Language-based security

- Language-based security: using language tools to specify and enforce security
  - End-to-end security specifications
  - Program analysis
  - Program transformation
- This seminar: explicitly integrating security policies into the programming model
  - Programmers need help writing secure applications
  - What's the right programming model to achieve this?

Explicit security models

- How can we specify security requirements?
  - Access control policies?
  - Confidentiality? Availability? Anonymity?
- Static or dynamic enforcement?
- How to show that complex systems/programs satisfy security requirements?
  - Formal validation
  - Scalable, modular analysis
- How should security requirements appear in or be connected to programs?
  - Program annotations? External specifications?

Plan of action

- Participants participate!
- Read recent papers on language-based security (+ a few seminal papers)
- Some lectures for background
  - 611 dependency only later in course
- 35-minute student presentation, 15-30 minute discussion
  - Presentation: review paper, kick off discussion
  - Each student: 1-2 presentations
- Readers:
  - Come prepared with issues, questions, criticisms
  - Speak up (constructively)
- Final project or survey
  - 10-minute presentation
  - One paragraph proposal due Nov. 3

Language-based security models

- Access control
  - “You can’t scram the core unless you are a reactor supervisor”
  - Principal/Authentication
  - Capabilities
  - Static access control
  - Java stack inspection
- Information flow control
  - “The plane’s location should only be known by traffic controllers”
  - Confidentiality, integrity
  - Absolute security?
- Need both and more
  - “The aggregate salaries in this demographic database are only accessible to subscribers who have paid”
  - Inference controls, quantitative information flow, intransitive noninterference

Not about:

- Buffer overruns
- Proof-carrying code
- Memory safety
- Type safety

Instead: How to prevent attacks that misuse or exploit application code but don’t violate “simple” safety properties?
**Computer security**

- Goal: prevent bad things from happening
  - Clients not paying for services
  - Critical services unavailable
  - Confidential information leaked
  - Important information damaged
  - System used to violate law

**When to enforce security**

Possible times to respond to security violations:

- Before execution: analyze, reject, rewrite
- During execution: monitor, log, halt, change
- After execution: roll back, restore, audit, sue, call police

**Conventional security mechanisms**

- Encryption, firewalls, memory protection
- Treat the program as a black box
  - Not fine-grained enough
  - No help with validation
  - Internal behavior of program is important!

**Conventional OS security**

- Program is black box
- Program talks to OS via protected interface (system calls)
  - Multiplex hardware
  - Isolate processes from each other
  - Restrict access to persistent data (files)

- Language-independent, simple

**OS: Coarse-grained control**

- Operating system enforces security at system call layer
- Hard to control application when it is not making system calls
- Security enforcement decisions made with regard to large-granularity operating-system abstractions
  - Files, sockets, processes, ports

**Need: fine-grained control**

- Modern programs make security decisions with respect to application abstractions
  - UI: access control at window level
  - Mobile code: no network send after file read
  - E-commerce: no goods until payment
  - Intellectual property rights management
- Need extensible, reusable mechanism for enforcing security policies
- Language-based security can support an extensible protected interface to control access
  - E.g., Java security
  - Capabilities, access control lists, stack inspection
- Language-based security can also support analyses of information security
End-to-end security

- Near-term problem: ensuring programs are memory-safe, type-safe so fine-grained access control policies can be enforced
- Long-term problem: ensuring that complex (distributed) computing systems enforce system-wide information security policies
  - Confidentiality
  - Integrity
  - Availability
- Confidentiality, integrity: end-to-end security described by information-flow policies

Information security: confidentiality

- Confidentiality: valuable information should not be leaked by computation
- Also known as secrecy; sometimes a distinction is made:
  - Secrecy: information itself is not leaked
  - Confidentiality: nothing can be learned about information
- Simple (access control) version:
  - Only authorized processes can read from a file
  - But... when should a process be "authorized"?
- End-to-end version:
  - Information should not be improperly released by a computation no matter how it is used
  - Requires tracking information flow in system
  - Encryption provides end-to-end secrecy—but prevents computation

Information security: integrity

- Integrity: valuable information should not be damaged by computation
- Simple (access control) version:
  - Only authorized processes can write to a file
  - But... when should a process be "authorized"
- End-to-end version:
  - Information should not be updated on the basis of less trustworthy information
  - Requires tracking information flow in system

Availability

- System is responsive to requests
- DoS attacks: attempts to destroy availability (perhaps by cutting off network access)
- Fault tolerance: system can recover from faults (failures), remain available, reliable
- Benign faults: not directed by an adversary
  - Usual province of fault tolerance
- Malicious or Byzantine faults: adversary can choose time and nature of fault
  - Byzantine faults are attempted security violations
  - Usually limited by not knowing some secret keys

Security Property Landscape

“System does exactly what it should”

- Privacy
- Digital rights
- Noninterference
- Mandatory access control
- Byzantine Fault Tolerance
- Discretionary access control
- Fault Tolerance
- Confinement
- Type safety
- Availability
- Memory safety
- Memory protection
- Confidentiality
- Liveness properties

Why put security in the program?

- Part of the programming model – can support conveniently
- Can tie program directly to policy and enforce
- Limits top properties enforceable through libraries and hardware
- Support separate compilation and modular analysis
- Why not?
  - separation of policy and program
Security specifications

- Is security proving that a program is correct?
- Ordinary correctness specifications:
  \( \{P\} S \{Q\} \)
  precondition \( P \) \implies postcondition \( Q \)
- How do we know the specification satisfies security requirements?
- Example:
  - Precondition: all salaries in the database are nonnegative
  - Postcondition: \( x \) contains the average salary
- Partial correctness assertions describe properties satisfied by every execution individually; information flow assertions compare every pair of executions