Module 2

Web Application Authentication

- Authentication Fundamentals
- Two Factor and Three Factor Authentication
- Web Application Authentication- Password Based, Built-in HTTP, Single Sign-on, Custom Authentication, Validating credentials
- Secured Password Based Authentication: Attacks against Password, Importance of Password Complexity
- Design Flaws in Authentication Mechanisms
- Implementation Flaws in Authentication Mechanisms
- Securing Authentication

Authentication Fundamentals

Authentication is the process in which a user proves that they are who they claim to be. Whether it involves a key card and PIN or a username and password, this process is composed of two steps: identification and confirmation.

Identification is the first step of claiming to be a certain person, and confirmation is the second step, which allows a subject to prove that claim.

When you log in to your online bank, you go through the authentication process by entering a username and password. Entering a username is how you *identify* yourself. In order to prove your claim, you must *confirm* by entering the password associated with the username.

The process of authentication is the first step in any access control mechanism. Authentication is important for confirming the user's identity. The next step is authorization, to determine whether or not a user has been given the rights to access certain data or perform specific operations.

A well-designed access control mechanism will first perform authentication and then perform authorization whenever access is requested to any protected resource.

Proving Your Identity

There are three classes that can be used to prove identity:

• Something you *know*
• Something you *are*
• Something you *have*

Something you *know* - providing the authentication with something that you *know.* Example – password, PIN or pass phrase, last four digits of your adhar card, your data of birth in a particular format.

Something you *have* - providing the authentication with something that you *have.* Example- a digital certificate, smart card, or a security token (for example, RSA SecurID), mobile authentication app etc. It is like having a key, to get past the locked door.

Something you *are* - authentication factors is based on something that's part of who you are. Example fingerprint, retinal pattern, hand geometry, or even the topography of your face. These factors are commonly referred to as *biometrics* because they're based on a person 's intrinsic physical qualities or behavioral characteristics.

Two-Factor Authentication (2FA)

Two-Factor Authentication (2FA) is a security process that requires users to provide **two different types of authentication factors** to verify their identity. This enhances security by making it harder for attackers to gain access to an account or system, even if they have stolen a password.

Example of 2FA in Action

- Logging into an online bank account:
    1. Enter your password (something you know).
    2. Receive a one-time code (OTP) on your phone and enter it (something you have).
- Web applications requiring smart card (something you have) and fingerprint (something you are).

Three-Factor Authentication (3FA)

Three-Factor Authentication (3FA) adds an additional layer of security by **requiring three different authentication factors.** It follows the same categories as 2FA but uses a third factor for even stronger security.

Example of 3FA in Action

- **Accessing a highly secure facility:**
    1. Enter a **PIN or password** (something you know).
    2. Scan an **access card or RSA token** (something you have).
    3. Provide a **fingerprint scan** (something you are).

Web applications requiring Government access, military, high-security environments

Web Application Authentication

There are several **authentication mechanisms** used in web applications, which can be categorized into different approaches based on security requirements and ease of implementation.

- Password Based
- Built-in HTTP
- Single Sign-on
- Custom Authentication
- Validating credentials

1. Password-Based Authentication

This is the most common and basic authentication method where users provide a username and password to access a web application.

Steps:

- The user enters a **username** and **password** in a login form.
- The system checks the entered credentials against a stored database of registered users.
- If the credentials match, the user is granted access; otherwise, authentication fails.

The HTTP specification provides two built-in authentication mechanisms, called Basic access authentication and Digest access authentication, where the **username and password** are encoded/ hashed and sent in the HTTP headers.

2. Built-in HTTP Authentication

A simple authentication mechanism provided by the HTTP protocol, which does not require a custom login page.

