REPORT TO THE CISA DIRECTOR

Technical Advisory Council

Memory Safety

December 5, 2023

Introduction

This report explores how CISA can engage in improving the resilience of computer, network, embedded, and consumer systems by taking advantage of, and promoting, memory safe technologies.

When memory is written or read in the wrong space or time, memory corruption can occur and lead to low-level vulnerabilities that can be exploited to compromise a system. Improving memory safety is a strategy to reduce the number of vulnerabilities capable of being exploited, thereby decreasing the risks of the technologies that society relies on to operate. This report focuses on memory safety, and specifically memory safe languages suitable for use in systems development.

Background

Vulnerabilities are constantly discovered, both by those working to improve system security, and by those attacking it. By gathering statistics and anecdotal evidence a pattern emerges: Memory safety accounts for approximately 70% of reported security issues according to Microsoft [1] and Google Chrome [2]. According to the public zero-day tracker 2021 [3] had the highest number of zero-day exploits on record with 88, the majority due to memory safety issues. This memory safety issue persists despite widespread use of security techniques such as vulnerability fuzzing, secure software development lifecycle processes, static code analysis, and penetration testing.

In other words, based on today's understanding, there is a 70/30 rule: 70% of all vulnerabilities reported to Microsoft and Google are due to memory safety issues, and 30% are in other categories, such as logic flaws. An important caveat to understand is that this trend does not necessarily hold for *exploited* vulnerabilities. We can reason about vulnerabilities as *potential risk* whereas the implementation of an exploit is an *actualized risk*.

The Problem of Memory Safety

Memory safety is an age-old problem with the first large scale internet worm leveraging a stack-overflow in the unix finger daemon in 1988 [4]. To understand the problem of mitigating memory safety risks, it is important to understand the broad categories of memory vulnerabilities, *Spatial* and *Temporal*, the broad categories of solutions, *Probabilistic* or *Deterministic*, and the major categories of programming languages, *System* and *Non-system*.

The primary focus for defenders over the last 20 years has been to address memory safety by targeting protections against individual exploit techniques and their variants rather than focusing on the underlying classes of vulnerabilities that make the exploits possible in the first place.

Exploit mitigation techniques like Address Layout Randomization (ASLR), Stack Cookies, and Control Flow Integrity (CFI) succeeded in temporarily driving up costs for exploitation but have failed to disrupt exploit development overall as evidenced by the ongoing linear growth in known memory safety exploits [5] with 2023 on track to be the highest year on record.

To address the issue of memory corruption exploits, and to break the cycle of forever adding specific exploit mitigations which introduce their own complexity and cost, a new strategy which focuses on defeating the vulnerability class itself is necessary. An approach which addresses all known classes with deterministic protection can be called memory safe and is the most promising way forward for long term software safety.

Memory Safe Languages

There are many languages that provide type and memory safety guarantees. Some of the most popular languages in use today such as JavaScript, Python, and Java provide memory or type safety by default, and their use comprises a majority of software development [6][7]. Unfortunately bypassing these protections is common when interoperability is required with legacy C/C++libraries or performance reasons.

In these languages memory safety is enforced by interpreters or virtual machines at runtime which comes at the cost of performance and resource efficiency. This has limited widespread use of memory safe languages to use cases where performance is not the most important factor. Key pieces of software such as Web Browsers, Office Applications, and Operating Systems are generally still implemented in so-called “native” languages such as C/C++ for performance and efficiency reasons.

A new generation of memory safe but high-performance memory safety languages has changed this trade-off calculus. Memory Safe Systems Languages (MSSL) such as Rust, Golang, and Swift have been used as system languages which can meet stringent performance and efficiency requirements while still maintaining memory safety. It is this new development that has seen the security and development community push for more widespread use of systems level memory safe languages.

The clear north star for security against memory corruption exploits is the broad usage of Memory Safe Languages that provide intrinsic, deterministic memory safety with performance and efficiency that make them practical for low-level and systems applications. Anything less is playing the whack-a-mole of exploit mitigation.

Spatial and Temporal

When considering the memory safety of a program there are two broad categories, *Spatial* and *Temporal* Memory safety [8][9]]. Developers employ many different strategies to try and protect against attacks against these two categories. For a program to have complete protection both categories must be protected against, in either software, hardware or a combination of both.

Spatial Memory Safety issues result from memory accesses performed outside of the "correct" bounds established for variables and objects in memory. Examples of spatial safety issues include stack or heap-based buffer overflows where adjacent memory is overwritten. Often, this exploitation of spatial memory safety issues directly overwrites objects in memory with attacker-controlled data leading to system compromise. An example of this would be a buffer overflow. Essentially you are writing or reading in the wrong space.

