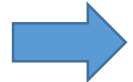


Access Control

Frank Piessens

Overview

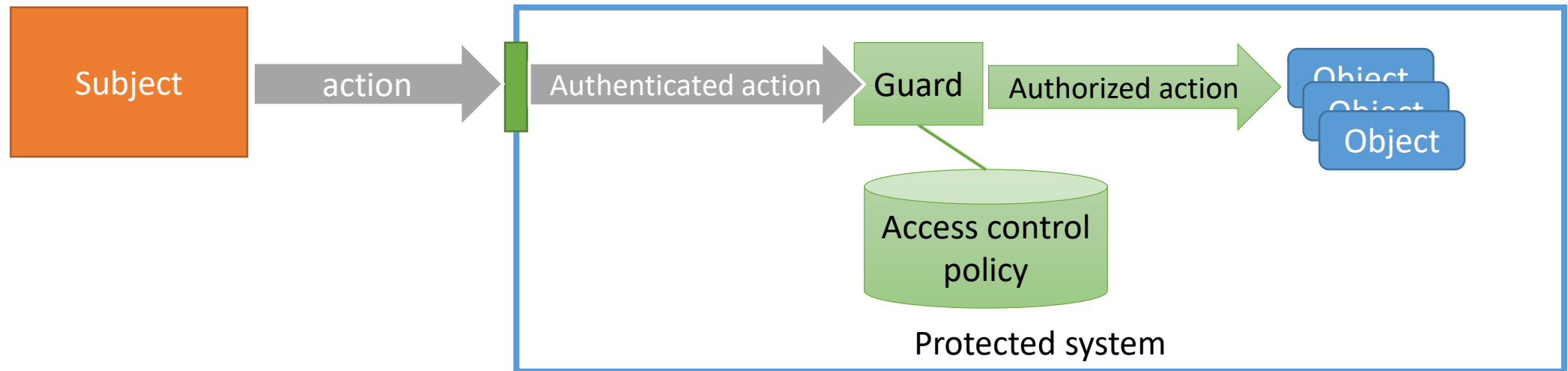
- 
- Introduction
 - System model, security objective and attacker model
 - Security automata
 - Access control models
 - Implementation considerations
 - Conclusions

Introduction

- Access control addresses the problem of specifying and enforcing rules on how a software application can be used
- We will assume that:
 - The software application has been adequately **isolated** such that interaction with the application is only possible through a well-defined interface
 - Users (subjects) interacting with the application have been adequately **authenticated** such that we know who is interacting over these interfaces
- We focus on enforcement by **run-time monitoring**

System model / attacker model

Security objective: the **guard** should block or allow actions as specified in the **access control policy**



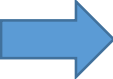
Examples

Subject	Action	Guard	Protected objects
Host	Packet send	Firewall	Intranet hosts
Process	Open file	OS kernel	Files / OS resources / ...
Java Module	Open file	Java Security Manager	Files / JVM resources
User	Query	DBMS	Databases / tables / ...
User	Get page	Web server	Web site contents
...

Application-level access control

- There is a trend to move access checks into applications in addition to OS / network level access control
 - More context-sensitive policies
 - E.g. Friends-based access control in online social networks
 - E.g. Task-based or workflow-based access control
 - Policies are more user-oriented
 - Many resources that need protection are application resources
 - E.g. a shopping basket

Overview

- Introduction
 - System model, security objective and attacker model
-  • Security automata
- Access control models
- Implementation considerations
- Conclusions

What should the guard do?

- A key question in access control is what the guard should do exactly
 - It should enforce some “access control policy”
 - But that policy should be configurable
 - By the owner of the protected system (“mandatory” policies)
 - But sometimes also by the users of the protected system (“discretionary” policies)
- We will introduce a general mechanism for specifying guard behavior
 - And then use it to specify some example guards
 - An **access control model** is a kind of “design pattern” for a guard

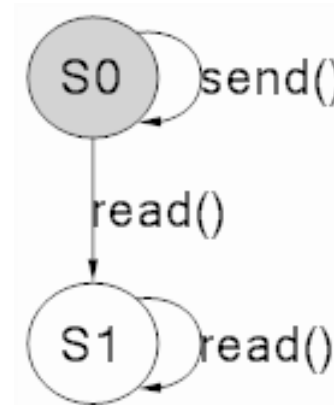
Specifying guard behavior

- We will consider guards that:
 - Can keep local state (the “protection state”)
 - On receipt of an authenticated action, the guard:
 - Decides whether to pass or drop it
 - Extensions are possible, for instance modify / sanitize the action
 - And possibly updates its protection state
- Such guards can be generally modeled as **security automata**:
 - A state set, with an initial state
 - A transition relation defined by predicates on actions and current state

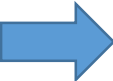
Specifying guard behavior

- Notation:
 - Security automaton is specified as a Scala object
 - State space is specified by declaration of typed state variables
 - Actions are specified as methods
 - Preconditions determine acceptability of action
 - Implementation body determines state update
- Example: no network send after file read

```
object ExPolicy {  
  var hasRead = false;  
  def send() {  
    require (!hasRead);  
  }  
  def read() {  
    hasRead = true;  
  }  
}
```



Overview

- Introduction
 - System model, security objective and attacker model
- Security automata
-  • Access control models
 - Discretionary access control (DAC)
 - Lattice-based access control (LBAC)
 - Role-based access control (RBAC)
 - Others
- Implementation considerations
- Conclusions

Discretionary Access Control (DAC)

- Objective = creator-controlled sharing of information
 - I.e. *Discretionary*: creator can set policy
- Key Concepts
 - Protected system manages objects, passive entities requiring controlled access
 - Objects are accessed by means of operations on them
 - Every object has an owner
 - Owner can grant right to use operations to other users
- Variants:
 - Possible to pass on ownership or not?
 - Possible to delegate right to grant access or not?
 - Constraints on revocation of rights.
 - Possible to create subjects with lesser privileges.

