

Chapter 22: MERN Security

22.1 Web Security Fundamentals

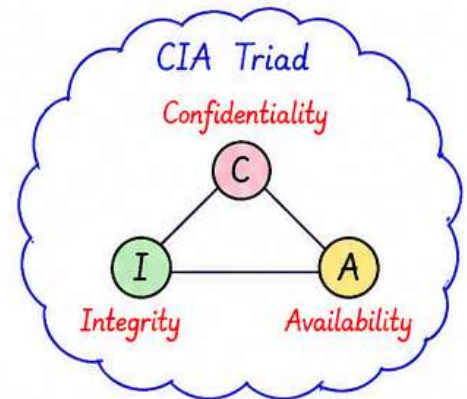
22.1.1 What is Web Security?

- * Web security refers to the practices, technologies and processes used to **protect websites** and **web applications** from attacks, unauthorized access, damage or data theft.
- * It ensures the **confidentiality**, **integrity** and **availability** of data exchanged between users and servers.
- * Web security is essential for building **trust** and providing a **safe experience** to users.



Key Points:

- ✓ Protects applications and data from malicious attacks.
- ✓ Prevents unauthorized access and data breaches.
- ✓ Ensures data privacy, integrity and availability,
- ✓ Builds user trust and maintains application reliability.



22.1.2 Why Web Security is Important?

- * Cyber attacks are increasing every day targeting websites and web applications.
- * Vulnerabilities can lead to **data theft**, **financial loss**, and **reputation damage**.
- * Secure applications protect **user data** and **business information**.

Impact of Insecurity

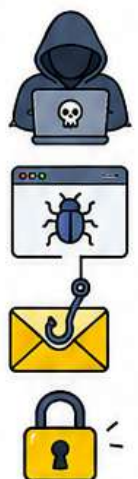
- Data Breach
- Financial Loss
- Reputation Damage
- Legal Consequences
- Loss of User Trust



22.1.3 Common Web Security Threats

Some of the most common web security threats are:

Threat	Description
SQL Injection	Attacker injects malicious SQL queries to access or manipulate database.
Cross-Site Scripting (XSS)	Attacker injects malicious scripts into web pages viewed by other users.
Cross-Site Request Forgery (CSRF)	Attacker tricks users into performing unwanted actions on a web application.
Authentication Attacks	Attacker tries to gain unauthorized access by breaking authentication.
Sensitive Data Exposure	Sensitive information is exposed due to poor security practices.



22.1.4 Web Security Best Practices

- * Always **validate** and **sanitize** user inputs.
- * Use **HTTPS** to encrypt data in transit.
- * Use secure **authentication** and **authorization**.
- * Keep dependencies and software **up to date**.
- * Implement proper **error handling** and **logging**.

Best Practices



HTTPS



Auth



Validation



Updates



Logging

Key Takeaway:



Web security is the foundation of any **safe** and reliable web application. Understanding threats and following best practices helps in building secure and **trusted** applications.

22.2 Common Web Vulnerabilities

22.2.1 What are Common Web Vulnerabilities?

- * Web vulnerabilities are **weaknesses** in **web applications** that can be **exploited** by attackers to gain unauthorized access, steal data or execute malicious code.
- * They occur due to **poor coding practices, misconfigurations**, or lack of security measures.
- * Common web vulnerabilities include **SQL Injection, XSS, CSRF, Broken Authentication**, etc.



Key Points:

- ✓ Web vulnerabilities can be exploited.
- ✓ They lead to data theft and system compromise.
- ✓ Prevention involves secure coding and regular audits.

22.2.2 Top Common Web Vulnerabilities (OWASP Top 5)

The OWASP (Open Web Application Security Project) identifies the most critical web vulnerabilities.

No.	Vulnerability	Description	Example	Impact	Icon
1.	Injection	Attackers inject malicious code into inputs that are sent to an interpreter (e.g., SQL, NoSQL, OS commands).	SQL Injection, NoSQL Injection, Command Injection	Data breach, data theft, system compromise.	
2.	Broken Authentication	Flaws in authentication allow attackers to compromise user accounts.	Weak passwords, Session hijacking, Credential stuffing	Unauthorized access, account takeover, data theft.	
3.	Sensitive Data Exposure	Applications fail to protect sensitive data such as passwords, tokens, or personal information.	Unencrypted data transmission, storing plaintext passwords	Data breach, privacy violation, financial loss.	
4.	XML External Entities (XXE)	Improper processing of XML input can allow attackers to interfere with the application.	XXE attacks, file read, SSRF	Data disclosure, server-side attacks, DoS.	
5.	Broken Access Control	Applications do not properly restrict access to authenticated users.	Privilege escalation, IDOR (Insecure Direct Object Reference)	Unauthorized access to data, privilege escalation.	

22.2.3 How to Prevent Vulnerabilities?

- * Validate and **sanitize** all user inputs.
- * Use **parameterized queries** to prevent injections.
- * Implement strong **authentication** and **authorization**.
- * Encrypt sensitive data using **HTTPS** and **hashing**.
- * Keep dependencies and frameworks **updated**.
- * Perform regular **security testing** and **code reviews**.



Key Takeaway:



Web vulnerabilities can put applications and users at risk. By understanding common vulnerabilities and following best practices, we can build **secure** and **resilient** web applications.

22.3 SQL Injection Concepts

22.3.1 What is SQL Injection?

- * SQL Injection is a **code injection** technique used by attackers to interfere with the queries that an application makes to its database.
- * Attackers insert **malicious SQL code** into input fields to manipulate the database.
- * It can lead to **unauthorized access, data theft, data modification** or even complete **database takeover**.

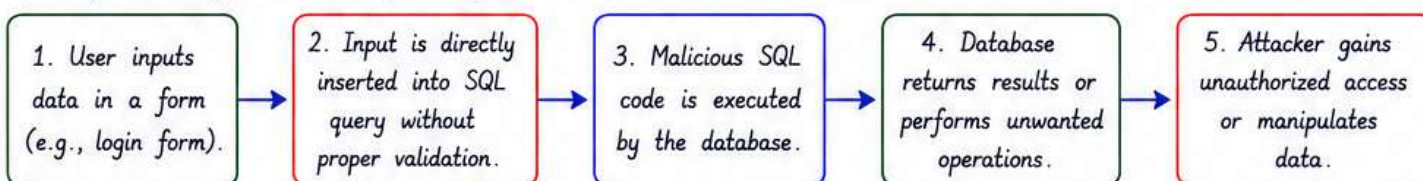


Key Points:

- ✓ Exploits vulnerabilities in database queries.
- ✓ Can bypass authentication and access sensitive data.
- ✓ Can modify or delete data.
- ✓ Prevention is better than cure.

22.3.2 How SQL Injection Works?

SQL Injection happens in the following flow:



22.3.3 Examples of SQL Injection

1. Normal Query:

```
SELECT * FROM users WHERE username = 'admin' AND password = 'password123';
```

2. Malicious Input:

If attacker enters → **' OR '1' = '1'**

3. Resulting Query:

```
SELECT * FROM users WHERE username = '' OR '1' = '1' AND password = '';
```

4. Effect:

- * **'1'='1'** is always true, so the query returns all rows from the users table, bypassing authentication.

