Chapter 2

Core Defense Mechanisms:

Web application security is fundamentally challenged by the fact that all user input is untrusted, leading applications to rely on common core defense mechanisms. These include controlling user access to data and functionality, validating and safely handling user input, responding appropriately to active attacks, and providing administrative controls for monitoring and configuration. While these mechanisms are designed to protect applications, they also form most of the application's attack surface, making a deep understanding of how they work—and where they fail—essential for both securing and effectively attacking web applications.

Handling User Access:

Web applications must control user access to data and functionality across different roles. This relies on:

- Authentication
- Session management
- Access control

and a weakness in any one of these can compromise the entire application.

Authentication:

Authentication is the process of verifying that a user is who they claim to be and is fundamental to controlling access in web applications. Most applications use username–password logins, sometimes enhanced with multi-factor methods like tokens, certificates, or smartcards for higher security, along with supporting features such as registration and password recovery. Despite appearing simple, authentication mechanisms often contain design or implementation flaws that allow attackers to guess credentials, discover usernames, or bypass login logic, making them a critical target for attacks.

Various methods of authentication are:

1. Password-based (cookies or JWT)
2. Single Sign On
3. MFA
4. Passwordless
5. Biometric

Session Management:

Session management tracks an authenticated user across multiple HTTP requests by creating a server-side session and issuing a unique session token. This token, usually sent via cookies, allows the application to associate each request with the correct user and enforce access control, and it should expire after inactivity. Session management is a major attack target because if attackers can guess, steal, or misuse session tokens, they can impersonate users without logging in, often due to weak token generation or insecure handling.

What happens after login (step by step)

1. User sends username + password
2. Application verifies credentials (authentication)
3. Application creates a session
4. Application sends the user a session token
5. Browser stores the token
6. Browser sends the token with every future request
7. Server uses the token to identify the user

What is a session?

A session is:

- A set of data stored on the server
- Linked to one logged-in user

What is a session token?

A session token is:

- A unique string (random value)
- Given to the user's browser
- Used as a key to find the session on the server

Access Control:

Access control is how an application decides whether a user is allowed to do something or see certain data after the user is identified. It checks the user's role, permissions, and limits to decide if a request should be allowed or denied. Because applications often have many users, roles, and rules, these checks can become complex.

Developers may forget to add checks in some places or make wrong assumptions, which can create security weaknesses. As a result, access control errors are common and can let attackers access things they should not.

Handling User Input:

All user input is untrusted, and many attacks happen when malicious input causes unexpected behaviour. Applications must handle input safely, but input problems can occur anywhere. There is no single, simple way to protect against all bad input.

Varieties of Input:

Web applications receive many types of input, so validation rules must vary. Some inputs can be tightly restricted, like usernames with fixed length and letters only. Others, such as addresses, need more flexibility but still require limits and safety checks. Some inputs, like blog posts, must allow almost anything and be handled safely instead of rejected. Applications also receive hidden or server-generated data like cookies, which must be carefully checked because attackers can change them, and suspicious changes should be rejected and logged.

Approaches to input handling:

1. Reject known bad

"Reject known bad" uses a blacklist to block input that matches known attack patterns and allows everything else. This method is weak because attacks can be written in many different ways that blacklists may miss, and new attack techniques appear all the time, making blacklists quickly outdated.

2. Accept known good

"Accept known good" uses a whitelist to allow only input that meets specific, safe criteria and blocks everything else. This is the most effective way to prevent attacks when feasible, because only approved input can pass. However, it isn't always practical—for example, names or other data may need characters that could also be used in attacks—so it can't solve all input-handling problems.

3. Sanitization

Sanitization handles input that can't be guaranteed safe by cleaning or encoding it to prevent harm. Dangerous characters are removed or escaped before processing. This approach is often effective, like encoding input to prevent cross-site scripting, but can be tricky when input needs to allow multiple types of potentially unsafe data, in which case stricter boundary checks may be better.

4. Safe data handling

Safe data handling means designing the way an application processes input so that it cannot cause harm, even if the input is malicious. Instead of just validating input, developers use safe methods—like parameterized database queries to prevent SQL injection—or avoid unsafe practices, such as sending user input directly to system commands. While not applicable to every situation, this approach is a powerful way to reduce vulnerabilities.

