CCN-CERT

BP/28

Recommendations on Secure Development

GOOD PRACTICE REPORT

JANUARY **2023**

ccn-cert centro criptológico nacional
CCN centro criptológico nacional

Edited by:

MINISTERIO DE DEFENSA
SUBSECRETARÍA DE DEFENSA
SECRETARÍA GENERAL TÉCNICA

CCN centro criptológico nacional

Paseo de la Castellana 109, 28046 Madrid

Date of issue: January de 2023

LIMITATION OF LIABILITY

This document is provided in accordance with the terms contained herein, expressly rejecting any type of implicit guarantee that may be related to it. Under no circumstances can the National Cryptologic Centre be held responsible for direct, indirect, fortuitous or extraordinary damage derived from the use of the information and software indicated, even when warned of such a possibility.

LEGAL NOTICE

General State Administration Publications Catalogue
https://cpage.mpr.gob.es

Index

Index

1. Introduction

Objetives

This document is intended to help development teams understand the most common security controls that should be applied during the software development lifecycle. The secure development guides provide developers with a set of recommendations to follow that enable them to build applications with security techniques.

Establishing security measures during application development in the design and coding phases not only lays the foundation for security against the most common vulnerabilities, but also reduces future costs, as fixing problems at a later stage is more costly.

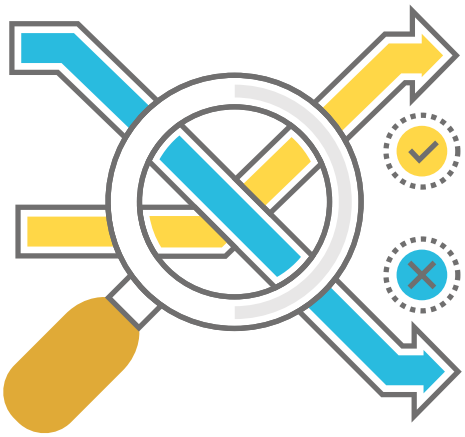
The secure development guides provide developers with a set of recommendations to follow that enable them to build applications with security techniques

Scope

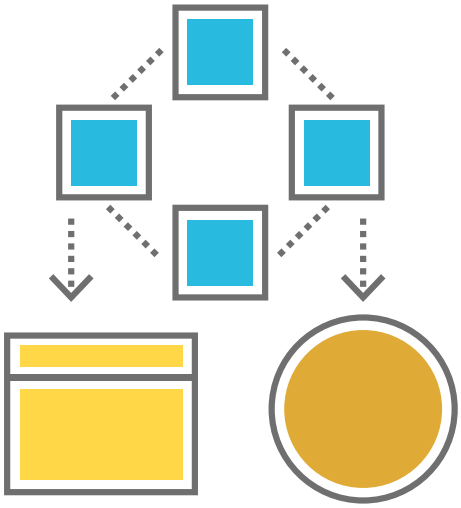
Regardless of the development methodology used, the definition of security controls should start at the design stage or even before, and continue throughout the application lifecycle to ensure that the application will continue to have the relevant security measures in case of changes to the initial arrangement due to business needs.

Building secure software involves activities on many levels, not only to comply with internal security policies, but also to follow the law and external regulations such as HIPAA, PCI or GDPR. The resulting software must implement security features that meet these requirements.

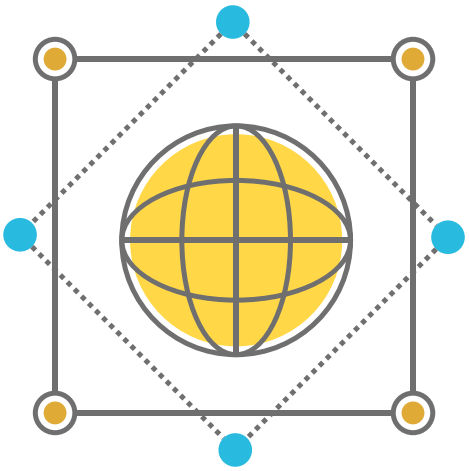
In addition, during the threat modelling process of a project, when properly combined with security requirements engineering and secure design principles, it allows the development team to identify the security features that are necessary to ensure the integrity, confidentiality and availability of the data involved in the project.



The guide has been classified as follows:



- **Security recommendations for the following aspects:**
  - Architecture
  - Authorisation
  - Authentication
  - Session management
  - Validation of input and output parameters
  - Error handling
  - Secure registration
  - Cryptography
  - Secure file management
  - Transaction security
  - Communications security
  - Data protection
- **Supplementary indications for Python**
- **Security Controls Checklist**
- **Security Vulnerabilities and Controls**
- **ENS Security Measures and Security Controls**
- **Annexes at the end of the document with Basic and Advanced Security Controls**



2. Architecture safe

A solid secure architecture is fundamental to the construction of the software as it is the foundation on which the software is built and developed.

To achieve this purpose, it is necessary to identify the components used, ensure that there are no known vulnerabilities and that they are properly updated.

The architecture must always be designed with security requirements in mind in order to avoid or limit potential security threats.

2.1. Potential risks

Potential threats to an application can be multiple and of high probability when:

- **Components with known vulnerabilities are used.**
- **Outdated, obsolete or unsupported components are used.**
- **Unidentified components are used.**
- **Expendable ports are kept open.**

2.2. Security recommendations

- **Identify all the components of the architecture.**
  - Components that have not been correctly identified are potential security risks.
- **Conduct a review of the basing of each hardware device from a security point of view:**
  - Review of configurations. Ensure that the configurations are the most secure: no debugging options enabled, no default users and passwords, etc.
  - Check ports. Make sure that only those communication ports that are strictly necessary are open.
  - Status of the latest system update.
  - Identification of all the system components: Libraries, Modules, *Frameworks*, Services, etc.
  - For each system component, perform the same baseline review of configurations and update status.
- **From the result of the previous review, obtain a report of the components in which there are vulnerabilities detected for which there is currently no security patch and analyse their level of risk within the application. It could be that certain vulnerabilities detected do not pose a real risk to the application or do not have a relevant impact.**
- **For those vulnerabilities that pose a real risk, keep a close watch on vulnerable components so that they are updated as soon as possible.**
- **Conduct a study of how the security problems created by these risk vulnerabilities could be avoided or mitigated by alternative security systems.**
- **Improve logical perimeter security by installing firewalls, IDS or similar devices, or by segmenting the network.**

Ensure that the configurations are the most secure: no debugging options enabled, no default users and passwords, etc

- **Ensure that data is protected by authorisation mechanisms between environments through physical or logical segregation and by backups to ensure availability.**
- **Use the most recent version of the programming language.**
- **Use a Virtual Environment as a project workspace if applicable according to the programming language.**
- **Correct import of packages according to programming language. Installed and imported packages, thoroughly checking the security of the packages to be installed.**
- **Disable all debugging options in Production that prevent information leakage in the detailed error messages.**
- **Use tools in the IDE that perform basic semantic and security analysis.**

2.3. References

Design Patterns:
https://refactoring.guru/es/design-patterns [1]

OWASP. Design Secure Web Applications:
https://owasp.org/www-pdf-archive/APAC13_Ashish_Rao.pdf [2]

Areas of Homeland Security: Critical Infrastructure Protection:
file:///Users/lagor/Downloads/BOE-400_Ambitos_de_la_Seguridad_Nacional_Proteccion_de_Infraestructuras_Criticas.pdf [3]

3. Authentication

Authentication is the verification of the identity of a user or device in order to grant access to its resources or information. This task usually requires the presentation of credentials, such as a username and password, to verify that the user is indeed who he/she claims to be.