Common Injection Strings

- **' OR '1'='1' --** → Always true
- **' OR '1'='1' /*** → Always true
- **' UNION SELECT null--** → Union attack
- **'; DROP TABLE users; --** → Delete table
- **' OR 'a'='a** → Bypass login



22.3.4 Types of SQL Injection

Type	Description
In-Band SQLi	Attacker uses the same channel to launch the attack and retrieve data.
Blind SQLi	Attacker cannot see query results but infers data through true/false responses.
Union-based SQLi	Uses UNION operator to combine results of attacker's query with the original query.
Error-based SQLi	Uses database error messages to extract information.
Out-of-Band SQLi	Data is extracted using channels like DNS or HTTP requests.

22.3.5 How to Prevent SQL Injection?

- * Use **parameterized queries** or **prepared statements**.
- * Validate and **sanitize** all user inputs.
- * Use **least privilege access** for database users.
- * Avoid **dynamic SQL queries**.
- * Use **stored procedures**.
- * Display **generic error messages** to users.
- * Keep database, libraries and frameworks **updated**.
- * Implement **Web Application Firewall (WAF)**.



Key Takeaway:



SQL Injection is a **dangerous** vulnerability that can compromise your application and data. Understanding how it works and following **secure coding practices** can protect your application from SQL Injection attacks.

22.4 NoSQL Injection



```
{ name: "admin",  
  role: "user" }
```



22.4.1 What is NoSQL Injection?

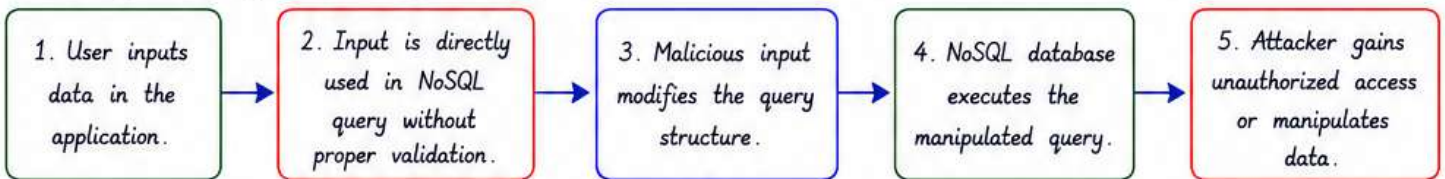
- * NoSQL Injection is a **code injection** technique that attacks NoSQL databases.
- * It occurs when user input is sent to a NoSQL query without proper **validation** and **sanitization**.
- * Attackers can manipulate queries to **bypass authentication**, access sensitive data, or perform unauthorized actions.

Key Points:

- ✓ Targets NoSQL databases like MongoDB, CouchDB, etc.
- ✓ Can bypass authentication mechanisms.
- ✓ Can access, modify or delete sensitive data.
- ✓ Proper validation and sanitization can prevent it.

22.4.2 How NoSQL Injection Works?

NoSQL Injection happens in the following flow:



22.4.3 Example of NoSQL Injection (MongoDB)

1. Normal Query (Login):

```
db.users.find({"username": "admin", "password": "pass123"})
```

2. Malicious Input:

If attacker enters:

```
username: { "$ne": null }, password: { "$ne": null }
```

3. Resulting Query:

```
db.users.find({"username": { "$ne": null }, "password": { "$ne": null }})
```

4. Effect:

- * The **\$ne** operator means "not equal".
- * Since all values are usually not null, the query returns **all documents**, bypassing authentication.

Common NoSQL Injection Operators

- **\$ne** - Not Equal
- **\$gt** - Greater Than
- **\$lt** - Less Than
- **\$in** - Matches any value in array
- **\$nin** - Not in array
- **\$regex** - Regular expression match
- **\$where** - Execute JavaScript



22.4.4 Types of NoSQL Injection

Type	Description
Authentication Bypass	Bypass login by manipulating query conditions.
Data Extraction	Extract sensitive data from the database.
Data Manipulation	Modify or delete data without authorized access.
Remote Code Execution	Execute JavaScript code on the database server using \$where.
Denial of Service	Send complex queries to overload the database.

22.4.5 How to Prevent NoSQL Injection?

- * Always **validate** and **sanitize** all user inputs.
- * Use **parameterized queries** or query builders.
- * Avoid using **\$where** operator.
- * Use **strict schema validation**.
- * Apply **least privilege access** to database users.
- * Keep database and libraries **updated**.
- * Implement input **whitelist** instead of blacklist.
- * Use Web Application Firewall (**WAF**).



Key Takeaway:



NoSQL Injection can be as dangerous as SQL Injection. It can lead to **authentication bypass**, **data theft**, **data manipulation** and even **remote code execution**. Secure coding practices and proper validation are the key to preventing these attacks.

22.5 XSS

22.5.1 What is XSS?

- * XSS (Cross-Site Scripting) is a **code injection** attack where attackers inject malicious scripts into web pages viewed by other users.
- * It occurs when an application includes **untrusted data** in a web page without proper validation or escaping.
- * Attackers can **steal cookies**, **session tokens**, or perform unauthorized actions on behalf of users.

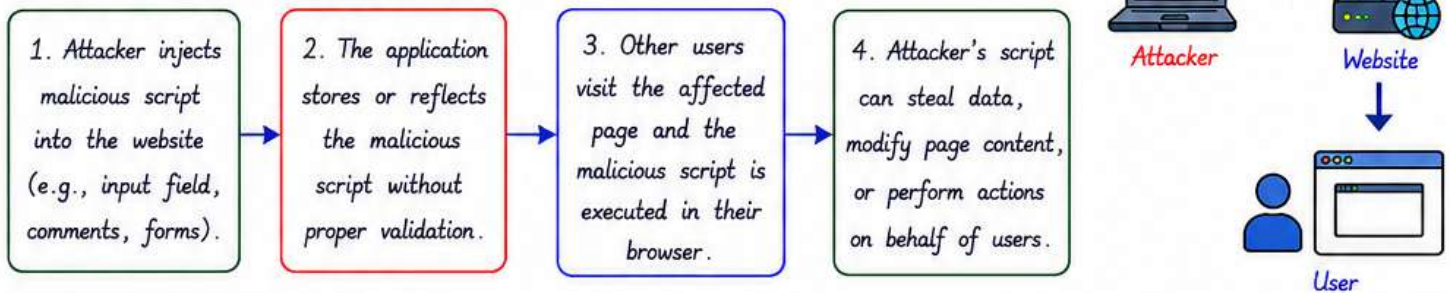


Key Points:

- ✓ Allows attackers to execute scripts in users' browsers.
- ✓ Can steal sensitive data and session information.
- ✓ Can deface websites or redirect users.
- ✓ Proper validation and encoding can prevent XSS.

22.5.2 How XSS Works?

XSS happens in the following flow:



22.5.3 Types of XSS

Type	Description	Example	Stored or Reflected	Risk Level
1. Stored XSS	Malicious script is permanently stored on the server (e.g., in database, comments).	<code><script> alert('XSS') </script></code>	Stored	High
2. Reflected XSS	Malicious script is reflected back to the user through URL or form input.	<code>http://site.com/search?q= <script> alert('XSS') </script></code>	Reflected	Medium
3. DOM-based XSS	Malicious script is executed by modifying the DOM using client-side scripts.	<code>document.getElementById('msg').innerHTML = location.hash;</code>	DOM-based	Medium

22.5.4 Example of XSS

1. User Input (Comment Box):

```
<script> alert('You have been hacked!') </script>
```

2. Stored in Database

3. When other users view the comment:

```
<script> alert('You have been hacked!') </script>
```

4. Effect: Script executes in user's browser.

- * Shows alert box.
- * Can steal cookies, session tokens, or redirect user.

