant evolution in the approach to software security, moving beyond existing measures to proactively prevent vulnerabilities by default during development at compile time and/or during runtime. The importance of memory safety cannot be overstated: a 2019 study estimated that 66% of Common Vulnerabilities and Exposures (CVEs) for iOS 12 and 71% of CVEs for Mojave were caused by memory safety issues. [8] The consequences of memory safety vulnerabilities can be severe, ranging from data breaches to system crashes and operational disruptions. For example, a Google Project Zero review of exploits detected in-the-wild estimates that 75% of CVEs used in those exploits were memory safety vulnerabilities. [9] Out of the 58 in-the-wild zero-days discovered in 2021, 67% were memory safety vulnerabilities. [10] As a result, the adoption of MSLs is regarded as a key strategy in improving software security and reducing the risk of costly security incidents. This aligns with CISA's Secure by Design principles, which advocate for reducing vulnerability classes by default. [11]

The Office of the National Cyber Director (ONCD) and CISA have strongly advocated for the adoption of MSLs with multiple publications emphasizing MSLs' importance. [2],[12],[13],[14],[15],[16],[17] CISA's Secure by Design program specifically calls for integrating proactive security measures throughout the software development lifecycle, with MSLs as a central component. [11] Consistent support for MSLs underscores the benefits of this transition for national security and resilience.

A balanced approach acknowledges that MSLs are not a panacea and that transitioning involves significant challenges, particularly for organizations with large existing codebases or mission-critical systems. However, several benefits, such as increased reliability, reduced attack surface, and decreased long-term costs, make a strong case for MSL adoption.

Memory vulnerabilities explained

Memory is where a computer stores and accesses data. Memory safety bugs occur when a computer program incorrectly uses memory. They often arise from languages that allow control over memory allocation and access, combined with improper memory management by developers. MSLs are designed to enforce memory safety by default, reducing the risk of security breaches caused by memory mismanagement.

Technical examples of memory bugs

These memory bugs are common in non-MSLs that use manually managed memory environments.

- **Buffer overflow:** a program allocates a fixed buffer size for data and writes data intended to be contained inside the buffer outside the bounds of the buffer, usually overwriting adjacent memory and corrupting data outside the allocated buffer.
- **Use-after-free:** a program allocates memory for an intended purpose, deallocates the memory and continues to use the memory after it was freed. If the memory was reclaimed and allocated for something else, the process of freeing and reallocating and/or the use of the same memory for different purposes causes data corruption.
- **Data races:** when two or more threads in a single application concurrently access the same memory location and one or more threads are writing data, the resulting value in memory or the value read can be timing and architecture specific, leading to data corruption.

- **Initialization safety:** occurs when a programmer assumes memory was initialized correctly and reads its data without proper initialization. The uninitialized data is essentially corrupted data compared to what the programmer expected.

Some memory bugs may be memory vulnerabilities exploitable by attackers. For example, suppose a programmer, by accident, allocates only 100 bytes of memory to store 300 bytes of public information. Because the memory holding the last 200 bytes of public data was never allocated, the program may later allocate that same memory to store confidential data. Later when the program intends to display the 300 bytes of public data, it could actually retrieve and divulge 100 bytes of public data and 200 bytes of confidential data. In other cases, memory corruption can allow an attacker to fully control program execution and actions or trigger crashes that impact system availability.

Key features addressing memory safety

MSLs use built-in mechanisms to prevent memory bugs. They embed safety features directly into the language by default to avoid most memory mismanagement vulnerabilities. Examples are:

- **Bounds checking:** prevents buffer overflows by keeping memory accesses within allocated boundaries. Some languages enforce bounds checking through type safety, which restricts the operations that can be performed on each data type. Type safety ensures the bounds and behavior of an object are known when the object is created and enforced whenever it is accessed.

- **Memory management:** minimizes the likelihood of manual memory management errors by forcing memory initialization before use and employing either garbage collection (e.g., as in Go or Java) or strict ownership and borrowing rules for each region of memory allocated by default (as in Rust). Garbage collection is a technique where while a program is active, a memory management engine automatically runs in the background (usually with compile time support) that manages memory allocation and periodically frees memory that is no longer being used. A way to determine whether memory is not used anymore is by verifying no program variables point to the memory anymore. Strict ownership ensures that only the data owner can modify the data at an acceptable time and that memory is freed when it no longer has an owner. Both approaches for memory management help prevent bugs, such as use-after-free ones.

