Web App Access Control Design

What is Access Control / Authorization?

- Authorization is the process where a system determines if a specific user has access to a particular resource
- The intent of authorization is to ensure that a user only accesses system functionality to which he is entitled
- Role based access c ontrol (RBAC) is commonly used to manage permissions within an application

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 workflow

Access Control Issues

- Many applications utilize an “all or nothing” approach
  - Once authenticated all users have equal privilege levels
- Authorization logic often relies on Security Through Obscurity (STO) by assuming:
  - Users won’t find unlinked or “hidden” paths/functionality.
  - Users will not find and tamper with “obscured” client side parameters (i.e. “hidden” form fields, cookies, etc)
- Applications with multiple permission levels/roles often increases the possibility of conflicting permission sets resulting in unanticipated privileges

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

Hard Coded Roles

```
if (user.isManager() ||
    user.isAdministrator() ||
    user.isEditor() ||
    user.isUser()) {

//  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
- 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 user 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

Access Control Issues

- Many administrative interfaces require only a password for authentication
- Shared accounts combined with a lack of auditing and logging make it extremely difficult to differentiate between malicious and honest administrators
- Administrative interfaces are often not designed as “secure” as user-level interfaces given the assumption that administrators are trusted users
- Authorization/Access Control relies on client-side information (e.g., hidden fields)

```
<input type=“text" name=“fname" value=“Derek”>
<input type=“text" name=“lname" value=“Jeter”>
<input type=“hidden" name=“usertype" value=“admin”>
```

Attacking Access Controls

- Elevation of privileges
- 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

Testing for Broken Access Control

- Attempt to access administrative components or functions as an anonymous or regular user
  - Scour HTML source for “interesting” hidden form fields
  - Test web accessible directory structure for names like admin, administrator, manager, etc (i.e. attempt to directly browse to “restricted” areas)
- Determine how administrators are authenticated. Ensure that adequate authentication is used and enforced
- For each user role, ensure that only the appropriate pages or components are accessible for that role
- If able to compromise administrator-level account, test for all other common web application vulnerabilities (poor input validation, privileged database access, etc)

Defenses Against Access Control Attacks

- Implement role based access control to assign permissions to application users for vertical access control requirements
- Implement data-contextual access control to assign permissions to application users in the context of specific data items for horizontal access control requirements
- Avoid assigning permissions on a per-user basis
- Perform consistent authorization checking routines on all application pages
- Where applicable, apply DENY privileges last, issue ALLOW privileges on a case-by-case basis

Defenses Against Access Control

- Where possible restrict administrator access to machines located on the local area network (i.e. it's best to avoid remote administrator access from public facing access points)
- Log all failed access authorization requests to a secure location for review by administrators
- Perform reviews of failed login attempts on a periodic basis
- Utilise the strengths and functionality provided by the SSO solution you chose, e.g. Netegrity

Best Practice: Code to the Activity

```
if (AC.hasAccess(ARTICLE_EDIT)) {
   //execute activity
}
```

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

Best Practice: Centralized ACL Controller

- Define a centralized access controller
  - `ACLService.isAuthorized(ACTION_CONSTANT)`
  - `ACLService.assertAuthorized(ACTION_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

Using a Centralized Access Controller

In Presentation Layer

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

In Controller

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

Best Practice: Verifying policy server-side

- Keep user identity verification in session
- Load entitlements server side from trusted sources
- Force authorization checks on ALL requests
  - JS file, image, AJAX and FLASH requests as well!
  - Force this check using a filter if possible

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>
```

Access Control Positive Patterns

- Code to the activity, not the role
- Centralize access control logic
- Design access control as a filter
- Deny by default, fail securely
- Build centralized access control mechanism
- Apply same core logic to presentation and server-side access control decisions
- Server-side trusted data should drive access control

Data Contextual Access Control

Data Contextual / Horizontal Access Control API examples

- `ACLService.isAuthorized(EDIT_ORG, 142)`
- `ACLService.assertAuthorized(VIEW_ORG, 900)`

Long form

- `isAuthorized(user, EDIT_ORG, Organization.class, 14)`

- Essentially checking 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

Role/Activity	
Role/Activity ID	Role/Activity Name

Entitlement / Privilege			
User ID	Role/Activity ID	Data Type ID	Data Instance Id

Data Type	
Data ID	Data Name