22.5.5 How to Prevent XSS?

- * Always **validate** and **sanitize** user inputs.
- * **Encode** special characters before displaying user input.
- * Use Content Security Policy (**CSP**).
- * Use **HttpOnly** and **Secure** cookies.
- * Avoid using **innerHTML**; use **textContent** instead.
- * Keep libraries and frameworks **updated**.
- * Use Web Application Firewall (**WAF**).



Key Takeaway:



XSS attacks can compromise users and applications by executing **malicious scripts**. Understanding the types of XSS and following **secure coding practices** helps in building safe, secure and **user-trusted** web applications.



22.6 CSRF

22.6.1 What is CSRF?

- * CSRF (Cross-Site Request Forgery) is an **attack** that tricks a user's browser into performing **unwanted actions** on a web application where the user is authenticated.
- * The attacker forces the user to execute a request **without** the user's knowledge or **consent**.
- * It can lead to actions like **changing email**, **transferring money**, **updating profile**, etc.

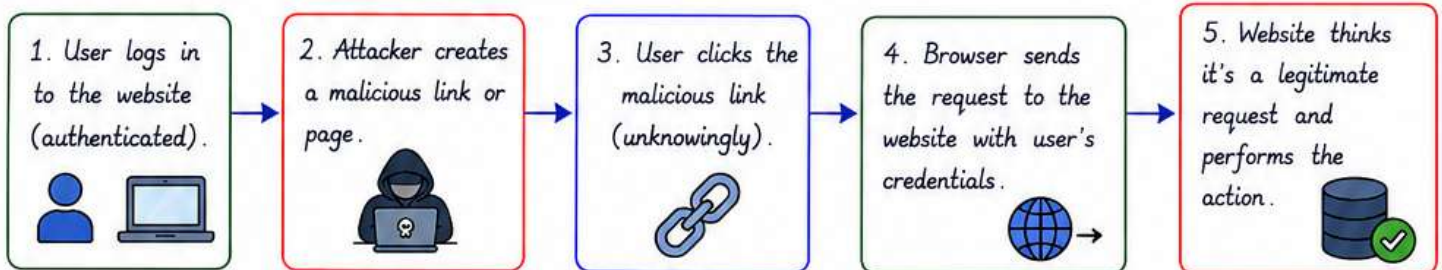


Key Points:

- ✓ Exploits the trust that a website has in the user's browser.
- ✓ Can perform unauthorized actions.
- ✓ Does not steal data, but performs actions.
- ✓ Can be prevented with tokens and proper validation.

22.6.2 How CSRF Works?

CSRF attack happens in the following flow:



22.6.3 Example of CSRF

1. User is logged in to his bank account.
2. Attacker sends a link like:
`http://bank.com/transfer?to=attacker&amount=1000`
3. If user clicks the link, the browser sends request to bank.com with user's session.
4. Bank website thinks it's a valid request and transfers money.

Example Request (Hidden from user):

```
POST /transfer HTTP/1.1
Host: bank.com
Cookie: sessionid=abc123...
Content-Type: application/x-www-form-urlencoded
to=attacker&amount=1000
```

Common CSRF Targets

- Changing password or email
- Fund transfer
- Updating profile details
- Placing orders
- Changing account settings
- Deleting data



22.6.4 Types of CSRF

Type	Description
Login CSRF	Forces the victim to log in to the attacker's account.
Action CSRF	Forces the victim to perform unwanted actions on the website.

22.6.5 How to Prevent CSRF?

- * Use **CSRF tokens** (unique, unpredictable tokens).
- * Validate tokens on every state-changing request.
- * Use **SameSite cookies** to restrict cross-site requests.
- * Use **POST requests** for state-changing actions.
- * Verify the **Referer** or **Origin** header.
- * Implement user **re-authentication** for critical actions.
- * Do not use **GET requests** for state-changing operations.
- * Use **secure coding practices** and keep frameworks **updated**.



Key Takeaway:



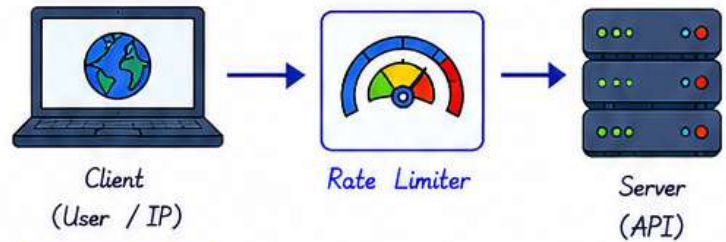
CSRF attacks exploit the trust between a user's browser and a website. Using **tokens**, proper **validation** and **secure practices** can protect applications from CSRF attacks.



22.8 Rate Limiting

22.8.1 What is Rate Limiting?

- * Rate Limiting is a **security** technique that limits the number of requests a user or IP can make to a server within a given time period.
- * It helps **protect** applications from **abuse**, **brute force** attacks, and **denial-of-service (DoS)** attacks.
- * It ensures **fair usage** of resources and improves the performance and availability of the API.

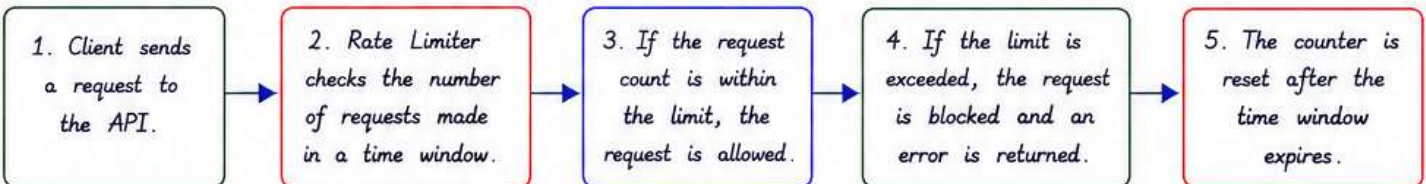


Key Points:

- ✓ Prevents abuse and excessive API usage.
- ✓ Protects against brute force and DoS attacks.
- ✓ Improves API performance and stability.
- ✓ Can be based on IP, user, or API key.
- ✓ Returns an error (e.g., 429 Too Many Requests) when the limit is exceeded.

22.8.2 How Rate Limiting Works?

Rate Limiting works in the following flow:



22.8.3 Example of Rate Limiting

- * Limit: 100 requests per minute per IP
- * Scenario:
A user with IP 192.168.1.10 makes 105 requests in 1 minute.
- * Result:
The first 100 requests are allowed.
The next 5 requests are blocked and the server returns:

```
HTTP/1.1 429 Too Many Requests
Content-Type: application/json
{
  "message": "Too Many Requests. Please try again later.",
  "retryAfter": 30
}
```

Common Rate Limiting Strategies

- **Fixed Window Counter** - Limit requests in fixed time windows.
- **Sliding Window Log** - Keeps a log of requests in a time window.
- **Token Bucket** - Tokens are added at a fixed rate; requests consume tokens.
- **Leaky Bucket** - Requests are processed at a fixed rate; excess requests are dropped.
- **IP-based Limiting** - Limit requests based on client IP address.
- **User-based Limiting** - Limit requests based on authenticated user.

22.8.4 Rate Limiting Headers

Header	Description
X-RateLimit-Limit	Maximum number of requests allowed in the time window.
X-RateLimit-Remaining	Number of requests remaining in the current time window.
X-RateLimit-Reset	Time (in seconds) when the limit resets.
Retry-After	Time (in seconds) to wait before making another request.