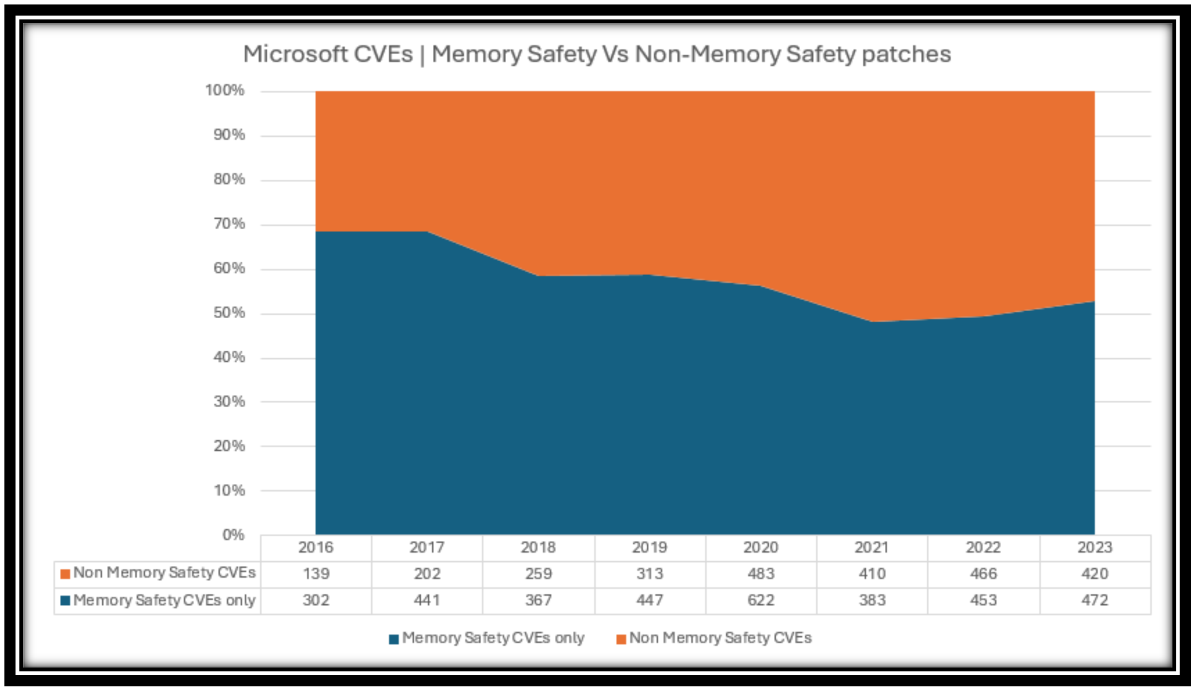
- **Data race prevention:** prevents unsynchronized concurrent access to a piece of data from two or more threads by default.

Memory safety in practice

Security by design

A core strength of MSLs lies in their proactive security by design. Unlike non-MSL approaches to detecting memory issues after the code is written, such as fuzzing or exploit mitigations, MSLs embed safety mechanisms directly into the language. This design prevents memory safety bugs from the outset.

This approach represents a paradigm shift in approaching security. It builds on proven successes in other domains, such as XSS and SQL Injection vulnerabilities, where some software manufacturers have implemented secure by design APIs and libraries that have virtually eliminated the occurrence of these flaws in their products.



	2016	2017	2018	2019	2020	2021	2022	2023
🟧 Non Memory Safety CVEs	139	202	259	313	483	410	466	420
🟦 Memory Safety CVEs only	302	441	367	447	622	383	453	472

Figure 1: Microsoft CVEs | *Memory Safety Vs Non-Memory Safety Patches*

Figure *1* shows all Microsoft's CVEs from 2016 through 2023 as a percentage of memory related issues versus the total number each year. In 2016, Microsoft attributed nearly 70% of their CVEs to memory safety. [18] In recent years the percentage has declined to approximately 50%. Despite improvements, approximately half of Microsoft's patches still address memory safety CVEs, which may be further reduced with greater use of MSLs.