Security automaton for DAC

```
import scala.collection.mutable._;
object DACPolicy {
  type U = String; // Users
  type O = String; // Objects
  type A = String; // Actions
  type R = (U,O,A); // Rights
  var users = Set[U]();
  var objects = Set[O]();
  var rights = Set[R](); // Access Control Matrix
  var ownerOf = Map[O,U]();
  // Invariant: for (u,o,a) in rights, u is in users and o in objects

  // Access checks
  def read(s: U, o:O) { require(rights.contains((s,o,"Read"))); }
  def write(s: U, o:O) { require(rights.contains((s,o,"Write"))); }
  // Actions that impact the protection state
  def addRight(s: U, r:R) { r match { case (u,o,a) =>
    require(users.contains(u) & objects.contains(o) & ownerOf(o) == s);
    rights += r; }
  }
}
```

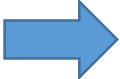
Security automaton for DAC (ctd)

```
def deleteRight(s: U, r: R) { r match { case (u,o,a) =>
  require(ownerOf(o) == s);
  rights -= r; }
}
def addObject(s: U, o: O) {
  require(!objects.contains(o));
  objects += o;
  ownerOf += (o -> s);
}
def delObject(s: U, o: O) {
  require(objects.contains(o) & ownerOf(o) == s);
  objects -= o;
  ownerOf -= o;
  rights.retain( {case (u,o2,a) => o2 != o } );
}
// Admin functions
def addUser(s: U, u: U) {
  require(!users.contains(u)); // Possibly require s is an admin
  users += u;
}
}
```

DAC Extensions

- Exercises: extend this simple specification with features such as:
 - Groups
 - Negative permissions
 - Mechanisms to start subjects with restricted privileges
 - ...

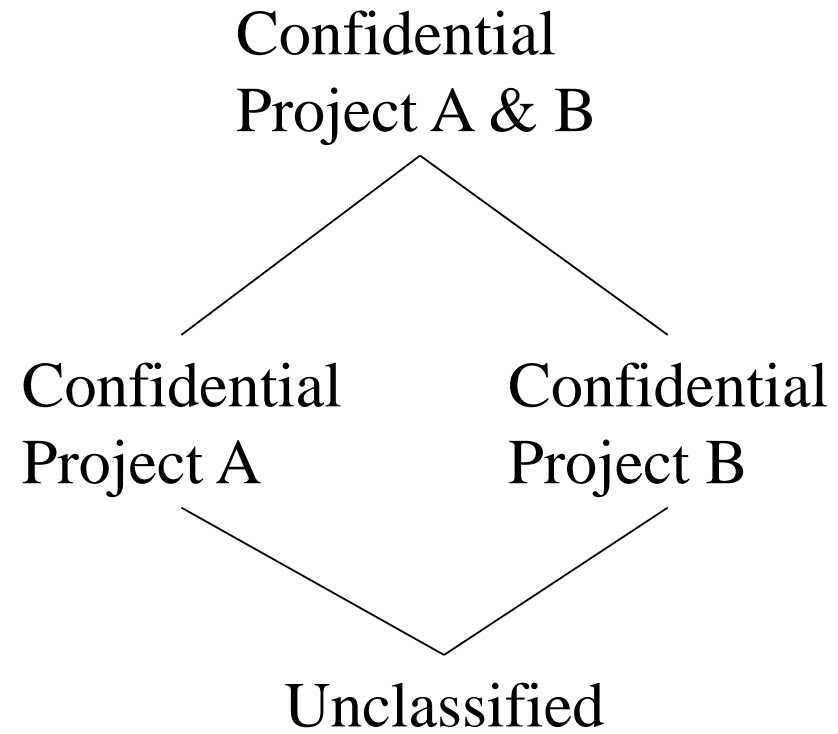
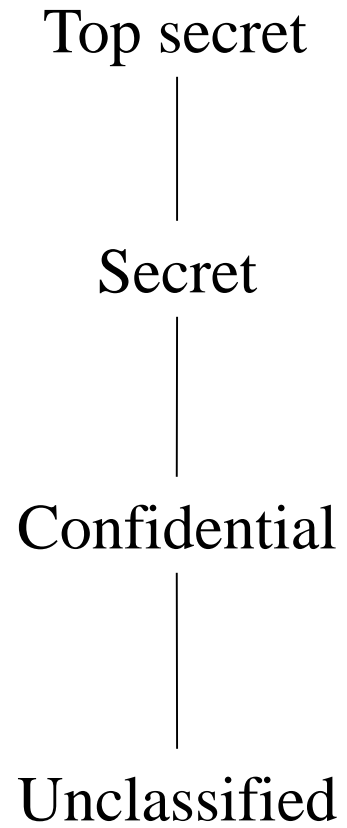
Overview