Types of Built-in HTTP Authentication:

a) **Basic Authentication**

Basic authentication begins when a user attempts to access a protected resource on a web server. When a request goes out for a file, such as http://www.website.cxx/secure/private-file.html, the web server will respond with the **401 Authorization Required** response code, shown here:

```
HTTP/1.1 401 Authorization Required
Server: HTTPd/1.0
Date: Sat, 27 Nov 2004 10:18:15 GMT
WWW-Authenticate: Basic realm="Secure Area"
Content-Type: text/html
Content-Length: 311

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/1999/REC-html401-19991224/loose.dtd">
<HTML>
  <HEAD>
    <TITLE>Error</TITLE>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html;
                                    charset=ISO-8859-1">
  </HEAD>
  <BODY><H1>401 Unauthorized.</H1></BODY>
</HTML>
```

When the browser sees this response, it will pop up a dialog box requesting the user to enter their credentials.

After the credentials are entered and the user clicks the OK button, the browser will take those values and combine them as –'username:password'. This concatenated value is then base64-encoded and submitted via a GET request to the web server under the Authorization header. For example, if the username is "stewie" and the password is "griffin," then the concatenated plaintext would be "stewie:griffin," and then after base64 encoding it would be "c3RJd2llOmdyaWZmaW4=" (attached to the Authorization header).

Authorization: Basic c3RJd2llOmdyaWZmaW4=

If the credentials are accepted by the server, then the protected resource is returned to the user.

Despite the fact that a username and password are required to access the resource, this method of authentication is insecure for several reasons.

Insecure Transmission Despite the fact that the username and password are base64 encoded, it's a trivial exercise for an attacker to intercept these values and decode them. This is because encoding is not encryption, so it lacks the security provided by encryption.

Repeated Exposure the credentials themselves must be submitted with every single request for a protected resource. The username and password are exposed over and over with each request to the web server.

Insecure Storage Because the username and password must be submitted to the web server with each request, the browser caches the authentication credentials.

To improve the security of an application already utilizing Basic access authentication, require that all communications occur over SSL. By using an encrypted channel such as SSL, you are able to mitigate the risk of plaintext transmission and the repeated exposure of the credentials.

Although this doesn't solve the issues related to insecure storage, it does mitigate the two most significant threats.

b) **Digest Authentication**

- A more secure version of Basic Authentication.
- Uses hashing (MD5 hashing algorithm) to prevent passwords from being transmitted in plain text.
- Uses a nonce (unique value) to make replay attacks difficult.

3. Single Sign-On (SSO) Authentication

Single sign-on (SSO) authentication allow a user to log in to a single interface and gain access to multiple, independently secured systems. With web applications, SSO allows a user to enter their credentials and access multiple web applications, without needing to re-enter credentials.

Google Accounts is an example of an Internet SSO system for Google services. By logging in to your Google account once, you're able to access multiple services provided by Google such as Gmail, Google Talk, YouTube, Drive etc.

As everyone continues to migrate more toward web-based applications, it will only become more and more difficult to manage credentials for the disparate systems. The adoption of SSO will help to solve this problem by reducing the number of credentials that must be remembered. Still, it's important to keep in mind that the convenience of a single set of credentials comes at the risk of a single point of failure. But if the credentials are compromised, then all the applications will be accessible.

Examples of **centralized authentication servers - OAuth, SAML, or OpenID Connect (OIDC)**.