It is the main area of security control, so the type and method of authentication must be part of the design.

Authentication is the verification of the identity of a user or device in order to grant access to its resources or information

3.1. Types of authentication

- **Network authentication:** the identity of a user or device is verified through the use of network credentials, such as a username and password, or through the verification of a digital certificate.
- **Password-based authentication:** the user provides a username and password to access a system or resource.
- ***Token*-based authentication:** the user receives a physical or digital *token* that must be provided in order to access a system or resource.
- **Two-factor authentication:** the user is required to provide two (2) forms of verification of their identity, such as a password and a code sent to their mobile phone.
- **Biometric authentication:** physical characteristics of the user, such as their fingerprint or voice, are used to verify their identity.

3.2. Authentication methods

3.2.1. Basic authentication

Basic authentication is a **network authentication** method. In this authentication method, the user provides a username and password in clear text form (base64) and is therefore considered an insecure form of authentication. Other methods, such as two-factor authentication or authentication based on digital certificates, are recommended.

RISKS

- **MitM (Man-in-the-Middle) attack:** authentication information is sent in clear text form, which means that it could be easily intercepted and read by anyone with access to the decrypted information on the network.
- **Key reuse:** If the same password is used for multiple sites or systems, an attacker who obtains the password through basic authentication can use it to access other protected resources.
- **Brute-force or dictionary attack:** This type of authentication does not provide adequate protection against these attacks, where an attacker attempts to guess passwords through the use of automated programs.
- **Weak authentication:** does not allow a user to prove his identity in a secure and reliable way, preventing the implementation of stronger security measures, such as two-factor authentication or authentication based on digital certificates.

3.2.2. Authentication via forms

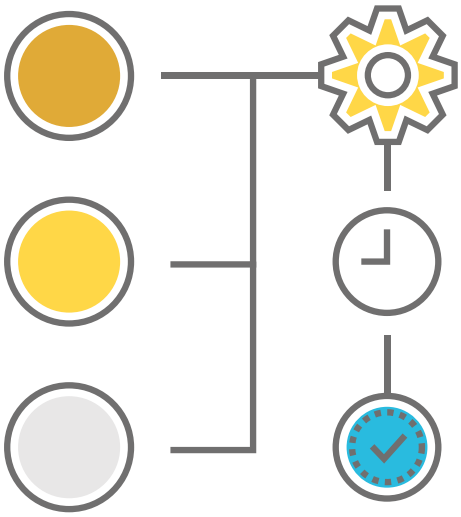
Forms-based authentication is a type of **password-based authentication** for a website. In this authentication method, the user provides his or her username and password through a form on a web page. The web server verifies the authentication information received and if it is correct, it will allow the user access.

Forms authentication can be a secure form of authentication if appropriate measures are used to protect the authentication information,

such as encryption of information in transit and the use of strong passwords. However, it also presents some risks, such as the possibility of an attacker obtaining the authentication information through social engineering techniques or through the use of brute force software.

RISKS

- **MitM (Man-in-the-Middle) attack:** authentication information could be intercepted and used to access the protected resource if appropriate techniques are not used to protect the information such as encryption of communications or the use of strong passwords.
- **Key reuse:** If the same password is used for multiple sites or systems, an attacker who obtains the password through forms authentication can use it to access other protected resources.
- **Brute-force or dictionary attack:** also, does not provide adequate protection against these attacks, where an attacker attempts to guess passwords using automated programs.
- **Weak authentication:** does not allow a user to prove his identity in a secure and reliable way, preventing the implementation of stronger security measures, such as two-factor authentication or authentication based on digital certificates.



3.2.3. Implicit authentication

The user does not have to explicitly provide credentials to access a protected resource. Instead, other forms of identification are used, such as the user's IP address, information stored in a *cookie* or a *token*. It is a type of **network** or ***token*-based authentication** depending on the way chosen to provide this identification.

The most secure way to use this method is to use a cryptographic *hash* based on it as a password to obtain a *token* that is sufficiently strong and long to make it useless to use a brute force attack against it. The server compares the *token* received with the *token* it has calculated and stored for the user requiring access.

RISKS

- **Unauthorised access:** If information such as the user's IP address is used to determine whether the user has access to a protected resource, an attacker can spoof this information and gain unauthorised access to the resource.
- **Unauthorised access:** If *cookies* are used to store authentication information, an attacker can gain access to the *cookie* and use it to access the protected resource.
- **Brute force or dictionary attack:** where sufficiently strong encryption algorithms, such as MD5, have not been used, it could be vulnerable to such attacks.
- **Weak authentication:** does not allow a user to prove his identity in a secure and reliable way, preventing the implementation of stronger security measures, such as two-factor authentication or authentication based on digital certificates.

3.2.4. Autenticación de cliente HTTP

A type of network-based authentication in which a cryptographic key pair, a public key and a private key, is used to verify the identity of a user or device. The public key is used to encrypt the information and the private key is used to decrypt it. The user sends a digital certificate containing his public key for the server to encrypt the content and then uses his private key to decrypt it.

Authentication by public key certificate is considered one of the most secure forms of authentication, as it allows a user to prove his identity in a reliable and secure way. This is provided that the encryption algorithm is sufficiently strong for this authentication method to be truly secure. Typically, this method is used for HTTPS (HTTP over SSL/TLS) communications. The only negative to this method is that it can be more complicated and costly to implement than other forms of authentication.

The client's public key certificate is issued by a trusted entity, such as a Certification Authority (CA), which also provides an identification for the bearer.

Authentication by public key certificate is considered one of the most secure forms of authentication, as it allows a user to prove his identity in a reliable and secure way

RISKS

- **Unauthorised access:** If an attacker manages to obtain a user's private key, he can use it to gain unauthorised access to that user's protected resources.

  If a fake certificate is issued with a fake public key, it could be used to gain unauthorised access to protected resources.
- **MitM attack:** by intercepting a certificate in transit, it could be used to access protected resources in an unauthorised manner.
- **Tampering with an SSL certificate:** the implementation of certificates that have not been verified by a trusted CA or self-signed certificates that could provide a false sense of security

3.2.5. Windows Authentication

A type of **password-based authentication** in which the identity of a user attempting to access a computer or resource protected by MS Windows is verified. It is done by using a username and password, which are verified against information stored on an authentication server or in a local directory. MS Windows uses two (2) authentication protocols Kerberos and NTLM.

User authentication is performed by using a centralised authentication server

Kerberos uses a centralised authentication server and encrypted keys to securely verify the identity of users and NTLM uses the computer's local file system.