- Introduction
 - System model, security objective and attacker model
- Security automata
- Access control models
 - Discretionary access control (DAC)
 -  • Lattice-based access control (LBAC)
 - Role-based access control (RBAC)
 - Others
- Implementation considerations
- Conclusions

Lattice-based Access Control

- Objective = strict control of information flow
 - I.e. *Mandatory*: policy is not configurable by users
- Objective =
 - A lattice of *security labels* is given
 - Objects and users are tagged with security labels
 - Enforce that:
 - Users can only see information below their clearance
 - Information can only flow upward, even in the presence of Trojan Horses

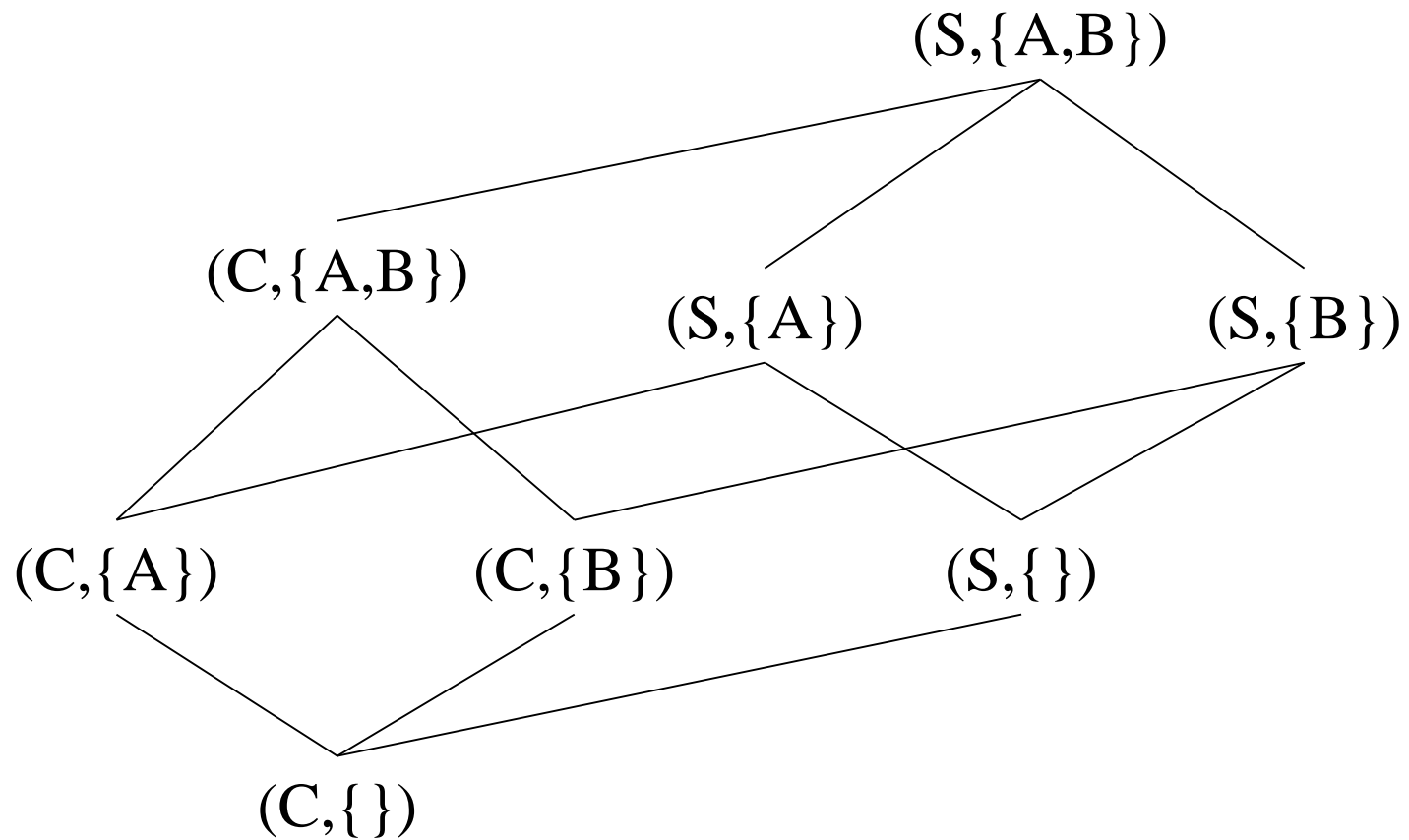
Example lattices



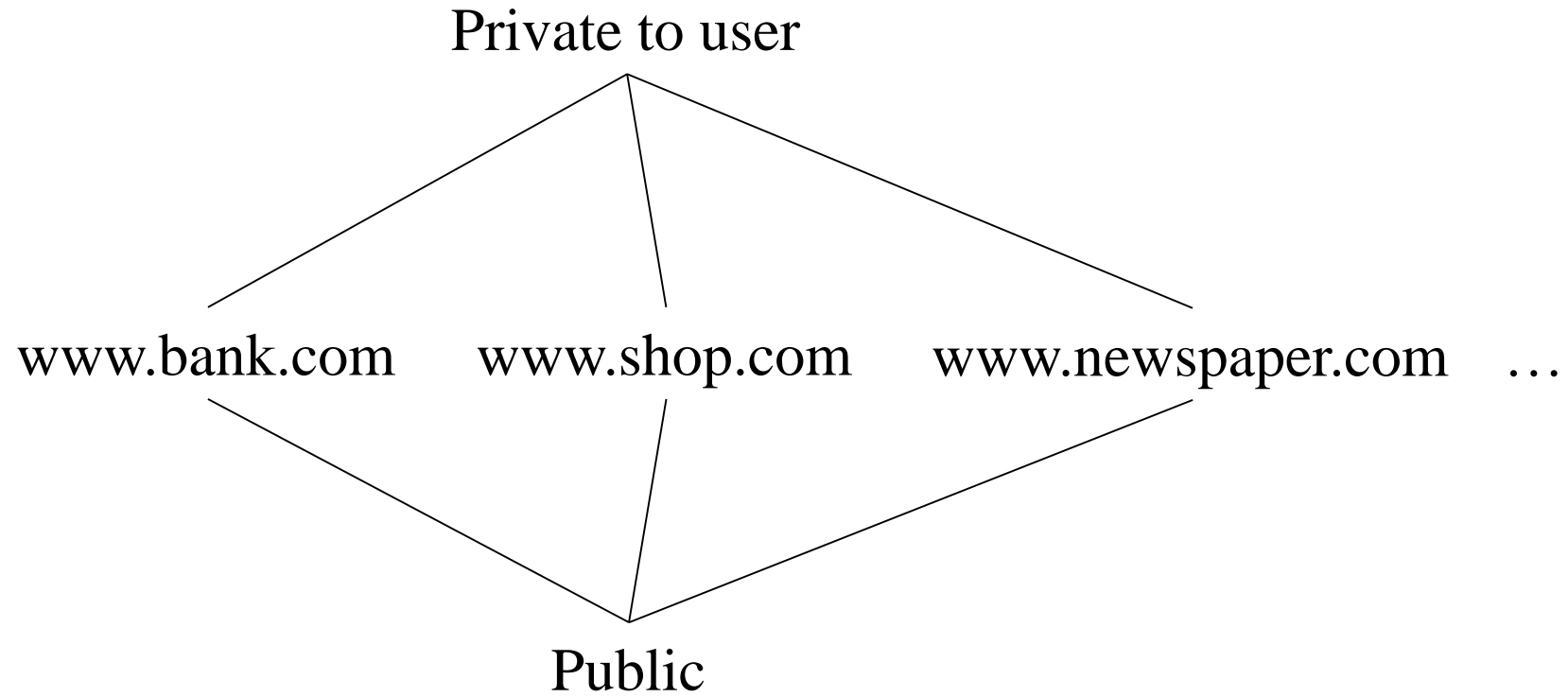
Typical construction of lattice

- Security label = (level, compartment)
- Compartment = set of categories
- Category = keyword relating to a project or area of interest
- Levels are ordered linearly
 - E.g. Top Secret – Secret – Confidential – Unclassified
- Compartments are ordered by subset inclusion

