Securing the web architecture means securing:

- The network
- The operating system
- The web server (Apache for instance)
- The administration server (SSH for instance)
- The database (Oracle for instance)
- The web application ← Our focus

Insufficient Transport Layer Protection a.k.a the need for HTTPs:

- While hackers can try to brute force a user's password/session ID, a more common and better method is to steal the user's password or session ID. Brute forcing usually doesn't work.
- Hackers can eavesdrop and/or tamper with the messages sent back and forth between your browser and the server. This is known as **man in the middle attack (MitM) attack**.
- MitM attacks consist of sitting between the connection of two parties and either observing or manipulating traffic. This could be through interfering with legitimate networks or creating fake networks that the attacker controls. Compromised traffic is then stripped of any encryption in order to steal, change or reroute that traffic to the attacker's destination of choice (such as a phishing log-in site). Because attackers may be silently observing or re-encrypting intercepted traffic to its intended source once recorded or edited, it can be a difficult attack to spot.
- A generic solution we can use is to use HTTPS over HTTP.
- Because HTTP was originally designed as a clear text protocol, it is vulnerable to man in the middle attacks. By including SSL/TLS encryption, HTTPS prevents data sent over the internet from being intercepted and read by a third party. Through public-key cryptography and the SSL/TLS handshake, an encrypted communication session can be securely set up between two parties via the creation of a shared secret key.
- Hypertext transfer protocol secure (HTTPS) is the secure version of HTTP, which is the primary protocol used to send data between a web browser and a website. HTTPS is encrypted in order to increase security of data transfer. This is particularly important when users transmit sensitive data, such as by logging into a bank account, email service, or health insurance provider.
- HTTPS = HTTP + TLS
- HTTPS uses an encryption protocol to encrypt communications. This protocol is called Transport Layer Security (TLS), although formerly it was known as Secure Sockets Layer (SSL). TLS secures communications by using an asymmetric public key infrastructure. This type of security system uses two different keys to encrypt communications between two parties:
    1. The private key: This key is controlled by the owner of a website and it's kept, as the reader may have speculated, private. This key lives on a web server and is used to decrypt information encrypted by the public key.
    2. The public key: This key is available to everyone who wants to interact with the server in a way that's secure. Information that's encrypted by the public key can only be decrypted by the private key.
- HTTPS includes robust authentication via the SSL/TLS protocol. A website's SSL/TLS certificate includes a public key that a web browser can use to confirm that documents sent by the server have been digitally signed by someone in possession of the corresponding private key. If the server's certificate has been signed by a publicly trusted **certificate authority (CA)**, the browser will accept that any identifying information included in the certificate has been validated by a trusted third party.
- **Note:** Self-signed certificates are not trusted by your browser.
- Your browser trusts many CAs by default.

- Transport Layer Security provides:
    1. confidentiality: end-to-end secure channel
    2. integrity: authentication handshake
- HTTPS prevents websites from having their information broadcast in a way that's easily viewed by anyone snooping on the network. When information is sent over regular HTTP, the information is broken into packets of data that can be easily "sniffed" using free software. This makes communication over an unsecure medium, such as public Wi-Fi, highly vulnerable to interception. In fact, all communications that occur over HTTP occur in plain text, making them highly accessible to anyone with the correct tools, and vulnerable to on-path attacks. With HTTPS, traffic is encrypted such that even if the packets are sniffed or otherwise intercepted, they will come across as nonsensical characters.
- HTTPS protects any data send back and forth including:
    - login and password
    - session ID
- HTTPS must be used during the entire session. This is because of **mixed-content attacks**. **Mixed-content** happens when an HTTPS page contains elements such as ajax, js, image, video, css, etc that is served with HTTP and an HTTPS page transfers control to another HTTP page within the same domain. Then, the authentication cookie will be sent over HTTP.
- In addition, we can create cookies with the secure flag. A cookie with the Secure attribute is sent to the server only with an encrypted request over the HTTPS protocol, and therefore can't easily be accessed by a man-in-the-middle attacker. Insecure sites, with http: in the URL, can't set cookies with the Secure attribute. However, do not assume that Secure prevents all access to sensitive information in cookies.
  I.e. The Secure attribute makes it so that the cookie will be sent over HTTPS exclusively and will prevent authentication cookies from leaking in case of mixed-content.
- Do/Don't with HTTPS:
    - Always use HTTPS exclusively in production.
    - Always have a valid and signed certificate (no self-signed cert).
    - Always avoid using absolute URL (mixed-content).
    - Always use a secure cookie flag with an authentication cookie.
- **Note:** HTTPS protects against man in the middle attacks but can't protect against hackers who are in your browser or are on the server.
- Other types of vulnerabilities:

Frontend Vulnerabilities	**Backend Vulnerabilities**
Content Spoofing	Incomplete mediation
Cross-Site Scripting	Information leakage
Cross-site Request forgery	SQL injection

<u>Incomplete Mediation:</u>

- Occurs when the application accepts bad/invalid/malicious data from the frontend and that data causes issues.
  I.e. It occurs when failure to perform "sanity checks" on data can lead to random or carefully planned flaws.
- Data coming from the frontend cannot be trusted.
- Sensitive operations must be done on the backend.

Information Leakage:

- **Information leakage** happens whenever a system that is designed to be closed to an eavesdropper reveals some information to unauthorized parties nonetheless.
  I.e. Information leakage occurs when secret information correlates with, or can be correlated with, observable information.
- In its most common form, information leakage is the result of one or more of the following conditions:
  1. A failure to scrub out HTML/script comments containing sensitive information.
  2. Improper application or server configurations.
  3. Differences in page responses for valid vs. invalid data.
- Sensitive information may be present within HTML comments, error messages, source code, or left in plain sight, and there are many ways a website can be coaxed into revealing this type of information. While information leakage doesn't necessarily represent a security breach, it gives an attacker useful guidance for future exploitation.
- A solution to information leakage is to use authentication (I.e. Who are the authorized users?) and to use authorization (I.e. Who can access what and how?).

SQL Injection:

- **SQL injection** is a type of an injection attack that makes it possible to execute malicious SQL statements. These statements control a database server behind a web application. Attackers can use SQL injection vulnerabilities to bypass application security measures. They can go around authentication and authorization of a web page or web application and retrieve the content of the entire SQL/NoSQL database. They can also use SQL injection to add, modify, and delete records in the database.
- An SQL injection usually occurs when you ask a user for input and the user gives you an SQL statement that you will unknowingly run on your database.

Content Spoofing:

- **Content spoofing** allows the end user of the vulnerable web application to spoof or modify the actual content on the web page. The user might use the security loopholes in the website to inject the content that he/she wishes to the target website. When an application does not properly handle user supplied data, an attacker can supply content to a web application, typically via a parameter value, that is reflected back to the user.
- An attacker can inject HTML tags in the page. They will add illegitimate content to the webpage (ads most of the time).
- A generic solution is to validate data inserted in the DOM.

Cross-Site Scripting (XSS):

- **Cross-Site Scripting (XSS) attacks** target scripts embedded in a page that are executed on the client-side rather than on the server-side. Cross-site scripting is one of the most common application-layer web attacks. XSS in itself is a threat that is brought about by the internet security weaknesses of client-side scripting languages, such as HTML and JavaScript. The concept of XSS is to manipulate client-side scripts of a web application to execute in the manner desired by the malicious user. Such a manipulation can embed a script in a page that can be executed every time the page is loaded, or whenever an associated event is performed.
- Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.
- An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute

the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page.

- An attacker can inject arbitrary javascript code in the page that will be executed by the browser. Then, the attacker can:
    - Inject illegitimate content in the page (same as content spoofing)
    - Perform illegitimate HTTP requests through Ajax (same as a CSRF attack)
    - Steal Session ID from the cookie
    - Steal user's login/password by modifying the page to forge a perfect scam
- Some variations on XSS attacks are:
    1. **Reflected XSS:** Malicious data sent to the backend are immediately sent back to the frontend to be inserted into the DOM.
    2. **Stored XSS:** Malicious data sent to the backend are stored in the database and later-on sent back to the frontend to be inserted into the DOM.
    3. **DOM-based attack:** Malicious data are manipulated in the frontend (javascript) and inserted into the DOM.
- A generic solution is to validate data inserted in the DOM.
- Another solution is to use the HttpOnly cookie flag. This makes it so that the cookie is not readable/writable from the frontend. This prevents the authentication cookie from being leaked when an XSS attack occurs.
  Note: The name is a little misleading. HttpOnly has nothing to do with HTTP or HTTPS.