22.8.5 How to Implement Rate Limiting in Node.js

```
const rateLimit = require('express-rate-limit');

const limiter = rateLimit({
  windowMs: 60 * 1000, // 1 minute
  max: 100, // Limit each IP to 100 requests
  message: {
    error: 'Too Many Requests',
    message: 'Please try again later.'
  },
  standardHeaders: true, // Return rate limit info in headers
  legacyHeaders: false // Disable the X-RateLimit-* headers
});

app.use('/api', limiter); // Apply to all /api routes
```

JS

22.8.6 Best Practices

- * Use rate limiting on all **public** APIs.
- * Set different limits for different endpoints.
- * Use **IP** and user-based rate limiting for better control.
- * Return **meaningful error messages** and retry time.
- * **Monitor** and **log** rate limit violations.
- * **Adjust** limits based on traffic and resource capacity.



Key Takeaway:

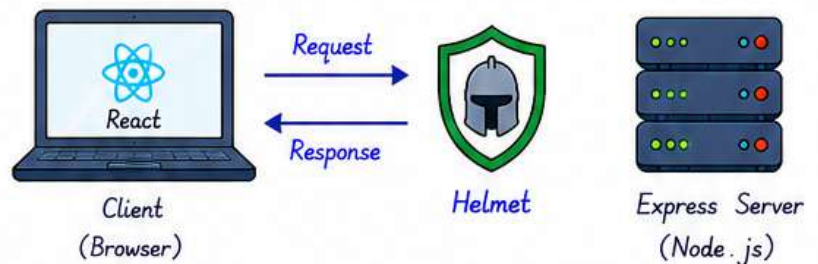


Rate Limiting is essential for **protecting** your MERN application from abuse and ensuring **fair usage** of APIs. Implement it with the right strategy and always **monitor** and **tune** it.

22.9 Helmet

22.9.1 What is Helmet?

- * Helmet is a **security middleware** for Express.js that helps secure your MERN application by setting various HTTP headers.
- * It helps protect the app from **common web vulnerabilities** by configuring secure HTTP headers.
- * It is **easy to use**, **configurable** and follows security best practices.

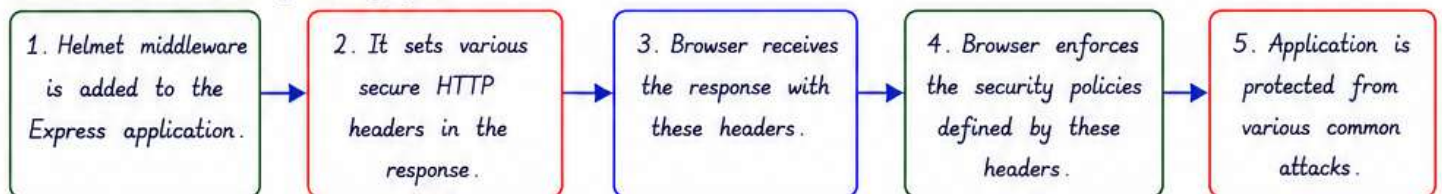


Key Points:

- ✓ Helmet sets secure HTTP headers automatically.
- ✓ Helps protect against common attacks.
- ✓ Easy to integrate with Express.js.
- ✓ Improves the security posture of your application.
- ✓ Recommended for all production applications.

22.9.2 How Helmet Works?

Helmet works in the following flow:



22.9.3 Common Helmet Headers

Header	Description
Content-Security-Policy	Helps prevent XSS attacks by controlling resources the browser is allowed to load.
X-Content-Type-Options	Prevents browsers from MIME type sniffing. Set to "nosniff".
X-Frame-Options	Prevents clickjacking attacks by controlling if the page can be framed.
Strict-Transport-Security	Enforces HTTPS by instructing browsers to only use HTTPS.
Referrer-Policy	Controls how much referrer information is sent with requests.
Permissions-Policy	Controls which browser features can be used by the site.
Cross-Origin-Opener-Policy	Helps isolate the browsing context to reduce certain attacks.

22.9.4 Example (Express.js)

```
const express = require('express');
const helmet = require('helmet');
const app = express();

// Use Helmet middleware
app.use(helmet());

app.get('/', (req, res) => {
  res.send('Helmet is protecting our app!');
});

app.listen(5000, () => {
  console.log('Server running on port 5000');
});
```

JS

22.9.5 Helmet Configuration Example

```
app.use(helmet({
  contentSecurityPolicy: {
    directives: {
      defaultSrc: ["'self'"],
      scriptSrc: ["'self'", "https://trusted.cdn.com"],
      styleSrc: ["'self'", "'unsafe-inline'"]
    }
  },
  crossOriginOpenerPolicy: { policy: "same-origin" },
  referrerPolicy: { policy: "no-referrer" }
}));
```

JS

22.9.6 Best Practices

- * Use Helmet in all Express applications.
- * Keep Helmet **updated** to get latest security headers.
- * Review and customize headers according to your needs.
- * Do not disable headers unless absolutely necessary.
- * Always **test** your application after configuration changes.



Key Takeaway:



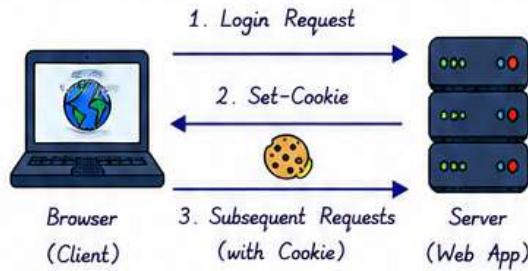
Helmet is a simple but powerful middleware that **secures** your MERN application by setting various HTTP headers. It helps protect against **common web attacks** and improves the overall security of your application with minimal effort.



22.10 Secure Cookies

22.10.1 What are Cookies?

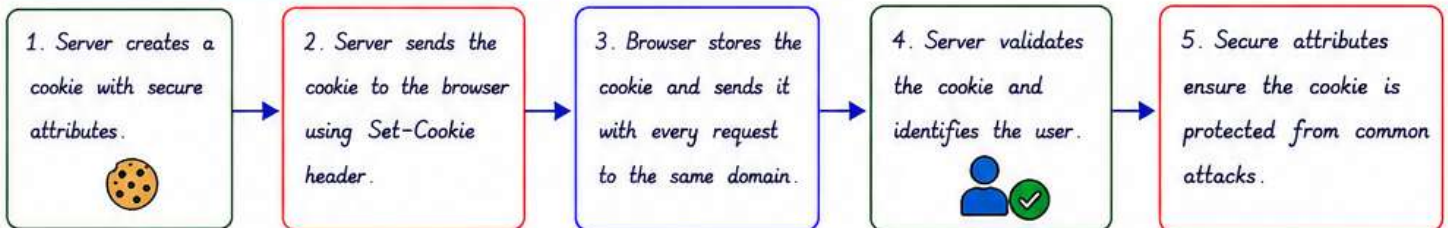
- * Cookies are small pieces of data stored in the user's browser and sent with every request to the server.
- * They are used for **session management**, user preferences, authentication, and tracking.
- * If not secured, cookies can be **stolen** or **misused** by attackers.



Key Points:

- ✓ Cookies maintain state in stateless HTTP.
- ✓ Secure cookies protect user sessions and data.
- ✓ Proper cookie settings prevent attacks like XSS, CSRF, and Session Hijacking.