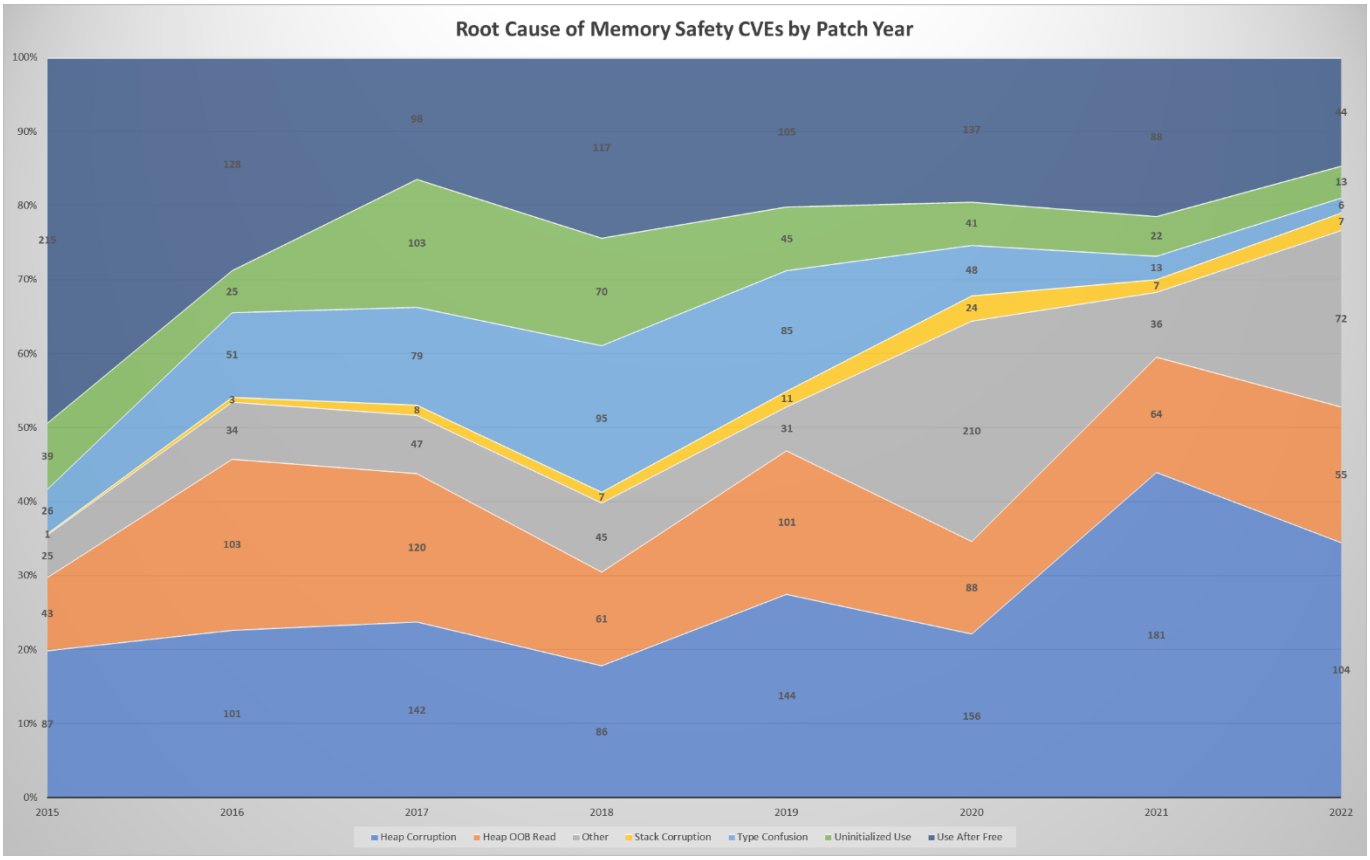
Temporal Memory Safety violations arise when memory is accessed outside of time or state, such as accessing object data after the object is freed. It could also happen when memory accesses are unexpectedly interleaved, such as when critical sections fail to properly lock which leads to concurrent data access. This can enable code execution or other security impacts when an attacker can replace memory with an unexpected state such as a pointer to memory containing malicious code. This is often achieved with techniques that cause the behavior or state of dynamic data structures such as the heap to become more predictable. Essentially you are reading or writing memory at the wrong point in time.

A variant of temporal safety issues can occur *without* violating object lifetimes, such as concurrent tasks concurrently accessing valid data with insufficient locking. For the purposes of this paper, we include these issues in Temporal Safety.

Both overarching categories, Spatial and Temporal, include, in turn, numerous "classes" of bugs that share common underlying characteristics. In general, these characteristics are associated with Common Weakness Enumeration (CWE) entries maintained by MITRE [10]

Categorization of Memory Safety Issues

Understanding the root-cause of memory corruption issues at a granular level is important when evaluating approaches to achieve memory safety. Each specific subclass of temporal and spatial issues may require specific solutions or imply less expensive and more practical solutions. Common discourse around memory safety strategies is often binary and focus on languages which are known to be completely memory safe (e.g. Rust). This approach can limit the investment in practical and effective solutions for eliminating bug classes in existing C/C++ code. The chart below provides the granular root-cause-analysis (RCA) for memory safety issues reported to Microsoft.



This data provides useful framing for the specific bug classes and their relative population in a common operating system. The TAC utilized this data to evaluate the relative impact and priority of recommendations throughout the document.

It is possible to pursue different technological approaches over time, each capable of reducing the percentage of memory safety vulnerabilities, with some eliminating a bug class entirely. Adopting memory safe technologies will reduce the number of vulnerabilities that can be considered a potential risk, and a drop in potential risk is expected to limit the number of exploits which can be thought of as actualized risk. This is because the pool of potential vulnerabilities to exploit is significantly smaller.

Probabilistic vs. Deterministic vs. Some of both

Memory protections (and mitigations in general) can be categorized as either *Probabilistic* or *Deterministic*.

Probabilistic memory protection is where protection against an unexpected behavior is likely but not guaranteed. This can occur for performance reasons such as having a security check that is asynchronous to avoid blocking.

An example of Probabilistic memory safety protection is GWP-ASan. GWP-ASan [11][12] performs tracking of userspace heap allocations. When GWP-ASan detects an error, it records a crash report and terminates the process. The bug reports contain additional allocation and deallocation information that make it easier to identify the root cause. For mobile devices that have limited resources GWP-ASan is randomly enabled at start-up for 1% of system processes, and in dedicated server infrastructure GWP-Asan could be enabled 100% of the time.

Attackers that need near certainty that their attacks will succeed and not be detected may be deterred by probabilistic defenses that allow for a low chance of success, and a successful attack may require defeating more than a single defense increasing the probability of detection even more. Because of these considerations *Probabilistic* protections, while not absolute like deterministic, can provide both a deterrent and a method, such as GWP-ASan, for detecting when an attack has failed. This creates an opportunity for the software developer to detect and improve their protections.

Probabilistic protection can in some cases be bypassed without being detected, either with brute force techniques or with the help of additional information leak vulnerabilities, where the underlying applications do not crash or report the attempts. Clever attackers will target where disruption (causing a crash) or detection are least likely.