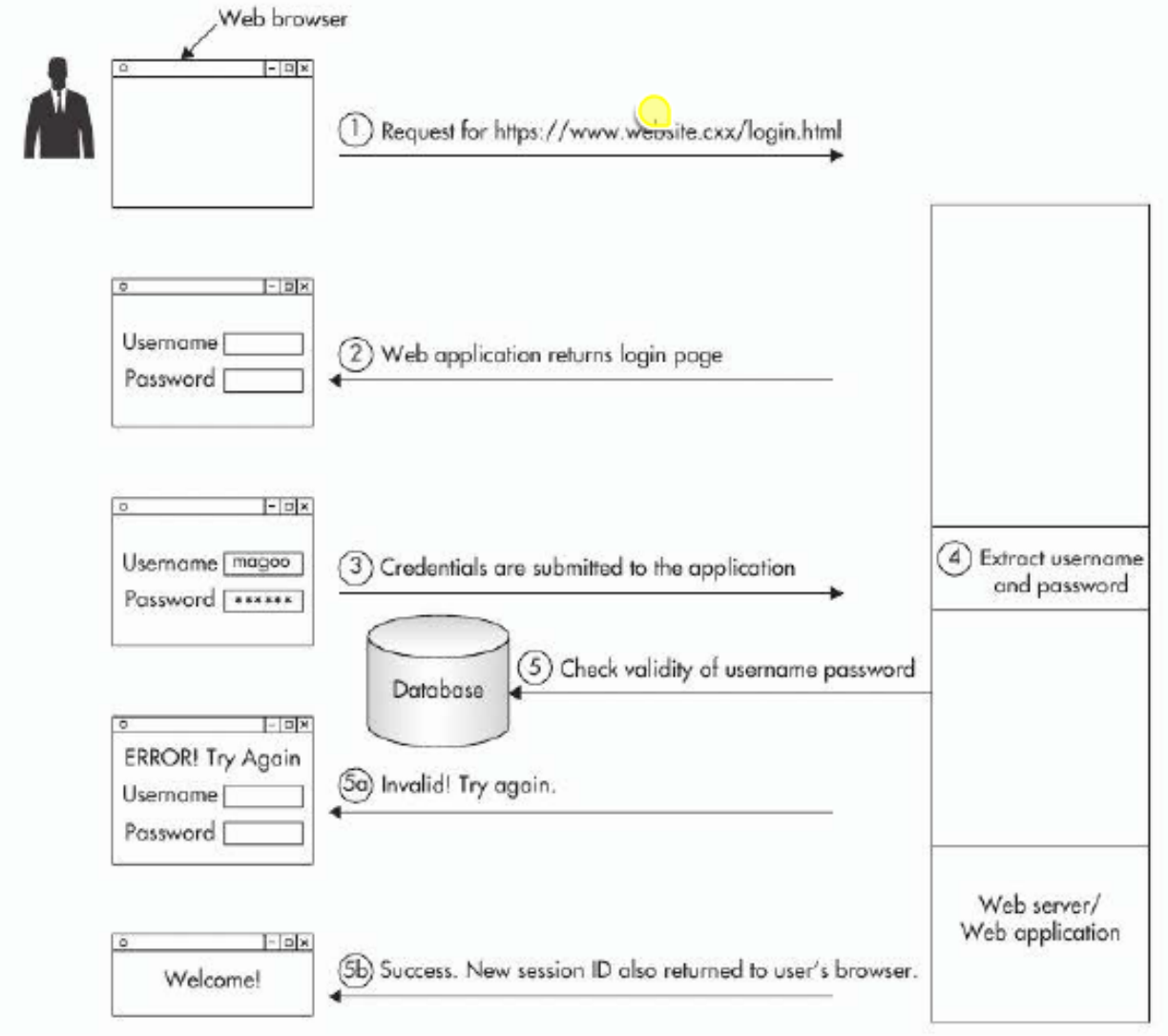
4. Custom Authentication

A **tailor-made authentication system** designed based on specific business or security needs instead of using pre-built methods. It is the most common authentication mechanism implemented in web applications.

Example – The login process for the majority of web applications -

1. Using the browser, a user requests the login page from the web application, such as http://www.website.cxxllogin.html.

2. The web application returns the login page to the browser.



3. The user enters their username and password into the input fields and submits the form, which now contains the credentials, to the web application. When the form is submitted to the web application, the browser will include the form fields as part of the HTTP POST request that is submitted.

4. Upon receiving the request, the web application parses the information in the HTTP message and extracts the values of the username and password.