5. Semantic checks

Semantic checks ensure that input not only follows correct syntax but also makes sense in context. Some attacks use input that looks normal but is used in the wrong way—for example, changing a hidden form field to access another user's account. Syntactic validation alone can't catch this; the application must verify that the data is valid for the specific user or situation.

Boundary Validation:

All user input is always untrusted, even if it looks valid, passes browser checks, or comes through HTTPS, because the moment the server receives it, it crosses a trust boundary. Client-side validation only improves user experience and can be easily bypassed, so it cannot be relied on for security.

Validating input just once at the server entry point is dangerous because the same data is reused in different places, transformed during processing, and exposed to different types of attacks (SQL injection, XSS, command injection, XML/SMTP injection), which often require different and sometimes conflicting defenses.

The correct approach is boundary validation, where every component treats its input as potentially malicious and validates or encodes data each time it crosses a boundary (browser → app, app → database, app → API, app → browser, app → email).

In short: never trust input because it was validated earlier—always validate data based on where and how it is used.

Multistep Validation and Canonicalization:

Problems occur when user input is validated in multiple steps or decoded after validation, because attackers can hide malicious content and make it appear later. By splitting dangerous input (like <script>) into parts, changing the order of validation steps, or using encoding tricks (%27, %%2727), attackers can bypass filters and re-create the attack after validation is done. Since data may look safe at one step but become dangerous after decoding or transformation, multistep cleaning is unreliable and complex to fix.

The safest approach is to decode input first, then validate it once in the correct context, and reject dangerous input instead of trying to clean it.

Handling Attackers:

When building a secure application, you must assume skilled attackers will try to break it. So, the application should not just try to prevent attacks, but also handle them properly when they happen. This includes safely handling errors without revealing sensitive details, keeping:

1. audit logs to record what attackers did,
2. alerting administrators when suspicious activity occurs, and
3. reacting to attacks (such as blocking accounts or IPs) to limit damage.

These measures help control attacks, frustrate attackers, and give owners clear evidence of what happened.

Handling Errors:

Even with good input validation, errors are unavoidable, especially when attackers interact with an application in unexpected ways. While normal user errors are usually found during testing, attacker-triggered errors often appear only after the application is live. For security, the application must handle errors safely, recover if possible, and show only simple, non-technical error messages to users. It should never display debug details or system error messages, because these can reveal sensitive information and help attackers plan further attacks. Instead, detailed error information should be logged internally for developers and administrators, so they can fix security weaknesses without exposing anything to users.

Maintaining Audit Logs:

Audit logs are records of important actions in an application and are mainly used to investigate attacks after they happen. A secure application should log key events like login attempts, password changes, money transactions, blocked access attempts, and requests that look malicious. Good logs record details such as time, IP address, user account, and session ID so owners can understand what happened, how the attack worked, and what data was affected.

These logs must be strongly protected, because if attackers can read or change them, they may gain sensitive information or hide their tracks, so logs are often stored securely or on separate systems to keep them safe.

Alerting Administrators:

Alerting administrators means detecting attacks as they happen and reacting quickly, instead of only investigating later using logs. When suspicious activity is detected, admins can block IPs or accounts, limit damage, or even temporarily shut down the application. A good alerting system avoids spamming alerts and focuses on real threats, such as unusual request patterns, strange business activity, known attack inputs, or tampered hidden data.

While tools like web application firewalls can catch obvious attacks, they often miss subtle ones, so the most effective alerts are built inside the application itself, using its own validation and logic checks to accurately detect malicious behaviour with fewer false alarms.

Reacting to Attacks:

Reacting to attacks means that an application automatically takes action when it detects suspicious or malicious behavior, instead of only logging or alerting it. Since attackers usually test an application by sending many crafted requests to find weaknesses, the application can slow down their requests, log them out, or block their session to make attacks harder.

These reactions won't stop very determined attackers, but they discourage casual attackers and buy time for administrators to respond. While this does not replace fixing vulnerabilities, it adds an extra defense layer that reduces the chance of remaining flaws being discovered and exploited.

Managing the Application:

Applications need admin features to manage users, roles, settings, and monitoring, and these features are an important part of security. Often, admin tools are built into the same web interface as normal features, which makes them a major target for attackers. If security is weak, attackers can gain admin access, create powerful accounts, exploit issues like cross-site scripting, or misuse poorly tested admin functions. Because admin tools can perform dangerous actions, compromising them can allow an attacker to take over the whole application or even the server.

--The End--