An example of a *Deterministic* memory safety protection would be an application written entirely in a MSSL without using any unsafe exceptions. The language provides for security guarantees of temporal and spatial memory protection. For example, with Rust's guaranteed ownership semantics when a dynamically allocated value is freed ownership is relinquished. Any subsequent attempts to use the value will produce an error. Spatial memory safety problems such as off-by-one, stack-based buffer overflow are not possible.

There exist situations where a technology does not cleanly provide either one or the other type of protection, but combination of *both Probabilistic and Deterministic*. For example, some Memory Tagging (MT) protections can be seen as both deterministic and probabilistic. For linear buffer overflows MT can be used as a deterministic protection [13] as well as for heap-use-after-free if combined with an additional GC-like pass [14]. For non-linear buffer overflows MT provides only a probabilistic protection, such that a given instance of a bug will be detected with ~90% probability.

Currently Arm has introduced Memory Tagging Extensions (MTE) in hardware which Google recently implemented in their Pixel 8 [15]. Apple is possibly working to implement their version [16] as well. MTE is clearly emerging as an improvement to current strategies but should not be considered as a replacement for the use of MSSL.

Systems Languages vs. Non-Systems Languages

Systems languages are used by operating systems, mobile and embedded device manufactures as well as cloud providers, aerospace, and in situations where performance and efficiency are of primary importance. Historically classical systems languages such as C and C++ were chosen primarily for their efficiency, performance, and flexibility but traded off inherent memory safety for performance. These languages, and others not mentioned, are generally accepted as providing performance advantages over classic interpreted or managed languages such as Java, C#, or Python and thus can be used for low level use cases.

Non-Systems languages would be exemplified by the classic managed languages that are suitable for "high level" development scenarios which don't have strict performance requirements, and existing software developers with experience in these languages can continue to safely use them.

For systems scenarios, MSSL provides a clear value from a security standpoint as they categorically eliminate a range of memory safety bugs using a combination of static and dynamic enforcement. Each MSSL will have its own strengths and weaknesses to address enforcement, and each will have performance tradeoffs that will need to be considered from an implementation perspective. However, adopting any of these languages would lead to a significant increase in memory safety.

Potential Solutions

Cost Considerations

There are real costs creating friction against making all code memory safe. Costs fall into three broad categories: *Cost of labor*, the *cost of performance*, and the *cost of certifications*.

The *cost of labor* relates to developer effort in reimplementing software components in memory safe languages and can include developer skills and training, cost and resources, adapting existing tooling and CI/CD pipelines, as well as Quality Acceptance (QA) testing changes necessary to support the new protection technologies.

In addition to the tooling to ease transitions discussed above, investment in interoperability tooling can help reduce friction. For example, the existing production-quality interoperability tooling for C++/Rust assumes a narrow API surface. While this has been sufficient for some ecosystems such as Android, other ecosystems have additional requirements [17].

The *cost of performance* relates to how much additional computational overhead or system resources are needed to implement the memory protection technology. In some embedded devices additional compute overhead may not be tolerable and the protection options will be largely determined by device performance. In system and application use cases additional overhead may not impact deployment requirements.

For example, if implementing a solution that provides complete spatial memory protection requires 5% more CPU overhead or 5% more memory then a company could calculate how many more servers would be needed to perform at current levels and then weigh it against the benefits such as increased security, fewer service interruptions and unexpected patching, smaller attack surfaces to manage, etc.

The *cost of certification* efforts, which are necessary in many regulated industries such as medical, aviation, automotive, oil and gas, chemical, etc. when making changes to already approved systems must be factored in.

In some cases, executives may find these costs of transition too high. Investments by the industry in developing tools that can reduce the friction, and therefore the cost, of transition will help change the decision calculus. Moreover, better software interoperability tooling can make it easier for software ecosystems to adopt memory safe code *incrementally*, even while some parts still use legacy code. Compared to complete rewrites, this may result in lower cost at a trade-off of reduced overall safety. Alternatively, memory safety can be adopted for *new* projects without impacting legacy code. When transitions to memory safe code are still not viable, hardware-based mitigation can help reduce risk, and CISA could help with encouraging hardware vendors to develop and deploy mitigations.

When and where to implement Memory Safety Protections

There are generally three places where memory safety protections can be applied: Before compiling the program, during the compile time of the program from source code into executable code, and during the run time execution of the program.

Before Compiling protections use formal verification tools that can detect and reject unsafe code prior to the compilation

phase. For the best and most consistent results these tools must be built into the Software Development Lifecycle (SDLC) so that developers cannot accidentally or intentionally bypass them.

Compile time protections typically can often make overarching guarantees about the resulting compiled binary and, downstream, provide the least performance loss. However, the application of compile-time protection often requires extensive refactoring and rewriting of existing code.

Runtime protections are generally less performant because CPU time must be spent in verifying program state integrity rather than executing program logic. The advantage of such protections is that they can be *retrofitted* into existing, even already-compiled code (e.g., temporal protection through the substitution of a secure dynamic memory allocator). MSSLs replace many (but not all) runtime protections with equivalent compile-time protections.

Incremental Rewrite vs. Complete Rewrite

Once the decision has been made to embrace memory safe languages the question becomes how do you integrate it into your development process. There are two approaches, *incremental rewrite* or *complete rewrite*. Each has strengths and benefits, and your correct answer will be based on your specific needs.

An *incremental* approach means you identify the most at risk portions of your code and implement memory safety there first. Static analysis tools such as CodeQL or semgrep can be used to help search source code for areas that would gain the most protection, such as areas dealing with certificates, authorization tokens, user credentials, API keys, etc.

We have seen that higher fidelity interoperability enables incremental adoption in additional ecosystems, as done for Swift [18] already, and explored for Rust in Crubit [19].