Example lattice



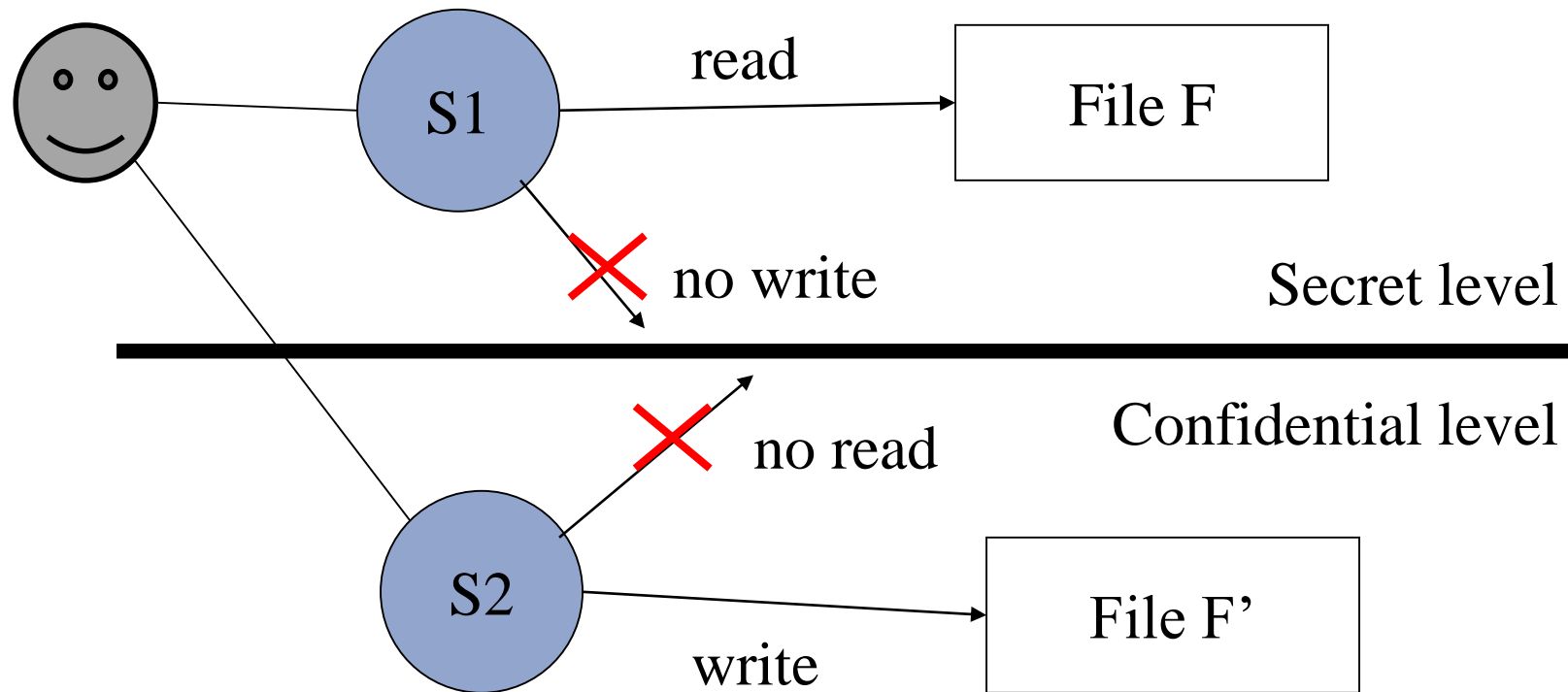
Web-based example



LBAC

- Key concepts of the model:
 - Users initiate *subjects* or *sessions*, and these are labeled on creation
 - Users of clearance L can start subjects with any label $L' \leq L$
 - Enforced rules:
 - Simple security property: subjects with label L can only read objects with label $L' \leq L$ (**no read up**)
 - *-property: subjects with label L can only write objects with label $L' \geq L$ (**no write down**)
 - The *-property addresses the Trojan Horse problem

LBAC and the Trojan Horse problem



Security automaton for LBAC

```
import scala.collection.mutable._;
object LBACPolicy {
  type U = String; // Users
  type O = String; // Objects
  type L = String; // Labels - in reality should be a more interesting lattice
  type S = Object; // Sessions

  // Stable part of the protection state
  var users = Set[U]("alice", "bob"); // some example users
  var ulabel = Map[U, L](("alice" -> "A"), ("bob" -> "B"));
  // Dynamic part of the protection state
  var objects = Set[O]();
  var sessions = Set[S]();
  var slabel = Map[S, L]();
  var olabel = Map[O, L]();

  // No read up
  def read(s: S, o: O) { require(slabel(s) >= olabel(o)); }
  // No write down
  def write(s: S, o: O) { require(slabel(s) <= olabel(o)); }
```

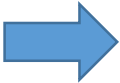

Security automaton for LBAC (ctd)

```
// Managing sessions and objects
def createSession(u:U,l:L) {
  require(ulabel(u) >= l);
  val s = new S();
  sessions += s;
  slabel += ( s -> l );
}
def addObject(s:S,o:O,l:L) {
  require(!objects.contains(o) & slabel(s) <= l);
  objects += o;
  olabel += (o -> l);
}
}
```

LBAC Extensions

- Exercises: extend this simple LBAC model with:
 - Declassification
 - Dynamic labels
 - Integrity protection instead of confidentiality protection
 - ...

Overview

- Introduction
 - System model, security objective and attacker model
- Security automata
- Access control models
 - Discretionary access control (DAC)
 - Lattice-based access control (LBAC)
 -  • Role-based access control (RBAC)
 - Others
- Implementation considerations
- Conclusions

Role-Based Access Control (RBAC)

- Main objective: *manageable* access control
 - Mandatory, but policy setting by some administrator
- Key concepts of the model:
 - **A role:**
 - many-to-many relation between users and permissions
 - Corresponds to a well-defined job or responsibility
 - Think of it as a named set of permissions that can be assigned to users
 - When a user starts a session, he can activate some or all of his roles
 - A session has all the permissions associated with the activated roles

Security automaton for RBAC

```
import scala.collection.mutable._;
object RBACPolicy {
  type U = String; // Users
  type R = String; // Roles
  type P = String; // Permissions
  type S = Object; // Sessions

  // Stable part of the protection state
  var users = Set[U]("alice", "bob");
  var roles = Set[R]("teacher", "researcher");
  var perms = Set[P]("accesslib", "bookroom");
  var ua = Map[U, Set[R]](("alice" -> Set("teacher")), ("bob" -> Set("teacher", "researcher")));
  var pa = Map[R, Set[P]](("teacher" -> Set("bookroom")), ("researcher" -> Set("accesslib")));
  // Dynamic part of the protection state
  var sessions = Set[S]();
  var session_roles = Map[S, Set[R]]();
  var session_user = Map[S, U]();

  // Access checks
  def checkAccess(s: S, p:P) {
    require(session_roles(s).exists(r => pa(r).contains(p)) ); }
}
```

Security automaton for RBAC (ctd)

