# 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** \*: 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 (wrong) |

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

- Reflected XSS: javascript injected into request and reflected immediately in response

- Stored XSS: script injected into request, stored somewhere in server and reflected repeatedly

Mitigation: sanitize input data, escape (transform “<” into “&lt;” so it is displayed as text and not executed)

## Broken access control

#### Cross site request forgery (CSRF)

Browser has the sessionID cookie stored for bank.com, and when user accesses evil.com, it can make browser execute a request to bank.com (like a money transfer), and it will be sent with the right cookie. Browser automatically sends the cookies on requests, and can’t be sure if the user intentionally made the request or not.

* Identify vulnerability: create HTML page separate to the testing web, include something like <img src="http://mysite.com/account/del" width="0" height="0">, log into account in the web and with the session active open the HTML page and check if account has been deleted.

|  |
| --- |
| 1.Never change data in GET requests, only POST + csrf\_token  2.Check for csrf middleware |

* Mitigation
  + Extra authentication
  + DSRF tokens
  + SameSite cookies: browser setting that prevents cross-site cookie usage (not let evil.com access cookies for bank.com)
  + Referer-based validation: verifies the request originates from own domain
  + Validation via CSRF token: add action token as hidden field in the body of the request to genuine forms, token that will not be sent by the browser accidentally and the attacker can’t access through evil.com

#### Server Side Request Forgery (SSRF) Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico El contenido generado por IA puede ser incorrecto.

Attacker writes URL with and embedded URL in it. The second URL would usually not be accessible for the attacker, but from the server it is possible (inside the trust zone).

* Countermeasures: no universal fix, highly depends on requirements
  + Whitelist and DNS resolution
  + Response handling
  + Disable unused URL schemas
  + Authentication on internal services
  + Network segregation

## Security misconfiguration

#### XML External Entities (XXE)

<!ENTITY name SYSTEM "URI"> is a way of defining an external resource to be reusable in the document, but can be vulnerable to attackers reading local files, internal services, doing SSRF or DoS with inputs like this:  
Logotipo

El contenido generado por IA puede ser incorrecto. Imagen que contiene Polígono

El contenido generado por IA puede ser incorrecto.

* Countermeasures:
  + Disable external entity resolution
  + Use safe XML parsers – defusedxml: fromstring(request.body).

|  |
| --- |
| Check for views that do uploads or similar (request.FILES[“upload”]) and parse it as XML without filtering or using defusedxml. |

## Software and data integrity failure

#### Insecure deserialization

Serialization: turning an object into a data format that can be sent over the network (so python class, for example, to JSON or XML)

Deserialization: converting serialized string back into an object. The problem comes when the server blindly deserializes input from the user and then trusts the resulting object. Attacker modifies the serialized data and then the deserialized object will be executed, leading to SQL injection, Remote Code Execution, Authentication bypass, Data tampering, Privilege escalation.

* Countermeasures:
  + Don’t deserialize untrusted input
  + Use data-only formats like JSON
  + Avoid pickle, yaml.load() on input (python libraries to serialize and deserialize data)
  + Validate input manually after deserialization
  + Use digital signatures (to check if content has been tampered with)
  + Run in low-privilege environments

|  |
| --- |
| Search for picke and yaml loads or raw use of deserialized data in code |

## Identification and authentication failure

#### Broken authentication

Authentication: process of verifying who you are by

* Something you know: passwords & security questions. Easy to implement and understand, but easy to crack and forget
* Something you have: bankID device, phone (SMS). Hard to crack, but can be broken, stolen and forged (cloned) and the strength only depends on how difficult it is to forge.
* Something you are: biometrics. Hard to crack and to steal, but not always accurate, privacy issues and hard to replace

Crack a password

* Vulnerable password storage: basic but vulnerable, all users compromised if hacker obtains password file.
  + Countermeasure: Hashing – compare hash of password when sent
* Dictionary attack: use words from dictionary and compute the hashes (online or offline)
  + Countermeasure:
    - Salting – include additional info in hash, hash password concatenated with salt (a random number) & store salt in password file. Good for online dictionary attack, but ineffective against online.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.\*Hashing salt does not make sense, because then you can’t reverse the salt and then can’t verify the password properly

* + - Password Pepper: like salt but all passwords have the same pepper value, which is stored in an encrypted form in another secure place. Defends better against dictionary attacks and also against offline attacks better.

Other password security techniques

* Filtering: guarantee strong password by setting a minimum length, requiring mixed characters and measuring the strength
* Limiting logins: allow 3-4 logins and lock account, inconvenient to forgetful user.
* Aging password: require to change passwords every once in a while (users will do workarounds and become more insecure)
* Last login / Protective monitoring: notify users of suspicious login, educate users to pay attention and report possible attacks
* One-time password: send one-time password through SMS
* Two-factor/two-channel authentication: combine different ways of authentication
* Password recovery with URL tokens, PINs, Offline methods and security questions

CAPTCHA and reCAPTCHA: Completely Automated Public Turing Test to Tell Computers and Humans Apart. Commonly used to block bots

## Security logging and monitoring failures

#### Insufficient logging and monitoring

* Not logged auditable events like logins and transactions
* Unclear/inadecuate/no log messages from warnings and errors
* Logs not monitored for suspicious activity
* Logs only stored locally
* Not effective alerting thresholds
* Unable to detect, escalate or alert for attacks in real time

## Insecure design

Clickjacking: attacker can place a layer with a URL that he wants the user to click that seems transparent by setting HTML iframe and opacity.

