RUNSAFE SECURITY

Safeguarding Code: A Comprehensive Guide to Addressing the Memory Safety Crisis

RunSafeSecurity.com

Table of Contents

An Introduction to Memory Safety

Memory safety is a foundational aspect of software development, referring to the protection of memory access operations against common vulnerabilities like buffer overflows, dangling pointers, and memory leaks. But while memory safety is a well-understood problem by both software developers and policymakers alike, memory safety vulnerabilities have persisted across the decades, from the infamous Morris Worm of 1988—one of the first major cyber incidents—that exploited a buffer overflow vulnerability to the present-day Volt Typhoon campaign leveraging memory safety vulnerabilities to target critical infrastructure.

Memory safety vulnerabilities remain an issue for four reasons:

1. **Legacy Codebase Dependency:** Critical infrastructure relies heavily on systems written in memory-unsafe languages, primarily C and C++. The codebase for a typical industrial control system can span millions of lines, making comprehensive rewrites into memory-safe languages like Rust, Go, or Java impractical.
2. **Performance Constraints:** Memory-safe languages often introduce runtime overhead that may be unacceptable for performance-critical systems, particularly in embedded environments with limited resources.
3. **Complexity of Modern Applications:** Modern software applications are increasingly complex and distributed across diverse environments (cloud, edge, IoT). Managing memory safety across these environments while ensuring interoperability and performance remains a challenge.
4. **Sophisticated Exploitation Techniques:** Cyber adversaries continually evolve their techniques to exploit memory safety vulnerabilities. Zero-day exploits targeting memory flaws can have severe consequences if not promptly identified and mitigated.

Despite the challenges, memory safety vulnerabilities must be addressed. In 2022, the National Security Agency (NSA) issued guidance emphasizing the severity of memory safety vulnerabilities, stating that they remain the most readily exploitable category of software flaws. Additionally, MITRE's Common Weakness Enumeration (CWE) consistently ranks memory corruption vulnerabilities among the top 25 most dangerous software weaknesses.

Memory safety vulnerabilities in particular pose a substantial and pressing threat to embedded software deployed across critical infrastructure, the automotive industry, medical devices, and more. Malicious actors can exploit these vulnerabilities to execute arbitrary code, compromise sensitive data, or cause system crashes. When we think about these attacks in the context of the energy grid, defense systems, and transportation, it's clear that it's time to address the memory safety crisis once and for all.

This guide is designed to empower software developers, product managers, and security professionals with the knowledge and tools needed to address memory safety challenges to protect embedded systems today and into the future.

We'll explore:

- Strategies for transitioning to memory-safe architectures
- Approaches for securing legacy systems
- The role of automated tools and deploying protection at runtime to prevent memory corruption

Mitigating Memory Safety Vulnerabilities: Where to Start

In 2023, CISA and international cyber agencies published guidance as part of their Secure by Design campaign on steps to take to address memory safety vulnerabilities, including using memory safe programming languages and writing and publishing memory safe roadmaps to "eliminate this class of vulnerability."

Following Secure by Design practices is an excellent approach, addressing memory safety issues throughout the development process rather than bolting on security measures after the fact. The goal is to reduce the likelihood of vulnerabilities by making security an inherent part of the system architecture.

In the case of memory safety, Secure by Design often involves transitioning to memory-safe languages like Rust and Go, which can drastically reduce the risk of memory-related bugs. However, transitioning entire codebases to these languages, which can take years and requires a significant amount of resources, is impractical for legacy systems and critical infrastructure. In the interim, immediate solutions are needed to bridge the gap.

Organizations are adopting various approaches to address memory safety:

1. **Selective Rewriting:** Identifying and rewriting critical components in memory-safe languages like Rust.
2. **Runtime Protection:** Implementing Address Space Layout Randomization (ASLR), Control Flow Integrity (CFI), and emerging solutions like Load-Time Function Randomization (LFR).
3. **Static and Dynamic Analysis:** Employing advanced static analysis tools to identify potential memory safety violations during development.