5. The application logic then queries a back-end data store (for example, a database) to determine whether or not the password entered is associated with the username entered.

a. If the matching is unsuccessful, then the web application will send the user an error message along with the login page again.

b. If the password is successfully matched to the associated username, then the application will establish a session with the user. Most applications establish a session with a user by

generating a session ID value and returning it to the user. The session ID value is returned by setting a cookie in the HTTP response.

Set-Cookie : JSESSIONID=vPGwNrxDzSshFhd

6. When the browser receives and parses the HTTP response, it will observe the SetCookie directive in the HTTP header and store the value of the session ID.

a. Because the session ID value was set in a cookie, the browser will now automatically submit the session ID value alongside all requests made to the web application.

This acts as a form of persistent authentication because the user no longer needs to enter their username and password to authenticate every request going to the application.

b. Whenever the web application parses an HTTP request, it will see the session ID value and know that an existing session has already been established. It will use this session ID to authorize each request within the application logic.

5. Validating Credentials

The process of validating credentials is more complex than simply "determining whether or not the supplied password is associated with the supplied username." The credentials must be stored and used securely and access must be permitted only after multiple authentications.

There are several common ways to determine whether the correct password has been entered, and they depend on how the passwords are being stored in the back-end system and where the validation is done.

There are generally two variables involved in the validation of credentials: the location of the comparison logic and how the password is stored. The first variable concerns the location of the comparison logic, and it is usually found either within the application or within the database. The second variable is how the password is stored, which usually is either plaintext or hashed (or encrypted). Taking the cross-product of these variables gives us the four different approaches:

1. Comparison logic in the application with plaintext passwords
2. Comparison logic in the database with plaintext passwords
3. Comparison logic in the application with hashed passwords
4. Comparison logic in the database with hashed passwords

Comparison logic in the Application with Plaintext Passwords

Here, the application sends a request (for example, SQL query or LDAP query) to the back-end database to retrieve the record associated with the username. If the username does not exist, then the validation process fails.

If a record is associated with the username, then the application-based logic compares the plaintext password provided by the user to the plaintext password from the retrieved record. If they match, then the credentials are valid.

Comparison logic in the Database with Plaintext Passwords

This technique involves sending a SQL query to the back-end system to return any records with matching fields that correspond to the supplied username and the supplied password. The database performs the username and password comparisons against the corresponding fields. If any records are returned, then the user has provided a legitimate set of credentials.

Comparison logic in the Application with Hashed Passwords

A request is first made to the back-end datastore to retrieve the record associated with the user-supplied username.

Because hashing is a one-way transformation, the user-supplied password must be hashed using the same algorithm that was used to hash the stored password in order to compare them. If the two hashed values are equal, then the user-supplied password was valid, and then authentication succeeds.

Comparison logic in the Database with Hashed Passwords

Comparing hashed passwords on the back end also involves hashing the user-supplied password before it is sent to the datastore. The back-end system logic attempts to match the supplied username and the hashed form of the supplied password with a record in the system.

If a match occurs, then the record is returned, and the application assumes that the credentials were valid.

Securing Password-Based Authentication

Passwords are the most popular way of confirming your identity to a web application. So we must be more cautious about the ways of password attacks.

a password-based authentication system and how you can successfully defend against them.

Common attack variations include:

• Dictionary attack

• Brute-force attack

• Precomputed dictionary attack

• Rubber-hose attack

Dictionary Attack

A *dictionary attack* is based on precompiled list of commonly used words, phrases, or passwords-often derived from real-world password leaks, through social media. In some cases, real dictionaries may be used, and creating permutations, such as appending a digit or special character at the end of each word, may be done as well.

The attacker will successively attempt each password until they have successfully guessed the password or the list is exhausted.

If the application is online, then time limitation is a concern. However, if offline attacks are done multiple dictionaries are used and their permutations to attack the password.

Brute-Force Attack

A *brute-force attack* is also referred to as an exhaustive key search, and in theory involves attempting every single possible key. In reality, limits are usually placed on a brute-force attack based on length and character set.

It is frequently used in offline attacks against hashed password values.

