# WEB SECURITY

ONE LOVE. ONE FUTURE.

# Learning Outcomes

**By the end of this lesson, students will be able to:**

- Identify and understand common web vulnerabilities (XSS, CSRF, SQL Injection).
- Implement secure authentication and authorization mechanisms.
- Apply JWT correctly for stateless authentication.
- Use HTTPS/TLS to protect data in transit.

# Content

1. Common Vulnerabilities
2. Indentiy and Access Management
3. Token-based Authentication
4. Protocol-level Security

HUST

hust.edu.vn fb.com/dhbkhn

1. Common Vulnerabilities

1.1. Cross-site Scripting (XSS)

1.2. Cross-site Request Forgery (CSRF)

1.3. SQL Injection & NoSQL Injection

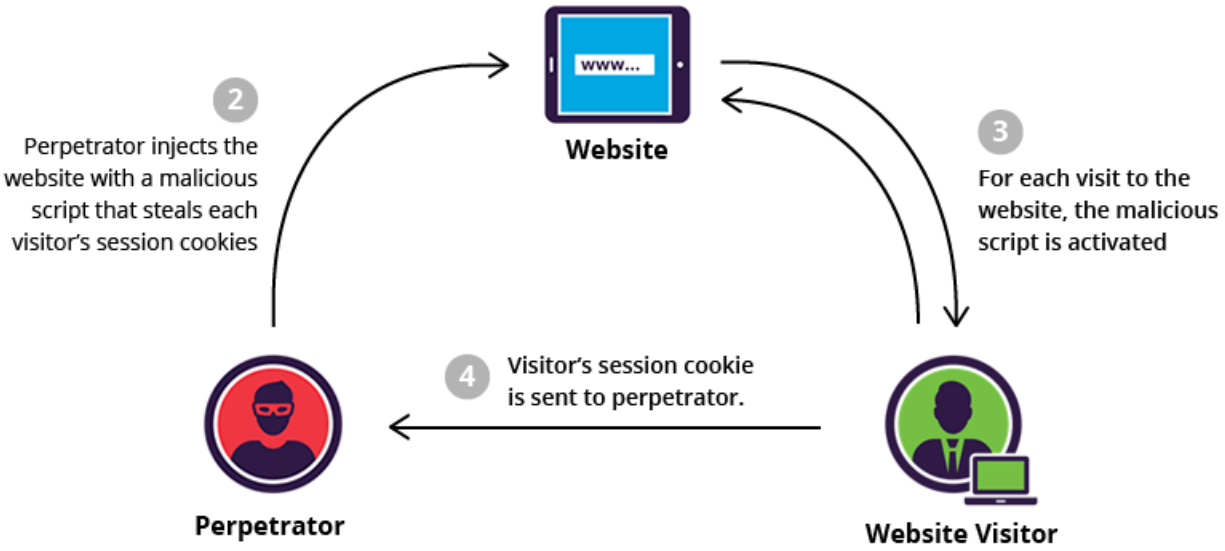
# 1.1. Cross-Site Scripting (XSS)

- A vulnerability that allows attackers to inject and execute malicious JavaScript in a victim's browser.
- Occurs when user-controlled input is rendered into a webpage without proper escaping or validation.


Source: Youtube

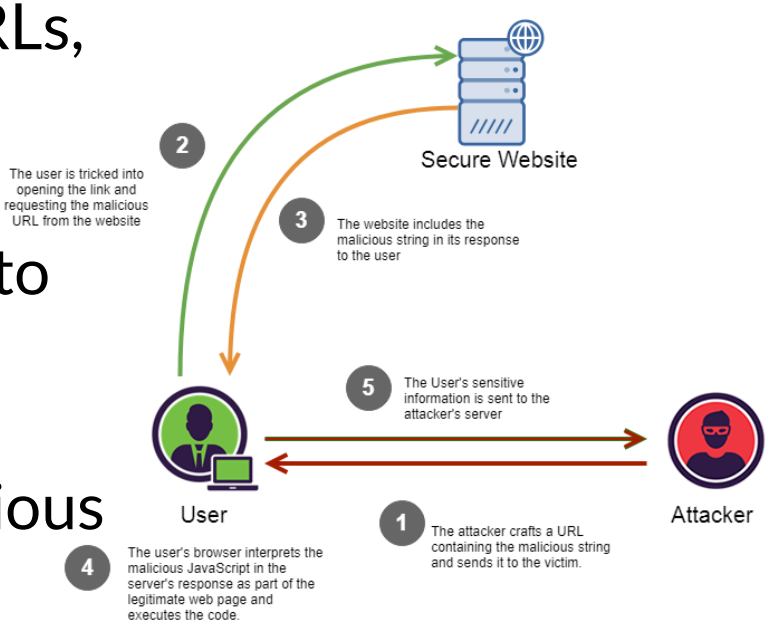
# Stored XSS

- Also known as **Persistent XSS**.
- Malicious script is **permanently stored** on the server
- How Stored XSS works
  - Attacker → submits malicious input (comment, message, username…).
  - Server **stores** this input without sanitization.
  - Users view the page:  the browser executes the injected script.