In the next sections, we'll address the practicalities of transitioning to memory-safe architectures and approaches to securing legacy systems without code rewrites.

Examples of Memory Safety Vulnerabilities

BUFFER OVERFLOWS

A buffer overflow occurs when a program writes more data to a buffer than it can hold, leading to adjacent memory being overwritten. This can cause unexpected behavior and provide an attack vector for malicious code execution.

USE-AFTER-FREE ERRORS

Use-After-Free errors occur when a program continues to use a pointer after the memory it references has been freed. This can lead to arbitrary code execution, data corruption, or system crashes.

MEMORY LEAKS

Memory leaks occur when a program fails to release memory that is no longer needed, leading to gradual memory consumption and potential system slowdowns or crashes.

DANGLING POINTERS

Dangling pointers arise when pointers are left pointing to memory locations that have been deallocated, potentially leading to access of invalid memory.

Adopting Secure Code Standards and Memory Safe Languages

Adopting strict coding standards, leveraging memory-safe programming languages, and strategically migrating legacy code are vital steps to take toward preventing memory safety vulnerabilities and securing software systems. This section outlines these best practices.

Secure Code Standards and Guidelines

Adhering to coding standards and guidelines is fundamental to writing secure code. These practices help ensure consistency, readability, and, most importantly, security in software development.

Input validation

Always validate and sanitize input data to prevent buffer overflow and injection attacks. Use libraries and frameworks that provide built-in input validation mechanisms.

Memory management

Manage memory explicitly and avoid common pitfalls like buffer overflows, dangling pointers, and memory leaks. Use static analysis tools to detect and correct memory management issues early in the development cycle.

Code reviews

Conduct regular code reviews to catch potential vulnerabilities and enforce coding standards. Incorporate security-focused code review checklists and involve multiple reviewers with different expertise.

Use of libraries

Only use well-maintained and widely-used libraries that follow security best practices. Keep dependencies updated to incorporate the latest security patches.

Memory-Safe Programming Languages

Memory-safe programming languages are designed to prevent common memory-related errors, reducing the risk of vulnerabilities and exploits. Below are a few of the most well-known ones.

Rust

Rust is known for its robust memory safety features, enforced through its ownership model and type system. Rust guarantees memory safety without a garbage collector, provides high performance, and supports concurrency.