Case study: Android's transition

The Android operating system, a complex platform with a vast codebase, provides a compelling illustration of the impact of MSLs. In 2019, memory safety issues accounted for 76% of all Android vulnerabilities—typical for projects predominantly developed in memory-unsafe languages. [19]

Recognizing the high concentration of memory-related vulnerabilities in new code, the Android team made a strategic decision to prioritize MSLs, specifically Rust and Java, for all new development. Rather than attempting a massive and complex rewrite of existing code, they focused on preventing new vulnerabilities from entering the system.

By 2024, memory safety vulnerabilities had plummeted to 24% of the total, representing an improvement that had not been seen with previous approaches to memory safety. This success underscores the effectiveness of MSLs in proactively building a more secure foundation for software. [19]

Reliability and productivity benefits

Adopting MSLs not only produces more secure systems but also more reliable and stable ones. By preventing memory bugs that often lead to crashes and unpredictable behavior, MSLs drive down costs that would otherwise be spent addressing vulnerabilities and contribute to increased uptime and smoother operations.

MSLs make it easier to write correct and reliable software, which improves software quality as well as developer productivity by eliminating entire classes of bugs and integrating runtime checks:

- **Elimination of bug classes:** By default, MSLs prevent certain classes of bugs from ever occurring. For instance, use-after-free bugs, a common source of crashes in non-MSLs, are impossible to have without overriding the default language settings. These built-in controls inherently lower the probability of incorrect program behavior or crashes. In addition, the higher specificity of MSL type systems can also prevent programs from entering invalid or unintended states.

- **Runtime safety checks:** MSLs incorporate runtime safety checks that detect and prevent memory corruptions, preventing potentially silent data corruption and leading to more informative error messages that aid in faster debugging.

These same advantages translate to increased developer productivity. Early error detection during compilation or runtime testing accelerates debugging, reduces troubleshooting time, and minimizes the risk of costly incidents. This quality improvement empowers developers to focus on innovation and feature development rather than constantly battling memory safety issues.

Fewer crashes and unexpected errors during operation also translate to reduced downtime and improved system availability, which are essential for businesses that rely on continuous operation.

Scalability

Adopting MSLs does not necessitate a complete rewrite of existing codebases. As demonstrated in Android, prioritizing MSL adoption in new code, and leveraging interoperability to integrate with existing codebases, offers a practical and cost-effective path toward enhanced security.

While not as effective as adopting an MSL, applications written in non-MSLs can also be made safer by:

- enabling bounds checking, [20]
- avoiding inherently unsafe functions,
- adopting smart pointers,
- using recommended compiler options, and
- performing static and dynamic analysis.

Deciding on a balanced adoption approach

When considering the adoption of MSLs, it is essential to weigh the benefits of reducing security incidents and achieving long-term cost savings against the initial investments that may be incurred. Additionally, aligning with security best practices, regulatory requirements, and industry standards, while carefully selecting an MSL that integrates with existing codebases, is crucial for a successful adoption strategy.

Adoption considerations

- Adopt MSLs to help reduce security incidents, minimize emergency patching, and achieve long-term cost savings.
- Invest initially in training, tools, and refactoring. This investment can usually be offset by long-term savings through reduced downtime, fewer vulnerabilities, and enhanced developer efficiency.
- Carefully select an MSL to help ensure it can integrate seamlessly with existing codebases, APIs, and external libraries.
- Align with security best practices
    - Approach security holistically using MSLs as one risk mitigation technique to be combined with others, such as practices in the National Institute of Standards and Technology (NIST) Secure Software Development Framework (SSDF). [21]
    - Align MSL efforts with industry standards, such as NIST, International Standards Organization (ISO) / International Engineering Consortium (IEC), and any applicable sector-specific guidance.
    - Factor in regulatory and internal compliance requirements throughout the product lifecycle. As necessary, find ways to satisfy language specific requirements, potentially through new compliance or conformance approaches, developing supportive tooling or by engaging with regulators.
- Begin adoption by starting new code or projects with MSLs because most memory vulnerabilities arise in new code. [9]
- Make increasing the usage of MSLs a company priority and plan investments accordingly. CISA urges software manufacturers to create and publish a memory safety adoption roadmap. [22]

Engineering the MSL adoption decision

For technical teams, adopting MSLs presents opportunities to improve security, reliability, and development efficiency while minimizing vulnerabilities related to memory safety issues. This approach aligns with the NIST SSDF by emphasizing proactive security measures, structured documentation, and incremental improvements for a secure by design development culture. [21],[23]

Strategic adoption

New development

Prioritizing MSL adoption in new projects is a relatively low-risk way to introduce memory safety benefits without overhauling the workflows of existing codebases. Starting fresh with MSLs improves code quality by avoiding technical debt and incorporating memory safety from the beginning.

