## Security concepts and principles

### Security goals

* **Confidentiality**: keep something secret, data in communication or at rest (cryptography, authentication…)
* **Integrity:** no corruption or control hijacking, who can write (message, data hashing)
* **Availability:** system uptime, response time, free storage…
* **Privacy:** right to be left alone, personal information
* **Accountability:** login and audit trails (secure timestamping, integrity in logs…)
* **Non-repudiation:** two parties can’t deny they have interacted (trust a 3rd party and generate evidence)

### Security guiding principles

* Secure the weakest link (attacker needs only one flaw)
* Practice defense in depth (use layers of defense)
* Fail securely
* Compartmentalize (separate code into parts)
* Be reluctant to trust
* Principle of least privilege (minimum access and time necessary)
* Keep it simple (to reduce attack surface, tradeoff with usability)
* Promote privacy
* Hiding secrets is hard (hiding details is not enough, attacker can have ways of finding them)
* Use community resources

There are also some attacks described here, come back to check after and see if they were covered.

## OWASP Top 10

## Information Gathering

## An attacker wants to find an easy way of attacking, and as a developer you want to decide test scope, coverage and prioritize

What to gather?

* Application structure - identify pages, external links, and trust zones (areas with varying access permissions)
* Data flow within the application (observe how data moves between client and server, focusing on GET/POST requests, responses, and parameters)
* Infrastructure or platform (web server, applications, entry points, execution path, framework…)

Tools: web debugging proxy (example: Burp Suite): this can be used to capture and examine requests and responses, manipulate payloads, attacks… Automated scanners can’t capture everything, so manual testing is recommended.

## Injection Attacks

#### SQL injection

An attacker manipulates user input to inject malicious SQL into a query. This allows unauthorized access to data, data corruption, privilege escalation, or even destruction of the database. Also possible in XML with Xpath injection.

* + Blind SQL injection: guess database schema with binary search after checking if its vulnerable to sql injection.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

* + Countermeasures
    - Blacklisting: filter quotes, semicolons, whitespace.. It can have problems with functional requirements and it is easy to bypass
    - Whitelisting: only allow well defined safe inputs. The problem is that RegExp (regular expressions) are hard to define for all safe values
    - Escape input: treating everything as strings and not as logic, like blacklisting, it could miss dangerous characters
    - Prepared statements & bind variables: decouple query statement and data input with a template, most robust.
      * ORM: Texto

        El contenido generado por IA puede ser incorrecto.
      * Manual: 
    - Mitigate impact
      * Avoid information leakage (don’t display errors and stack traces)
      * Limiting privileges
      * Encrypt sensitive data
      * Key management precautions (don’t store encryption key in DB)
      * Hash password

#### Session Management Attacks

HTTP is stateless, so unless you reauthenticate for every request , you can’t know for sure if the requests are from the same clients. With session management, you can assign session tokens and then validate them for every request. You can store session tokens by:

* Embedding in url: https://site.com/checkout?sessionToken= 1234. Exposes the token everywhere, basically)
* In hidden form field: <input type= “hidden” name = “sessionToken” value = “1234”> Must be included in every form manually, relies on client-side data.
* In browser cookie: setcookie: sessionToken = 1234 (these are set in headers of HTTP requests and responses). Safer, but server only sees cookie, not domain who sent the cookie.

##### Attacks

* **Session token theft**
  + Methods
    - **Network sniffing:** an attacker can see someone’s cookie with the session ID when they make requests to sites with HTTP (not HTTPS) and use it (since the server doesn’t check the domain from where it is being sent)

|  |
| --- |
| Find in settings.py  SECURE\_SSL\_REDIRECT = True (to force HTTPS and cookie settings)  SESSION\_COOKIE\_SECURE = True  SESSION\_COOKIE\_HTTPONLY = True  CSRF\_COOKIE\_SECURE = True  'django.middleware.security.SecurityMiddleware' (in MIDDLEWARE) |

* + - **Logout problem:** logging out should destroy the token in both client and server. Usually browsers delete the cookie (session expired or logout), but the server keep the session valid, so attackers who stole the token can still use it.

|  |
| --- |
| Interfaz de usuario gráfica, Texto  El contenido generado por IA puede ser incorrecto.This is wrong, in Django use  logout(request) + short session lifetimes in session.py |

* + - Cross-site scripting\*: attacker crafts malicious URL with script inside that returns the session cookie from the client’s browser (where the script is executed when client clicks into the malicious URL)

|  |
| --- |
| Django has built-in auto-escaping for XSS, so you should be fine. It could be unsecure if it has the “| safe” in html, which should only be used if 100% the data has been sanitized previously |

* + Solutions
    - Once user logged in (token around), communication through HTTPS
    - Remember to log out
    - Time-out session ID and delete expired session ID
    - Bind session token to client’s IP or computer
* **Session token predication attack**: some tokens can be predicted, seeing one or more tokens shouldn’t be able to predict others. Solution: use token generators from frameworks (in Django - check code for manually set session keys. Correct: request.session['key'] = value.
* **Session fixation attack:** server sends anonymous token to browser (not logged in yet), which will elevate privileges when the client logs in. If attacker overwrites the token before logging in, he will have an elevated token after logging in.
  + Methods
    - Tampering through network: client visits server through HTTP, attacker can intercept and alter the HTTP traffic, and he injects into response an overwritten cookie (session token).
    - Cross-site scripting (XSS): attacker crafts malicious URL with script inside that establishes the anonymous session cookie before logging in, and then when the client clicks it and logs in (elevates the session), the attacker has a logged in session.
  + Solution/Mitigation: always issue a new session token, when elevating from anonymous token to logged-in token.

|  |
| --- |
| Does request.session.flush(). Correct.  Look for manually setting session variables without regenerating sessionId |

\*Cross-site scriptiong (XSS) – attacker can insert javascript scripts that will be run in the victim’s browser