In a Windows NT domain or Active Directory environment, user authentication is performed by using a centralised authentication server. When a user attempts to log in, his or her computer sends an access request to the authentication server and verifies the user's identity using the credentials stored in the Active Directory. If the credentials are valid, the authentication server issues an access ticket allowing the user access.

This authentication method is most suitable in corporate environments (Intranet) where centralised control over access to network resources is required. In addition, by using an Active Directory, it is possible to centrally manage user accounts and their access permissions to different network resources.

There are quite a few security risks with this authentication method, many of them dependent on the security policies that have been set by the MS Windows server administrator.

RISKS

- **Dictionary attacks:** these are based on guessing passwords by using programs that attempt common combinations of letters and numbers.
- **Brute-force attacks:** attempt to guess a user's password by using programs that generate and test combinations of characters until the correct one is found.
- **MitM attacks:** Intervene the communication between the user and the authentication server in order to obtain the user's credentials.
- **Replay attacks:** capture and reuse valid *login* tickets issued by the authentication server, which would allow an attacker to gain access without having to know the user's password. MS Windows operating systems include security measures that prevent the reuse of access tickets.
- **Phishing attacks:** these are based on sending fake emails that appear to come from a trusted source with messages that use social engineering techniques in order to obtain a user's credentials.
- **Software vulnerabilities:** MS Windows operating systems and other software used for authentication contain vulnerabilities that can be identified and exploited by attackers.
- **Human security failures:** mistakes or oversights by users, such as using weak passwords, sharing their credentials or allowing them to be discovered.

3.2.6. Passport Authentication

Microsoft Passport is an authentication service used by some Windows systems and Microsoft applications to verify the identity of users. This service uses a Microsoft account, such as an Outlook account or a Skype account, to verify the user's identity.

It is based on the use of one-time credentials, which are sent to the authentication server instead of the user's password. This prevents attackers from guessing the user's password by using brute force or dictionary techniques. In addition, it uses encryption to protect against the risk of MitM attacks.

Starting with MS Windows 10, the Passport system has evolved using two-factor authentication. One is device registration and the other is biometric authentication using a gesture (Windows Hello) or a PIN.

Microsoft Passport is an authentication service used by some Windows systems and Microsoft applications to verify the identity of users

RISKS

- **Dependence on a Microsoft account:** To use Microsoft Passport, you must have a Microsoft account.
- **Software vulnerabilities:** Microsoft Passport may have vulnerabilities that could be exploited by attackers.
- **Phishing and human error risks:** the same problems as indicated for MS Windows authentication.

3.3. Potential risks

In summary, the most common threats and risks due to lack of security authentication controls in applications or insufficiently secure authentication methods are:

- **Brute-force attacks:** using programmes that generate and test combinations of characters to guess users' passwords and gain access to their accounts. If passwords are weak or easy to guess, these attacks could be successful.

- **Dictionary attacks:** using programs that attempt to guess passwords by using lists of words most commonly used as passwords using variations with letters and numbers. If the passwords are weak or typical, these attacks may succeed in less time.

- **MitM attacks:** if communications are intercepted, credentials, *token* or access ticket could be obtained (replay attack). If, in addition, communications are insecure or clear text forms of authentication are used, attacks could be more successful.

- **Replay attacks:** once an attacker has obtained an access ticket, it could be reused to gain access.

- **Software vulnerabilities:** operating systems and applications used in authentication may have vulnerabilities that can be exploited by attackers.

- **User enumeration:** the attacker could find registered users on the system by proving identities and analysing server responses, either by response time or error messages.

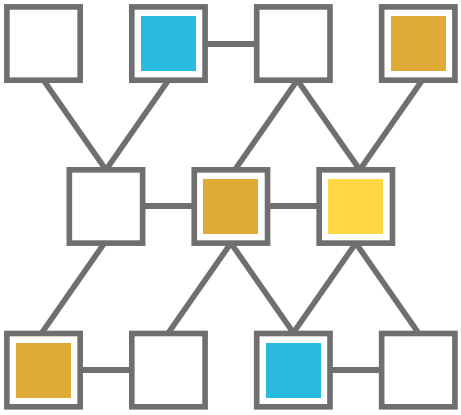
- **Spoofing:** obtaining credentials by impersonating the authentication server, making the victim believe that they are in the right place to enter their credentials to gain access. The service could even forward the information to the real server and redirect it authenticated so that the victim does not notice the deception.

- **Authentication *bypass*:** includes techniques that allow authentication by bypassing the associated security measures. For example, access to a user account by code injection.

- **Identity theft:** this is when the attacker obtains the victim's identity and uses it to perform some actions. This can be done, for example, by stealing a session *cookie* from the victim.

3.4. Security recommendations

- Ensure that passwords are not stored in a readable format, so that if the system or resource containing the passwords is compromised, the malicious user is still unable to use them. A good method could be the use of functions that guarantee the irreversibility of the operation, such as the use of strong *hash* functions.
- Ensure that each page of the application has a logout link, that the session expires when the user logs out, and that the session expires after a reasonable period of non-activity.
- Never expose credentials in the URL.
- When using forms, use POST methods for sending information between the client and the server.
- Using multi-factor authentication to increase security for operations with access to sensitive resources or publicly accessible applications.
- Use two-factor authentication to the user for critical functions in the application, such as changing passwords or accessing particularly sensitive resources.
- Implement account lockout after three (3) failed *login* attempts and a way to contact the administrator to unlock the account.
- Implement CAPTCHA to mitigate brute force attacks on applications exposed on the internet.
- Avoid enumeration of users, providing generic error messages in case of authentication failure, such as "The user and/or password provided are not correct".



- Avoid enumeration of users by providing different response times in case of non-existent user or incorrect password. It is required to include random timeouts in both cases so that it is impossible to recognise the cases by measuring response times.

- Avoid enumeration of users in password recovery procedures that require entering the username.

- Disable the "auto-complete" attribute of the password field in the application, as it is enabled by default.

- Enforce password complexity requirements set by policy or regulation, and ensure that these requirements follow minimum security rules such as:
  - It should be a minimum of eight (8) characters and a sensible maximum.
  - It must contain at least three (3) of the following characters:
    - A capital letter.
    - A lower case letter.
    - A number.
    - A special character.

- Not to persistently store the authentication *cookie* on the customer's computer, and not to use it for other purposes such as personalisation.

- Ensure that the session is different each time a user has successfully logged in to the application.

- Record all successful and unsuccessful authentication attempts without exposing the key used.

- Recording malicious access attempts in a specific security log: Detection of multiple failed authentication attempts, detection of injection attempts, such as SQL or LDAP, detection of multiple users for the same IPs, etc.

Enforce password complexity requirements set by policy or regulation, and ensure that these requirements follow minimum security rules

3.5. Example

This example creates a *hash* and *salt* for a password by calling **getSaltedHash(String password)**. This method will return a string containing the *salt* and *hash* separated by a | (pipe).

To check if a given password is correct, call **check(String password, String stored)**, passing the password being checked along with the stored *hash*/*salt*. This method will return true if the password is correct.