Cross-Site Request Forgery (CSRF):

- The **same origin policy** is a critical security mechanism that restricts how a document or script loaded from one origin can interact with a resource from another origin. It helps isolate potentially malicious documents, reducing possible attack vectors. Two URLs have the same **origin** if the protocol, port, and host are the same for both.
  This means https://api.mydomain.com and https://mydomain.com are actually different origins and thus impacted by same-origin policy. In a similar way, http://localhost:9000 and http://localhost:8080 are also different origins.
- **Note:** The path or query parameters are ignored when considering the origin.
- **Note:** Internet Explorer has an exception to the definition of origin. IE treats all ports the same way. This is non-standard and no other browser behaves this way.
- Elements under control of the same-origin policy include:
    - Ajax requests
    - Form actions
- Elements not under control of the same-origin policy include:
    - Javascript scripts
    - CSS
    - Images, video, sound
    - Plugins
- **Cross-Site Request Forgery (CSRF)** is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. CSRFs are typically conducted using malicious social engineering, such as an email or link that tricks the victim into sending a forged request to a server. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application.

- Consider this example:
  Suppose you log into https://bank.com and a cookie is stored. While logged in, suppose you unknowingly browse a malicious website. Without the same origin policy, the malicious website could make authenticated malicious AJAX calls to https://bank.com/api to POST /withdraw even though the hacker website doesn't have direct access to the bank's cookies.

  This is because many websites use cookies to keep track of authentication or session info. Those cookies are bound to a certain domain when they are created. On every HTTP call to that domain, the browser will attach the cookies that were created for that domain. Furthermore,the browser automatically attaches any cookies bound to https://bank.com for any HTTP calls to that domain, including AJAX calls from the malicious website. By restricting HTTP calls to only ones to the same origin, the same-origin policy closes some hacker backdoors such as around CSRF attacks.
- **Cross-origin resource sharing (CORS)** is a security mechanism that allows a web page from one origin to access a resource with a different domain. CORS is a relaxation of the same-origin policy implemented in modern browsers. Without features like CORS, websites are restricted to accessing resources from the same origin through what is known as same-origin policy.
- There are legitimate reasons for a website to make cross-origin HTTP requests.
  A single-page app at https://mydomain.com could need to make AJAX calls to https://api.mydomain.com.
  Furthermore, some websites, such as Reddit, allow users to embed images from other websites, like Imgur.
- There are 2 types of CORS requests:
  1. **Preflighted Requests:**
  - For Ajax and HTTP request methods that can modify data, the specification mandates that browsers preflight the request, solicit supported methods from the server with an HTTP OPTIONS request method, and then, upon approval from the server, send the actual request with the actual HTTP request method.
    I.e. When performing certain types of cross-domain Ajax requests, modern browsers that support CORS will initiate an extra "preflight" request to determine whether they have permission to perform the action. Cross-origin requests are preflighted this way because they may have implications to user data.
  - A **preflighted request** is a CORS request where the browser is required to send a preflight request (a preliminary check) before sending the request being preflighted to ask the server permission if the original CORS request can proceed.
  - E.g.
    This is the preflight request:
    OPTIONS /
    Host: service.example.com
    Origin: http://www.example.com
    Access-Control-Request-Method: PUT

    If service.example.com is willing to accept the action, it may respond with the following headers:

    Access-Control-Allow-Origin: http://www.example.com
    Access-Control-Allow-Methods: PUT, DELETE

Then, the browser will then make the actual request. If service.example.com does not accept cross-site requests from this origin then it will respond with error to the OPTIONS request and the browser will not make the actual request.

2. **Simple Requests:**
    - A **simple request** is a CORS request that doesn't require a preflight request before being initiated.

- **JSON with Padding (JSONP)** is another way to circumvent same-origin policy. JSONP is a historical JavaScript technique for requesting data by loading a <script> element. Because JSONP is vulnerable to CSRF attacks, it is very dangerous to use and has been replaced with CORS.
- We can protect legitimate requests with a CSRF token.
- We can also prevent CSRF attacks using the SameSite cookie flag. If you're using the SameSite flag, the cookie will not be sent over cross-site requests. This prevents forwarding the authentication cookie over cross origin requests.