22.10.2 How Secure Cookies Work?



22.10.3 Important Cookie Attributes

Attribute	Description	Why It's Important
Secure	Ensures the cookie is sent only over HTTPS connections.	Prevents cookies from being intercepted over HTTP.
HttpOnly	Makes the cookie inaccessible via JavaScript (document.cookie).	Protects against XSS attacks.
SameSite	Controls when cookies are sent. • Strict - Never send on cross-site requests. • Lax - Send on top-level navigation. • None - Send on all requests (Requires Secure).	Helps prevent CSRF attacks.
Max-Age / Expires	Defines the lifetime of the cookie.	Limits how long a cookie remains valid.
Path	Specifies the URL path for which the cookie is valid.	Limits cookie scope to a specific path.
Domain	Specifies the domain for which the cookie is valid.	Allows or restricts cookie sharing across subdomains.

22.10.4 Example (Express.js)

```
const express = require('express');
const app = express();

app.get('/set-cookie', (req, res) => {
  res.cookie('token', 'abc123', {
    httpOnly: true,
    secure: true,           // sent only over HTTPS
    sameSite: 'Strict',    // prevent CSRF
    maxAge: 1000 * 60 * 60, // 1 hour
    path: '/',
  });
  res.send('Secure cookie has been set!');
});
```

JS

22.10.5 Example Set-Cookie Header

```
Set-Cookie: token=abc123; HttpOnly; Secure; SameSite=Strict; Path=/; Max-Age=3600
```

22.10.6 Risks of Insecure Cookies

- **Session Hijacking** - Attackers steal session cookies to impersonate users.
- **XSS Attacks** - Cookies accessible via JavaScript can be stolen.
- **CSRF Attacks** - Cookies sent automatically with cross-site requests.
- **Man-in-the-Middle (MITM)** - Cookies intercepted over HTTP.

22.10.7 Best Practices

- * Always use HTTPS in production.
- * Set **Secure** and **HttpOnly** flags.
- * Use **SameSite** to prevent CSRF.
- * Set appropriate **Max-Age / Expires**.
- * Avoid storing sensitive data in cookies.
- * Regularly review and rotate session cookies.



22.10.8 Summary



Secure cookies are essential for protecting user sessions and sensitive data in MERN applications. Using attributes like **Secure**, **HttpOnly**, and **SameSite** helps prevent common web vulnerabilities and ensures a safer user experience.



22.11 Password Security

22.11.1 What is Password Security?

- ✳ Password security refers to the practices and techniques used to **protect** passwords from being guessed, stolen, or cracked.
- ✳ Strong password security helps protect user accounts, data, and systems from **unauthorized access**.
- ✳ Weak passwords are one of the most common causes of **security breaches**.



Key Points:

- ✓ Strong passwords prevent unauthorized access.
- ✓ Use long, random, and complex passwords.
- ✓ Store passwords securely (hashed + salted).
- ✓ Regularly update and manage passwords.
- ✓ Enable additional security like MFA.

22.11.2 Characteristics of a Strong Password

- At least 12-16 characters long.
- Includes uppercase letters (A-Z).
- Includes lowercase letters (a-z).
- Includes numbers (0-9).
- Includes special characters (!, @, #, \$, %, etc.).
- Not a common word or easily guessable.
- Unique for each account.



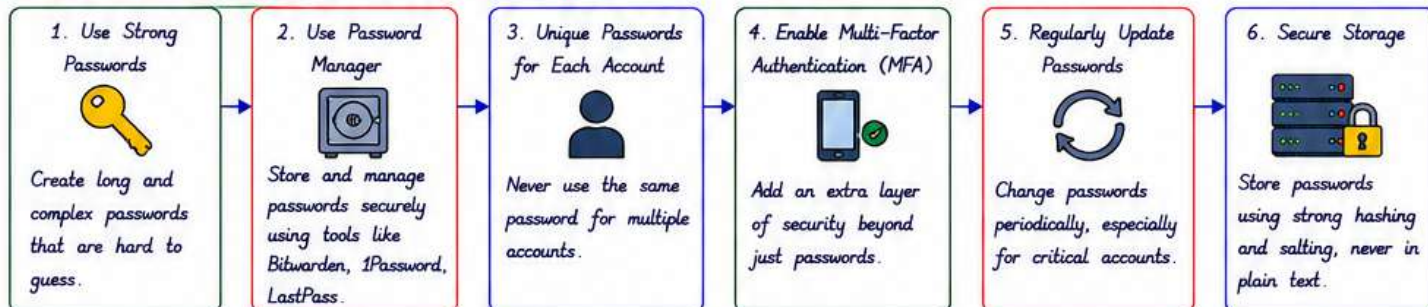
22.11.3 Weak Password Examples

- 123456
- password
- qwerty
- admin
- letmein
- password123
- welcome@123



Avoid using common or predictable passwords!

22.11.4 Best Practices for Password Security



22.11.5 Password Storage Best Practices

Practice	Description
Hashing	Convert passwords into fixed-length hash values using strong algorithms.
Salting	Add a unique random value (salt) to each password before hashing.
Key Stretching	Use algorithms like bcrypt, scrypt, or Argon2 to make brute-force attacks harder.
No Plain Text	Never store passwords in plain text.
Secure Transmission	Use HTTPS to protect passwords in transit.

22.11.6 Example (Node.js with bcrypt)

```
const bcrypt = require('bcrypt');

// Hashing Password
const saltRounds = 12;
bcrypt.hash('MySecureP@ss123!', saltRounds, (err, hash) => {
  if (err) throw err;
  console.log('Hashed Password:', hash);
});

// Comparing Passwords
bcrypt.compare('MySecureP@ss123!', storedHash, (err, result) => {
  if (err) throw err;
  console.log('Password Match:', result);
});
```

JS

22.11.7 Common Attacks on Passwords

- Brute Force Attack - Trying many combinations.
- Dictionary Attack - Using a list of common passwords.
- Credential Stuffing - Using stolen credentials from other breaches.
- Phishing Attack - Tricking users to reveal their passwords.
- Keylogging - Capturing keystrokes to steal passwords.

22.11.8 Additional Security Tips

- ✳ Do not share your passwords with anyone.
- ✳ Be cautious of phishing emails and fake login pages.
- ✳ Logout after use, especially on public devices.
- ✳ Monitor account activity for any suspicious behavior.
- ✳ Report and update immediately if a password is compromised.



22.11.9 Summary



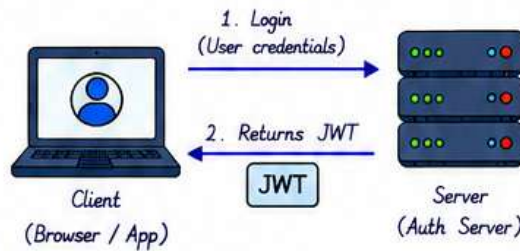
Strong password security is essential to protect user accounts and sensitive data. Use strong passwords, store them securely, and follow best practices to prevent attacks and keep your systems safe.



22.12 JWT Security

22.12.1 What is JWT?

- JWT (JSON Web Token) is a secure, compact and self-contained token used for **authentication** and **information sharing**.
- It is an open standard (RFC 7519) and is widely used in modern web applications.
- JWTs help in securely transmitting **user identity** and **claims** between client and server.



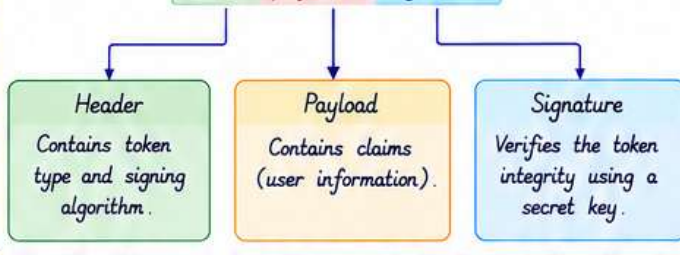
Key Points:

- ✓ Stateless authentication.
- ✓ Compact and URL safe.
- ✓ Self-contained token.
- ✓ Digitally signed for security.
- ✓ Works across different systems and domains.

