When Good Components Go Bad

What are the security guarantees of compartmentalization?

Cătălin Hrițcu (Inria Paris)

HOPE Project
Devastating low-level vulnerabilities

• Languages like C/C++ sacrifice security for efficiency
  – type and memory unsafe:
    • e.g. any buffer overflow is catastrophic
  – root cause, working on fixes, but it's challenging:
    • efficiency
    • precision
    • scalability
    • backwards compatibility
    • deployment
Compartmentalization = Practical mitigation

• Main idea:
  – break up security-critical applications into **mutually distrustful components** with clearly specified privileges

• Enforce components can only interact in a safe way:
  – component separation, call-return discipline, ...

• ... by building secure compilation chain:
  – compiler, linker, loader, runtime, system, hardware

• ... targeting various mechanisms:
  – tagged architecture (micro-policies) — software fault isolation (SFI)
  – hardware enclaves (SGX) — capability machines (CHERI)
What are the security guarantees of compartmentalization?
Challenge

• **Source reasoning**
  = want compartmentalization to enable reasoning formally about security with respect to source language semantics

• **Undefined behavior**
  = can't be expressed at all by source language semantics!

• **Many different examples in a usual C compiler**
  – out of bounds array accesses
  – use after frees and double frees
  – invalid unchecked casts
  – (often even) signed integer overflows,
  – ...
Restricting undefined behavior

• **Limit spatial scope** of undefined behavior
  – mutually-distrustful components
    • each component protected from all the others, in particular from already compromised components

• **Limit temporal scope** of undefined behavior
  – dynamic compromise
    • each component gets guarantees as long as it has not encountered undefined behavior
    • i.e. the mere existence of vulnerabilities doesn't immediately make a component compromised
∀ attack trace $t$, if $\trianglerightrightarrow t$ then

∃ a dynamic compromise scenario explaining $t$ in source language ... for instance:

(0) $\trianglerightrightarrow* m_1;\text{Undef}(C_1)$

(1) $\exists A_1. \trianglerightrightarrow* m_2;\text{Undef}(C_2)$

(2) $\exists A_2. \trianglerightrightarrow t$

When Good Components Go Bad (arXiv:1802.00588)
Building secure compilation chain

When Good Components Go Bad (arXiv:1802.00588)