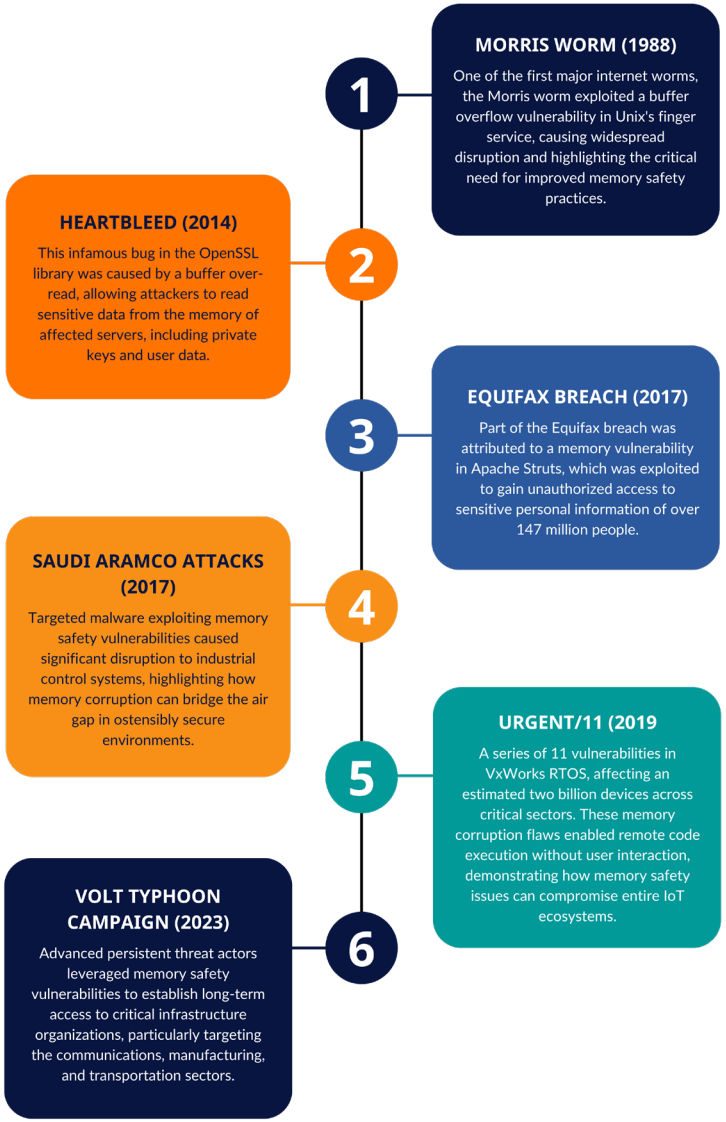
Go

Go simplifies memory management with garbage collection and strong typing. Go offers fast compilation, ease of use, and built-in concurrency support which makes it ideal for scalable applications.

Swift

Swift offers automatic memory management through reference counting, designed for developing iOS and macOS applications. Benefits include high performance, safety features like optionals to handle null values, and an expressive syntax.

From Vulnerabilities to Weaponized Exploits: Case Studies of Significant Memory Safety Breaches



Tips for Migrating Legacy Code to Memory-Safe Languages

Migrating legacy code to memory-safe languages can be complex but offers long-term security benefits. Below are some strategies to facilitate this process.

Assess and plan

Evaluate the existing codebase to identify critical components and dependencies. Develop a detailed migration plan that includes timelines, resource allocation, and risk mitigation strategies. Use static analysis tools to map out the code structure and identify areas needing significant changes.

Incremental migration

Break down the migration process into smaller, manageable phases. Start with critical modules that benefit most from enhanced memory safety. Establish clear milestones and continuously test each migrated component to ensure functionality and security.

Use automated refactoring tools

Use automated tools to assist in refactoring code, which can help in translating legacy code into a memory-safe language. Combine these automated tools with manual code reviews to ensure the accuracy and security of the migrated code.

When Code Rewrites Are Impractical: Securing Legacy Systems Today

While organizations work toward the long-term goal of transitioning to memory-safe languages, adopting effective patching strategies, implementing runtime protection measures, and establishing continuous monitoring and vulnerability management processes all support in reducing the attack surface. This section outlines best practices to enhance the security of existing systems.

Strategies for Patching and Updating Software

Conduct regular patch management
Develop and adhere to a regular patch management schedule to ensure that all software components are up-to-date with the latest security patches. Automate the patching process using tools like WSUS (Windows Server Update Services) or third-party patch management solutions to ensure timely updates.

Prioritize patches based on severity
Evaluate the severity and exploitability of vulnerabilities to prioritize patch deployment. Use vulnerability scoring systems like CVSS (Common Vulnerability Scoring System) to assess risk levels and prioritize accordingly.

Test patches before deployment
Conduct thorough testing of patches in a staging environment to identify potential issues before rolling them out to production. Use automated testing tools to simulate patch deployment and detect any adverse effects on system stability and functionality.

Implementing Runtime Protection Measures

Integrate Runtime Application Self-Protection (RASP)
Integrate RASP solutions to monitor and protect applications during runtime by detecting and mitigating attacks in real-time. Be sure to choose RASP tools that provide visibility into application behavior and can dynamically adapt to new threats.

Enable Data Execution Prevention (DEP)
Enable DEP to prevent the execution of code from non-executable memory regions, reducing the risk of certain types of attacks. Ensure that DEP is configured correctly and consistently across all systems and applications.

