Manifest Safety and Security

Robert Harper
Carnegie Mellon University
Collaborators

This work is, in part, joint with Lujo Bauer, Karl Crary, Peter Lee, Mike Reiter, and Frank Pfenning at Carnegie Mellon.

Work on manifest security is partly in collaboration with Benjamin Pierce, Stephanie Weirich, and Steve Zdancewic at UPenn.

Thanks especially to our students!
Safe and Secure Extensible Systems

- Extensible systems are prevalent:
  - Volunteer networks.
  - Browsers, operating systems.
  - Virtual communities (e.g., Second Life)

- Provides adaptability through customization, but threatens safety and security.
Safe and Secure Extensible Platforms

How can we build extensible systems without compromising integrity?

Using **manifest security**, which means:

- Rigorously specified policies.
- Guaranteed compliance with policy.
- Direct relationship to running code.
Current Approaches

Extensible systems rely on two main methods for ensuring safety and security:

- **Restriction**: limit potential damage by limiting capabilities of extensions.

- **Detection**: monitor execution to detect violations.

These are means ... but to what ends?
Current Approaches

- Restriction limits both good and bad behavior.
- In the limit, extensibility is disallowed.
- In practice, extensions have very limited capabilities.
- Tension between expressiveness and safety & security of extensions.
Current Approaches

- Detection requires run-time monitoring, and provides only a post-mortem analysis.
- Overhead can be significant.
- Little help with ensuring good behavior.
- Applies only to conditions that can be checked at run-time!
- eg, information flow vs access control
What’s Really At Stake?

Current methods attempt to address a high-level problem using low-level methods.

Violates the “end-to-end” principle.

Cannot define “security” at the level of bits, bytes, packets, address spaces, ....

Safety and security are governed by principals and policies, not bits and bytes.
Fundamentally, we wish to prove a theorem about a program.

- Does not violate API restrictions.
- Does not leak sensitive information.
- Complies with access control policies.

How can we state and prove such theorems about practical systems?
Implementing Manifest Safety and Security

ConCert Project: Trustless Grid Computing

- Manifest safety for mobile code.


- Manifest security for access control.


- Manifestly secure extensible systems.
Trustless Grid Computing in ConCert

- A general framework for grid computing.
- Loosely coupled volunteer network.
- Work-stealing scheduler.
- Manifest safety: verification, not trust.
- Hosts specify safety policy.
- Clients must prove compliance.
The ConCert Grid
Manifest Safety

- Logical specification of safety properties.
- Execution safety: no illegal instructions, no branches to unsafe code.
- Memory safety: no out-of-bounds array accesses, no stack violations.
- Logics include assembly-level type systems and Hoare-like annotations.
Manifest Safety

- Enforcement by proof- and type checking.
- Reject programs that do not pass checks.
- Compliance ensured by certifying compilers.
- Transfer source-level safety properties to object-level code.
- Produce formal certificates of compliance with host policy.
Certification and Verification Methods

- Proof-Carrying Code.
- VCGen + Theorem Proving for certification.
- LF representation of proofs.
- Typed Assembly Language.
- Typed compilation and type checking.
- Type annotations on object code.
A TAL-R Snippet

;; stack is described by S
;; sp : S
;;
;; virtual clock reads N+k+1
;; vck : N+k+1
;;
;; ebx contains an int->int that runs in at most k steps
;; ebx : ALL i:Nat. ALL r:ST.
;;
;; { eax:int,
;;   sp:{ eax:int, sp:r, vck:i }->0 * r
;;   vck:i+k }->0

add eax, eax, edx ;; consume one clock tick

;; vck : N+k

call ebx [N'] [S] ;; instantiate i=N and r=S,
;; place retaddr on stack, jump

;; vck : N
How TAL Defends Against Safety Attacks

- Malicious source code. 
  
  `loadFile “accounts.qdf”` is rejected.

- Malicious hand-written assembly code.
  
  `call loadFile` is ill-typed.

- `mov sp[0], 0xfe00b0c4; ret` is also ill-typed
How TAL Defends Against Attacks

One can think up more and more “tricks” ...
- Indirect jumps, stack over-runs, etc.

But it is a theorem that no well-typed assembly program can violate the safety policy.
- No attack will pass type checker!
How TAL Defends Against Attacks

Aha! What if we change the type system?

Nope, must supply a proof of soundness with respect to the safety policy!

Rats! Is there no way to defeat it?