Precomputed Dictionary Attack

One downside of dictionary attacks and brute-force attacks is that every time you want to crack a password, you must rehash all the dictionary or brute-force values. (If the actual password is hashed and stored).

This hashing of all the dictionary and Brute-force values takes a lot of time. Hence, all the dictionary values or combination of characters are pre-hashed and stored, by the attacker. So that only hashed values similarity need to be checked. This method of cracking passwords is called a *precomputed dictionary attack.*

Rubber-Hose Attack

In "rubber-hose" attack intruder uses any sort of physical coercion to extract the value of the password from an individual.

Design Flaws in Authentication Mechanisms

Design Flaws in Authentication Mechanisms

Bad Passwords

Brute-Forcible Login

Verbose Failure Messages

Vulnerable Transmission of Credentials

Password Change Functionality

Forgotten Password Functionality

“Remember Me” Functionality

User Impersonation Functionality

Incomplete Validation of Credentials

Non-Unique Usernames

Predictable Usernames

Predictable Initial Passwords

Insecure Distribution of Credential

Design Flaws in Authentication Mechanisms

Authentication mechanisms are crucial for securing applications and preventing unauthorized access. However, design flaws can weaken security and expose systems to attacks. Shortcomings in the design of a simple authentication model, can leave the application highly vulnerable to unauthorized access.

Below are some common authentication design flaws:

1. Bad Passwords

- Weak or commonly used passwords (e.g., "password123", "admin", "qwerty") make it easier for attackers to guess them.
- Lack of password complexity requirements (e.g., length, special characters, numbers) increases vulnerability.
- Reusing old passwords across multiple sites poses a risk if one is compromised.
- Still set to default password.
- **Solution**: Enforce strong password policies, use password managers, and implement multi-factor authentication (MFA).

2. Brute-Forcible Login

- Systems that do not limit failed login attempts are vulnerable to brute-force attacks, where attackers try numerous password combinations.
- **Solution**: Implement rate limiting(limit the number of login), CAPTCHA, account lockout after multiple failed attempts, and MFA.

3. Verbose Failure Messages

- Detailed error messages (e.g., "Incorrect password" or "Username does not exist") help attackers confirm valid usernames.
- When a login attempt fails, attacker can infer that at least one piece of information was incorrect. However, if the application informs which piece of information was invalid, this reduces the effectiveness of the login mechanism.

Solution: Use generic messages like "Invalid username or password" to avoid leaking information.

4. Vulnerable Transmission of Credentials

- Sending passwords in plaintext over HTTP or email exposes them to interception via packet sniffing or man-in-the-middle (MITM) attacks.

Even if login occurs over HTTPS, credentials will be disclosed to unauthorized parties if the application handles them in an unsafe manner, as given below:

If credentials are transmitted as query string parameters, then the credentials will be available in various places—like, within the user's browser history and within the web server logs.

- **Solution**: Use HTTPS/TLS encryption, avoid storing plaintext passwords, and implement secure authentication protocols like OAuth or OpenID Connect.

5. Password Change Functionality

- If poorly designed, attackers can exploit weak authentication steps (e.g., not requiring the old password) to hijack accounts.

Many web applications do not provide any way for users to change their password. However, this functionality is necessary for a well-designed authentication mechanism for two reasons:

- Periodic enforced password change mitigates the threat of password compromise.
- Users who suspect that their passwords may have been compromised need to be able to quickly change their password to reduce the threat of unauthorized use.

- **Solution**: Require authentication before changing a password, notify users of changes via email, and implement logging for password changes.

6. Forgotten Password Functionality

- Weak recovery mechanisms (e.g., security questions with predictable answers) allow attackers to reset passwords.

The forgotten password recovery functionality often involves presenting the user with a question to recover the password. Questions about mothers' maiden names, memorable dates, favorite colors, and the like will generally have a much smaller set of answers than the set of possible passwords, and will be easier for the attacker to discover with the information from social media.