Incremental adoption for existing code

Completely rewriting existing codebases is often impractical. Instead, an incremental adoption strategy for existing systems is often more feasible:

- Write new components and features in MSLs. When tight coupling becomes unwieldy, break down tightly coupled codebases into self-contained components. Modular design simplifies integrating new MSL code with non-MSL systems, allowing for smoother transitions through well-defined APIs.
- Identify high-risk components or modules that have significant attack surfaces or are operationally critical. Rewriting these parts using MSLs helps mitigate vulnerabilities.
    - Examples of high-risk areas include network-facing services, file parsers, codecs, and cryptographic operations.

Interlanguage integration and API considerations

When transitioning to MSLs, managing interlanguage integration is critical. Establishing robust APIs for communication between MSL and non-MSL components helps ensure secure and efficient interoperability. Data marshaling—converting data formats between languages—is commonly used to maintain compatibility and preserve memory safety across different components.

Adoption challenges

Selecting the appropriate MSL depends on multiple factors that influence the requirements and constraints of the software project. Key determinants include:

- The need for low-level access and high efficiency, especially in media processing or cryptographic functions.

- Whether systems are highly concurrent, such as servers handling numerous simultaneous requests.
- The ease of learning and adopting a new language, which might delay onboarding and productivity.
- The need to integrate seamlessly with existing codebases or third-party libraries.
- The availability of tools, libraries, and community support is critical for effective development.

Managing dependencies and ecosystem maturity

While MSLs enhance security, challenges remain around dependency management, such as when critical external libraries were not developed using MSLs. Managing dependencies in a memory safe way is crucial to minimizing security risks. Teams can implement strict version control practices and dependency review processes to help ensure long-term supply chain and overall safety in MSL projects.

Handling legacy systems and tightly coupled code

Existing systems often consist of tightly coupled code, which can make adoption challenging. To address this, focus on breaking down existing codebases into smaller, modular components. This approach allows for easier isolation of high-risk areas and facilitates targeted rewrites using MSLs. Establishing well-defined APIs facilitates compatibility and helps gradually replace non-MSL components without causing significant disruptions.

Performance and scalability considerations

Interlanguage communication between MSL and non-MSL components can introduce performance overhead. To mitigate this, organizations can carefully design interoperability layers and conduct rigorous performance testing to identify and address potential bottlenecks. The adoption of MSLs does not need to compromise the performance or scalability of critical systems.

Training and upskilling teams

Adoption may require equipping teams with relevant skills. Effective training programs can:

- Include memory safety as a core aspect of developer training, focusing on secure by design practices.
- Adapt training paths to the experience of developers. For those familiar with writing low-level firmware or high-performance code in C or C++, highlight similarities to an MSL with manual memory management capabilities. Others might benefit from starting with higher-level MSLs that automatically manage memory allocation and garbage collection.

Language considerations

Selecting an MSL includes several key aspects for technical teams to consider:

- Adopting a language solely because it is currently in vogue can be risky, as it may not have the long-term support or community backing needed for sustained use. The concept of memory safety and other inherent protections within a language is not a passing trend. However, adopting a language that has only a niche market or lacks a strong community can bring significant challenges, even if that language has certain technical advantages.
- Many MSLs, particularly those that are relatively new, evolve at a faster rate than more mature languages. This rapid evolution can be both an advantage and a challenge, as it may lead to frequent changes in language features, tooling, and best practices.
- The ecosystem of tools and libraries available for an MSL can significantly impact its adoption. While MSLs have growing ecosystems, they may still lack extensive tooling and library support found in more established languages. Addressing this gap is an area of ongoing work for various organizations, which are studying and developing recommendations on tooling, library support, and other aspects needed for a competitive MSL ecosystem.

Organizational roles

Organizations in academia, the U.S. Government, and private industry play key roles in the long-term adoption of MSLs. These organizations:

- increase awareness of memory safety,
- develop materials to promote MSLs,
- invest in key programs, and

- create demand for MSL skillsets as industry integrates MSLs into real-world applications.

Academia

While there are many pathways to becoming a software developer, including independent learning, academia can play a key role in increasing awareness of memory safety and the importance of adopting MSLs. Software developers could be taught both about MSLs that they will use professionally and about the fundamentals of memory safety as a cybersecurity concern.