# Reflected XSS

- Also known as **Non-persistent XSS.**
- Malicious input is immediately **reflected** in the server's response.
- The payload is delivered through URLs, query parameter
- How Reflected XSS works
  - Attacker → sends malicious link to victim
  - Victim → clicks the link
  - Server → reflects back the malicious payload
  - Browser → executes injected script
  - User information → sends to Attacker



# DOM-based XSS

- A type of XSS that occurs entirely on the client side (browser).
- The malicious payload never reaches the server.
- Triggered when client-side JavaScript modifies the DOM using untrusted data
- How DOM-based XSS works
  - Attacker → sends malicious URL
  - Victim → opens the link Browser
  - JS → reads untrusted data from URL
  - Vulnerable script → writes data into DOM (innerHTML)
  - Browser → executes the injected script
  - Data → sends to Attacker

# XSS Demo

- 1. Stored XSS
  - <?= $comment ?>
  - <script>alert('Stored XSS')</script>

- 2. Reflected XSS
  - You searched: <?= $_GET['q'] ?>
  - https://example.com/?q=<script>alert('Reflected')</script>

- 3. DOM-based XSS
  - document.body.innerHTML = location.hash.substring(1);
  - https://example.com/#<img src=x onerror=alert('DOM XSS')>

# Preventing XSS

- Output Escaping (encode before rendering)
  - Convert special characters so they cannot be interpreted as HTML/JS.
  - `<` → `<` `"` → `&quot;`
- Input Sanitization (remove unsafe content)
  - Remove dangerous characters before processing.
  - Library: DOMPurify (client-side), sanitize-html (server-side)
  - `const clean=DOMPurify.sanitize(userInput);`
- Content Security Policy (CSP)
  - Browser-level defense that restricts where scripts can be loaded/executed.
  - `Content-Security-Policy: script-src 'self';`

# 1.2. Cross-site Request Forgery (CSRF)

- Your browser does something on your behalf, but you never intended it
  - Attacker tricks the browser into sending unwanted requests
  - The request includes valid authentication cookies.
- Why does CSRF happen?
  - Browsers automatically include cookies with every request to a website's domain
  - Logined bank.com: browser stores cookie sessionId=abc123
  - Evil website (e.g., evil.com): <img src="https://bank.com/transfer?to=attacker&amount=100">

  => browser sends sessionId=abc123

# CSRF Attack Example

- 1. Vulnerable Bank website

  POST https://bank.com/transfer

  Body: to=attacker&amount=100

  Bank uses **session cookie** for authentication.
- 2. Attacker website

```
<form action=https://bank.com/transfer method="POST">
  <input type="hidden" name="to" value="attacker">
  <input type="hidden" name="amount" value="1000">
</form>
<script>document.forms[0].submit();</script>
```

- 3. Flows
  - Victim is logged into bank.com in another tab
  - Browser automatically attaches bank.com session cookies
  - Form submits without victim interaction

# Preventing CSRF

- **SameSite Cookies (Browser-Level Protection)**
  - Controls whether cookies are sent cross-site
  - `Set-Cookie: sessionId=abc123; SameSite=Strict; Secure; HttpOnly`

| Mode | Cookie Sent Cross-Site | CSRF Protection | Use Cases |
|---|---|---|---|
| **Strict** | 🚫 Never | Strongest | Banking, Admin Panels |
| **Lax** (default) | Only on top-level GET | Good | Most websites |
| **None; Secure** | ✅ Always (HTTPS only) | Weak | OAuth, iframes, APIs |

- **Origin/Referer Validation (Server-Side Protection)**
  - Server checks whether request came from a trusted domain
  - Prevents unauthorized cross-site POST/PUT/DELETE requests
  - ```
    if(req.headers.origin !== "https://bank.com") {
        return res.status(403).end();
    }
    ```
    ```

# Preventing CSRF

- **CSRF Token**
  - A secret value generated by the server and required on every state-changing request.
  - Server verifies: request.token == **server.stored.token** ?
  - Purpose:
    - Attacker cannot read the token → cannot forge a valid request
    - Protects against all CSRF attacks
- **Double-Submit Cookie Pattern**
  - Token is sent **both as a cookie and as a request value**
  - Server sets a cookie, and sends to Client
  - Client-side JS reads the cookie and sends the token in a header/form field
  - Server does NOT need to store the token
  - Server verifies: cookie.token == header/form.token ?

# Comparison

| Technique | Feature | Strengths | Best Use Cases |
|---|---|---|---|
| SameSite Cookies | Browser-level | Strong protection | Web apps don't require cross-site requests |
| Origin/Referer Validation | Server-level, trusted domains | Simple to implement; effective for POST/PUT/DELETE | APIs, banking sites, sensitive operations |
| CSRF Token | Server stores token | Strongest protection | Traditional apps |
| Double-Submit Cookie Pattern | Token is sent twice | Works with SPAs & microservices; compatible with JWT | Single-page apps, cross-domain APIs |

# 1.3. SQL Injection & NoSQL Injection

- SQL Injection (SQLi) is a code injection attack
  - Input is not validated
  - Database executes the attacker's injected SQL

```
const q = "SELECT * FROM users WHERE
username = '" + user + "'";
```

```
Attacker user = ' OR '1'= '1
```

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

=> Always true → login bypass

SQL Injection happens when user input is treated as SQL code instead of data.

# SQL Injection Attack

Attacker can modify an SQL query to return additional results.