It is important to use **SecureRandom** to generate the *salt* **in** a secure, random and unique way for each password. A sufficiently high number of iterations is used for the *hash* calculation (10000 in this example) to increase security and make it more difficult for brute force attackers to calculate the *hash* of a given password. It is also important to use a secure *hash* function such as SHA-512, which is resistant to collision attacks and is relatively fast to compute.

```
import java.security.MessageDigest;
import java.security.SecureRandom;
import java.util.Arrays;
import java.util.Base64;

public class SecureAuth {

   private static final int ITERATION_COUNT = 10000;
   private static final int KEY_LENGTH = 512;

   public static String getSaltedHash(String password) throws IllegalStateException {
      byte[] salt = SecureRandom.getInstanceStrong().generateSeed(KEY_LENGTH / 8);
      return Base64.getEncoder().encodeToString(salt) + "|" + hash(password, salt);
   }

   public static boolean check(String password, String stored) throws IllegalStateException{
      String[] saltAndPass = stored.split("\\|");
      if (saltAndPass.length != 2) {
         throw new IllegalStateException("Use '<salt>|<hash>'");
      }
      byte[] salt = Base64.getDecoder().decode(saltAndPass[0]);
      String hashOfInput = hash(password, salt);
      return hashOfInput.equals(saltAndPass[1]);
   }

   private static String hash(String password, byte[] salt) throws IllegalStateException {
      MessageDigest md = MessageDigest.getInstance("SHA-512");
      md.update(salt);
      byte[] hashedPassword = md.digest(password.getBytes("UTF-8"));
      int iterations = ITERATION_COUNT;
      while (iterations-- > 0) {
         hashedPassword = md.update(hashedPassword);
      }
      hashedPassword = md.digest(hashedPassword);
      return Base64.getEncoder().encodeToString(hashedPassword);
   }
}
```

3.6. References

Java Secure Authentication:
https://docs.oracle.com/javaee/5/tutorial/doc/bncbe.html#bncbn [4]

Python: Library to implement OTP:
https://pypi.org/project/pyotp/ [5]

4. Authorisation

Authorisation is responsible for determining what actions a user or system is allowed to perform, therefore, authentication is a prerequisite for authorisation, as it is necessary to verify the identity of a user or system before determining what actions they are allowed to perform.

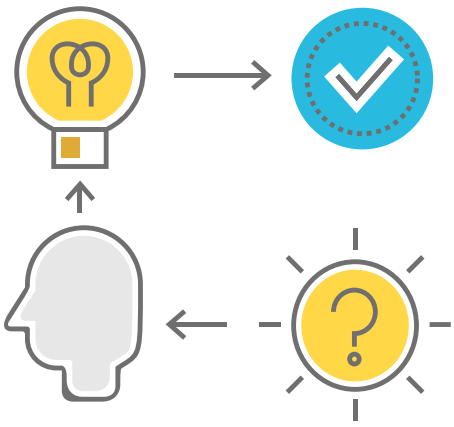
It is the second security control after authentication and is closely linked to authorisation over the business functions of the application.

4.1. Potential risks

There are several vulnerabilities that can arise in the absence of adequate security controls when implementing authorisation. The following are some of the main and common risks identified in applications due to lack of security controls on authorisation.

- **Unauthorised access:** if authorisation is not properly implemented, it may result in unauthorised users having access to resources and operations to which they should not have permission.
- **Vertical privilege escalation:** one user can access the functionality of another user with higher privileges.
- **Horizontal privilege escalation:** a user can access the functionality of another user with the same level of access.

- **Disclosure of sensitive information:** an application gives more information than necessary to the user and could be used by an attacker for malicious purposes.
- **Privacy breach:** if the attacker ends up having access to other users' privacy-related data.
- **Identity theft:** this is when the attacker obtains the victim's identity and uses it to perform some actions.
- **Data theft:** is when an attacker illegitimately obtains data, whether confidential or not.
- **Service availability:** unauthorised access to critical elements of the application could allow an attacker to disrupt or damage the service.
- **Data manipulation:** an attacker is able to intercept and manipulate the data exchanged between the client and the server.
- **Log modification:** when an attacker manages to access and edit application or system logs, compromising the integrity of log traceability.
- **Path traversal:** the attacker moves through the server's file system, beyond the scope in which he is supposed to act and outside the context that would require authorisation.
- **Business logic:** if the application was not well designed, it could happen that, if a user does not follow the normal application logic for his business functions, he could encounter unexpected behaviour that could be exploited to perform operations that should not have been allowed.

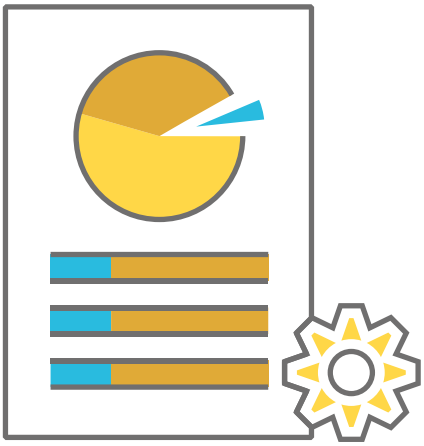


4.2. Security recommendations

- Ensure the implementation of the principle of least privilege: users have restricted access only to the functions and data they really need to perform their work normally.
- Assign permissions and privileges to application roles, never directly to users. Users must have roles and their privileges are taken from these roles.
- Check that the access to confidential records is protected, so that only authorised objects or data accessible to each user can be reached (e.g. by protecting against users manipulating a parameter to view or modify another user's account).
- Verify that browsing in the directory is disabled, unless deliberately enabled. In addition, applications should not allow the discovery or disclosure of files or directories, such as Thumbs.ds, .DS_Store, .git or .svn folders.
- Ensure that the access control rules are applied on the server side.
- Verify that all the user attributes, data and policy information used by access controls cannot be manipulated by end-users unless specifically authorised.
- Verify that there is a centralised mechanism (including libraries calling external authorisation services) to protect access to each type of protected resource.
- Ensure that the application uses strong anti-CSRF random *tokens* or implements another transaction protection mechanism.

Check that the access to confidential records is protected, so that only authorised objects or data accessible to each user can be reached

- Record all the operations on sensitive data in a specific security log containing at least: the date/time of the operation, the operation (read, create, delete, update), the name of the data, the process, function or service that generated the operation, the user, the role of the user who has the privilege for the operation, the result of the operation (whether it was successful or not) and an optional message about the result of the operation (in case of error).



4.3. Example

Using the Spring Security *framework* which provides a comprehensive set of security features, including authentication and role-based authorisation. Annotations are used to control the access to certain parts of your application.

In the following example, the **listUsers** method shall only be accessible to users with the **ROLE_ADMIN** role.

```
@PreAuthorize("hasRole('ROLE_ADMIN')")
@GetMapping("/admin/users")
public String listUsers(Model model) {
   // Código para listar a los usuarios
}
```

4.4. References

Java. Secure Authorisation:
https://docs.oracle.com/en/java/javase/19/security/java-authentication-and-authorization-service-jaas1.html [6]

Python. Simple Authorisation Secure Library:
https://pypi.org/project/python-authorization/ [7]

5. Session management

A session is an active connection between a user and the system. Session management consists of determining actions on sessions that serve to ensure the security of authorisation, integrity and privacy of information between the user and the system. Each authenticated and authorised access to the system creates a new session in the system that stores temporary information about the user's unique operations and business logic in the application for the duration of the user's access. Each session is identified by a unique identifier.