Implement Address Space Layout Randomization (ASLR)
Implement ASLR to randomize memory addresses used by system and application processes, making it harder for attackers to predict target addresses. Verify that ASLR is enabled and functioning correctly on all supported systems.

Load-Time Function Randomization (LFR)
Adopt Load Time Function Randomization to protect against both known and unknown vulnerabilities at runtime. LFR randomizes code layouts in memory, preventing attackers from predicting where code is located, making it significantly harder to exploit memory vulnerabilities like buffer overflow attacks and Return-Oriented Programming (ROP).

Control Flow Integrity (CFI)
Implement Control Flow Integrity (CFI) to prevent attackers from hijacking a program's execution flow, making it much harder for malicious actors to redirect program execution to arbitrary code locations.

Implementing Continuous Monitoring and Vulnerability Management Strategies

This section explores essential strategies for effective patching and updating of software, as well as critical runtime protection measures to enhance application security and resilience against threats.

Implement automated vulnerability scanning
Employ automated tools to continuously scan for vulnerabilities in the system, network, and applications. Use comprehensive vulnerability management platforms like Qualys or Nessus to schedule regular scans and receive real-time alerts.

Deploy Security Information and Event Management (SIEM)
Deploy SIEM solutions to collect, analyze, and respond to security events and incidents across the organization. Integrate SIEM with other security tools to create a cohesive incident response strategy and ensure that it can correlate events from multiple sources for better threat detection.

Conduct regular security audits and penetration testing
Conduct regular security audits and penetration tests to identify and remediate vulnerabilities that may have been missed by automated tools. Go the extra step by partnering with external security experts to perform thorough assessments and provide an objective evaluation of the system's security posture.

Tools and Techniques for Detecting Memory Safety Issues

Static vs. Dynamic Analysis Tools

Both static and dynamic analysis tools play a significant role in identifying vulnerabilities early in the development cycle.

Static analysis tools analyze code without executing it. They are excellent for detecting potential memory safety issues by examining code patterns and syntax.

Examples include:

- **Clang Static Analyzer:** An open-source tool that integrates with the Clang compiler to find bugs in C, C++, and Objective-C code.
- **Coverity:** A commercial tool that uses static analysis to detect defects in source code, including buffer overflows and memory leaks.
- **Code Secure:** A SAST tool that scans source code to identify security vulnerabilities, compliance issues, and potential weaknesses in web applications and enterprise software.

Dynamic analysis tools analyze the behavior of running software. They are particularly effective at identifying runtime errors such as use-after-free and buffer overflows.

Examples include:

- **Valgrind:** An open-source tool that detects memory management and threading bugs in C and C++ programs.
- **AddressSanitizer (ASan):** A fast memory error detector that can find out-of-bounds access and use-after-free errors.
- **AppScan:** A web and mobile application security testing suite that identifies vulnerabilities through multiple testing methods.
- **Veracode:** A cloud-based platform for continuous application security testing throughout the software development lifecycle.
- **Checkmarx:** A static application security testing tool that analyzes source code to detect vulnerabilities across various languages.
- **Fortify:** An end-to-end application security platform offering static, dynamic, and runtime testing to identify and remediate security issues.

Automated Tool and Manual Testing Best Practices

Integrate tools early and often
Integrate both static and dynamic analysis tools into the continuous integration/continuous deployment (CI/CD) pipeline to catch issues as early as possible.

Combine automated and manual testing
While automated tools are powerful, manual code reviews and testing are essential to identify complex vulnerabilities that automated tools might miss. Pairing both approaches ensures comprehensive coverage.

Regularly update tools
Keep analysis tools up-to-date to benefit from the latest detection capabilities and fixes for known issues.

Customize tool configurations
Tailor the configuration of analysis tools to suit your project's specific needs and coding standards to reduce false positives and enhance detection accuracy.

Comparative Analysis of Available Tools

Below is a comparative analysis of various tools available for software analysis and debugging, highlighting their strengths and ideal use cases.