The benefit of this approach is reduced cost and improved security while avoiding a total rewrite of the code. The downside of this approach is that deterministic safety across all memory safety categories is not possible until all code is rewritten.

For longer discussion about a practical approach to these issues please see the Internet Security Research Group (ISRG) project [20] and Prossimo [21].

A *complete* rewrite is just that, a fresh start where the developers reimplement the functionality of the old code in a memory safe language. This gives the developers an opportunity to make large changes that may not be possible in an incremental approach, but also can be more expensive if you must also continue to maintain the old codebase.

There is evidence that the incremental approach of code rewrite is the most viable for companies with large legacy codebases. Google has observed a steady drop of memory safety vulnerabilities in the Android operating system as they gradually replace their codebase with memory safe languages. [22]

In particular, safe, performant and ergonomic interoperability is a key ingredient for an incremental approach. Both Android and Apple are following a transition strategy centered around interoperability, with Rust and Swift [23] respectively.

For software developers who cannot wholesale replace existing codebases with an MSSL it is possible to use existing C/C++ static and runtime technologies to address specific bug classes. Here is an example of a roadmap to address the majority of memory safety issues within existing native code:

- Uninitialized Use can be mitigated in C/C++ code with automatic initialized technologies [24].
- Heap Corruption can be mitigated using MTE or other memory tagging approaches for spatial safety.
- Type Confusion can be partially mitigated with technologies such as CastGuard [25] for preventing unsafe downcasts.

- Heap OOB can be improved by performing Array Access through gsl::span which guarantees memory access will occur within safe bounds.

For MSSL by replacing components one-by-one, security improvements are delivered continuously instead of all at once at the end of a long rewrite. Note that a full rewrite may eventually be achieved with this incremental strategy, but without the risks typically associated with complete rewrites of large systems. Indeed, during that time, the system remains a single code base, continuously tested and shippable.

Costs and Deployability

One of the criteria in the viability analysis is deployability of an incremental mitigation, which in this context means it prevents an entire class of memory safety issues, while not requiring a full rewrite of existing native (C/C++) code. Solutions which require partial changes, but full recompilation are considered deployable. In addition, the mitigation must also be viable from a performance and compatibility perspective. An example of deployable C/C++ memory safety mitigation is automatic initialization of stack-based variables [26]. This mitigation deterministically prevents a common class of memory safety issues called uninitialized stack variables [27]. This is a compiler-based mitigation that can be enabled for an existing code base without a rewrite. This approach also has low to neutral performance overhead and no known compatibility issues since the behavior of uninitialized variables is undefined.

Conversely, an example of a class of issues without a known deployable solution is the category of temporal safety. The known approaches for addressing temporal issues in C/C++ code require garbage collection which lacks a generic, system wide approach. This would require developers to implement bespoke solutions to existing code that would require a significant rewrite and potentially introduce substantial overhead. A deployable temporal safety solution for existing code is an open problem.

Safer languages subsets such as Apple's Firebloom [28], C++ GSL::Span [29] or Microsoft's Checked C [30] offer memory safety to existing systems languages like C and C++. Safer language subsets can target both spatial (bounds constrained) and temporal vulnerabilities (garbage collection) and can be applied to existing code bases without a complete rewrite and arguably better performance. Unfortunately, they offer less complete coverage of memory safety issues. These implementations also have significant cost in both rewrite and performance overhead.

Another proposed approach to adding memory safety to non-memory-safe languages is to use formal verification techniques. The core concept is to make logical proofs of the correctness of software by modeling the software as a set of logic statements that can be formally proven by automated systems. Formal verification techniques can theoretically provide memory safety for non-memory-safe languages but also are able to prove other kinds of correctness properties that can increase the quality of security-sensitive software. To be effective, however, these tools and techniques must be integrated into the daily workflow of the engineering team working on the software so that all code changes are analyzed, and any bugs identified prior to next steps such as compilation. Furthermore, scaling these techniques to entire code bases, and ensuring sound detection of memory safety vulnerabilities remains an open research problem and, thus, these techniques may not be applicable in many use-cases without further research. [31]

Finally, there are Architectural improvements such as CPU based technologies like CHERI [32], ARM [33] MTE, Intel Memory Tagging [34]. These CPU based architectural improvements range from complete implementations (CHERI) to non-deterministic targeted approaches (MTE). CPU based implementations have the general property of being high-performance and can be applied to existing code bases (CHERI Linux/BSD). Unfortunately, they require new hardware which varies across vendor implementations. It also has the high cost of allocator and possible key API rewrites and ultimately may have no deterministic guarantees.

General Conclusions

The TAC believes that memory safe technologies exist to address multiple use cases. From performance constrained

embedded devices to general use systems and applications, memory safe hardware and language options now exist where they may not have in the past. There is now a path to address existing legacy needs while simultaneously developing new components in safer languages.

Workforce development and education will play a key role in the adoption of skills that are needed for the long-term transition to memory safe technologies. In cases where performance and form factor constraints are not an issue there are many more existing options today with commonly used system and application-level languages. In cases where performance and form factor constraints exist and room for change is limited, the hardware and performance-oriented systems-based language solutions are currently the only options.

Different ecosystems and form factors will require different short-term, mid-term and long-term solutions depending on performance, manpower, and regulatory needs. In some cases, implementing memory safe technologies with existing languages may expedite improved memory safety in the near term.

