Safe Extension

Dawn Song
dawnsong@cs.berkeley.edu

Part II OS & Web Security

• OS Security
• Web Security
• More esoteric topics
  – Click fraud, etc.
  – Reputation systems & trust metrics
  – Few papers, but local experts
    » Guest lectures from Google, etc.

In the World of Extensions

• Today’s systems are designed to be extensible
  – OS kernel module/drivers
  – Browser plug-ins
• Extension accounts for over \( x\% \) of Linux kernel code
  – \( x=70 \) [Chou et. al.]
• Windows XP desktops
  – Over 35,000 drivers with over 120,000 versions [Swift et. al.]
• Drivers cause 85% of reported failures in Windows XP [Swift et. al.]
Desired Properties of Extensible Architecture

- Efficiency
- Protection
  - Extension should not read and/or write to certain regions in host → Isolation, sandbox
    - Do no harm to others
  - Extension should satisfy certain memory safety properties
    - Doesn’t shoot itself in the foot
  - Other more sophisticated security policies
- Security model
  - Malicious
  - Buggy

Enforcing Isolation (I)

- Hardware protection: process

- Disadvantages
  - Coarse grained
  - Performance hit on cross-domain calls
    - Context switches

Enforcing Isolation (II)

- Safe languages
- Advantages
  - Fine-grained protection
  - Ok performance overhead?
- Disadvantages
  - Legacy code
Enforcing Isolation (III)

- **Interpreter/emulator**
  - Inspect every instruction to be executed

- **Advantages**
  - Fine-grained protection
  - Works for legacy code

- **Disadvantages**
  - Prohibitively expensive
  - Although optimizations & code caching help a lot

- **Examples**
  - Program shepherding
  - Dynamic taint analysis

Enforcing Isolation (IV)

- **In-line reference monitors/dynamic checks**
  - IRMs enforce security policies by inserting into subject programs the code for validity checks and also any additional state that is needed for enforcement

- **Idea**
  - Add dynamic checks to enforce properties at run time
  - Combine with static analysis to reduce dynamic checks
  - Ensure dynamic checks are not by-passed
    - Control & data property enforcements are intertwined
  - Verifier:
    - Ensure dynamic checks are properly inlined

A Whole Spectrum

- **Tradeoff**
  - Complexity of properties enforced
  - Runtime overhead
  - Assumptions required
  - Complexity of priori analysis needed

- **Properties enforced entail**
  - What dynamic checks to add
  - How to add these dynamic checks

- **The spectrum**
  - SFI, CFI, DFI, XFI, ...
  - Interpreter/emulator is one end of the spectrum
SFI

- SFI [Wahbe et. al. 93]
  - Software fault isolation
  - Extension code only writes and jumps to dedicated data and code region
  - What’s the simplest checks can you insert?
  - How do you ensure checks are not by-passed?
    » Dedicated registers (5)
- SFI for CISC architectures [McCamant et al. 06]
  - Pad code blocks to be well aligned
  - Ensure jumps always to beginning of blocks

CFI

- Control-flow integrity [Abadi et al. 05]
- Enforce execution must follow a path of a CFG determined ahead of time
  - Obtain CFG via static analysis, execution profiling, or explicit security policies
- What checks to insert? How to ensure checks are not by-passed?
  - Assign unique IDs to equivalence classes of destination instructions
  - Source instruction includes IDs
  - Indirect jumps require ID-checks

DFI

- Data-flow integrity [Costa et al. 06]
- Enforce certain def-use relationship
  - Statically identify def-use relationships
  - For each use, enforce its def set
**XFI**

- Extensive property enforcement
  - Memory-access constraints
    - Only to certain regions
  - Interface restrictions
    - Control can only flow out of module via calls to stubs & returns to external call-sites
  - Scoped-stack integrity
  - Certain instructions disallowed
  - Certain registers cannot be modified
  - Control-flow integrity
  - Data integrity
    - Certain globals & locals can only be accessed via static references from proper instructions
- Why this set of properties?

**Mechanisms to Insert Checks**

- Source to source transformation
  - CIL
- Compiler-based approach
  - Gcc extensions
- Assembly -> binary code (statically)
  - Python :)
- Dynamic binary instrumentation
  - Rio, Valgrind, QEMU, Bocs, Plex86
- Static binary rewriting
  - Usually with debugging info/PDBs
  - Vulcan

**Discussions**

- Why do we need the verifier?
  - Smaller TCB
- How does XFI performance compare with SFI?

- What classes of properties can XFI/IRM enforce?
  - Can: safety properties
  - Cannot: Liveness properties, non-interference properties

- Does XFI prevent extensions from exploiting kernel vulnerabilities?
- How may attacker get around?
- How would you apply this approach to browser plug-ins?
  - What issues to consider?