From this point onwards, the way in which the user identifies himself to the server for all other operations may vary:

- **Through *cookies*:** this is the most common and insecure mechanism. A unique identifier is generated in the browser *cookie* that is used to identify the user's session on the server as an active, authenticated session. It is more insecure because it requires delegating responsibility for *cookie* persistence to more or less secure browser implementations.

- **Through *tokens*:** this is equivalent to the previous system where the *token* is an access identifier that allows access to the system after the *token* has been associated with the corresponding session ID. This *token* usually travels in the request headers. It is more secure because it allows access *tokens* to be modified from time to time, by means of a *token* exchange, without the need to modify the session ID, which would always be kept safe on the server side.

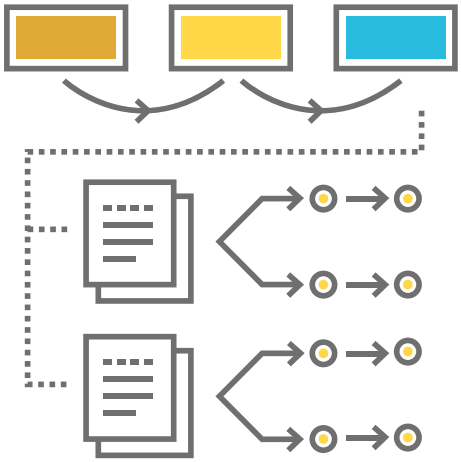
Session management consists of determining actions on sessions that serve to ensure the security of authorisation, integrity and privacy of information between the user and the system

5.1. Security aspects

The first basic aspects to consider in session management security are:

- **Securely authenticate users to ensure that only authorised persons have access to the system.**
- **Appropriately authorise the user who has obtained access.**
- **Encrypt communications between client and server with a robust algorithm.**

5.1.1. Client-side sessione



The client-side session is something that is often given little or no attention and is as important as or more important than the session that is created on the server. If this session does not exist, the following insecure situations, among others, could be created by way of example:

- A user finishes his task in the application, opens another tab and switches to another task. The server session ends, but the client does not know that the session has ended as long as it does not generate activity in the forgotten window. This tab could expose sensitive information that anyone tampering with the device could see and copy without any activity.
- A user has to fill in a form with a lot of information that takes time to complete, and may even be filled in from queries to other sources that require their attention. If the timeout of a server session was 30 minutes due to inactivity, after 2 hours filling in the form, the user would press submit and as the session expired, all his work is lost and he would find himself with a *login* on the screen asking for credentials for a new access. The client had activity for 2 hours, but the server could not perceive it.

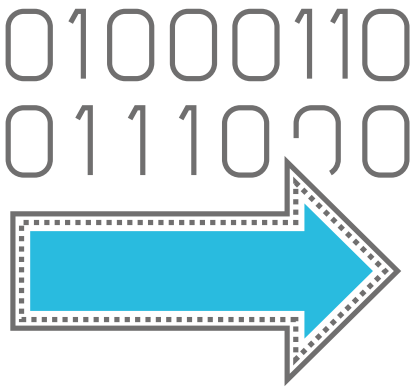
In addition, implementing the session concept on the client side would allow expiry warnings to be displayed some time before they occur. The client-side session must therefore meet some minimum security requirements:

- The persistent information of the client session ID must be stored associated to the tab, so that when the tab disappears, all the information associated to this session disappears. This can be implemented through code within the page itself or through the **sessionStorage** object.
- Persistent client session information must always be deleted at the *login* screen and created when the session is created on the server, not before.
- Allow the client session to make empty transactions against the server simply to tell the server that the session should remain active and not expire. This behaviour should be on-demand on screens that require a lot of client-side attention, never as a default behaviour.
- As with server sessions, they should control the maximum inactivity time with a similar or lesser time than server sessions. In case of inactivity on the client side, the client session should launch a logout request to the server and redirect to the *login* screen. Setting an absolute maximum session timeout may also be advisable.
- The information stored in these client sessions should be minimal and necessary for the navigation logic and should not contain sensitive information. It could contain the server access *token* to avoid dragging it in all requests (if any).
- *Login* timeout, to prevent session fixation attacks.
- Force client and server logout on browser window or tab close events.

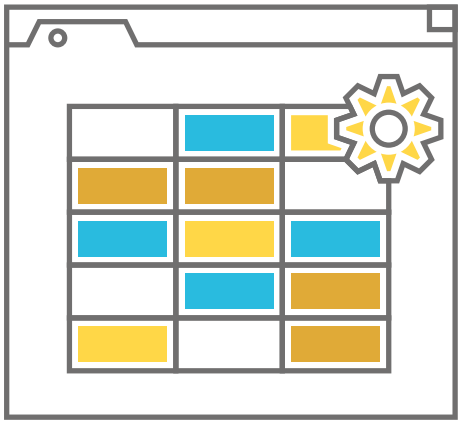
5.1.2. Session ID

The minimum security controls associated with the session ID are:

- Any session identifier must be unique, sufficiently random and of a suitable length as provided by a cryptographically secure *hash* from a random number, with a significant key length.
- The random number generator to create the session ID must be a secure random number generator.
- IDs must be validated by the server to ensure that they are in valid format and are part of active and valid sessions.
- Session IDs must not be registered. If necessary for session traceability, use a different ID, never use the ID used for session identification in client-server traffic.



- A new session ID must be generated at each *login* or when there is a change in the user's privilege level, to prevent session fixation attacks.
- The storage and monitoring of active session IDs must be secure to prevent unauthorised access by unauthorised personnel.



5.1.3. Logging out

When a server session is closed, the following security controls must be taken into account:

- Always redirect the user to the *login* page.
- Ensure that the customer will delete all *cookie* or access *token* information.
- Ensure that the client will delete all information from the client session (if applicable).
- Logging logouts in the security register

5.1.4. Expiry of the session

The minimum security controls on session expiry are:

- Every session should have a reasonable maximum lifetime in case of inactivity to avoid the session remaining permanently active.
- When the server session expires you must delete all information related to this session.
- Record session expirations in the security log.
- Set an absolute maximum time for the duration of a session.

5.1.5. Session management in mobile applications

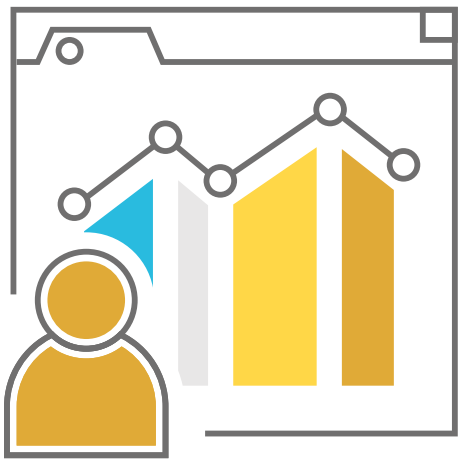
For usability reasons, mobile applications often require sessions to last longer than web applications. Recommendations for managing sessions in mobile applications are:

- Use *tokens* that can be removed if the device is lost or stolen.
- The session timeout of the mobile application shall be configured and expire depending on the sensitivity of the application.
- Use a server-based data store to facilitate the use of the session on multiple pages.
- Never use a device identifier as a session *token*.