- **Solution**: Use secure password reset links via email or SMS with a short expiration time, enforce MFA during resets, and limit reset attempts.

7. "Remember Me" Functionality

- Storing persistent login credentials insecurely (e.g., in plaintext cookies) can allow attackers to hijack sessions.
- **Solution**: Use secure cookies with HTTPOnly and Secure flags, and avoid storing credentials in cookies.

8. User Impersonation Functionality

- Admins or support staff having the ability to log in as other users without proper logging can lead to misuse.
- **Solution**: Log impersonation events, notify the affected user, and restrict impersonation privileges to authorized personnel.

9. Incomplete Validation of Credentials

- Systems that do not fully validate passwords or allow partial matches make authentication less secure.
- **Solution**: Ensure complete validation of usernames, passwords, and MFA tokens during authentication.

10. Non-Unique Usernames

- Allowing duplicate usernames can create confusion and security risks, leading to unintended account access.
- **Solution**: Enforce unique usernames during registration and prevent duplicates in the database.

11. Predictable Usernames

- Usernames based on easily guessable patterns (e.g., first initial + last name, employee ID) make targeted attacks easier.
- **Solution**: Allow users to choose their own usernames and use randomization techniques for system-generated usernames.

12. Predictable Initial Passwords

- Default passwords (e.g., "123456", "password") or predictable ones (e.g., username + "123") are easy for attackers to guess.
- **Solution**: Generate random initial passwords and require users to change them on first login.

13. Insecure Distribution of Credentials

- Sending passwords via email or storing them in insecure locations can lead to unauthorized access.
- **Solution**: Use secure channels (e.g., encrypted communication) for credential distribution and avoid storing credentials in plaintext.

By addressing these authentication flaws, systems can significantly enhance their security and protect user accounts from common attacks.

Implementation Flaws in Authentication

Implementation flaws in authentication occur when security mechanisms are improperly coded or configured, leading to vulnerabilities that attackers can exploit. Below are some common implementation flaws:

1. Fail-Open Login Mechanisms

- A **fail-open** authentication system allows access when an error occurs instead of denying it.
- This can happen due to:
    - Improper error handling (e.g., if a database connection fails, the system grants access by default).

  - Logic flaws in authentication flow (e.g., skipping password validation under certain conditions).
  - API authentication failures (e.g., if an external authentication service is unavailable, access is granted instead of denied).
- **Example**: If an application relies on a third-party authentication provider and does not handle timeout errors properly, it may allow users to log in without verifying credentials.
- **Solution**: Implement **fail-secure** mechanisms where errors result in access denial, use robust error handling, and log authentication failures for security monitoring.

2. Defects in Multistage Login Mechanisms

- Multistage login involves multiple steps (e.g., entering username, then password, then MFA).
- Implementation defects can include:
  - **Skipping authentication steps**: Attackers can bypass some stages due to improper session handling.
  - **Session fixation issues**: If a session token is set before authentication is complete and not reset afterward, an attacker could hijack the session.
  - **Improper validation of previous steps**: If a system does not verify earlier stages, users may be able to skip required authentication steps.
- **Example**: A banking website asks for a username, then a password, then an OTP. If the OTP validation step does not verify the previous steps, an attacker might submit an OTP without entering the correct password.
- **Solution**: Ensure that each stage validates the previous step, regenerate session tokens after login, and test for bypass vulnerabilities.

3. Insecure Storage of Credentials

- Storing credentials improperly can lead to account takeovers if attackers gain access.
- Common storage flaws:
  - **Plaintext storage**: Storing passwords without encryption makes them readable if a database is compromised.
  - **Weak hashing algorithms**: Using outdated hashing techniques (e.g., MD5, SHA-1) makes password cracking easier.
  - **Lack of salting**: If passwords are hashed without a unique salt, attackers can use precomputed hash tables (rainbow tables) to crack them.
- **Example**: If an application stores passwords as plaintext in a database or logs them in error messages, an attacker who gains database access can steal all user credentials.