22.12.2 Structure of a JWT

A JWT has three parts, separated by dots (.):

header.payload.signature

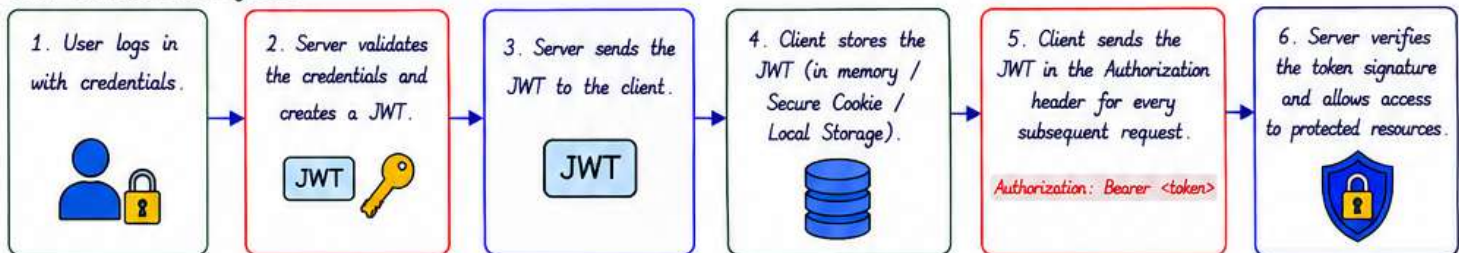


22.12.3 Types of Claims

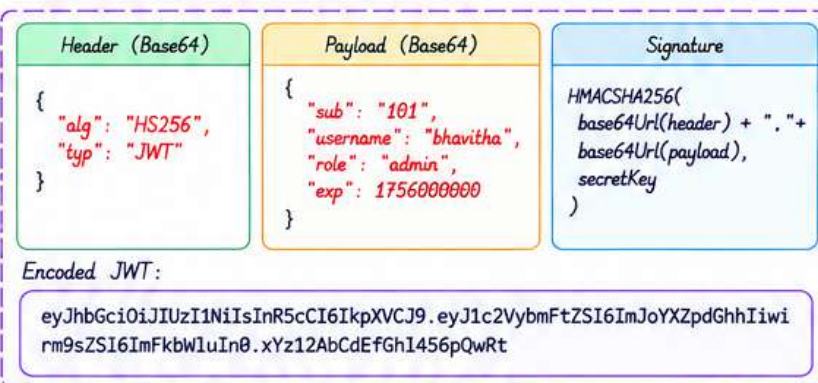
Type	Description	Examples
Registered Claims	Standard claims defined by JWT.	iss (issuer), exp (expiry), sub (subject), aud (audience)
Public Claims	Custom claims defined by the application.	role, department, username, email
Private Claims	Custom claims for specific use cases.	isAdmin, userType

22.12.4 How JWT Works?

JWT authentication flow:



22.12.5 Example JWT Token



22.12.7 Best Practices for JWT Security

- Always use **HTTPS** to transmit JWTs.
- Use **short expiry time** for tokens.
- Store tokens in memory or **httpOnly + Secure Cookies** (avoid plain localStorage).
- Use strong signing algorithms like **HS256** or **RS256**.
- Implement **token refresh** mechanism.
- Validate token on every protected request.
- Do not store sensitive information in the payload.

22.12.8 Common Attacks on JWT

- Token theft (via XSS, browser storage).
- Token **tampering** (changing payload without valid signature).
- None algorithm attack (setting "alg": "none").
- Token replay attack (reusing an old token).
- Long expiry tokens (greater risk if compromised).

22.12.6 Advantages of JWT

- ✓ Stateless and scalable (no server-side session storage needed).
- ✓ Works well with microservices and distributed systems.
- ✓ Compact and lightweight token format.
- ✓ Can contain custom information (claims).
- ✓ Supports single sign-on (SSO) and API authentication.



22.12.9 Summary



JWT is a powerful and secure mechanism for **authentication** and **authorization** in modern web applications. When implemented with best practices, it ensures **stateless**, **scalable** and **secure** access control.

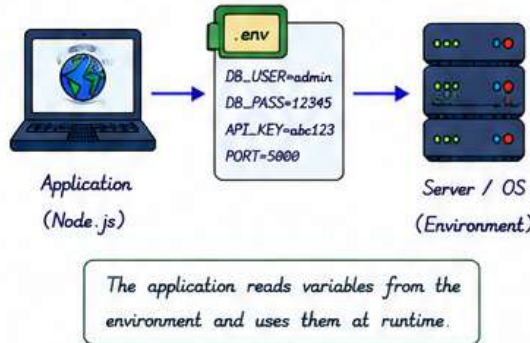


22.13 Environment Variables

22.13.1 What are Environment Variables?

How Environment Variables Work

- Environment variables are dynamic values that store **configuration settings** in the environment where an application runs.
- They are used to keep **sensitive information** (like API keys, database passwords) and environment-specific settings outside the code.
- Using environment variables improves **security**, **flexibility**, and **portability**.



Key Points:

- ✓ Store sensitive data securely.
- ✓ Avoid hardcoding values in code.
- ✓ Different values for different environments (dev, test, prod).
- ✓ Makes deployment easier.
- ✓ Improves collaboration and code maintainability.

22.13.2 Benefits

- Enhances security by keeping secrets out of code.
- Allows different configurations per environment.
- Simplifies code changes (no need to modify code).
- Improves portability across different systems.
- Supports 12-Factor App methodology.

22.13.3 Example .env File

```
# .env file
DB_HOST=localhost           # Database host
DB_PORT=5432                # Database port
DB_USER=myuser              # Database username
DB_PASS=mypassword123      # Database password
JWT_SECRET=superSecretKey  # Secret key for JWT
API_KEY=abc123xyz           # External API key
PORT=5000                   # Server port
NODE_ENV=development        # Environment mode
```

22.13.4 Accessing Environment Variables in Node.js

```
// 1. Install dotenv
// npm install dotenv

// 2. Load environment variables
require('dotenv').config();

// 3. Access variables
const dbHost = process.env.DB_HOST;
const dbUser = process.env.DB_USER;
const port = process.env.PORT;

console.log('Database Host :', dbHost);
console.log('Database User :', dbUser);
console.log('Server Port :', port);
```

Loads variables from .env file into process.env.