Timeline	Embedded Devices & RTOS	General Computing OS and apps
Short-Term 1-2 Years	• Build a Memory Safe Roadmap • Use safer libraries and extensions in C and C++ • Use MSL verification tools that can be run as part of the build step [35] • Conduct performance cost comparisons between safe features of legacy languages (e.g. C/C++) vs. MSSL	• Build a Memory Safe Roadmap • Use safer libraries and extensions in C and C++ • Use MSL verification tools that can be run as part of the build step [35] • Begin to build the necessary toolchains and integrations
Mid-Term 3-5 Years	• Write new code in MSSL that support embedded use cases (e.g. Rust) • Adopt hardware that has memory safe capabilities (CHERI/MTE/etc.)	• Use MSSL for all new projects where appropriate • Incrementally rewrite the most critical code in a MSSL • Incorporate memory safety testing tools in the build process • All major CPU vendors support memory tagging
Long-Term 5+ Years	• The MSSL toolchain is readily adoptable by all software development teams, all major IDEs and static tools have first-class support for MSSL • Replace existing RTOS with memory safe RTOS such as Tock [36] • Implement a plan to upgrade or replace old and legacy EOL products with memory safe products	• The MSSL toolchain is readily adoptable by all software development teams, all major IDEs and static tools have first-class support for MSSL • Companies use MSSL for all new code with attack surface • All major OS system allocators support memory tagging and readily available for use for legacy C/C++ code

CISA Questions

Broadly speaking, to achieve the objectives implicit in the six questions posed by CISA, there are six major themes our recommendations fall under: Collaboration with a broad ecosystem of strategic partners, inform and influence standards, create and promote tools, academic and education engagement, create pilot programs, and instill a memory safety mindset in the procurement process. The recommendations were informed by a series of briefings involving participants from various cybersecurity communities.

Question 1: *How can CISA help technology manufacturers (including open-source projects) migrate toward using memory-safe code as a default?*

Discussion:

Making the decision to switch to a MSSL requires that all the prerequisites are in place for a successful transition. A decision maker may wish to use a MSSL, but still needs to make sure all the current practices and tooling they currently use support it. That could mean a company's build, test, deployment, and production environments all have the required toolchains and integrations required to replace the current solutions. Essentially the build environment needs to be mature enough for a switch to occur. Because of the many moving parts there are opportunities for CISA to identify and improve slowdowns or roadblocks to MSSL adoption.

CISA, as part of its Secure By Design initiative [37] has started publishing guidance for "Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Secure by Design Software" and released its first document in October 2023 that includes recommendations for prioritizing the use of MSSL. [38] This initiative covers more than just the use of MSSL and has three key principles: (1) Taking ownership of their security outcomes, (2) Adopting radical transparency, and (3) Taking a top-down approach to developing secure products.

Findings:

Technology creators need stronger market signals, including from the government, to help them understand why enabling and migrating to MSSL is vital to the safety and security of society. Incentives and consistent messaging across all areas of government that this issue needs to become a requirement would be one way of showing the market that the use of MSSL will be prioritized.

The Secure By Design initiative has recommendations for companies to develop and adopt a memory safe roadmap that includes:

- Defined phases with dates and outcomes: Evaluation of memory safe programming languages, a pilot phase to test writing a new component in an MSL or incorporating MSL into an existing component, threat modeling to find the most dangerous memory unsafe code, and refactoring memory unsafe code.
- A date for memory safe programming language use in new systems.
- Internal developer training and integration plan, an external dependency plan, a transparency plan and developing a CVE support program plan.

Example roadmaps can act as templates for companies, which would help accelerate the decision-making process and speed MSSL adoption.

CISA has a GitHub account [39] and should use it to greater effect to publish resources related to the adoption of MSSL and the Secure By Design initiative.

Recommendations:

1. Work with the R&D ecosystem across government and industry to create and update tools that enable usage of memory safe features of existing languages and hardware.
2. Add memory safe tools to CISA's existing GitHub to highlight open-source solutions.
3. Advocate for memory safety in Computer Science, Embedded Systems and Engineering education curricula, for example by maintaining a list of university curricula that include memory safety coursework as well as incentivizing ongoing education of memory safe languages.
4. Continue to develop and expand the existing CISA "Secure By Design" initiative, publishing memory safe migration roadmaps and supporting materials, including cost/benefit analysis to help inform company transition plans.
5. Conduct performance studies and comparison between MSSL vs. legacy languages to help answer performance and cost concerns within embedded device communities.

Question 2: *What key partnerships should CISA forge to promote memory safety?*

Discussion:

CISA can leverage its unique role sitting at the nexus between commercial, civil-society, regulatory, non-regulatory, law enforcement, military and intelligence organizations to drive a consistent message and tooling to accelerate manufacturers towards safer and more secure practices.

Within the government, CISA can encourage regulatory partners to address memory safety as an imperative in their public notices, which can help shape dialogue and market forces to prioritize this issue across multiple sectors.

Within industry, CISA can work with major hardware, firmware and software manufacturing companies to help drive them towards increased memory safety practices. Those improvements can then be adopted by the rest of their supply chain.

Finally, partnerships with industry and open-source projects developing memory safe technologies and tools will help build relationships that can be used to better provide advice when producing the types of documents suggested by this report and the Secure By Design initiative.

Findings:

Building trust and increasing collaboration will be necessary ingredients for better policy outcomes with the various manufactures, associations, and standards bodies and could shift the landscape in a memory safe direction at scale. The capacity and resources of such a collaborative group could address market, technology, supply chain, regulatory and policy issues more comprehensively than addressing them all individually.

A memory safety council, populated by stakeholders throughout the ecosystems, could provide a useful place to help coordinate these efforts. Similar projects have been done in the past via the Enduring Security Framework (ESF) [40] to get TPM chips in most general computing products for example. Critical infrastructure sectors have the Critical Infrastructure Partnership Advisory Council (CIPAC) that helped drive sector councils, national response strategies, information sharing and improvements to infrastructure security practices across all critical infrastructure sectors. Outreach could start with the Prossimo Project, and other projects currently building operating systems entirely in Rust [41]. Their experience and insights will be invaluable.

CISA, working with universities and other research institutions, to foster investment in research and development on temporal protections could lead to new techniques to resolve the missing path forward for temporal safety. Finally, there needs to be a pipeline of programmers ready and able to work with memory safe technologies. CISA can work with universities and academics to determine the need for further educational programs that will increase the base of programmers able to work on memory safe projects and transition legacy code.