- **Solution**:
    - Store passwords using strong hashing algorithms like **bcrypt, Argon2, or PBKDF2** with a unique salt.
    - Never store or log plaintext credentials.
    - Use **HSMs (Hardware Security Modules)** or dedicated key management services to store cryptographic keys securely.

Securing Authentication

To protect user accounts and prevent unauthorized access, authentication mechanisms must be designed with strong security measures. Below are key principles for securing authentication:

1. Use Strong Credentials

- Weak passwords are easily guessed or cracked. Ensuring strong credentials is the first layer of defense.
- **Best Practices:**
    - Require passwords to be **long (12+ characters)** and include **uppercase, lowercase, numbers, and special characters**.
    - Implement a **password blacklist** to prevent commonly used or breached passwords.
    - Encourage or enforce **passphrases** (e.g., "CorrectHorseBatteryStaple").
    - Use **Multi-Factor Authentication (MFA)** to add an extra layer of security.

2. Handle Credentials Secretively

- Credentials must be protected from exposure, both in storage and transmission.
- **Best Practices:**
    - Store passwords using **strong hashing algorithms** (e.g., **bcrypt, Argon2, PBKDF2**).
    - Do **not store passwords in plaintext** or logs.
    - Use **secure password managers** for managing credentials.
    - Never transmit credentials **over unencrypted channels** (always use **HTTPS/TLS**).

3. Validate Credentials Properly

- Incorrect validation can allow attackers to bypass authentication.
- **Best Practices:**
  - Ensure the **username and password** match exactly, preventing partial matches.
  - Validate **Multi-Factor Authentication (MFA) codes** properly, using secure time-based tokens.
  - Prevent **timing attacks** by using constant-time comparison functions.
  - Enforce **case-sensitive usernames and passwords** unless explicitly designed otherwise.

4. Prevent Information Leakage

- Error messages and responses should not disclose sensitive information that attackers can exploit.
- **Best Practices:**
  - Use **generic authentication failure messages** (e.g., "Invalid username or password").
  - Avoid revealing **whether a username exists** in password reset forms.
  - Do not expose credentials or session tokens in **URL parameters**.
  - Mask password input fields to prevent **shoulder surfing**.

5. Prevent Brute-Force Attacks

- Brute-force attacks involve repeatedly trying different passwords until the correct one is found.
- **Best Practices:**
  - Implement **account lockouts** after multiple failed attempts.
  - Use **progressive delays** or CAPTCHA to slow down attacks.
  - Log and monitor **failed login attempts** for anomaly detection.
  - Require **MFA** to mitigate brute-force risks.

6. Prevent Misuse of the Password Change Function

- Attackers may exploit weak password change mechanisms to take over accounts.
- **Best Practices:**
  - Require **authentication (current password or MFA)** before allowing a password change.
  - Notify users via **email or SMS** when their password is changed.
  - Enforce **password complexity rules** to prevent weak new passwords.
  - Do not allow reusing previous passwords.

7. Prevent Misuse of the Account Recovery Function

- Attackers can exploit weak recovery mechanisms to reset passwords and take over accounts.
- **Best Practices:**
  - Use **secure password reset links** (one-time, time-limited tokens sent via email).
  - Do not allow users to reset passwords using easily guessable **security questions**.
  - Limit the number of **password reset attempts** to prevent abuse.
  - Notify users of **any password reset attempts**, even if unsuccessful.

8. Log, Monitor, and Notify

- Security incidents must be tracked and responded to in real time.
- **Best Practices:**
  - Log **all authentication-related events** (logins, failed attempts, password resets).
  - Monitor for **suspicious login activity** (e.g., multiple failed attempts, logins from unusual locations).
  - Implement **real-time notifications** for critical actions like **password changes, failed login attempts, or new device logins**.
  - Use **SIEM (Security Information and Event Management) systems** to analyze and detect threats.