```sql
-- with x is value passed
SELECT * FROM products WHERE category = 'x' AND released = 1
-- x=Gifts
SELECT * FROM products WHERE category = 'Gifts' AND released = 1
-- x=Gifts'--
SELECT * FROM products WHERE category = 'Gifts'--' AND released = 1
-- x=Gifts' OR 1=1--
SELECT * FROM products WHERE category = 'Gifts' OR 1=1--' AND released = 1
```

# SQL Injection Attack

Attacker can change a query to interfere with the application's logic.

```
1 -- x and y are values passed
2 SELECT * FROM users WHERE username = 'x' AND password = 'y'
3 -- x=wiener, y=bluecheese
4 SELECT * FROM users WHERE username = 'wiener' AND password = 'bluecheese'
5 -- x=administrator'--
6 SELECT * FROM users WHERE username = 'administrator'--' AND password = ''
```

# UNION-based SQL Injection

- Attacker can retrieve data from different database tables.

```
1 -- x is value passed
2 SELECT name, description FROM products WHERE category = 'x'
3 -- x=Gifts
4 SELECT name, description FROM products WHERE category = 'Gifts'
5 -- x=Gifts' UNION SELECT username, password FROM users--
6 SELECT name, description FROM products WHERE category = 'Gifts'
  UNION SELECT username, password FROM users--
```

# Blind SQL Injection

- Database executes injected SQL, but the application does NOT return visible error messages or query results.
- Attacker must infer information indirectly.
- **Boolean-based Blind SQLi**: uses boolean conditions
  - Server: `SELECT * FROM users WHERE id = '$id';`
  - Hacker:
    - `?id='5' AND 1=1--'` => TRUE
    - `?id='5' AND 1=2--'` => FALSE
    - `?id='5' AND SUBSTRING((SELECT password FROM users WHERE id='5'), 1, 1) = 'a'--'` => guess password's first character

# Blind SQL Injection

- **Time-based Blind SQLi:** uses boolean conditions
  - Server: `SELECT * FROM users WHERE id = '$id';`
  - Hacker:
    - `?id='5' AND IF(1=1, SLEEP(5), 0)--`
      => TRUE, delay ~ 5s
    - `?id='5' AND IF(1=2, SLEEP(5), 0)--`
      => FALSE, response immediately
    - `?id='5' AND IF(SUBSTRING(password,1,1)='a', SLEEP(5), 0)--` => guess password's first character

- SQLi is dangerous: allows extraction of entire databases slowly but reliably

# Preventing SQL Injection

- Use parameterized queries (prepared statements)

```
const query = "SELECT * FROM users WHERE id = ?";
db.execute(query, [req.params.id]);
```

  - Treats user input as **data**, not SQL commands.
  - Prevents attacks like OR 1=1, UNION SELECT, --
- Use an ORM (Object–Relational Mapping)
  - Developers work with a relational database using **objects**
  - `User.findAll({where:{age: {[Op.gt]:18}} });`

  → `SELECT * FROM users WHERE age > 18;`
- Input Validation & Allowlist
  - 
```
if (!/^[a-zA-Z0-9_]+$/.test (req.body.username)) {
 return res.status(400).send("Invalid");
 }
```

# NoSQL Injection (MongoDB Operators)

- Attackers inject MongoDB operators into the query

| Operator | Meaning | Example Attack |
|---|---|---|
| $ne | Not equal | { "password": { "$ne": null } } |
| $gt / $lt | Greater / Less than | { "age": { "$gt": 0 } } |
| $regex | Regex match | { "username": { "$regex": ".*" } } |
| $or | OR condition | { "$or": [ {}, { "admin": true } ] } |

```
const user = await users.findOne({
  username: req.body.username,
  password: req.body.password
});
```

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

Always TRUE

Attacker logs in without password

# Preventing NoSQL Injection

- Validate & sanitize input

```
if (typeof req.body.username !== 'string')
throw Error("Invalid input");
```

- Use middleware to remove dangerous operators

```
const mongoSanitize = require('express-mongo-sanitize');

app.use(mongoSanitize());
```

- Use schema validation (Mongoose)

