Web App Access Control Design

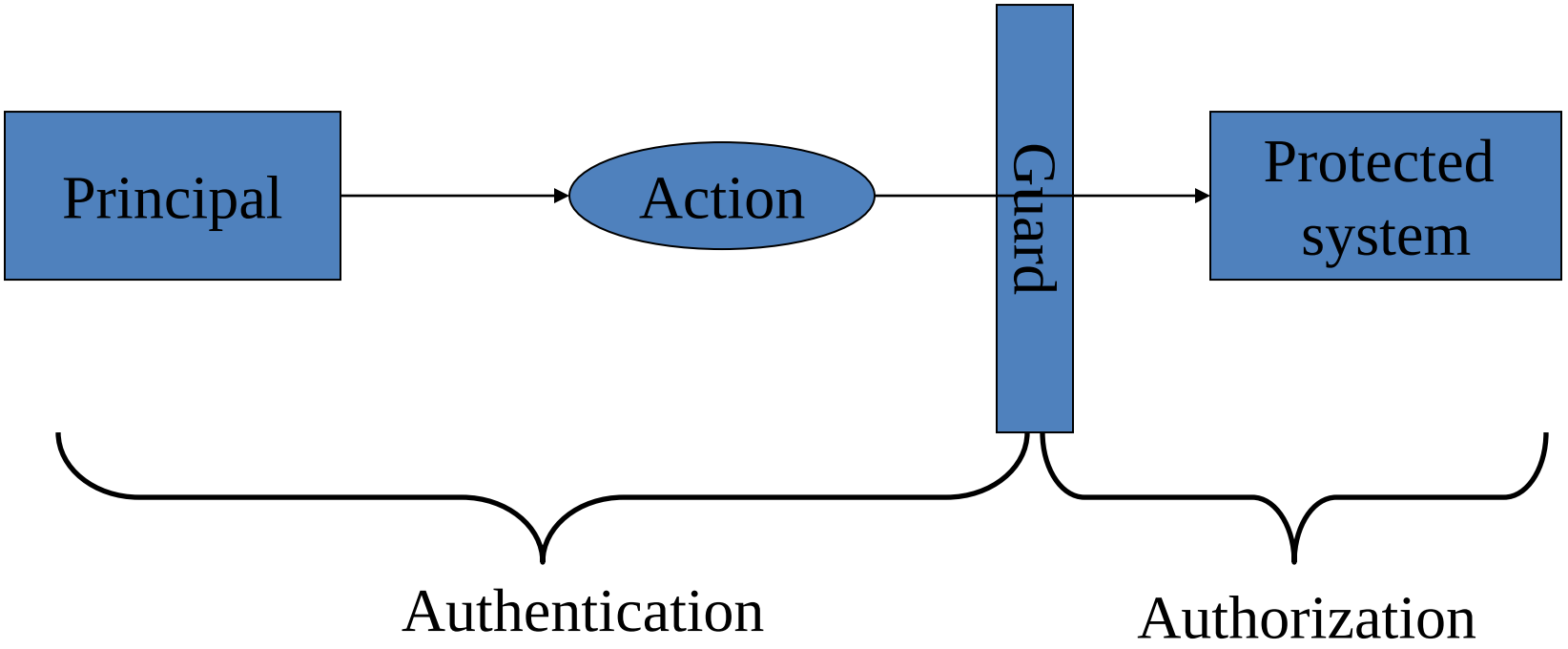
Access Control Best Practices

- Build a centralized AuthZ mechanism
- Code to the permission, not the role
- Design AuthZ as a filter
- Deny by default, fail securely
- Server-side trusted data should drive AuthZ
- Be able to change entitlements in real time
- Design standardized data contextual AuthZ
- Build grouping for users and permissions

Access Control Anti-Patterns

- Hard-coded role checks in application code
- Lack of centralized access control logic
- Untrusted data driving access control decisions
- Access control that is “open by default”
- Lack of addressing horizontal access control in a standardized way (if at all)
- Access control logic that needs to be manually added to every endpoint in code
- Access Control that is “sticky” per session
- Access Control that requires per-user policy

General Access Control Model



What is Access Control?