Use a server-based data store to facilitate the use of the session on multiple pages

5.2. Potential risks

- **Session prediction:** focuses on predicting session ID values that allow an attacker to *bypass* an application's authentication scheme.
- **Session hijacking:** The session hijacking attack consists of exploiting the web session control mechanism, usually managed by a session *token*.
- **Session fixation:** this consists of obtaining a valid session ID (e.g. by establishing a connection to the application), inducing a user to authenticate with that session ID, and then hijacking the user's validated session to learn the session ID used.
- **Session spoofing:** when the attacker gains the identity of another entity to commit some kind of fraud. For example, an attacker who generates a malicious website under the guise of a legitimate bank to deceive victims through phishing.



5.3. Security recommendations

- It is crucial to ensure that the session ID is never exposed in unencrypted traffic.
- Implement secure headers with directives such as **cache-control** or **strict-transport security**.
- Check that sessions are invalidated when the user logs out.
- Sessions must expire after a specified time of inactivity.
- Ensure that all pages requiring authentication have easy and user-friendly access to the logout functionality.
- Verify that the session ID never appears in URLs, error messages or logs.
- Ensure that every successful authentication and re-authentication generates a new session and a new session ID, destroying the old one.
- Check that the session ID stored in the *cookies* is set using the **HttpOnly** and Secure attributes.
- Set the Path attribute of session *cookies* appropriately to prevent access to other domains.
- Check that the application keeps track of all the active sessions and allows the user to end sessions selectively or globally from their account.
- In the case of high-value applications, ensure that the user is required to close all the active sessions if the password has just been successfully changed.

It is crucial to ensure that the session ID is never exposed in unencrypted traffic.

- Limit access to protected URLs, roles, application data, user attributes and access configuration data only to authorised users.
- Record in a security log all the sessions that are created, manually closed or expired from the client or from the server identified by the username and the server's internal session ID (not shared with the client) including the date/time of the event.

5.4. Example

In Java, session management can be performed by using the **javax.servlet.http.HttpSession** interface. This interface provides methods to store and retrieve session attributes, set and get the session timeout, and invalidate the session.

To use the **HttpSession** interface, an instance must first be obtained:

```
HttpSession session = request.getSession();
```

Then, the interface methods can be used to carry out session management:

```
session.setAttribute("user", "John");
```

To retrieve a session attribute:

```
String user = (String) session.getAttribute("user");
```

To set the session timeout time (in seconds):

```
session.setMaxInactiveInterval(3600);
```

To invalidate the session:

```
session.invalidate();
```

It is important to note that to ensure the security of session management, secure HTTPS *cookies* must be used and unique and unpredictable session identifiers must be generated. Also make sure to validate the request and the session state for each request to avoid session spoofing.

```
import java.security.MessageDigest;
import java.security.SecureRandom;
import java.util.Arrays;
import java.util.Base64;
...

SecureRandom random = new SecureRandom();
byte[] bytes = new byte[32];
random.nextBytes(bytes);
MessageDigest digest = MessageDigest.getInstance("SHA-256");
byte[] hashedSessionId = digest.digest(bytes);
String sessionId = Base64.getEncoder().encodeToString(hashedSessionId);

Cookie sessionCookie = new Cookie("SESSIONID", sessionId);
sessionCookie.setHttpOnly(true);
sessionCookie.setSecure(true);
sessionCookie.setMaxAge(3600); // expira en 1 hora
response.addCookie(sessionCookie);
```

5.5. References

OWASP. Secure Session Management:
https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html [8]

Java. SpringSession Framework:
https://www.baeldung.com/spring-session [9]

6. Validation of input and output data

It is one of the most important critical areas of application security. Most application vulnerabilities arise from incorrect or insufficient validation of input or output data that is exploited for attacks such as *cross-site scripting*, general injections, exposure of sensitive data or DoS.

Although validation of output data is not a very common activity among developers, it is equally important, and could be exploited by malicious users if adequate security measures were not included. The output returned by an application could be used to perform the same types of attacks, albeit in a more sophisticated way.

Implementing validations of all input and output data does not guarantee that the application will be free of vulnerabilities as such validations may be insufficient and may not have taken into account other types of functional or business logic issues that could affect the normal behaviour of the application. An example of this would be validating a numeric input where it is checked that it is a number, that it is positive and that it is not 0. However, if the application uses it within an iterative loop, it could be called with a huge value and put the application in an almost infinite loop; which would be a DoS attack.

Most application vulnerabilities arise from incorrect or insufficient validation of input or output data that is exploited for attacks such as *cross-site scripting*, general injections, exposure of sensitive data or DoS

6.1. Validation Techniques

6.1.1. Sanitisation

It is a process for converting data that has more than one possible representation into a standard, canonical or normalised format, reducing the input/output to a single converted and/or reduced fixed form. This technique alone avoids many validation problems and many types of attacks.

There are libraries for each type of language that already incorporate sanitisation methods.

6.1.1.1. SANITISATION OF ROUTES

All the file or directory paths are normalised.
For example, a UNIX path such as **/home/user/myFile.txt**, could be defined as **/var/log/.../.../home/user/myFile.txt**. This could be used by a malicious user to browse the file system and obtain unauthorised information. To avoid this, the path is canonicalised to a fixed form or the "../" and "./" substrings are removed from the path directly.

6.1.1.2. SANITISATION OF SPACES

All the input parameters remove spaces, tab characters or other white characters (160) before and after the data.
Some also choose to remove all white characters within the data and convert them directly to a space code (32).

6.1.1.3. SANITISATION OF CHARSET

It is verified that all the characters entered in the data correspond to the expected charset. Those that do not, are deleted or transformed to a white space. Characters that may arrive encoded as "0xA0", "/xA0", "%A0", etc. are taken into account to be decoded before being processed.

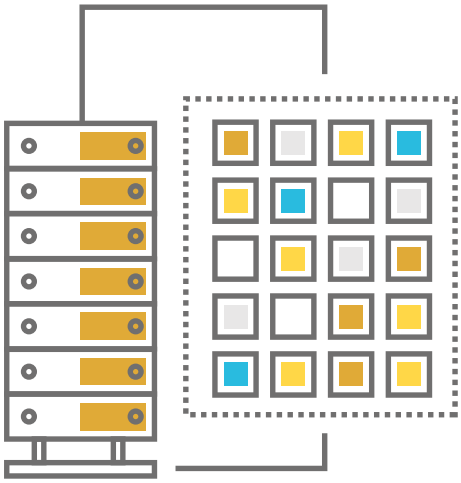
6.1.1.4. CASE SANITISATION

All the characters are converted to upper or lower case, as required by the defined parameter.

