Top 1: Broken Access Control

Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification, or destruction of all data or performing a business function outside the user's limits. Common access control vulnerabilities include:

- Violation of the principle of least privilege(**do not provide unnecessary access**) or deny by default (**Newly created user will not get any access unless assign any role**), where access should only be granted for particular capabilities, roles, or users, but is available to anyone.
- Bypassing access control checks by **modifying the URL** (parameter tampering or force browsing), internal application state, or the HTML page, or by using an attack tool modifying API requests.
- Permitting viewing or editing someone else's account, by providing its unique identifier (**insecure direct object references**). For example, consider a web application where users can view their account details by providing their user ID as a parameter in the URL. If the application doesn't verify whether the user making the request is the owner of the account corresponding to that user ID, an attacker could change the user ID in the URL to access and modify someone else's account.
- Accessing API with missing access controls for POST, PUT and DELETE. Each method has a specific purpose:
    - GET: Used to retrieve data from the server.
    - POST: Used to submit data to the server to create a new resource.
    - PUT: Used to update or replace an existing resource on the server.
    - DELETE: Used to request the removal of a resource from the server.

  Access controls, or authorization mechanisms, determine **who is allowed to perform** specific actions (e.g., create, update, or delete resources) within an **application or API**.
- Elevation of privilege. Acting as a user without being logged in or acting as an admin when logged in as a user.
- Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token, or a cookie or hidden field manipulated to elevate privileges or abusing JWT invalidation.
- CORS misconfiguration allows API access from unauthorized/untrusted origins.
- Force browsing to authenticated pages as an unauthenticated user or to privileged pages as a standard user.

How to Prevent

Access control is only effective in trusted server-side code or server-less API(Popular serverless platforms include AWS Lambda, Azure Functions, Google Cloud Functions, and various serverless frameworks for different programming languages. Developers can use these platforms to create, deploy, and manage serverless APIs without the need to worry about server provisioning, maintenance, or scaling), where the attacker cannot modify the access control check or metadata.

(why it is called **Serverless API**/stateless?

- No tension with OS server.
- No tension for load/concurrency. Auto Scalable.
- No tension with programming language. Commonly supported languages include JavaScript/Node.js, Python, Ruby, Java, and C#.
- Built-in security features, including Identity and Access Management (IAM) for access control and execution roles. You can define fine-grained permissions to control what functions can access and modify.)

- Except for public resources, deny by default.
- Implement access control mechanisms once and re-use them throughout the application, including minimizing Cross-Origin Resource Sharing (CORS) usage.
- Model access controls should enforce record ownership rather than accepting that the user can create, read, update, or delete any record.
- Unique application business limit requirements should be enforced by domain models.
- Disable web server directory listing and ensure file metadata (e.g., .git) and backup files are not present within web roots.
- Log access control failures, alert admins when appropriate (e.g., repeated failures).
- Rate limit API and controller access to minimize the harm from automated attack tooling.
- Stateful session identifiers should be invalidated on the server after logout. Stateless JWT tokens should rather be short-lived so that the window of opportunity for an attacker is minimized. For longer lived JWTs it's highly recommended to follow the OAuth standards to revoke access.
  Note: **Stateless** JWT (JSON Web Token) tokens are JWT tokens that do not require any server-side storage or tracking of token state. In a stateless JWT-based authentication system, all the information needed to verify and authenticate a token is contained within the token itself.

Developers and QA staff should include functional access control unit and integration tests.

Example Attack Scenarios

Scenario #1: The application uses unverified data in a SQL call that is accessing account information:

```
pstmt.setString(1, request.getParameter("acct"));
ResultSet results = pstmt.executeQuery( );
```

An attacker simply modifies the browser's 'acct' parameter to send whatever account number they want. If not correctly verified, the attacker can access any user's account.

```
https://example.com/app/accountInfo?acct=notmyacct
```

Scenario #2: An attacker simply forces browses to target URLs. Admin rights are required for access to the admin page.

```
https://example.com/app/getappInfo
https://example.com/app/admin_getappInfo
```

If an unauthenticated user can access either page, it's a flaw. If a non-admin can access the admin page, this is a flaw.

Notable Common Weakness Enumerations (CWEs) included are

- *CWE-200: Exposure of Sensitive Information to an Unauthorized Actor*,
- *CWE-201: Insertion of Sensitive Information Into Sent Data*, and
- *CWE-352: Cross-Site Request Forgery*.

Content Security Policy (CSP):

Here is an example of a Content Security Policy (CSP) policy:

Content-Security-Policy: default-src 'self'; script-src 'self' https://example.com; style-src 'self' https://example.com; img-src 'self' https://example.com;

This policy restricts the following:

default-src: This directive specifies the allowed sources for all resources, except for scripts, styles, and images. In this policy, only resources from the same origin (self) are allowed.
script-src: This directive specifies the allowed sources for scripts. In this policy, only scripts from the same origin (self) and scripts from the https://example.com domain are allowed.
style-src: This directive specifies the allowed sources for styles. In this policy, only styles from the same origin (self) and styles from the https://example.com domain are allowed.
img-src: This directive specifies the allowed sources for images. In this policy, only images from the same origin (self) and images from the https://example.com domain are allowed.
You can also use other CSP directives to restrict additional resources, such as fonts, media, and frames.

- Top 1: Hands-on