```
 const UserSchema = new mongoose.Schema({
  username: String,
  password: String
}, { strict: "throw" });
```

HUST

2. Indentiy and Access Management

2.1. Authentication Basics

2.2. Session Security

2.3. Password & Account Security

2.4. Authorization & Access Control

# 2.1. Authentication Basics

- Authentication is the process of verifying a user's identity proving ***who you are*** before accessing a system.
  - Prevents unauthorized users
  - The first step before authorization
- Methods
  - Passwords (most common)
  - Multi-Factor Authentication (MFA: TOTP, SMS, email OTP, hardware key)
  - Biometrics (fingerprint, face ID)
  - Tokens (Session cookies, JWT)
  - OAuth Login (Google, Facebook, Microsoft)

# Session-Based Authentication

- Server-side authentication method
  - User logs in → sends username/password…
  - Server verifies credentials
  - Server creates a session and **sessionID**
  - Server sends a **sessionID** as a cookie to the browser
  - Browser sends **sessionID** cookie with every request
  - Server looks up **sessionID** to identify the user
- Pros
  - Easy to implement
  - Simple to revoke (delete session)
- Cons
  - Server must store **sessionID** state → not stateless
  - Harder to scale horizontally

# Token-Based Authentication

- A stateless authentication method
  - User logs in → sends username/password...
  - Server verifies credentials
  - Server generates a **token** (e.g., JWT)
  - Client stores the **token** (HttpOnly cookie)
  - Client sends **token** in request headers: Authorization: Bearer <token>
  - Server verifies **token** signature → grants access
- Pros
  - Scalable
  - Ideal for mobile apps, SPAs, microservices
- Cons
  - Hard to revoke tokens once issued
  - Requires short expiration + refresh tokens

# Session-Based vs Token-Based Authentication

| Feature | Session-Based Auth | Token-Based Auth (JWT) |
|---|---|---|
| **State** | Stateful (server stores session data) | Stateless (server stores nothing) |
| **Scalability** | Harder to scale (requires shared session store) | Very scalable (no shared session needed) |
| **Storage** | HttpOnly cookie | HttpOnlycookie (or localStorage - not secure) |
| **Revocation** | Easy (delete session on server) | Hard (token valid until expiration unless blacklist used) |
| **Best for** | Classic web apps, server-rendered pages | SPAs, mobile apps, microservices |

Session auth = secure, simple, but requires server memory
Token auth = scalable, flexible, but requires token protection

# 2.2. Session Security - Session Fixation

**Session Fixation**

- Attacker obtains a valid sessionID (e.g., via login).
- Attacker sends a link to victim with this sessionID https://example.com/login?sessionId=ABC123
- Victim logs in with that same sessionID.
- Attacker has full access.
- Prevention:
  - Always regenerate sessionID after login
  - Reject sessionIDs from URLs
  - Set SameSite to limit cross-site usage

# 2.2. Session Security - Session Hijacking

## Session Hijacking

- Attacker steals an existing sessionID and impersonates the user.
  - XSS (read cookies)
  - Network sniffing (without HTTPS)
  - Malware / browser extensions
  - Predictable sessionIDs
- Prevention
  - Use HTTPS everywhere
  - Use HttpOnly cookies (block JS access)
  - Set Secure + SameSite

# Session Timeout, Regeneration, Re Authentication

- Session Timeout
  - Sessions should expire automatically after a period of inactivity.
  - Reduces risk if a device is lost, shared, or left unattended.
- Session ID Regeneration
  - Regenerate the sessionID on:
    - Successful login (to prevent Session Fixation)
    - Privilege escalation (e.g., user becomes admin)
  - Old session ID must be invalidated immediately.
- Re-Authentication
  - Users must re-enter credentials (or MFA) before performing high-risk actions, such as:
    - Changing password or email
    - Making financial transactions
  - Ensures that the user is still the legitimate owner.

# Cookie Security + MFA

- Cookie Security Flags
  - HttpOnly → JavaScript cannot read cookie → protects against XSS
  - Secure → Cookie sent only via HTTPS → prevents sniffing
  - SameSite → Controls cross-site cookie sending → protects against CSRF
    - Strict = strongest
    - Lax = default
    - None;  Secure = needed for cross-site (OAuth)
- MFA (Multi-Factor Authentication)
  - Requires 2+ factors
  - Strong additional protection even if password is stolen
- Key Point → Use secure cookies + MFA to protect accounts and sessions.

# 2.3. Password & Account Security

- Passwords must never be stored in plaintext
- Use bcrypt or Argon2 to transform password into an irreversible value.
- Example
  - `const bcrypt = require('bcrypt');`
  - `const hash = await bcrypt.hash("mypassword123", 10);`
  - `console.log(hash);`
  - `$2b$10$E9qe3VdSahz6Lfl4ovRpBOsR0xg2DzPq4g95EZqXbkcNoFg0t4leW`
  - `$2b$ → version`
  - `10 → cost factor`
  - `E9qe3VdSahz6Lfl4ovRpBO → salt(random string, generated by bcrypt)`
  - `remaining part → hashed password`

# 2.3. Authorization and Access Control

- Authorization determines what a user is allowed to do after they have been authenticated.
  - Answers question: “What actions can this user perform?”
  - Enforced on the server side
  - Controls access to routes, resources, APIs, and operations

- RBAC (Role-Based Access Control)
  - Assigns permissions based on roles, not individual users.
  - Roles → Permissions
    - User → read own data
    - Editor → read/write content
    - Admin → full access
    - SuperAdmin → system-level permissions

# IDOR (Insecure Direct Object Reference)

- An application exposes internal identifiers (e.g., user IDs, order IDs) without checking permission.
- Attacker changes an ID in the URL or request to access someone else's data.
- Example
  - Vulnerable endpoint: GET /api/users/123/profile
  - Attacker: GET /api/users/124/profile
    - Unauthorized data access
    - Viewing other users' orders, invoices, profiles
    - Account takeover in severe cases

# IDOR (Insecure Direct Object Reference)

- Vulnerable Code