Access using process.env.VARIABLE_NAME

JS

22.13.5 Example .env Usage in Code

```
require('dotenv').config();

const express = require('express');
const app = express();
const PORT = process.env.PORT || 5000;
const API_KEY = process.env.API_KEY;

app.get('/', (req, res) => {
  res.send('App is running...');
});

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

JS

22.13.6 Common Use Cases

Use Case	Example Variables
Database Configuration	DB_HOST, DB_PORT, DB_USER, DB_PASS
Authentication	JWT_SECRET, SESSION_SECRET
External APIs	API_KEY, THIRD_PARTY_KEY
Application Settings	PORT, NODE_ENV, APP_NAME
Email / SMTP	SMTP_HOST, SMTP_USER, SMTP_PASS

22.13.7 Best Practices

- Never commit .env files to version control.
- Add .env to .gitignore.
- Use strong, unique secrets for sensitive values.
- Use different .env files for different environments.
- Validate required environment variables on startup.
- Use default values only when safe.



22.13.8 .gitignore Example

```
# .gitignore
.env
.env.local
.env.*.local
node_modules/
```

Note:
This prevents your environment files from being pushed to GitHub.



22.13.9 Environment Types

Environment	Purpose	Icon
development	For local development and testing.	💻
testing	For automated tests and QA.	🔍
staging	For pre-production testing.	⚙️
production	For live application.	🚀

22.13.10 Summary



Environment variables help separate configuration from code.
They improve security, flexibility, and portability of applications.
Always store sensitive data in environment variables and follow best practices.



22.14 Input Validation

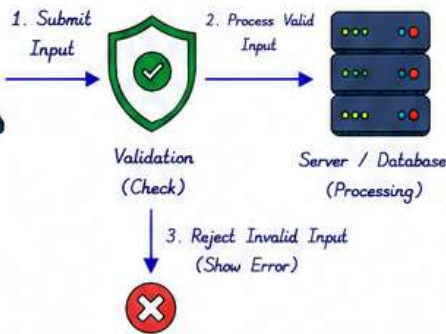
22.14.1 What is Input Validation?

- Input validation is the process of checking user input to ensure it is valid, safe, and expected before processing it.
- It helps prevent attacks like **SQL Injection**, **XSS**, **Command Injection**, and more.
- Always validate input on the **server side**, never trust client-side validation.



User
(Input)

How Input Validation Works



Key Points:

- ✓ Validate all input from users.
- ✓ Validate on the server side.
- ✓ Use allow-lists instead of deny-lists.
- ✓ Sanitize or escape input when needed.
- ✓ Provide meaningful error messages.
- ✓ Keep validation rules simple and consistent.

22.14.2 Why Input Validation is Important?

- Prevents security vulnerabilities.
- Ensures data integrity.
- Avoids unexpected errors or crashes.
- Enhances user experience.
- Helps maintain application reliability.



22.14.3 Common Attacks Due to Invalid Input

Attack	Description
SQL Injection	Malicious SQL queries are injected through user input to access or manipulate the database.
Cross-Site Scripting (XSS)	Malicious scripts are injected and executed in other users' browsers.
Command Injection	Attackers execute arbitrary system commands on the server.
Path Traversal	Attackers access files outside the intended directory.
NoSQL Injection	Manipulates NoSQL queries to bypass authentication or access data.

22.14.4 Types of Input Validation

Type	Description	Example
Presence Validation	Checks if the required field is not empty.	username, password should not be blank.
Type Validation	Ensures input is of the expected data type.	Age must be a number.
Format Validation	Checks if input matches a specific format.	Email must match abc@gmail.com
Length Validation	Ensures input length is within allowed limits.	Password length 8 to 16 characters.
Range Validation	Checks if input falls within a valid range.	Age must be between 18 and 100.
Whitelist Validation	Allows only known safe values.	Role can be only 'admin' or 'user'.
Sanitization	Removes or encodes harmful characters from input.	Remove <script> tags from comments.

22.14.5 Example (Node.js)

```
const express = require('express');
const validator = require('validator');
const app = express();
app.use(express.json());

app.post('/register', (req, res) => {
  const { email, age, password } = req.body;

  // Presence Validation
  if (!email || !age || !password) {
    return res.status(400).json({ error: 'All fields are required' });
  }

  // Format Validation
  if (!validator.isEmail(email)) {
    return res.status(400).json({ error: 'Invalid email format' });
  }

  // Type & Range Validation
  if (!Number.isInteger(age) || age < 18 || age > 100) {
    return res.status(400).json({ error: 'Age must be between 18 and 100' });
  }

  // Length Validation
  if (password.length < 8 || password.length > 16) {
    return res.status(400).json({ error: 'Password must be 8-16 characters' });
  }

  res.send({ message: 'Input is valid. Registration successful!' });
});
```

JS

22.14.6 Best Practices

- Validate all inputs on the server side.
- Use allow-lists, never deny-lists.
- Sanitize or escape output before displaying.
- Provide generic error messages (avoid exposing details).
- Keep validation rules consistent across the application.
- Log invalid input attempts for monitoring.



22.14.7 Example (SQL Injection Prevention)

```
// Using Parameterized Query (Node.js with mysql2)
const mysql = require('mysql2');
const connection = mysql.createConnection({ host: 'localhost', user: 'root',
  password: 'pass', database: 'db' });

const userId = req.body.userId; // untrusted input
const sql = 'SELECT * FROM users WHERE id = ?'; // parameterized query
connection.execute(sql, [userId], (err, results) => {
  if (err) throw err;
  res.send(results);
});
```

JS

22.14.8 Summary



- Input validation ensures only valid and safe data is processed.
- It prevents many common web vulnerabilities.
- Always validate, sanitize, and handle errors properly to build secure and robust applications.

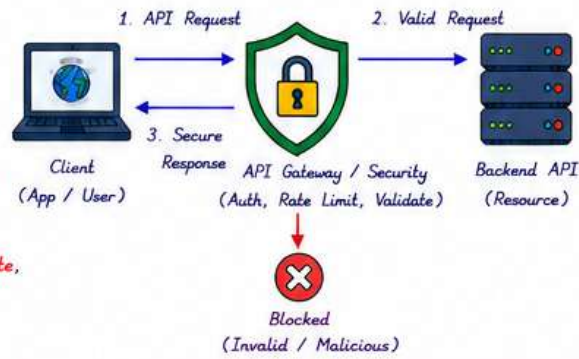


22.15 API Security

22.15.1 What is API Security?

- * API Security refers to **protecting** APIs from unauthorized access, abuse, and attacks.
- * APIs expose data and functionality, so securing them is critical.
- * Good API security ensures **confidentiality**, **integrity**, and **availability** of your application and data.
- * Never trust requests by default, always **authenticate**, **authorize** and **validate**.

How API Security Works



Key Points:

- ✓ Authenticate every request.
- ✓ Authorize access to resources.
- ✓ Validate all inputs.
- ✓ Use HTTPS everywhere.
- ✓ Limit and monitor API usage.
- ✓ Log and detect suspicious activity.
- ✓ Keep tokens and keys secure.

22.15.2 Common API Security Threats

Threat	Description
Broken Authentication	Weak or missing authentication allows unauthorized access.
Broken Authorization	Users can access resources they should not have.
Injection Attacks	SQL, NoSQL, Command injection through API inputs.
Sensitive Data Exposure	APIs return more data than necessary.
Broken Object Level Authorization	Accessing objects / records of other users.
Excessive Data Exposure	Poor filtering, pagination or field selection.
Rate Limit Abuse	Attackers overload APIs causing DoS.
Security Misconfiguration	Default configurations, open CORS, verbose errors, etc.

22.15.3 Best Practices for API Security

1. Use HTTPS	Always use HTTPS to encrypt data in transit.
2. Strong Authentication	Use API keys, OAuth2, or JWT for authentication.
3. Authorization	Implement role-based access control (RBAC).
4. Input Validation	Validate and sanitize all inputs on the server.
5. Rate Limiting	Limit requests to prevent abuse and brute-force attacks.
6. Use Least Privilege	Provide only the minimum access required.
7. Secure Error Handling	Return generic error messages, avoid exposing details.
8. Logging & Monitoring	Log API activity and monitor for suspicious behavior.
9. Keep APIs Updated	Regularly update dependencies and patch vulnerabilities.
10. Data Minimization	Return only required data, avoid overexposure.

22.15.4 Authentication Methods

- * **API Key**
Simple key sent in headers or query.
Example: `x-api-key: YOUR_API_KEY`
- * **OAuth 2.0**
Industry standard for delegated access.
- * **JWT (JSON Web Token)**
Token-based authentication using signed tokens.
- * **Basic Authentication**
Username and password encoded in base64.



22.15.6 Secure Headers for APIs

Header	Purpose
Content-Security-Policy	Prevents XSS attacks.
X-Content-Type-Options	Prevents MIME type sniffing.
X-Frame-Options	Protects against clickjacking.
Strict-Transport-Security	Forces HTTPS connections.
X-XSS-Protection	Enables browser XSS filtering.
Referrer-Policy	Controls referrer information.

Use **Helmet.js** (Node.js) to set secure headers easily.
`app.use(require('helmet')());`

22.15.7 Rate Limiting Example

