Easy-to-adopt security profiles for preventing RCE (remote code execution) in existing C++ code

Ulfar Erlingsson, Google Cloud, USA

P3627
Presented at WG21, SG23, Feb 11th, 2025, Hagenburg, Austria

Motivation

RCEs have given C++ (and C) a bad security reputation

Goal: Data-corruption C++ bugs rarely enable RCE attacks

- Doable by protecting stack, heap, and pointer data
- Doable with (almost full) compatibility with existing code
- Doable with mechanisms already implemented in LLVM etc.

Red-team hackers empirically confirm the above benefits

Reducing the RCE risk from the stack

Approach: Eliminate data pointers into the stack

- Move all address-taken stack variables to the heap
- Protects func params/vars, register spills, return addrs

Refinements

- Perf: Allow immutable stack ptrs on stack (for references)
- Perf: Put moved variables in fast thread-local heap arena
- Security: Randomize stack location or isolate the stack

Old idea (e.g., CCured); Implemented as SafeStack in LLVM

Reducing the RCE risk from the heap

Approach: Use disjoint heap arenas for different data

- Eliminates some use-after-free / overflow heap corruption
- Prevents heap feng shui / spooky action at a distance

Different implementations possible:

- Decent defense to partition heaps by libraries (DLLs/.so)
- Partition by types & check high bits for stronger protection

Widely implemented and used for security benefits

- Recent efforts: PartitionAlloc, kalloc_type, etc.

Reducing the RCE risk from pointers

Approach: Make pointers be more like unforgeable capabilities

- Make pointers hard to guess by randomization (e.g., ASLR)
- And/or add checks using high-order bits in pointers

Implementation is confined to backend (and runtime libraries)

Already supported in software/hardware:

- ASLR is pretty universal & a very useful server-side defense
- ARM CPU “ptr auth” checks integrity using high-order bits

Reducing the RCE risk from control-flow hijacking

Special protection for code pointers

Approach: Dynamically check control flow is to valid targets

- Ensure funcptr/vtable calls always target start of functions
- Ideally restrict also with arity+types
- Rely on stack integrity mechanisms for returns

Already supported in LLVM/gcc/MSVC and x86/ARM

- Some differences in check restrictions and enablement
- CPU overhead is very low, even without hardware support

Last 10+ years of attack data supports this approach

C++ RCE attacks do heap feng shui, hijack pointers & stack

These RCE defenses are complementary & “better together”

- (Already, ASLR itself is biggest hurdle for G red teams)

Widespread adoption is possible

- Performance overhead is very low (around 1% to 5%)
- Defenses work even on embedded CPUs without MMUs
- Can be adopted selectively, e.g., per DLL or .so

Potential realization in C++ source code / standard

Seems compatible with idea of a “more secure profile”

- Perhaps a declaration when importing modules?
- Or at top of source files?

(Above might be much like “use strict” in JavaScript.)

Unclear how to standardize

- How to phrase restrictions in abstract C++ machine?
- (Partitioning dynamic variables seems easy, at least)