```
app.get("/api/users/:id", (req, res) => {
  const user = db.getUser(req.params.id);
  res.json(user); // No ownership check!
});
```

- Secure Code: requestedUserId === loggedInUserId

```
app.get("/api/users/:id", authMiddleware, (req, res) => {
  const user = db.getUser(req.params.id);

  if (user.id !== req.user.id) {
    return res.status(403).json({ error: "Forbidden" });
  }

  res.json(user);
});
```

HUST

# 3. Token-based Authentication

## 3.1. JWT Fundamentals

## 3.2. JWT Flow & Verification

## 3.3. JWT Security

# 3.1. JWT Fundamentals

- JWT (JSON Web Token): a compact, digitally signed token used to securely transmit information
- Key characteristics
  - Stateless → server does not store session data
  - Self-contained → carries user info inside the token
  - Signed, not encrypted → payload is visible, but cannot be modified
  - Easy to send via headers or cookies

# JWT Structure

A JWT consists of 3 parts: Header.Payload.Sigature

- Header → algorithm + token type

  {"alg":"HS256","typ":"JWT"}

- Payload → claims (user id, role, exp…)

  {"user":"alice","role":"admin"}

- Signature → verifies token integrity

  data=base64url(header) + "." + base64url(payload)

  S = HMACSHA256(data, secret_key);

Server checks: HMACSHA256(data, secret_key) === S ?

# JWT Payload

- JWT payload is only Base64URL encoded
- Anyone who gets the token can decode it.
- JWT does NOT hide data — it only protects integrity.
  - Never put sensitive information inside a JWT payload.
  - e.g: private user info, confidential fields

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "iat": 1516239022
}
```

