Keeping Safe Rust Safe with Galeed

Elijah Rivera\textsuperscript{2}, Samuel Mergendahl\textsuperscript{1}, Howard Shrobe\textsuperscript{2}, Hamed Okhravi\textsuperscript{1}, and Nathan Burow\textsuperscript{1}

1: MIT Lincoln Laboratory 2: MIT CSAIL

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

© 2021 Massachusetts Institute of Technology.

Delivered to the U.S. Government with Unlimited Rights, as defined in DFARS Part 252.227-7013 or 7014 (Feb 2014). Notwithstanding any copyright notice, U.S. Government rights in this work are defined by DFARS 252.227-7013 or DFARS 252.227-7014 as detailed above. Use of this work other than as specifically authorized by the U.S. Government may violate any copyrights that exist in this work.
Unsafe languages like C/C++ contributed to a large fraction of vulnerabilities.

Codebases have started *incrementally* porting to safe languages like Rust.

We show that:

Incremental deployment of safe languages ≠ Incremental security

Code in an unsafe language can break the safety of code in a safe language.

We design, implement, and evaluate Galeed to:

- *Prevent unintended* interactions between languages
- *Secure intentional* interactions between languages

Galeed keeps Safe Rust Safe
Memory Corruption Attacks

Spatial Memory Violation

Temporal Memory Violation

Buffer

Function Pointer

Heap

Function Pointer

Heap

Attacker

Attacker

Attacker
Google Says 70% Security Bugs In Chrome Are ‘Memory Safety Problems’

Microsoft: 70 percent of all security bugs are memory safety issues
Percentage of memory safety issues has been hovering at 70 percent for the past 12 years.


1: Source: Matt Miller (Microsoft Security Response Center), “Trends, challenge, and shifts in software vulnerability mitigation landscape”, BlueHat IL, 2019
Rust: Memory-Safe Programming Language

- A systems programming language that is memory-safe
- Small language runtime: is translated to instructions directly; no need for language VMs

- Spatial safety (no buffer overflows):
  - Statically-sized objects: compile-time checks
  - Dynamically-sized objects: runtime bounds checks

- Temporal safety (no use-after-frees):
  - Ownership: only one owner of object at a time
  - Burrowing: ownership can be temporarily transferred
Focus on Safe Rust

• Rust’s checks can be disabled by using the `unsafe{}` keyword

• Done when Rust’s checks are too restrictive

• Example: manipulating raw bits for interfacing with hardware devices in device drivers

• Unsafe Rust is trivially vulnerable to memory corruption like C/C++

• We focus on Safe Rust
Problem Statement

- All C/C++ code cannot be immediately ported to Rust
- Real codebases *incrementally* port to Rust
- Rust code often exists alongside other languages, primarily C/C++
- Examples: Mozilla (Firefox), DropBox, Microsoft, Amazon, Discord, Facebook, etc.

![Diagram of intended and unintended interactions between Rust and C++ code.](https://via.placeholder.com/150)

**Unintended Interaction**
- Rust allocated memory
- C++ allocated memory
- Rust code
- C++ code

**Intended Interaction**
- Rust allocated memory
- C++ allocated memory
- Rust code
- C++ code

**Unsafe**
- Red arrows

**Safe**
- Green arrows
Sketch of Our Solution: Galeed

Component 1: Heap Isolation
- Need to isolate Rust heap when running C++ code → Heap Isolation

Component 2: Pseudo-Pointers
- Need to avoid passing actual pointers to C++ → Pseudo-Pointers

Unintended Interaction
- Rust code
- C++ code
- Rust allocated memory
- C++ allocated memory
- Heap

Intended Interaction
- Rust code
- C++ code
- *p
- *p
- Heap
- Rust allocated memory
- p

Safe  Unsafe
Galeed Heap Isolation: Preventing Unintended Interactions

- Uses Intel Memory Protection Keys (MPK) to isolate Rust heap from C++ heap
- Modified Rust standard allocator
- Code to switch permission included around all external call sites
- Implemented using libmpk
Heap Isolation Implementation

Permission Switching Code

```assembly
asm! (" rdpkru ", in(" ecx") ecx, lateout (" eax") eax, lateout (" edx");
eax = ( eax & !PKRU_DISABLE_ALL ) | PKRU_ALLOW_READ ;
asm! (" wrpkru ", in(" eax") eax, in(" ecx") ecx, in(" edx ") edx );
```

[Diagram showing Rust code and C++ code separated by a lock, indicating heap isolation.]
Galeed Pseudo-Pointers: Securing Intended Interactions

- Replace real pointers with pseudo-pointers (identifiers)
- Pass pseudo-pointers to C++
- Replace C++ pointer operations with calls to getter/setter methods (an LLVM pass)
- Let Rust handle actual access to memory

**Diagram:**

- Rust code
  *p
- Heap
  Rust allocated memory
  p
- C++ code
  *p

**No Protection**

- Rust code
  id(p)  pointer
- Heap
  p
- C++ code
  id(p)

**Protected with Pseudo-Pointers**

- Rust code
  id(p)  pointer
- Heap
  MPK Protection
  Rust allocated memory
  p
- C++ code
  id(p)
Pseudo-Pointer Implementation

No Protection

Protected with Pseudo-Pointers

```c
int add5 (MyStruct * const p) {
    p->x += 5;
}

int add5 (ID < MyStruct > const p) {
    x = get_x_in_MyStruct (p);
    set_x_in_MyStruct (p, x +5);
}
```
Evaluation: Micro-Benchmarking

- **Heap Isolation**
  - Average ~50 cycles

- **Pseudo-Pointers**
  - Average ~100 cycles
Evaluation: Macro-Benchmarking on Firefox libperf

Cycle Overhead (count)

Runtime Overhead (%)
Lessons Learned

- Rust is being actively developed; releases matter
- Inline assembly still only available in “nightly” builds
- Current MPK interfaces are in C and un-optimized; there is a need for implementing them safely and optimally
- Mixed-language application security is a growing problem and an open area of research
Conclusion

- Incrementally deploying Rust does not necessarily mean incremental security
- Unsafe components of an application can endanger safe components
- Galeed prevents unintended interactions
- Galeed also secures intended interactions
- There is significant space for new research in this area