No! Not within the confines of the policy.

But the policy may be “wrong” (more on this later).
What Can Be Certified?

- How far can we take this? What sort of properties can we certify?
- Short answer: anything for which one can devise a type system!
  - eg, TAL-R precludes certain DoS attacks
- Long answer: limited by how hard it is to generate and check proofs.
From Safety to Security

- Code safety is necessary for security.
- Precludes violation of language semantics.
- Source-level reasoning, not object-level enforcement.
- Can we extend manifest safety to manifest security?
Manifest Security

- Security policies are stated in a formal logical system.
- Augmented by certificates to identify principals and sign assertions.
- Assertions involve accessibility, ownership, delegation, etc.
- No fundamental limits on expressive power!
Manifest Security

- Compliance is demonstrated by a proof.
- eg, principal must prove that his/her access to a resource is entailed by the policy.
- Compose rules of deduction, starting with policy axioms and external certificates.
- Unforgeable, mechanically checkable.
Manifest Security

- Enforcement is by proof checking and cryptography.
- Present proof to reference monitor.
- Proof checker verifies evidence.
- Proof provides an “audit trail”.
- Direct expression and enforcement of intended security constraint!
Manifest Security

- Policies are formally analyzable.
  - eg, using cut elimination to investigate existence of proofs of certain assertions
  - provides a mathematical foundation for understanding consequences of a policy.

- Security policies can be very hard to understand!
Proof-Carrying Authorization Logic

Client

access request for resource R
“access(C,R)”

access theorem
“?: may-access(C,R)”

access proof
“P : may-access(C,R)”

resource monitor

proof and certificate checking

distributed theorem proving

“capability”

policy
Proof-Carrying Authorization Logic

A simple policy (all axioms are signed):
reg says class \((s, c)\) ...
prof says
  if reg says class\((s, c)\), then
  mayacc \((s, r)\)

A proof of mayacc \((s, r)\) involves:
  Certificate acquisition to est. identity.
  Logical inference from axioms.
How PCA Defends Against Attacks

- Replay attacks: client attempts to re-use previous authorization.
- Access control theorem and capability are time-stamped.
- Fraudulent assertions by principals.
- Requires breaking digital signatures.
How PCA Defends Against Attacks

- Misapplication of policy rules.
  - Prevented by proof checker, which ensures validity of all proofs.
- Fraudulent policies.
  - All axioms are signed, so must break cryptographic framework.
How PCA Defends Against Mistakes

- A principal may sign an assertion with unexpected consequences.

  - eg, a quantifier rotation $\forall \exists$ vs $\exists \forall$

- Requires policy analysis to validate.

- Instance of mechanized meta-reasoning.
How PCA Defends Against Mistakes

- Proofs provide an audit trail for analyzing attacks.
- Reveals who said what and why this was sufficient for access.
- Facilitates tracking errors in policy.
- Meaningful at the level of the policy, not at the level of some enforcement mechanism!
Secure Extensibility

How can we use manifest safety and security to implement safe extensibility?

Testbed: extensible browser architecture.

Safety against low-level attacks.

Security against unauthorized access and insecure information flows.
Manifestly Secure Extensibility

- Extend logics beyond safety and access control.
  - privacy and integrity
  - epistemic logic for info flow?
- Integrate security obligations into the programming language.
  - track proofs in the type system
Manifestly Secure Extension Architecture

program proofs types

program verifier

security policy

user

policy verifier

static
dynamic

runtime

executing program

request & proof

reference monitor

state

error
Manifest Security Infrastructure

- Logical frameworks.
- Specifying and analyzing security logics and programming languages.
- Representing and checking proofs.
- Certifying theorem provers.
- Finding proofs of logical assertions.
Manifest Security Infrastructure

- Theoretical investigations.
- Logics to express security policies.
- Analysis of logics and languages.
- Algorithms for proof checking and proof search.
- Informed by and informing practice!
Manifest Safety and Security

- Make safety and security policies explicit.
  - Rigorously specified in a suitable logic.
  - Analyzable and mechanizable.
- Enforce compliance of extensions with policy.
  - Require explicit proofs of compliance.
  - Verify using proof- and type checking.
Manifest Safety and Security

- Validate policies by meta-theoretic analysis.
- Ensure that policies capture intentions.
  - eg, not too restrictive, not too permissive
- Validate languages by semantic analysis.
- Ensure that accepted programs are indeed well-behaved.