6.1.2. Data type

Validates that the received data is of the expected type with the appropriate format. Data types can be simple or complex depending on the definitions implemented in the application. Simple data types would be, for example: integer, decimal, boolean, boolean, character string, and complex data types would be e-mail, NIF, address, name, name, CP, date, time, URL, etc..

6.1.3. Format



It allows to validate data with more or less complex and variable formats but which can perfectly well be defined within a regular expression. Many of the complex data types validated in the previous point are likely to be done using this technique.

The only thing to keep in mind is to create a regular expression so complex that it could be vulnerable to a ReDoS attack. To ensure that this does not happen, you should validate the regular expression with a tool that guarantees its security and suitability, such as **RegEx 101** or **RegEx Testing**.

6.1.4. Minimum and maximum sizes

Validates that the size (length) of the data exceeds a minimum number of characters or does not exceed a maximum number of characters. This check would prevent sending data so large that the application would be "frozen" simply processing the input.

6.1.5. Minimum and maximum values

Unlike the previous point, this validation applies to numeric values that must be within a range of maximum and minimum.

6.1.6. White list

Validates that the data is within a list of prefixed data. This validation is widely used on data that is part of enumerations..

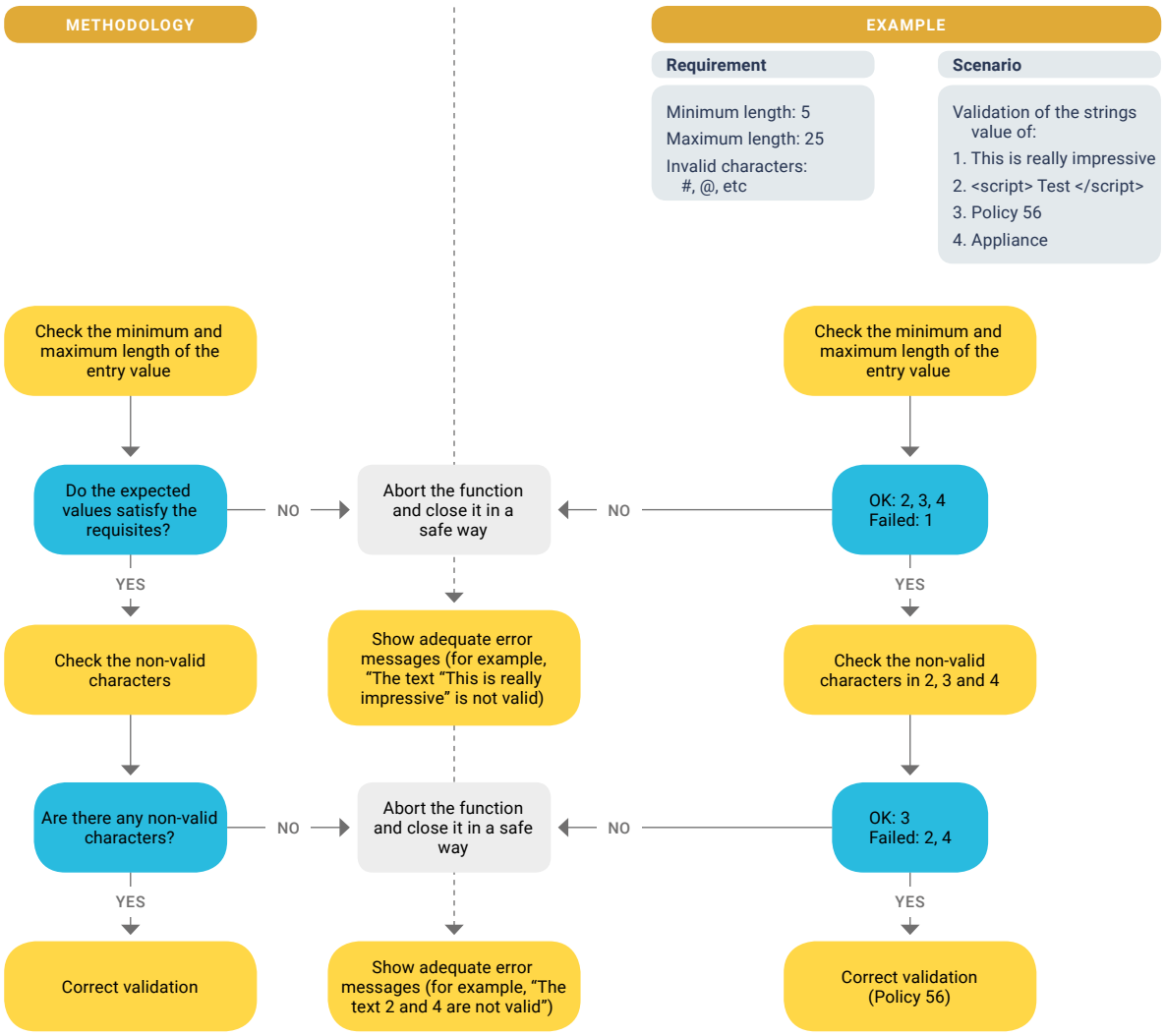
6.1.7. Blacklist

Validates that the data is not found within a list of prefixed data. This type of validation is often insufficient and dangerous because it can hardly cover all malicious possibilities.

However, it can be useful when deciding to reject texts containing offensive words that exist within a blacklist.

6.2. Flowchart

The following diagram is an example to illustrate the workflow for validating free text values. It includes four (4) input values that are checked against some simple requirements. When a value does not satisfy any requirements, the input data must be rejected.



6.3. Potential risks

Data validation is the source of many vulnerabilities that an attacker can exploit. The following are some of the vulnerabilities related to insufficient or missing input validation:

- **Cross-Site Scripting:** is a type of attack that allows a malicious user to inject code into the victims' web browser.
- **SQL Injection:** exploits application programming flaws at the input validation level to perform operations on a database illegitimately.
- **LDAP Injection:** consists of injecting arbitrary LDAP queries to access forbidden data or even obtain additional privileges.
- **Log Injection:** consists of injecting execution commands into the system or any other type of problem using the log system that records data from input parameter information.
- **XEE Injection:** An XPATH injection is the injection of arbitrary XML code with the intention to access data that should not be accessed or to obtain information about the XML tree structure.
- **XML bomb:** This attack attempts to overload XML by exceeding the memory resources of an application to cause a denial of service.
- **DoS, DDoS or ReDoS:** denial of service attacks due to system error in processing unvalidated input causing system failures.