Recommendations:

1. Create a memory safety council that invites both embedded and general computing stakeholders and educators to the table. Should include silicon, RTOS and general computing stakeholders to ensure coverage of both ICS OT and IT ecosystems.
2. Encourage industry standards groups to take on memory safety standardization efforts. One example could be funding research projects with legacy unsafe language standards groups such as C and C++ to update standards and tools to default to memory safe features.
3. As noted in Question 1, Recommendation 3, advocate for memory safety in education curricula.
4. Advocate within government for funding to help support the foundations that support key MSSL projects, such as the Rust Foundation.

Question 3: *Are there areas where the Federal government is holding adoption of memory-safe programming languages back? If so, how can CISA help address these areas?*

Discussion:

Collaborate with regulatory agencies to ensure their efforts are not preventing regulated markets from migrating to memory safe technologies due to regulatory hurdles to make major changes to operational environments. Oftentimes some industries are reluctant to make major changes, such as regulated critical infrastructure sectors, because of potential impacts to or conflicts with existing or changing regulations from government regulatory agencies.

CISA could leverage cross agency forums to help guide regulatory agencies in putting out memory safety friendly orders, notices and public position statements that encourage or provide guidance to regulatory sectors. Furthermore, government agency procurement practices could slow down or have conflicting requirements that could negatively impact or potentially deter adoption of memory safe technologies on existing or near future projects.

Findings:

In the past CISA collaborated with the National Labs, MITRE, JHU APL and others to help drive the Operating Technology market towards creating specific extensions of existing IT defensive tools to support OT.

CISA also has had success in working with this same ecosystem to push for SBOM tools and standards. [42]

Taking the lessons learned for these two past efforts, and others, CISA could collaborate with the Federally Funded Research & Development Centers (FFRDCs) ecosystems such as the national labs and MITRE to create and release tools that enable the usage of safer libraries and functions in existing legacy unsafe languages such as C and C++. Portability tools could be created to convert libraries and functions of unsafe languages into memory safe language code such as migrating C language developed drivers and compilers to Rust. These tools should then be released to the public to expedite the open-source community's ability to adopt MSSL faster. CISA has produced guidance for SBOM and tools plus guidance for ICS OT before leveraging the FFRDC community and we are confident there is a role for CISA to play in memory safe technologies as well.

Recommendations:

1. Work to ensure that independent regulatory agencies like FERC, FCC, FTC, FDA, EPA adopt cyber regulations that enable and do not degrade memory safety efforts throughout the supply chain. This may involve assisting in reviewing existing regulations for items that may run counter to the adoption of MSSLs.
2. Work with government standards organizations to develop standards that include the use of memory safety technologies and assist in reviewing existing standards to identify any that run counter to memory safety adoption.
3. Ensure existing FFRDC efforts implement CISA memory safety recommendations and practices and produce or

update tools to further enable memory safe adoption.

Question 4: *How should CISA factor memory safety into federal government procurement processes?*

Discussion:

The federal procurement process provides an opportunity to promote memory safety in government systems, as well as provide a market incentive to developers. Federal Acquisition Regulations (FAR), Office of Management and Budget (OMB) circulars, agency specific directives and executive orders drive much of the existing procurement practices of the federal government.

CISA should use its relationships with federal civilian agencies and systems, in collaboration with DoD, DHS Procurement leadership, OMB and the Office of the National Cyber Director (ONCD), to drive memory safety requirements into improved procurement practices and secure by design and secure by default requirements that will be implemented, which would align well with the 2023 National Cybersecurity Strategy Implementation Plan.

CISA can recommend to Federal procurement councils and decision makers that memory safety requirements, developed using multi-stakeholder public-private partnerships, are included in all the updated cybersecurity requirements to improve federal procurement practices. This will send a significant market signal that this is an imperative, and a requirement, not an option. This is similar to the current government efforts for requiring SBOM or government Cloud security requirements being enforced via FedRAMP.

It is important to have a whole of government approach to make this an effective strategy. If a few agencies insist on memory safe code, while most do not, a supplier could decide that it simply is not a large enough market to put in the effort to improve memory safety. Likewise, this may require discussions with software developers to develop a realistic roadmap, and close attention to reducing the cost of transitions. As discussed in this report, cost of transition creates friction and the procurement tool will work best in combination with efforts to reduce friction and cost.

Findings:

The federal government has tremendous purchasing power, and thus its procurement is a powerful tool to align commercial incentives with the public need to enhance memory safety in our nation's systems. The ability to sell to the government may change the equation when a product vendor or software developer is considering the costs and benefits of implementing memory safe hardware, firmware and software. Once developed, that improved technology will be available to other customers lowering the barriers for others to enter the market.

To this end, CISA should collaborate with ONCD and OMB to study requiring government contractors to use memory safe products including hardware and programming languages, as well as memory safe functions in unsafe languages, for all updates to old products provided to the government, as well as any new products. Making changes to FAR and collaborating with DoD on DFAR would drive requirements in federal acquisition regulatory requirements. In addition, implementing an SBOM requirement to identify where memory-unsafe code may be included in a technology project.

***Recommendations*:**

1. CISA should make recommendations to DHS and OMB procurement councils to include piloting memory safe product requirements in their cyber security purchasing requirements for providing products and services to all federal agencies.
2. Collaborate with NIST for the consideration of memory safety updates to the existing requirements in NIST SP 800-53 Security and Privacy Controls for Information Systems and Organizations, SP 800-161 Cybersecurity Supply Chain Risk Management Practices for Systems and Organizations, SP 800-218 Secure Software Development Framework (SSDF) and similar guidelines.
3. CISA should recommend to DHS S&T to fund pilot projects to help industry create memory safe tools and

products to increase the availability of memory safe products on the market.