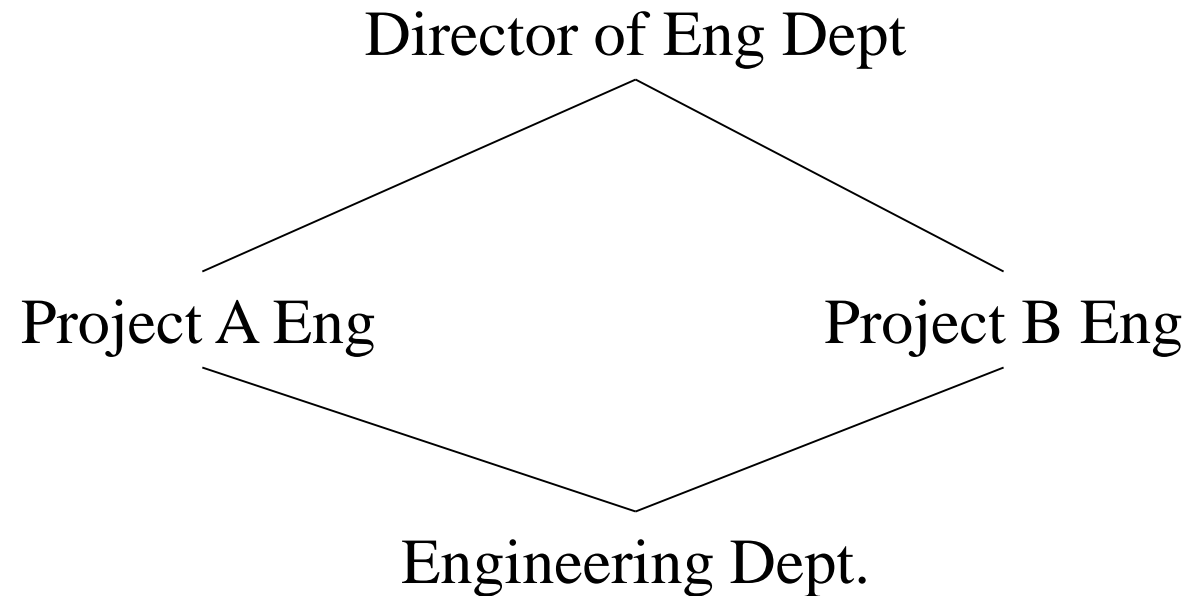
```
// Managing sessions
def createSession(u:U,rs:Set[R]) {
  require(rs.subsetOf(ua(u)));
  val s = new S();
  sessions += s;
  session_roles += ( s -> rs );
  session_user += (s -> u);
}

def dropRole(s:S,r:R) {
  require(session_roles(s).contains(r));
  session_roles(s) -= r;
}

def activateRole(s:S,r:R) {
  require(ua(session_user(s)).contains(r));
  session_roles(s) += r;
}
}
```

RBAC - Extensions

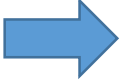
- Hierarchical roles: senior role inherits all permissions from junior role



RBAC - Extensions

- Constraints:
 - Static constraints
 - Constraints on the assignment of users to roles
 - E.g. Static separation of duty: nobody can both:
 - Order goods
 - Approve payment
 - Dynamic constraints
 - Constraints on the simultaneous activation of roles
 - E.g. to enforce least privilege

Overview

- Introduction
 - System model, security objective and attacker model
- Security automata
- Access control models
 - Discretionary access control (DAC)
 - Lattice-based access control (LBAC)
 - Role-based access control (RBAC)
-  • Others
- Implementation considerations
- Conclusions

Other models

- Lattice-based access control for integrity
 - Labels represent a level of *trustworthiness* of the integrity of information
 - Information should only flow from high-integrity objects to low-integrity objects – not the other way around
- Dynamic access control models
 - Where the protection state changes with work being done
 - E.g. enforcing a workflow, enforcing Chinese walls



Conclusions

- Think of all these models as *design patterns* for policies
 - Specific applications will generally need a custom policy
 - **Exercise:** come up with policies for
 - A discussion forum
 - A social network site
 - ...
- Security automata are a good tool to reason about and specify your policies
 - Some limited forms of automatic analysis might even be possible
 - E.g. safety analysis: can this user ever access this object?
 - But many of these questions will be undecidable in general

Overview

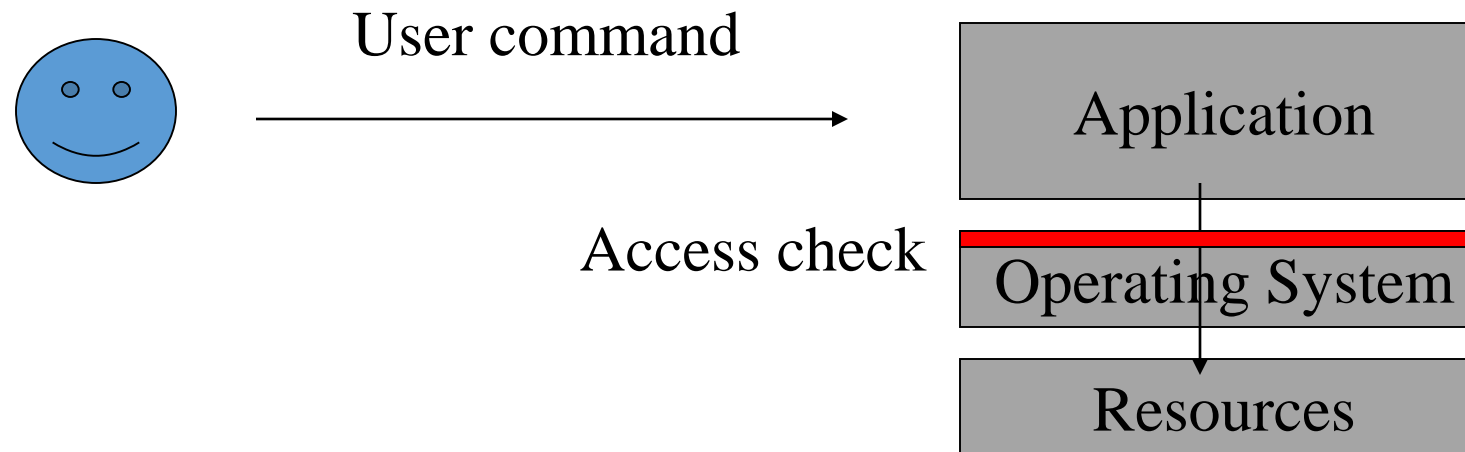
- Introduction
 - System model, security objective and attacker model
- Security automata
- Access control models
- ➔ • Implementation considerations
- Conclusions