* Countermeasures
  + X-Frame-Options: deny – completely disables the loading of the page in a frame
  + X-Frame-Options: sameorigin only embed from the same server
  + X-Frame-Options: allow-from – embed only from whitelist

+ OWASP web testing guide

# Cryptography

Secure communication: establish a shared secret key (through face-to-face, handshake algorithms…) and then transmit the data using the shared key. Cryptography is the basis for many security mechanisms, but not the solution to all security problems.

Kerckhoff’s principle: the encryption algorithm is open, the only secret is the key, that should be chosen at random and kept secret

#### Step 1. Establish shared secret key

* Public key encryption: system where you don’t use the same key to encrypt and to decrypt. To encrypt, you use the public key of the receptor, and then only the receptor can decrypt it with his secret key. Mathematically speaking, this happens with a trapdoor one-way function, that is easy to go forward, but not to invert. Y = F(PK,x) easy; F^-1 (SK,y) = x only possible with SK
* Digital signature: bind document to author, in digital world it will depend on the content of the document.  
   Interfaz de usuario gráfica, Texto, Aplicación

  El contenido generado por IA puede ser incorrecto. Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

  El contenido generado por IA puede ser incorrecto.
* Certificate Authority: third party that ensures that when Bob sends his public key to Alice, it is a verified and real key
* SSL/TLS handshake

Interfaz de usuario gráfica, Aplicación, Word

El contenido generado por IA puede ser incorrecto.TLS 1.2 is insecure, use TLS 1.3  
  
RSA – prime factorization, standardized, simple, effective, widely used

ECDSA – limited support (y^2 = x^3 +ax + b), higher complexity, faster, shorter keys and limited support. Uses points on elliptic curves

#### Step 2. Transmit data using a shared secret key

* Confidentiality
  + Substitution cipher: letter matrix, each letter defined by letter row and letter column

Imagen que contiene Escala de tiempo

El contenido generado por IA puede ser incorrecto.

* + Shift cipher: associate letter with number, choose one and shift every letter by that position with wraparound. Only 26 possible keys, insecure.

Gráfico de cajas y bigotes

El contenido generado por IA puede ser incorrecto. Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

* + The Vigenere cipher: key is a string, shift each character by the amaunt dictated by the character of the key. Much more keyspace 26^n (n length of key), but also insecure, since you can crack it this way: Guess the length of the key (it is correct if the letters obtained from the same character of the key follow the normal frequency of letters in English) and then guess each character of the key by substituting the most used characters in the text with the most used characters in english.

(encrypt HOLA with key LUNA : (H)7+(L)11 = 18 = S; 14(O)+20 (U) = 34 mod 26 = 8 = I …  
  
Texto

El contenido generado por IA puede ser incorrecto.

* + Interfaz de usuario gráfica, Texto

    El contenido generado por IA puede ser incorrecto.One time pad: use random key of the same size of the plaintext used only once, with the same logic as Vigenère. Perfectly secure if used correctly, but key must be as long as plaintext and be used once. It has no integrity protection, if something is change in the cipher, it’s changed in the plain too.  
    Basically, with the XOR, the attacker can know the differences between the two original texts, and by applying some mechanisms on the similar parts they can identify both messages

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

* + Stram cipher: use short key as seed to generate a pseudo random key as long as the message to make OTP encryption
  + Block cipher: cut text into blocks and encrypt each using short keys. In DES (Data Encryption Standard) you have each block 64 bits, k=56 bits and 16 rounds, and its reversible round function. Applies a 48-bit key to the rightmost 32 bits to produce a 32-bit output. Then, rightmost 32 bits are swapped with leftmost 32 bits
    - ECB (Electronic code book) is wrong, it can disclose plaintext information (same blocks represented same way)
    - CBC (Cipher Block Chaining) with random initialization vector, each block XORs with the previous before ciphering.
* Integrity
  + Interfaz de usuario gráfica, Texto, Aplicación

    El contenido generado por IA puede ser incorrecto.MAC (Message Authentication Code): with message and shared secret key a MAC function can be applied to generate a tag sent with the message. The receiver then can recalculate the MAC with the same key and compare the result (if it is the same, its integrity hasn’t been modified)
  + HMAC - uses hash functions like SHA-256 or SHA-3
  + Interfaz de usuario gráfica

    El contenido generado por IA puede ser incorrecto.ECBC – Enhanced CBC mode that adds an integrity verification system.
* Combination
  + MAC then Encrypt (TLS)
  + Encrypt and MAC (SSH)
  + Encrypt then MAC (IPsec)