Software development curricula

MSL education in academic institutions for computer science majors almost universally includes at least one garbage collected MSL, such as Python or Java. As such, most developers with formal education are already taught to use an MSL. Most institutions also offer courses on “systems programming,” which involves teaching non-garbage collected memory-unsafe languages.

Academics are beginning to develop curricular materials to promote MSL courses within academic institutions. Non-garbage collected MSLs are relatively new, so few institutions teach them. In the case of Rust, most available learning resources are either online textbooks on Rust’s website, Google's training materials, or vocational training books for sale. [24],[25]

Memory safety education

Separate from individual MSLs, developers can be educated in the general principles of memory safety as a cybersecurity concern that spans all programming languages. Research has shown that understanding memory safety is a key part in helping developers use non-garbage collected MSLs. [26] Many institutions offer a cybersecurity course that includes discussion of memory bugs, such as buffer overflows. However, two factors limit the impact of these courses:

1. Many professional developers have not taken these courses. They are often designated as electives rather than required.
2. These courses rarely describe how to prevent memory bugs by design. [27]

One promising direction is an increased focus on "secure coding” within cybersecurity curricula. Secure coding teaches how developers can architect their software to reduce the chance of security vulnerabilities, as opposed to relying on post hoc security measures, such as stack layout randomization.

U.S. Government

The U.S. Government has long invested in many areas to advance and secure computer science. Some recent programs are:

- The new Safety, Security, and Privacy of Open-Source Ecosystems (Safe-OSE) program, by the National Science Foundation (NSF) Pathways to Enable Open-Source Ecosystems (POSE) team, which includes a focus on funding for safety-oriented projects. [28]
- The Defense Advanced Research Projects Agency (DARPA) Translating All C to Rust (TRACTOR) program, which aims to automate the translation of existing C code to Rust. [29]
- DARPA's Verified Security and Performance Enhancement of Large Legacy Software (V-SPELLS) program, which aims to create practical tools to help developers with legacy software modernization. For example, many vulnerabilities in software occur where untrusted inputs, such as from the network, are initially being parsed and understood. V-SPELLS tools will aid developers in replacing hand-written parsing code with machine-generated parsers that are proven to be free of vulnerabilities. [30]
- Safe Documents (SafeDocs), a 2018 DARPA program, which is finding success at the intersection of MSLs and data format parsing. [31]

Industry

Along with the many roles private industry plays, one to highlight is their ability to create demand. Companies can set and advertise job requirements that include MSL expertise. This action will signal demand for these skills that is felt not only throughout the job market, but also in studies and in future funding of academic and certification programs.

Prossimo, a project of the Internet Security Research Group (ISRG) and the Open Source Security Foundation (OpenSSF), states that it plans to transition the Internet's critical infrastructure to memory safe code and develop memory safe essential software. [32] The OpenSSF, which focuses on enhancing the security of open-source software, promotes the use of MSLs to address vulnerabilities across software repositories and supply chains. [33]

Open questions and study areas

This section outlines the ongoing and future areas of research, highlighting open questions and topics that are critical for understanding and adopting MSLs.

- Choosing the right MSL for different requirements
    - How can organizations assess which MSL is best suited for different applications (e.g., high-performance systems vs. web development)?
    - What trade-offs exist between security, performance, and ease of use for different MSLs?
- Incremental adoption and prioritization
    - What are the best strategies for prioritizing MSL adoption within complex legacy codebases?
    - How can organizations plan incremental transitions while balancing business requirements?
- Handling constrained environments
    - What challenges do MSLs face in specialized environments like industrial control systems, embedded systems, or other resource-constrained environments?
    - How can the performance and compatibility issues in such systems be mitigated?
- Non-MSL safety enhancements
    - For cases where adoption is not feasible, what alternatives exist to enhance safety in non-MSL environments? Can implementing new security features in existing languages enhance safety when MSL adoption is not an option?
        - Alternative vulnerability mitigation approaches include hardware capabilities, such as the Memory Tagging Extension (MTE), and compiler controls.
- Training and tooling challenges
    - What are the barriers to effective MSL education, and how can the availability of training resources be improved?