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX
VCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiw
ibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4
iOnRydWUsImlhdCI6MTUxNjIzOTAyMn0
.KMUFsIDTnFmyG3nMiGM6H9FNFUROf3w
h7SmqJp-QV30
```

https://www.jwt.io/

# JWT Login Flow

- User sends login request
- Server verifies credentials
- Server generates tokens
  - creates access token (short-lived)
  - creates refresh token (long-lived, HttpOnly cookie)
- Client stores tokens
  - access token → memory or localStorage
  - refresh token → HttpOnly + Secure cookie
- Client sends request with access token authorization
- Server validates token signature
  - if valid → grants access
  - if expired → client uses refresh token to get a new one

# Access Token vs Refresh Token

| Feature | Access Token | Refresh Token |
|---|---|---|
| Lifetime | Short | Long |
| Sent with each request | Yes | No |
| Stored where | Memory / Cookie | HttpOnly Cookie |
| Contains user claims | Yes | Usually minimal |
| Used for | API access | Renewing access tokens |
| Risk if stolen | Medium (expires soon) | High (long validity) |

# Refresh Token Rotation

- A security mechanism where every time the client uses a refresh token, the server:
  - Issues a new refresh token, and
  - Invalidates the previous one.

=> prevents attackers from reusing a stolen refresh token.

- Flow
  - Client sends:  refresh_token: R1
  - Server validates R1 → returns:
    - New Access Token (A2)
    - New Refresh Token (R2)
  - Server stores R2 and marks R1 as used (invalid)

# Token Storage: localStorage vs HttpOnly Cookies

localStorage

- Pros
  - Persistent across page reloads
  - No automatic sending → no CSRF risk
- Cons
  - Vulnerable to XSS (JavaScript can read token)
  - Not recommended for Refresh Tokens