Clang Static Analyzer vs. Coverity
While Clang Static Analyzer is an excellent free option for developers, Coverity provides more advanced features and support, making it suitable for large enterprises seeking in-depth analysis.

Valgrind vs. AddressSanitizer (ASan)
Valgrind is comprehensive and versatile but can be slower compared to ASan, which offers faster detection at the cost of higher memory usage. ASan is integrated into the LLVM and GCC compilers, providing seamless adoption for C and C++ projects.

SonarQube
SonarQube is a versatile platform that offers static analysis capabilities for multiple languages. It provides detailed reports and integrates with CI/CD pipelines, making it a strong choice for continuous code quality assurance.

Leveraging a mix of static and dynamic analysis tools, combined with best practices in automated and manual testing, significantly enhances an organization's ability to detect and address memory safety vulnerabilities. This comprehensive approach ensures robust security and reliability for your team's software development.

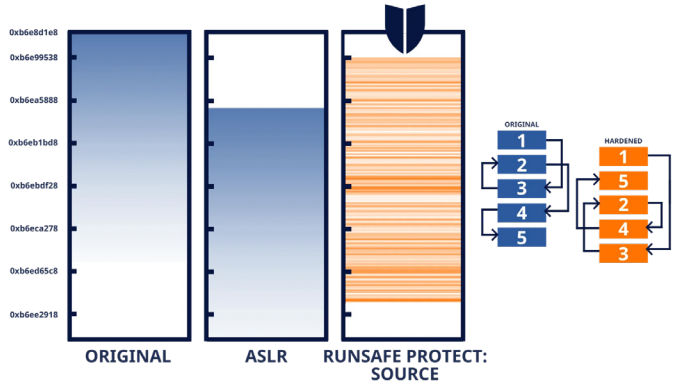
The Future of Memory Safety: A Time for Action, Not Just Awareness

Securing embedded systems in critical infrastructure has never been more pressing, and addressing memory-safety vulnerabilities will go a long way toward decreasing the attack surface of the technology, systems, and products that power everything from our power grid to our defense and transportation systems.

While the complete replacement or rewrite of legacy systems into memory safe languages may be the ideal long-term solution, practical constraints necessitate implementing security measures that can tackle memory safety vulnerabilities today and into the future.

- **Enhanced Runtime Protection:** Implement security measures that can protect systems without source code modification, providing security for legacy systems.
- **Binary-level Security:** Adopt techniques that improve the security posture of existing binaries without impacting system performance.
- **Architectural Resilience:** Develop systems with built-in security features that reduce vulnerability to exploitation.

These approaches offer practical solutions that can be implemented alongside longer-term strategies like code modernization while reducing the costs and challenges of scanning, patching, and monitoring.



How RunSafe Approaches Memory Safety

RunSafe provides memory-based vulnerability protection without the need for costly and time-consuming code rewrites. Through **Load-Time Function Randomization,** our Protect platform relocates software functions in memory every time the software is run, resulting in a unique memory layout, preventing attackers from exploiting memory-based vulnerabilities.

RunSafe's approach maintains system performance and functionality without modifying the original software. Additionally, RunSafe offers a repository of pre-hardened open-source packages and containers, providing immediate protection against attacks even without modifying source code.

Further Resources and Glossary

Further Resources

Learn more about the memory safety crisis and protecting embedded systems across critical infrastructure with resources from RunSafe Security.

Interview with Kiersten Todt, a former Chief of Staff at the Cybersecurity and Infrastructure Security Agency (CISA), emphasizes the importance of addressing memory safety and shifting liability from asset owners to product manufacturers to mitigate cybersecurity threats. (Kiersten Todt: "The Value of People" | RunSafe Security).

Visit the RunSafe blog. https://runsafesecurity.com/blog/
Register for upcoming events. https://runsafesecurity.com/events/
Learn about the RunSafe platform. https://runsafesecurity.com/platform/

Glossary

A

AddressSanitizer (ASan): A fast memory error detector that can find out-of-bounds access and use-after-free errors. Integrated into the LLVM and GCC compilers.