Introduction

- Implementing access control is challenging, because:
 - It must be **efficient**
 - It must implement a possibly complicated evolving high-level policy
 - How can you make access control *configurable* by high-level rules?
 - It must be secure:
 - Complete mediation: all accesses must be checked
 - Safety: it should be possible to check that the configured rules are adequate

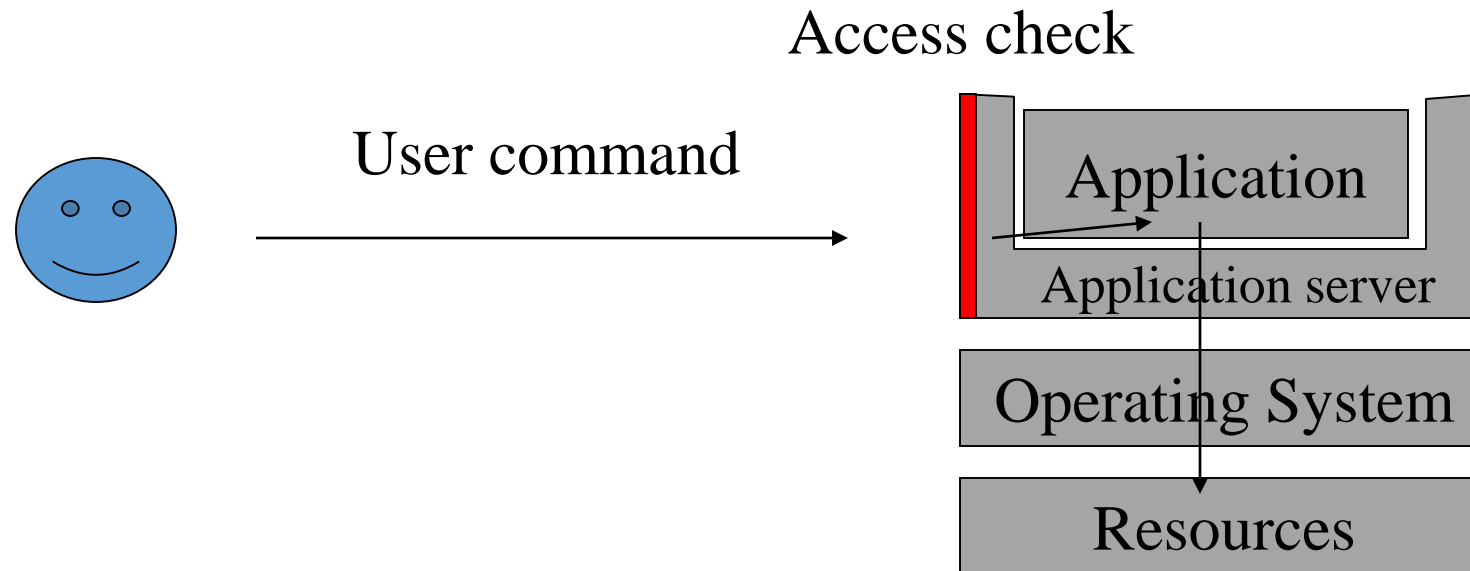
Approach #1: delegate to the OS

- If application resources can be mapped to OS resources, the OS access control can be reused
- E.g. in Windows:
 - Server authenticates client, and puts access token on the thread servicing the request