Question 5: *What are the major problems that CISA can advise the technology industry to solve to make all products memory safe? Examples might include prioritizing investment decisions to move towards memory-safe code, ensuring the next generation of computer scientists train in memory safe languages, and understanding software supply chains (often open-source) that often include memory-unsafe code.*

Discussion:

Existing solutions that can address multiple memory safety issues exist. However, not all existing solutions address all memory safety issues at scale. Some problems can be solved in hardware and others can currently only be solved by programming languages. Spatial safety is generally believed to be scalable with existing solutions.

However, there are no scalable, system-wide protection mechanisms or clear paths forward for temporal memory protection for legacy code without complete software rewrites in MSSLs. This is exacerbated by complex applications with unique methods for managing their object lifetimes, complicating adoption of temporal safety retrofitting. Microsoft's memGC for Edge would be an example of implementing bespoke garbage collectors in C/C++.

Memory tagging is an example of an available and impactful tool against memory safety issues. The introduction of tagging via MTE into mainstream mobile phones such as the Pixel 8 are a concrete demonstration that this technology can be scalably adopted.

There are past successful efforts CISA can be informed by and emulate for the adoption of MSSL. For example, the President's National Strategy to Secure Cyberspace (2003) [43] clearly stated the need to "secure the mechanisms of the Internet by improving protocols and routing."

To meet this need DHS S&T, created the DNSSEC Deployment Initiative. "DNSSEC has been developed to provide cryptographic support for domain name system (DNS) data integrity and authenticity. DHS sponsors a community-based, international effort to transition the current state of DNSSEC to large-scale global deployment, including sponsorship of the DNSSEC Deployment Working Group, a group of experts active in the development or deployment of DNSSEC. It is open for anyone interested in participation." [44]

This work is credited with saving 5-10 years of DNSSEC adoption time. Doing the same for MSSL now could yield equally large benefits.

Findings:

There are many edge cases and outstanding issues that can benefit from additional study. Working with the MSSL communities CISA can act to catalog these issues, priorities, and fund work to accelerate solutions and adoption much like the DHS S&T DNSSEC example above.

- Further academic and industry research would help determine what is technically and theoretically possible. In particular, answering the following questions would help resolve or mitigate the issues with temporal or spatial safety, and determine which effort will be most effective. Does adding temporal protections to hardware involve many incremental and dedicated CPU features that when combined with compilers grant a level of temporal safety?
- Are certain temporal problems ripe for hardware runtime optimizations, or must they all be mitigated with no hardware assistance?
- What work has been done in this space? What were the outcomes?

Moving forward, all major silicon vendors and architectures should accelerate plans to include tagging in silicon. OS and

application vendors should follow suit and commit to adopting these capabilities. Accelerated adoption of memory tagging and other mitigations by hardware manufacturers would help mitigate memory safety risks, especially for those which have a clear path forward in hardware. In software, stopgap measures can be adopted. For example, C++ developers should default to GSL::SPAN, ASan, Cast Guard, and other appropriate tools to gain memory bounds checking enforcement

For Rust there remain open questions which additional investment would help resolve, for example, when integrating an unsafe language. How to guarantee C++ code does not violate Rust code's exclusivity rule, which would create new forms of undefined behaviors [45][46].

Finally, MSSLs are being introduced into the software development lifecycle without a commensurate effort in earlier Computer Science education. There is very little coursework dealing with memory safety in academic institutions, and memory safety is not included in accreditation criteria for Computer Science.

Recommendations:

1. CISA should request the inclusion of memory tagging technologies into the roadmap of all major silicon vendors powering cloud, pc, mobile, and IoT devices leveraged in national infrastructure.
2. Request compiler and IDE vendors, both open-source and commercial, to default to secure options such as Span, InitiAll, and Castguard for C/C++ compilers and common libraries (C/C++ Standards).
3. Collaborate and fund workforce projects with academia and open-source communities for migrating to MSSL, and related tools necessary to implement memory safety features of legacy languages in C and C++ libraries.
4. Encourage accreditation bodies to include memory safety concepts in Computer Science and Systems Engineering degree program guidelines (This complements the education Recommendations in Question 1 and 2).
5. Help identify and fund or study outstanding issues slowing adoption in MSSL.

Question 6: How do we ensure that CISA's memory safety guidance gets traction beyond technical leaders working at software manufacturers, including with business leaders and other U.S. government agencies?

Discussion:

In our discussions with the technical community, a universal observation was that there is a need for memory safety. The community recognized that failure to address memory safety, in products and in the supply chain, could lead to production downtime, reputation hits to business or direct impacts to product quality and safety for customers. In some cases, these issues may also involve product recalls, impact to stock prices and regulatory fines or legal proceedings against their business.

The primary challenge is getting executive business leaders to prioritize and execute on a transition to memory safety. Thus, the technical community needs assistance in amplifying the importance of memory safety, so that business leaders will appropriately weigh the need to move to MSSL sooner rather than later when balancing competing priorities. This will include reaching out to startups, venture capital and incubators to help them understand the value of transitioning to memory safety in existing projects, as well as including memory safety in a Secure by Design new project.

In addition to a strong message on the importance of memory safety, executive decision makers will need the information necessary for them to have the confidence that a transition to memory safe code is a good decision. This information will include implementation studies and lessons learned from transitioning a project into a MSSL. In addition, decision makers could use research papers and other documentation that shows that an investment in memory safe code is fiscally responsible, including helping show the impact that technology, suppliers, providers and partners can have on their business and customers.

Furthermore, the academic, research and standards communities could help by adapting existing research and standards towards enabling the use of memory safe technologies. R&D funded projects and international engineering standards need to enforce and endorse memory safe technologies and practices.

Findings:

CISA can play an important role in helping the technical community persuade business leaders and non-technical executives to prioritize memory safety. First, CISA, in coordination with other government agencies, should help provide a strong signal highlighting the importance of memory safety, which will amplify the voices within the technical community, and help get key non-technical decision makers to give those voices more credence. This will help distinguish the memory safety effort from all the other initiatives competing for attention.