B

Buffer overflow: A vulnerability that occurs when a program writes more data to a buffer than it can hold, leading to adjacent memory being overwritten.

C

Clang Static Analyzer: An open-source tool that integrates with the Clang compiler to find bugs in C, C++, and Objective-C code through static analysis.

Common Vulnerability Scoring System (CVSS): A system used to evaluate the severity and exploitability of vulnerabilities to prioritize patch deployment.

Control Flow Integrity (CFI): Control Flow Integrity prevents attackers from hijacking a program's execution flow, making it much harder for malicious actors to redirect program execution to arbitrary code locations.
Coverity: A commercial static analysis tool that detects defects in source code, including buffer overflows and memory leaks.

D

Dangling pointer: A pointer that references a memory location that has been deallocated, potentially leading to invalid memory access.

Data Execution Prevention (DEP): A security feature that prevents the execution of code from non-executable memory regions, reducing the risk of certain types of attacks.

E

Equifax breach (2017): A major data breach exposing sensitive information of over 147 million individuals, partially attributed to memory management flaws in Apache Struts.

F

Federal Information Security Management Act (FISMA): A U.S. regulation requiring federal agencies to implement comprehensive information security programs, including secure software development practices.

G

General Data Protection Regulation (GDPR): An EU regulation mandating stringent data protection and privacy measures for organizations handling personal data of EU citizens.

H

Heartbleed (2014): A bug in the OpenSSL library caused by a buffer over-read, allowing attackers to read sensitive data from the memory of affected servers.

Health Insurance Portability and Accountability Act (HIPAA): A regulation setting standards for protecting sensitive patient data in the healthcare sector.

L

Load Time Function Randomization (LFR): Load Time Function Randomization protects against both known and unknown vulnerabilities at runtime. LFR randomizes code layouts in memory, preventing attackers from predicting where code is located, making it significantly harder to exploit memory vulnerabilities like buffer overflow attacks and Return-Oriented Programming (ROP).

M

Memory leak: Occurs when a program fails to release memory that is no longer needed, leading to gradual memory consumption and potential system slowdowns or crashes.

Memory safety: The protection of memory access operations against vulnerabilities such as buffer overflows, dangling pointers, and memory leaks.

Morris worm (1988): One of the first major internet worms, exploiting a buffer overflow vulnerability in Unix's finger service.

P

Payment Card Industry Data Security Standard (PCI DSS): A standard for organizations processing credit card transactions, including specific requirements for securing software systems to prevent data breaches and fraud.

R

Runtime Application Self-Protection (RASP): Solutions that monitor and protect applications during runtime by detecting and mitigating attacks in real-time.

Rust: A programming language known for its robust memory safety features, enforced through its ownership model and type system.

S

SonarQube: A versatile platform offering static analysis capabilities for multiple languages, providing detailed reports and integration with CI/CD pipelines.

Static Analysis Tools: Tools that analyze code without executing it, excellent for detecting potential memory safety issues by examining code patterns and syntax.

RUNSAFE SECURITY

ABOUT RUNSAFE SECURITY, INC.

RunSafe Security is the pioneer of a unique cyberhardening technology designed to disrupt attackers and protect vulnerable embedded systems and devices. With the ability to make each device functionally identical but logically unique, RunSafe Security renders threats inert by eliminating attack vectors, significantly reducing vulnerabilities, and denying malware the uniformity required to propagate. Based in McLean, Virginia, with an office in Huntsville, Alabama, RunSafe Security's customers span the Industrial Internet of Things (IIoT), critical infrastructure, automotive, and national security industries.

U

Use-after-free error: An error occurring when a program continues to use a pointer after the memory it references has been freed, potentially leading to arbitrary code execution, data corruption, or system crashes.

V

Valgrind: An open-source tool that detects memory management and threading bugs in C and C++ programs through dynamic analysis.

www.RunSafeSecurity.com

571.441.5076

sales@RunSafeSecurity.com