    - How mature are the current tooling ecosystems for various MSLs, and what are the gaps that need to be addressed?
    - How can the cybersecurity community ensure that the benefits of MSLs do not create complacency among developers, particularly when addressing non-memory-related vulnerabilities?
- Ecosystem problems
    - How do the training, library, and tool gaps for MSLs affect their broader adoption?
    - What incentives can be provided to developers and institutions to accelerate MSL ecosystem growth?
- Software supply chains
    - How can supply chain security investments and resources be adapted to MSLs?
        - Supply chains, including Software Bill of Materials (SBOM), continue to be a consideration for secure software development processes. Significant resources may have already been invested in non-MSL supply chain security that may need to be adapted to support MSLs. There are many reasons to have a strong supply chain initiative and memory safety is one.
- Secure by demand
    - What role can customers play in creating incentives for software manufacturers to adopt MSLs?
    - How can customers create demand for greater transparency into the use of secure software development practices in the products they buy?
- Transpiler use cases
    - When should transpilers be used during the MSL adoption process?
        - A transpiler is software that translates source code from one language to another at approximately the same level of functionality and level of abstraction. [34],[35]
        - There are positive industry reports of this assistive technology, but they are not included in the body of this report because:
            - They may be niche.
            - Recent technology trends may have unpredictable impact.
            - It may be a distraction from the other, more scientifically sound lessons.

- Automated transpiling offers several advantages:
  - Transpilers can provide a starting point to move to an MSL when development teams want to maintain the behavior of a legacy component.
  - Transpiling provides operational code in the new language immediately. However, the code requires further review and probable intervention to ensure quality consistent with the original.
  - Developers can iteratively improve code quality while maintaining functionality.
- These advantages may allow teams to balance immediate functional outcomes and gradual improvement in code quality and safety, making automated transpiling a potential practical bridge during the MSL adoption process.

Conclusion

Memory vulnerabilities pose serious risks to national security and critical infrastructure. MSLs offer the most comprehensive mitigation against this pervasive and dangerous class of vulnerability. Adopting MSLs can accelerate modern software development and enhance security by eliminating these vulnerabilities at their root.

Strategic MSL adoption is an investment in a secure software future. By defining memory safety roadmaps and leading the adoption of best practices, organizations can significantly improve software resilience and help ensure a safer digital landscape.

Acknowledgements

The NSA Cybersecurity Collaboration Center, along with CISA, acknowledges the Communications Sector Coordinating Council (CSCC), the DIB Sector Coordinating Council (DIBSCC), and the IT Sector Coordinating Council (ITSCC) for their collaboration on this guidance.

Works cited

[1] National Security Agency (NSA). Software Memory Safety. 2023. https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

[2] Cybersecurity and Infrastructure Security Agency (CISA) et al. The Case for Memory Safe Roadmaps. https://www.cisa.gov/case-memory-safe-roadmaps

[3] The White House. Back to the Building Blocks: A Path Toward Secure and Measurable Software. 2024. https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

[4] TIME. Report: Devastating Heartbleed Flaw Was Used in Hospital Hack. 2014. https://time.com/3148773/report-devastating-heartbleed-flaw-was-used-in-hospital-hack/

[5] The Register. AVG on Heartbleed: It's dangerous to go alone. Take this (an AVG tool). 2014. https://www.theregister.com/2014/05/20/heartbleed_still_prevalent/

[6] Claroty. What You Need to Know About BadAlloc and OT. 2021. https://claroty.com/team82/blog/what-you-need-to-know-about-badalloc-and-ot

[7] MITRE. Common Weakness Enumeration (CWE): CWE CATEGORY: Comprehensive Categorization: Memory Safety. 2025. https://cwe.mitre.org/data/definitions/1399.html

[8] Kehrer, Paul. Memory Unsafety in Apple's Operating Systems. 2019. https://langui.sh/2019/07/23/apple-memory-safety

[9] Google. Safer with Google: Advancing Memory Safety. 2024. https://security.googleblog.com/2024/10/safer-with-google-advancing-memory.html

[10] Google. The More You Know, The More You Know You Don’t Know. 2022. https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html

[11] CISA. Secure by Design. 2023. https://www.cisa.gov/securebydesign

[12] The White House. Press Release: Future Software Should Be Memory Safe. 2024. https://bidenwhitehouse.archives.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/

[13] The White House. Summary of the 2023 Request for Information for Information on Open-Source Software Security. 2024. https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/08/Summary-of-the-2023-Request-for-Information-on-Open-Source-Software-Security.pdf