- Use for: *Access Tokens (short-lived)*, (recommend in *memory)*

HttpOnly Cookies

- Pros
  - Protected from XSS (JS cannot read cookie)
  - Secure + SameSite can block cookie attacks
- Cons
  - Can be vulnerable to CSRF if not configured correctly
  - Must use HTTPS (Secure flag)
- Use for: *Refresh Tokens (long-lived)*

HUST

hust.edu.vn fb.com/dhbkhn

# 4. Protocol-level Security

## 4.1. HTTPS & TLS

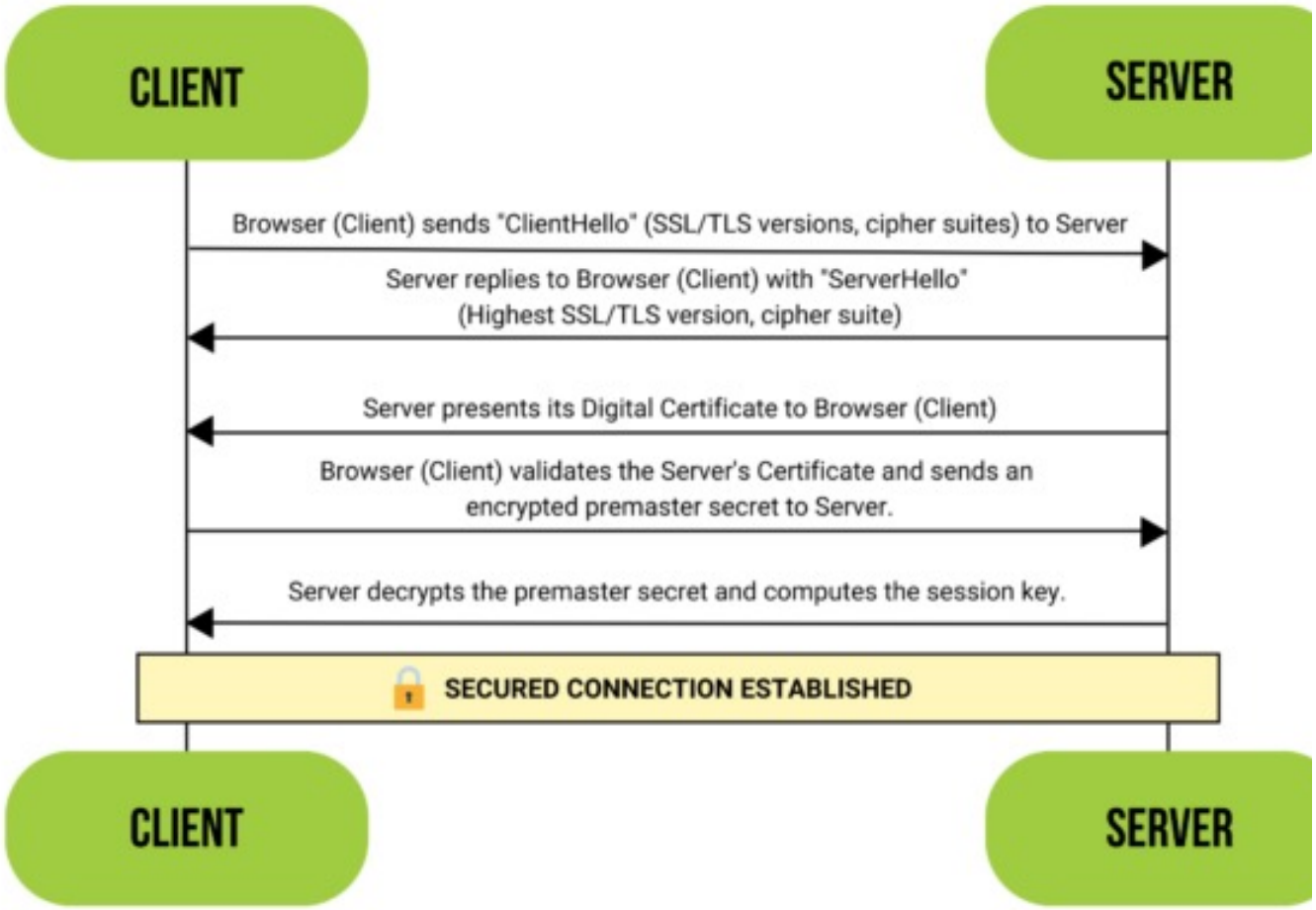
## 4.2. SOP & CORS

## 4.3. API Security

# 4.1. HTTP & TLS

- TLS (Transport Layer Security)
  - Cryptographic protocol secures communication over the Internet
  - Successor to SSL (SSL 3.0 → TLS 1.0 → … → TLS 1.3)
- What TLS Provides
  - Encryption: prevents attackers from reading data
  - Integrity: ensures data is not modified during transmission
  - Authentication: verifies the server's identity
- TLS  used in
  - HTTPS (web browsing)
  - API calls (REST, GraphQL)
  - Email (SMTP over TLS)
  - Mobile apps communicating with servers

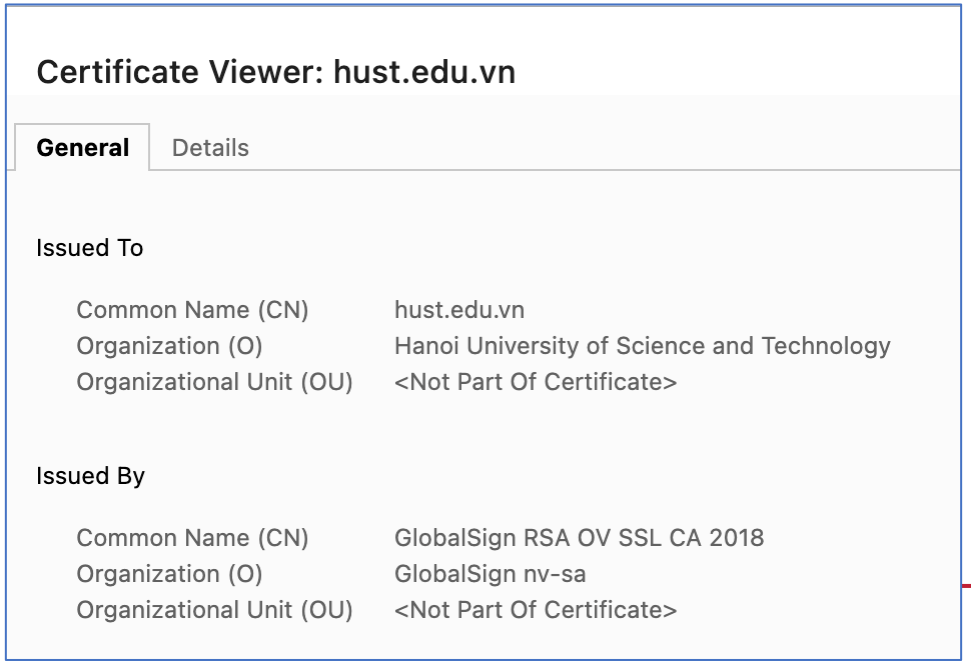
# TLS Handshake

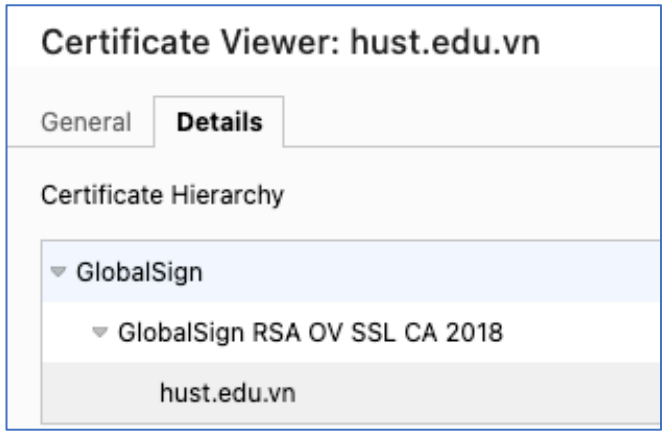



SSL.com

# Certificate

- A digital certificate proves the server's identity.
  - Issued to a domain (e.g., example.com).
  - Contains: Public key, Issuer information, Validity period, Signature by a Certificate Authority (CA)
- Certificate chain
  - Hierarchical sequence of certificates
  - RootCA -> Intermediate CA -> Server Certificate



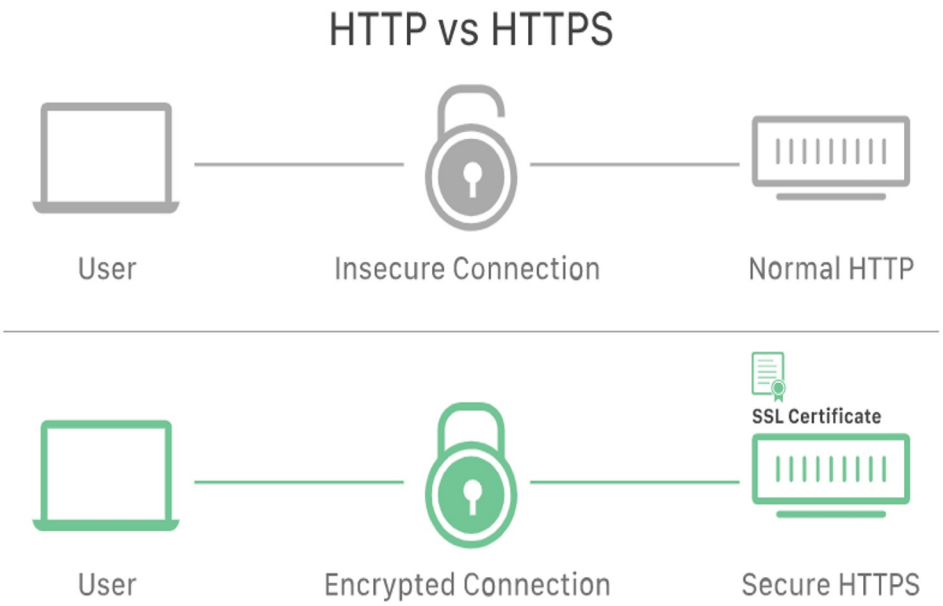



# HTTPS

- HTTPS = HTTP + SSL/TLS encryption.
- HTTPS use TLS to encrypt normal HTTP requests and response, making it safer and more secure.
- A website that uses HTTPS has https:// in the beginning of its URL instead of http://
- Almost browsers as Chrome, Firefox,... mark all HTTP websites as “Not secure”


# HTTPS

- When attacker try to intercept a request:




👉HTTP:

```
POST /login  HTTP/1.1
User-Agent: curl/7.63.0
libcurl/7.63.0 OpenSSL/1.1.l
zlib/1.2.11
Host: www.example.com
Accept-Language: en
{
    “username”: “admin”,
    “password”: “1234”
}
```

👉HTTPS:

```
t8Fw6T8UV81pQfyhDkhebbz7+oiwldr1j2gHBB3L3
RFTRsQCpaSnSBZ78Vme+DpDVJPvZdZUZHpzbbc
qmSW1+3xXGsERHg9YDmpYk0VVDiRvw1H5miNi
eJeJ/FNUjgH0BmVRWII6+T4MnDwmCMZUI/orxP
3HGwYCSIvyzS3MpmmSe4iaWKCOHQ==
```

# 4.2. SOP & CORS

## Same-Origin Policy (SOP): browser's security feature

- Prevents malicious sites from reading or manipulating data from another site

Origin

https://www.example.com:443

scheme host name port

| Origin A | Origin B | Explanation of whether Origin A and B are "same-origin" or "cross-origin" |
|---|---|---|
| https://www.example.com:443 | https://**www.evil.com**:443 | cross-origin: different domains |
| | https://**example.com**:443 | cross-origin: different subdomains |
| | https://**login**.example.com:443 | cross-origin: different subdomains |
| | **http**://www.example.com:443 | cross-origin: different schemes |
| | https://www.example.com:**80** | cross-origin: different ports |
| | **https://www.example.com:443** | **same-origin: exact match** |
| | **https://www.example.com** | **same-origin: implicit port number (443) matches** |

# Prevent info leaks

- Generally, embedding a cross-origin resource is permitted, while reading a cross-origin resource is blocked.

| | |
|---|---|
| iframes | Cross-origin embedding is usually permitted (depending on the `X-Frame-Options` directive), but cross-origin reading (such as using JavaScript to access a document in an iframe) isn't. |
| CSS | Cross-origin CSS can be embedded using a `<link>` element or an `@import` in a CSS file. The correct `Content-Type` header may be required. |
| forms | Cross-origin URLs can be used as the `action` attribute value of form elements. A web application can write form data to a cross-origin destination. |
| images | Embedding cross-origin images is permitted. However, reading cross-origin image data (such as retrieving binary data from a cross-origin image using JavaScript) is blocked. |
| multimedia | Cross-origin video and audio can be embedded using `<video>` and `<audio>` elements. |
| script | Cross-origin scripts can be embedded; however, access to certain APIs (such as cross-origin fetch requests) might be blocked. |

# Prevent info leaks

❑Example 1: A webpage on the **web.dev** domain includes this iframe:

```
<iframe id="iframe" src="https://example.com/some-page.html" alt="Sample iframe">
</iframe>
```

The webpage's JavaScript includes this code to get the text content from an element in the embedded page. Is this JavaScript allowed?

```
const iframe = document.getElementById('iframe');
const message = iframe.contentDocument.getElementById('message').innerText;
```

?No. Since the iframe is not on the same origin as the host webpage, the browser doesn't allow reading of the embedded page.

# Prevent info leaks

❑Example 2: A webpage on the **web.dev** domain includes this form. Can this form be submitted?

```
1 <form action="https://example.com/results.json">
2   <label for="email">Enter your email: </label>
3   <input type="email" name="email" id="email" required>
4   <button type="submit">Subscribe</button>
5 </form>
```

?Yes. Form data can be written to a cross-origin URL specified in the action attribute of the <form> element.

# Prevent info leaks

❑Example 3: A webpage on the **web.dev** domain includes this iframe. Is this iframe embed allowed?

```
1 <iframe id="iframe" src="https://example.com/some-page.html" alt="Sample iframe">
  </iframe>
```

?Usually. Cross-origin iframe embeds are allowed as long as the origin owner hasn't set the X-Frame-Options HTTP header to deny or sameorigin.

# Prevent info leaks

Example 4: A webpage on the **web.dev** domain includes this canvas:

```
1 <canvas id="bargraph"></canvas>
```

The webpage's JavaScript includes this code to draw an image on the canvas. Can this image be drawn on the canvas?

```
1 const context = document.getElementById('bargraph').getContext('2d')
2 const img = new Image()
3 img.onload = function () {
4   context.drawImage(img, 0, 0)
5 }
6 img.src = 'https://example.com/graph-axes.svg'
7
```

?Yes. Although the image is on a different origin, loading it as an img source does not require CORS. However, accessing the binary of the image using JavaScript such as getImageData, toBlob or toDataURL requires an explicit permission by CORS.

# CORS

- CORS (Cross-Origin Resource Sharing) is a browser security mechanis to enable cross-origin requests
- It is a controlled way to relax the Same-Origin Policy (SOP).
- Flow
  - Browser sends request with Origin header.
  - Server responds with Access-Control-Allow-Origin headers
  - Browser allows or blocks the response based on these headers.

# Preflight Request (OPTIONS)

- Browser asks the server whether a cross-origin request is allowed before sending the actual request.
- Browser sends (OPTIONS)

  OPTIONS /api/update

  Origin: https://frontend.com

  Access-Control-Request-Method: PUT

  Access-Control-Request-Headers: Authorization
- Server must reply

  Access-Control-Allow-Origin: https://frontend.com

  Access-Control-Allow-Methods: PUT

  Access-Control-Allow-Headers: Authorization

  Access-Control-Max-Age: 600

# 4.3. API Security - Rate Limiting

- Why Rate Limiting?
  - Prevent brute-force attacks
  - Protect APIs from abuse
  - Mitigate DDoS bursts
- **Fixed Window Algorithm**
  - Count requests within a fixed time window
  - Reset counter when the window ends
  - Simple, fast to implement
- **Token Bucket Algorithm**
  - A bucket has a max capacity (N tokens)
  - Tokens refill at a steady rate
  - Each request consumes 1 token
  - If bucket is empty → request denied or delayed
  - Smoother + fairer than Fixed Window

# API Authentication - OAuth2

- OAuth2 lets applications access data on behalf of the user using access tokens, not passwords.
- Flow
  - User logs in at Authorization Server e.g, Google
  - Server returns Authorization Code

    ```
    https://yourapp.com/callback?code=ABC123
    ```
  - App exchanges the code for an Access Token

    ```
    App: POST https://oauth2.server.com/token
         grant_type=authorization_code
         code=ABC123
         client_id=xxx
          client_secret=yyy
     Authorization Server: sends Access Token
    ```
  - App uses the Access Token to call APIs

# Exercise

- https://portswigger.net/web-security/all-labs