- Authorization is the process where a system determines if a specific user has access to a resource

- **Permission** : Represents app behavior only
- **Entitlement**: What a user is actually allowed to do
- **Principle/User**: Who/what you are entitling
- **Implicit Role**: Named permission, user associated
  - if (user.isRole(“Manager”));
- **Explicit Role**: Named permission, resource associated
  - if (user.isAuthorized(“report:view:3324”);

Attacks on Access Control

- Vertical Access Control Attacks
  - A standard user accessing administration functionality
- Horizontal Access Control attacks
  - Same role, but accessing another user's private data
- Business Logic Access Control Attacks
  - Abuse of one or more linked activities that collectively realize a business objective

Access Controls Impact

- Loss of accountability
  - Attackers maliciously execute actions as other users
  - Attackers maliciously execute higher level actions
- Disclosure of confidential data
  - Compromising admin-level accounts often results in access to user's confidential data
- Data tampering
  - Privilege levels do not distinguish users who can only view data and users permitted to modify data

Hard Coded Roles

```
void editProfile(User u, EditUser eu)
  if (u.isManager()) {
     editUser(eu)
  }
}
```

How do you change the policy of this code?

Hard Coded Roles

```
if ((user.isManager() ||
    user.isAdministrator() ||
    user.isEditor() ||
    user.isUser() &&
    user.id() != 1132))
{
    //execute action
}
```

Hard Coded Roles

- Makes “proving” the policy of an application difficult for audit or Q/A purposes
- Any time access control policy needs to change, new code need to be pushed
- RBAC is often not granular enough
- Fragile, easy to make mistakes

Order Specific Operations

Imagine the following parameters

```
http://example.com/buy?action=chooseDataPackage
http://example.com/buy?action=customizePackage
http://example.com/buy?action=makePayment
http://example.com/buy?action=downloadData
```

Can an attacker control the sequence?

Can an attacker abuse this with concurrency?

Never Depend on Untrusted Data

- Never trust request data for access control decisions
- Never make access control decisions in JavaScript
- Never make authorization decisions ***based solely on***
  - hidden fields
  - cookie values
  - form parameters
  - URL parameters
  - anything else from the request
- Never depend on the order of values sent from the client

Best Practice: Centralized AuthZ

- Define a centralized access controller
  - `ACLService.isAuthorized(PERMISSION_CONSTANT)`
  - `ACLService.assertAuthorized(PERMISSION_CONSTANT)`
- Access control decisions go through these simple API's
- Centralized logic to drive policy behavior and persistence
- May contain data-driven access control policy information

Best Practice: Code to the Activity

```
if (AC.hasAccess(“article:edit”))
{
   //execute activity
}
```

- Code it once, never needs to change again
- Implies policy is centralized in some way
- Implies policy is persisted in some way
- Requires more design/work up front to get right

Using a Centralized Access Controller

In Presentation Layer

```
if (isAuthorized(Permission.VIEW_LOG_PANEL))
{
  <h2>Here are the logs</h2>
  <%=getLogs();%/>
}
```

In Controller

```
try (assertAuthorized(Permission.DELETE_USER))
{
  deleteUser();
}
```

SQL Integrated Access Control

Example Feature

```
http://mail.example.com/viewMessage?msgid=2356342
```

This SQL would be vulnerable to tampering

```
select * from messages where messageid = 2356342
```

Ensure the owner is referenced in the query!

```
select * from messages where messageid = 2356342 AND
messages.message_owner = <userid_from_session>
```

Data Contextual Access Control

Data Contextual / Horizontal Access Control API examples:

- `ACLService.isAuthorized(“car:view:321”)`
- `ACLService.assertAuthorized(“car:edit:321”)`

Long form:

- `isAuthorized(user, Perm.EDIT_CAR, Car.class, 14)`

- Check if the user has the right role in the context of a specific object

- Protecting data a the lowest level!

Data Contextual Access Control

User	
User ID	User Name

Permission	
Permission ID	Permission Name

Data Type	
Data ID	Data Name

Role	
Role ID	Role Name

Entitlement				
User ID	Permission ID	Role ID	Data Type ID	Data Instance Id

Questions?

jim@owasp.org

jim.manico@whitehatsec.com