[14] CISA et al. Exploring Memory Safety in Critical Open Source Projects. 2024. https://www.cisa.gov/sites/default/files/2024-06/joint-guidance-exploring-memory-safety-in-critical-open-source-projects-508c.pdf

[15] CISA Cybersecurity Advisory Committee. Report to the CISA Director: Memory Safety. 2023. https://www.cisa.gov/sites/default/files/2023-12/CSAC_TAC_Recommendations-Memory-Safety_Final_20231205_508.pdf

[16] CISA and Federal Bureau of Investigation (FBI). Product Security Bad Practices. 2024. https://www.cisa.gov/sites/default/files/2024-10/joint-guidance-product-security-bad-practices-508c.pdf

[17] The White House. National Cybersecurity Strategy Implementation Plan. 2023. https://bidenwhitehouse.archives.gov/wp-content/uploads/2023/07/National-Cybersecurity-Strategy-Implementation-Plan-WH.gov_.pdf

[18] Microsoft. A proactive approach to more secure code. 2019. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/

[19] Google. Eliminating Memory Safety Vulnerabilities at the Source. 2024. https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html

[20] LLVM Project. Hardening Modes. 2025. https://libcxx.llvm.org/Hardening.html

[21] National Institute of Standards and Technology (NIST). NIST Special Publication 800-218: Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities. 2022. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-218.pdf

[22] CISA. The Case for Memory Safe Roadmaps. https://www.cisa.gov/case-memory-safe-roadmaps

[23] Secure Software Development Framework. 2025. https://csrc.nist.gov/Projects/ssdf

[24] The Rust Foundation. Learn Rust. 2025. https://www.rust-lang.org/learn

[25] Google. Comprehensive Rust. 2025. https://google.github.io/comprehensive-rust/

[26] Crichton, Will, et al. A Grounded Conceptual Model for Ownership Types in Rust. 2023. https://dl.acm.org/doi/abs/10.1145/3622841

[27] The Linux Foundation. Secure Software Development Education 2024 Survey: Understanding Current Needs. 2024. https://www.linuxfoundation.org/hubfs/LF%20Research/Secure_Software_Development_Education_2024_Survey.pdf

[28] U.S. National Science Foundation. Safety, Security, and Privacy of Open-Source Ecosystems (Safe-OSE). 2024. https://www.nsf.gov/funding/opportunities/safe-ose-safety-security-privacy-open-source-ecosystems

[29] Defense Advanced Research Projects Agency (DARPA). TRACTOR: Translating All C to Rust. https://www.darpa.mil/program/translating-all-c-to-rust

[30] DARPA. V-SPELLS: Verified Security and Performance Enhancement of Large Legacy Software. https://www.darpa.mil/program/verified-security-and-performance-enhancement-of-large-legacy-software

[31] DARPA. SafeDocs: Safe Documents. https://www.darpa.mil/program/safe-documents

[32] Internet Security Research Group (ISRG). Building a Better Internet: 2024 Annual Report. https://www.abetterinternet.org/documents/2024-ISRG-Annual-Report.pdf

[33] Open Source Security Foundation. OpenSSF Responds to US Federal Government RFI on Open Source Software Security. 2023. https://openssf.org/blog/2023/11/08/openssf-responds-to-us-federal-government-rfi-on-open-source-software-security

[34] ISRG. A Safer High Performance AV1 Decoder. 2023. https://www.memorysafety.org/blog/safer-av1-decoder/

[35] ISRG. AWS commits $1M to bring memory safety to critical parts of the Web. 2023. https://www.memorysafety.org/blog/aws-funding

Disclaimer of endorsement

Purpose

This document was developed in furtherance of the authoring agency's cybersecurity missions, including its responsibilities to identify and disseminate threats and to develop and issue cybersecurity specifications and mitigations. This information may be shared broadly to reach all appropriate stakeholders.

Contact

NSA

Cybersecurity Report Feedback: CybersecurityReports@nsa.gov
Defense Industrial Base Inquiries and Cybersecurity Services: DIB_Defense@cyber.nsa.gov
Media Inquiries / Press Desk: 443-634-0721, MediaRelations@nsa.gov

CISA

Organizations are encouraged to report suspicious or criminal activity related to information in this guide to CISA via CISA's 24/7 Operations Center (report@cisa.gov or 888-282-0870) or your local FBI field office. 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.