```
// Install: npm i express-rate-limit
const rateLimit = require('express-rate-limit');
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // limit each IP to 100 requests
  message: 'Too many requests, please try again later.'
});
app.use('/api/', limiter);
```



22.15.5 Example: Secure API with JWT (Node.js)

```
// Install: npm i express jsonwebtoken
const express = require('express');
const jwt = require('jsonwebtoken');
const app = express();
app.use(express.json());

// Secret key (keep it strong and private)
const SECRET = 'SuperSecretKey123';

// Login - Generate Token
app.post('/login', (req, res) => {
  const { username, password } = req.body;
  if (username === 'admin' && password === 'pass123') {
    const token = jwt.sign({ userId: 1, role: 'admin' }, SECRET, { expiresIn: '1h' });
    return res.json({ token });
  }
  return res.status(401).json({ error: 'Invalid credentials' });
});

// Protected Route
function authenticateToken(req, res, next) {
  const authHeader = req.headers['authorization'];
  if (!authHeader) return res.sendStatus(401);
  const token = authHeader.split(' ')[1];
  jwt.verify(token, SECRET, (err, user) => {
    if (err) return res.sendStatus(403);
    req.user = user;
    next();
  });
}

app.get('/profile', authenticateToken, (req, res) => {
  res.json({ message: 'Welcome ' + req.user.role });
});

app.listen(5000, () => console.log('API running on port 5000'));
```

JS

22.15.8 Summary

- * Secure your APIs by authenticating, authorizing and validating requests.
- * Use HTTPS, strong tokens, rate limiting and proper error handling.
- * Follow least privilege and monitor API usage regularly.
- * Keep your APIs and dependencies updated.
- * Security is not optional - it is essential for every API.

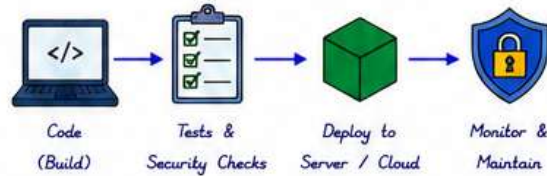


22.16 Secure Deployment

22.16.1 What is Secure Deployment?

- Secure Deployment is the process of deploying applications to production in a way that **protects** data, services, and infrastructure from attacks and **misconfigurations**.
- It ensures **confidentiality**, **integrity**, **availability**, and **reliability** of your application.
- A secure deployment process reduces risks and improves **trust** in your application.

Secure Deployment Flow



Secure deployment = Secure Code + Secure Config + Secure Infrastructure + Monitoring

Key Points:

- ✓ Keep secrets and keys safe.
- ✓ Use secure configurations.
- ✓ Regularly update and patch.
- ✓ Use HTTPS everywhere.
- ✓ Limit access and follow least privilege.
- ✓ Monitor, log, and respond quickly.
- ✓ Always backup before deployment.

22.16.2 Why Secure Deployment is Important?

- Protects from data breaches and attacks.
- Prevents downtime and service disruptions.
- Ensures compliance with security standards.
- Builds user trust and brand reputation.
- Helps in quick recovery if something goes wrong.



22.16.4 Secure Deployment Checklist

- ✓ Code reviewed and tested.
- ✓ All dependencies updated.
- ✓ Secrets stored securely.
- ✓ Environment variables configured.
- ✓ HTTPS enabled with valid SSL certificate.
- ✓ Unnecessary services and ports disabled.
- ✓ Firewall rules configured.
- ✓ Logs and monitoring enabled.
- ✓ Backups configured and tested.
- ✓ Rollback plan ready.



22.16.3 Best Practices for Secure Deployment

Practice	Description
Use HTTPS	Always use HTTPS to encrypt data in transit.
Environment Variables	Store sensitive data (keys, passwords) in environment variables, not in code.
Strong Secrets Management	Use tools like AWS Secrets Manager, HashiCorp Vault, or Azure Key Vault.
Least Privilege Access	Give only the minimum access required to users, services, and processes.
Secure Configuration	Disable unnecessary services and ports. Use secure default configurations.
Keep Dependencies Updated	Regularly update libraries, frameworks, and system packages.
Input Validation	Validate and sanitize all inputs even in production.
Logging & Monitoring	Monitor application logs, errors, and security events.
Backup & Recovery	Take regular backups and test recovery process.
Use Firewalls & WAF	Use firewalls and Web Application Firewalls to block malicious traffic.
Automated CI/CD Security	Include security scans (SAST/DAST) in CI/CD pipeline.

22.16.5 Example: Using Environment Variables (Node.js)

```
// .env file
DB_HOST=localhost
DB_USER=admin
DB_PASS=StrongPass@123
JWT_SECRET=SuperSecretKey123
PORT=5000

// app.js
require('dotenv').config();
const express = require('express');
const app = express();
const PORT = process.env.PORT || 5000;
const DB_HOST = process.env.DB_HOST;
app.get('/', (req, res) => {
  res.send('Application is running...');
});
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

JS

22.16.6 Enable HTTPS (Express with Helmet)

```
const express = require('express');
const https = require('https');
const fs = require('fs');
const helmet = require('helmet');
const app = express();

app.use(helmet()); // Set secure HTTP headers
app.get('/', (req, res) => {
  res.send('Secure deployment with HTTPS!');
});

const options = {
  key: fs.readFileSync('/etc/letsencrypt/live/yourdomain.com/privkey.pem'),
  cert: fs.readFileSync('/etc/letsencrypt/live/yourdomain.com/fullchain.pem')
};
https.createServer(options, app).listen(443, () => {
  console.log('HTTPS server running on port 443');
});
```

JS

22.16.7 Common Deployment Mistakes

Mistake	Impact
Hardcoding secrets in code	Exposure of sensitive information.
Using HTTP instead of HTTPS	Data can be intercepted.
Not updating dependencies	Known vulnerabilities can be exploited.
Over-permissive access	Attackers can misuse excess permissions.
No backups	Data loss in case of failure or attack.
No monitoring or logging	Hard to detect and respond to issues.

22.16.8 Recommended Tools

Category	Tools
Secrets Management	AWS Secrets Manager, HashiCorp Vault, Azure Key Vault
CI/CD	GitHub Actions, GitLab CI/CD, Jenkins
Security Scanning	Snyk, OWASP ZAP, SonarQube
Monitoring	Prometheus, Grafana, New Relic
Web Application Firewall	Cloudflare WAF, AWS WAF, ModSecurity
Container Security	Docker Bench, Trivy, Aqua Security

22.16.9 Summary



- Secure deployment protects your application, data, and users.
- Follow best practices, use strong configurations, and keep everything updated.
- Monitor, backup, and have a rollback plan for safe and reliable deployments.

