CS487 Building Secure Computer Systems

Memory Safety Bugs Keynote Report

Robert D. Hernandez rherna70@uic.edu

What (new things) you have learned?

1. **Memory Safety Bugs as Dominant Security Issues**: Memory safety issues, such as use-after-free and buffer overflow, are pervasive across software, accounting for about 70% of security vulnerabilities. These issues are not merely linked to low-level languages but are viewed as fundamental problems in memory abstractions themselves.

2. **Classification of Memory Safety Violations**: Memory safety issues are categorized into spatial bugs (accessing out-of-bounds or unauthorized memory) and temporal bugs (accessing memory after it has been freed). This classification helps in understanding how vulnerabilities manifest and provides a basis for targeted mitigation.

3. **Format String Vulnerabilities**: Even languages like Python, which are not typically considered memory-unsafe, have vulnerabilities in format strings that can lead to unauthorized data access. This suggests that memory safety issues are not restricted to traditional low-level languages and require broader attention across language types.

4. **Rust's Approach to Memory Safety**: Rust's programming model emphasizes memory safety through an expressive type system and lifetime checks, which prevent spatial and temporal bugs by enforcing rules on variable ownership, borrowing, and scope. Rust's restrictions against multiple mutable or overlapping references offer a practical way to avoid memory safety issues.

5. **Explicit Opt-In for Unsafe Operations**: Rust's approach requires developers to explicitly mark unsafe operations, making potential memory safety risks an intentional decision. This practice significantly reduces the likelihood of accidental vulnerabilities, as developers must carefully evaluate any operation that could compromise safety.

Briefly describe 1~2 key messages you learned that make you excited. Why?

Message 1: *Memory safety is a core abstraction problem, not just a language-level issue*. Recognizing that memory safety is fundamentally about how memory is abstracted rather than simply the result of language choice is crucial. It highlights the need for robust memory management principles across all levels of software development, not only in low-level systems programming. This broad perspective encourages solutions that can apply to a range of languages, beyond just C or C++.

Message 2: *Rust's enforced ownership and borrowing rules are impactful in eliminating classes of memory safety bugs*. Rust's strict, compile-time checks for ownership, lifetimes, and borrowing help prevent common errors associated with memory management. By eliminating the possibility of overlapping mutable

references or accessing invalid memory regions, Rust offers a model that could be inspirational for future language designs, especially in safety-critical domains. This model shows that memory safety can be enforced practically, not just in theory, and encourages the use of safer high-level language features.

These two messages are impactful because they reshape our understanding of memory safety as a structural issue rather than a problem inherent to specific languages. Rust's model shows that memory safety can be practically enforced through language features, encouraging the adoption of similar principles in both existing and new languages.

Thoughts on Eliminating Memory Safety Issues with Unlimited Research Funds

Tackling memory safety with unlimited resources would involve research into language and hardware design, widespread educational initiatives, and partnerships to create a culture that prioritizes memory safety across software development. With unlimited funds, a multi-layered, systematic approach is appropriate, combining advances in technology with developer-friendly tools and education, to truly eliminate memory safety issues. My funding allocation plan would consist of:

1. **Research in Language Design and Memory Abstractions**:
    - *Memory Abstraction and Pointer Safety Research*: Fund research on improving memory abstractions at the compiler and runtime levels, making it feasible to enforce boundaries on memory access dynamically or statically in real-time. This would include an investigation into pointer safety, treating pointers as explicit capabilities that enforce access permissions based on strict region boundaries, similar to capabilities in hardware or operating system security.
2. **Advanced Static and Dynamic Analysis Tools**:
    - *Develop tools to detect memory safety issues before they become a problem*: Static and dynamic analysis tools could help enforce memory safety across various languages, especially those not inherently designed for it. With extensive funding, creating sophisticated tools that integrate with IDEs and codebases to catch and automatically suggest fixes for unsafe memory operations would make a significant difference.
    - *Universal Memory Checker Libraries and Middleware*: These libraries could allow programs written in different languages to leverage memory safety checks across layers, making it easy to adopt memory-safe practices even in legacy or low-level codebases. They would offer features like automatic boundary checks, lifetime tracking, and secure handling of pointers or references.
3. **Hardware-Level Safety Features**:
    - *Research hardware solutions for memory safety*: Modern hardware could offer built-in safety checks for spatial and temporal memory safety violations, similar to how some processors now include support for detecting buffer overflows or stack corruption. Investing in partnerships with hardware manufacturers could create new processors that natively support memory-safe programming models, allowing developers to leverage these features directly from software. For example, if CPUs were built to recognize Rust-like ownership models, they could prevent unsafe operations at a hardware level. Such processors could support tagged pointers or

permissions-based memory models directly, reducing the performance overhead of software-enforced memory safety.

4. **Memory Safety Education and Open-Source Initiatives**:
    - *Expand developer training on memory safety principles*: Memory safety isn't just a technical challenge but also a cultural one. Funding global training programs to educate software engineers, from beginners to seasoned professionals, on memory safety would help drive the adoption of safe practices. Open-source repositories, code examples, and comprehensive courses could encourage best practices universally.
    - *Open-source libraries and frameworks for memory safety*: Providing open-source libraries to facilitate memory-safe practices in languages with traditionally weak memory safety (such as C/C++) could have a wide-reaching impact. These libraries could introduce safe memory abstractions, automated memory management features, and robust error-checking mechanisms, making memory safety more accessible to all developers. See #2 "Advanced Static and Dynamic Analysis Tools"
5. **Industry Partnerships for Widespread Adoption**:
    - *Work with software companies to encourage adoption*: Partnerships with major software developers (e.g., Microsoft, Google) to pilot memory-safe languages or models across large codebases could set a standard in the industry. Encouraging companies to adopt RustC over C for new projects, and supporting retrofitting of memory safety into existing code through grants or shared resources, could make memory safety an industry standard.