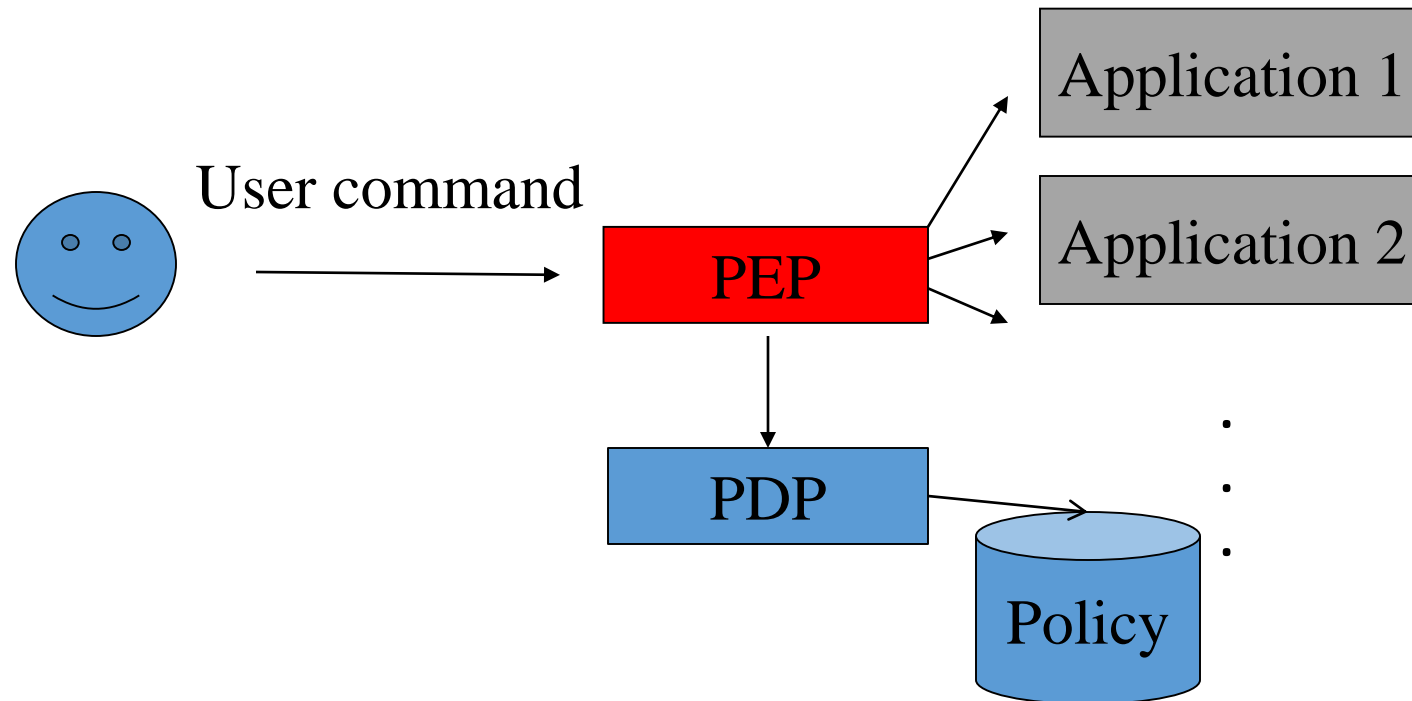
Approach #2: application servers

- Application server intercepts commands and performs access check
- E.g. in Windows COM+:
 - Look for a local group SID corresponding to a role in the client access token



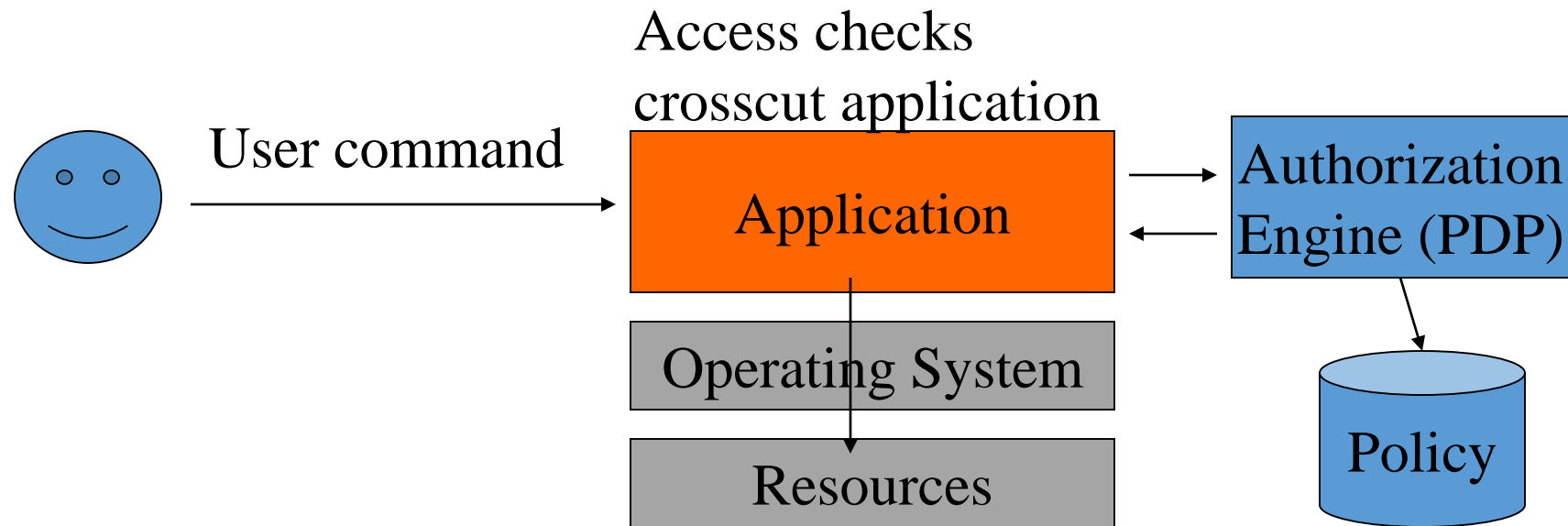
Approach #3: security middleware

- Reverse proxy intercepts commands and performs access check
- E.g. IBM WebSEAL



Approach #4: in the application

- Application performs explicit checks in the application code
- It makes sense to externalize at least access decision logic to an authorization engine



Conclusion

- Access control is a core mechanism for enforcing application-level security policies
- Enforcing access control is error-prone
 - Policies are often left implicit
 - We have proposed security automata as a policy specification method
 - Complete mediation is hard to achieve
 - We have discussed several approaches for implementing access control with a reasonable assurance of complete mediation
 - Users have difficulties setting policies
 - E.g. Facebook privacy controls