Second, CISA can work with industry and academia to help encourage the development of white papers, research reports and other documents that can provide examples and analysis which the technical community can use to show decision makers that an investment in memory safety is fiscally responsible. CISA could also provide a central repository to make these documents more easily available.

Third, CISA should leverage existing sector leadership coordination forums to drive home that memory safety needs to be considered in third party risks, just as with the software bill of materials efforts. The additional transparency provided by efforts such as the SBOM can help identify what components use memory safe technologies, their versions, and settings.

Finally, key efforts CISA will need to focus on with executives include:

- Shaping the narrative so that executives understand why this is important and not just a technical problem.
- Pointing out how addressing the attack surface can decrease business disruptions.
- Pointing out the supply chain risk to their businesses and industries.
- Educate executives that memory safe requirements and questions should be added to existing third party risk. questionnaires, practices, policies, and procedures as additional third-party risk line items needing to be accounted for.

Recommendations:

1. CISA, in coordination with others, should publicly signal to decision makers that memory safety is important, and decision makers should pay attention, through actions such as those identified in this report.
2. Encourage the development of white papers, research reports, and supporting documentation to assist decision makers in having the information necessary to justify investing in memory safety and provide a public repository for this documentation.
3. Educate executives in existing critical infrastructure sector coordinating councils on the need for MSSL, including in the supply chain.
4. Update the existing CISA SBOM and HBOM guidance to require disclosure of details (all, some, or none as an example) if each component was developed with MSL or technologies to better inform consumers of product capabilities.
5. CISA should recommend that incentives like legal safe harbors for following best security practices that can help encourage industry to move toward memory safety, be included in legal and regulatory decisions that may adopt best security practices. CISA should advise legal and regulatory agencies to be sure that these practices are standards agnostic, future focused, and do not create perverse incentives, such as abandoning software projects to avoid potential liability.
6. As part of the Secure By Design initiative create or support a public tracker that lists what important software is available in a MSSL to create public awareness and peer pressure.
7. Include the academic community and connect them with industry and government through efforts such as sponsoring workshops on memory safety with the top academic conferences.

Appendix A: Referenced in the Document

1. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/
2. https://www.chromium.org/Home/chromium-security/memory-safety/
3. https://www.zero-day.cz/
4. https://en.wikipedia.org/wiki/Morris_worm
5. 2021 has broken the record for zero-day hacking attacks | MIT Technology Review
6. https://www.infosecurity-magazine.com/news/mwise-zero-days-highest-year-record/
7. https://www.tiobe.com/tiobe-index/
8. https://octoverse.github.com/2022/top-programming-languages
9. https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/
10. https://cwe.mitre.org/
11. https://google.github.io/tcmalloc/gwp-asan.html
12. https://www.chromium.org/Home/chromium-security/articles/gwp-asan/
13. https://source.android.com/docs/security/test/memory-safety/arm-mte
14. https://www.cl.cam.ac.uk/~tmj32/papers/docs/ainsworth20-sp.pdf
15. https://googleprojectzero.blogspot.com/2023/11/first-handset-with-mte-on-market.html
16. https://github.com/apple-oss-distributions/xnu/blob/1031c584a5e37aff177559b9f69dbd3c8c3fd30a/osfmk/vm/vm_memtag.h#L91
17. https://security.googleblog.com/2021/06/rustc-interop-in-android-platform.html
18. https://www.swift.org/documentation/cxx-interop/
19. https://github.com/google/crubit/blob/main/docs/design.md#crust-interop-requirements
20. https://www.abetterinternet.org/
21. https://www.memorysafety.org/about/
22. https://security.googleblog.com/2022/05/retrofitting-temporal-memory-safety-on-c.html
23. https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html
24. https://msrc.microsoft.com/blog/2020/05/solving-uninitialized-stack-memory-on-windows/
25. https://i.blackhat.com/USA-22/Thursday/US-22-Bialek-CastGuard.pdf
26. https://reviews.llvm.org/D54604
27. https://cwe.mitre.org/data/definitions/457.html
28. https://saaramar.github.io/iBoot_firebloom/
29. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2771r0.html
30. https://github.com/microsoft/checkedc
31. https://onlinelibrary.wiley.com/doi/10.1002/spe.2949
32. https://www.arm.com/architecture/cpu/morello
33. https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/enhancing-memory-safety
34. https://d1io3yog0oux5.cloudfront.net/_6f1902c731ed10bd8538c1c8c9ca7ca1/intel/db/861/8422/pdf/Intel-Architecture-Day-2020-Presentation-Slides.pdf
35. https://github.com/diffblue/cbmc/
36. https://tockos.org/
37. https://www.cisa.gov/resources-tools/resources/secure-by-design
38. https://www.cisa.gov/sites/default/files/2023-10/SecureByDesign_1025_508c.pdf
39. https://github.com/cisagov
40. Enduring Security Framework ESF (nsa.gov)
41. https://github.com/flosse/rust-os-comparison
42. https://www.cisa.gov/sbom
43. https://www.dhs.gov/science-and-technology/csd-sp
44. https://www.dhs.gov/xlibrary/assets/pso_cat_st.pdf
45. https://dl.acm.org/doi/pdf/10.1145/3428204
46. https://cs.stanford.edu/~aozdemir/blog/unsafe-rust-syntax/

Acknowledgements

Technical Advisory Council Subcommittee Members:

Mr. Jeff Moss, Subcommittee Chair, DEF CON Communications
Mr. Dino Dai Zovi, CashApp
Mr. Luiz Eduardo, Aruba Threat Labs
Mr. Royal Hansen, Google
Mr. Isiah Jones, Applied Integrated Technologies
Mr. Kurt Opsahl, Electronic Frontier Foundation
Mr. Stephen Schmidt, Amazon
Mr. Yan Shoshitaishvili, Arizona State University
Mr. Kevin Tierney, General Motors
Ms. Rachel Tobac, SocialProof Security
Mr. David Weston, Microsoft