Data validation is the source of many vulnerabilities that an attacker can exploit

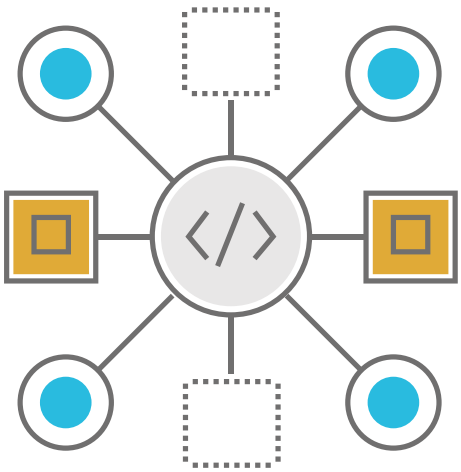
6.4. Security Recommendations

- Validations should always be performed on the server side. Client-side validations can also be useful but are only recommended.
- Failing that, use standard input validation mechanisms provided by technology-specific libraries (Spring Validator, etc.).
- Fully cover **data validation through validation schemes** or standard mechanisms that ensure data entry through: sanitisations, data type, format, lengths, values, whitelists, blacklists, etc.
- Checking that structured data is strongly typed and validated according to a defined schema, including allowed characters, length and pattern, e.g. credit card or telephone numbers, or validating that two related fields are reasonable, such as validating suburbs and postcodes.
- Verify that unstructured data is sanitised to impose generic security measures, such as allowed characters and length, and avoid potentially harmful characters.
- Ensure that all unreliable input is properly sanitised using a sanitisation library.
- Avoid displaying sensitive information because of a validation error of a received parameter.
- Accept only the expected data at each entry point of the application that comes from the user, from the end process of all input fields, forms, URLs, application *cookies*, etc. Any unexpected data should be rejected.

Validations should always be performed on the server side. Client-side validations can also be useful but are only recommended

- Verify that server-side input validation errors result in rejection of the request.
- Ensure that all **database** queries are **protected using parameterised queries** to prevent SQL injection.
- Check that the application is not susceptible to command injection.
- Verify that all string variables located within HTML or other web client code are correctly hand-coded in context or use templates that automatically encode the context to ensure that the application is not susceptible to cross-site scripting (XSS) reflected, stored or DOM.
- Check that the application does not contain mass parameter assignment vulnerabilities (AKA automatic variable binding). Ensure that all input data is validated, not just HTML form fields, but all input sources such as REST requests, query parameters, HTTP headers, *cookies*, batch files, RSS feeds, etc., using whitelists, lesser forms of validation such as greylists (which remove the known bad strings) or blacklists (which reject bad input).
- Verify that the application restricts XML parsers to use only the most restrictive settings possible and ensure that dangerous functions, such as external entity resolution, are disabled.
- Verify that deserialisation of unreliable data is prevented or largely protected when deserialisation cannot be avoided.

Verify that server-side input validation errors result in rejection of the request



6.5. Example

In Java Spring, annotations are used to validate input parameters to methods and allowed values in the members of a class:

```
@PostMapping("/users")
public ResponseEntity<User> createUser(@Valid @RequestBody User user) {
  ...
  return ResponseEntity.ok(user);
}

public class User {
  @NotBlank
  private String name;

  @Email
  private String email;

  ...
}
```

The **@Valid** annotation is used in conjunction with a validation object to validate the input parameters of a controller method.

The **@NotBlank** and **@Email** annotations to indicate that the **name** field must not be blank and that the **email field** must be in a valid email address format.

6.6. References

OWASP. Parameter Validation: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html [10]

OWASP. Data Validation - ESAPI Library: https://owasp.org/www-project-enterprise-security-api/ [11]

Validation of Web Forms in Javascript: https://www.tutorialspoint.com/javascript_form_validation_web_application/index.asp [12]

Java. Spring Boot Validation: //www.baeldung.com/spring-boot-bean-validation [13]

XSS Prevention: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html [14]

Python. Colander Validation: https://pypi.org/project/colander/ [15]

Python. Cerberus Validation: https://docs.python-cerberus.org/en/stable/ [16]

Python. Validation Schematics: https://schematics.readthedocs.io/en/latest/ [17]

Python. Schema Validation: https://pypi.org/project/schema/ [18]

Python. JSON Schema Validation: https://pypi.org/project/jsonschema/ [19]

7. Error management

Error handling is about how to avoid displaying relevant or sensitive information to users that could be used to launch other types of sophisticated attacks against the application and how to handle uncontrolled errors within the application to provide safe exits that do not allow unexpected situations to be exploited by malicious users.

If the applications have not implemented proper error handling, they may inadvertently reveal information about their configuration, internal state, debugging messages, sensitive data, or even violate privacy

7.1. Confidentiality of messages

If the applications have not implemented proper error handling, they may inadvertently reveal information about their configuration, internal state, debugging messages, sensitive data, or even violate privacy.

Other ways of disclosing information include, for example, the time taken to process certain transactions, or offering different codes for different inputs.

All this information could be exploited to launch, or even automate, very powerful attacks, which makes good error handling essential.

POTENTIAL RISKS

The following are the most common potential risks if the error handling is not properly implemented within the application:

- **Leakage of sensitive information:** server version, database engine, sensitive documents, file structure, etc.
- **Denial of service:** when errors forced through abuse can cause system downtime.
- **Cross-site scripting:** when error messages show input parameters that have not been correctly escaped.

7.2. Uncontrolled errors

When situations where exceptions or errors at points in the application could occur have not been contemplated, and do occur, the application would cause an unexpected exit from the business logic that could leave it in a vulnerable state to subsequent user activities.

It is therefore imperative that all transactions are fully analysed in terms of the various exceptions that may occur and that these exceptional outflows are dealt with appropriately.

Another way to exploit unchecked errors is when resources are not properly shut down, which can eventually cause a denial of service due to excessive consumption of resources that are not shut down as a result of forcing errors in the application. To avoid this vulnerability, it is essential to ensure that all resources are shut down when they are no longer used, regardless of any errors that may appear during operations. It is recommended to use **try / finally** statements to ensure the closure of resources.

POTENTIAL RISKS

The following are the most common potential risks if error handling is not properly implemented within the application.

- **Denial of service:** when errors forced through abuse can cause system downtime.
- **Bypassing business logic:** when unexpected exits from the business logic occur by forcing exceptions or uncontrolled errors.

7.3. Security recommendations

- Use generic error messages that do not give clues to end-users about any sensitive aspects of the application.
- Use centralised exception handling.
- The application must handle errors without relying on server error messages displayed to users.
- Any access control logic that leads to an error must deny access by default.
- Analyse in detail all exceptions that may occur due to the use of system libraries or third-party libraries in the application, handle them appropriately and provide a safe exit to the application.
- Use **try/catch/finally** to ensure the shutdown of all resources in case of error.
- Log all unexpected exceptions in a specific log indicating, among others, the date/time of the failure, the user who caused it and the method where the failure occurred, as well as exception information. This log should be located in a secure environment accessible only to authorised users.

7.4. Example

The following example would be the typical code block that should be found in any method where you want to control errors that may appear in a code block.

Any unhandled exception within the **try** block will be caught by **catch** from where a handled exception will be thrown. And, in either case, the **finally** block will be executed to close open resources before the **try** or after the **try**

```
try {
   ...
   // code that can throw an exception
   ...
} catch (Exception e) {
   // exception handling
   log.error("ERROR", e);
   throw new CustomException("ERROR", e);
} finally {
   // closure or release of open resources
   ...
}
```

7.5. References

Java. Try - Catch - Finally:
https://www.w3schools.com/java/java_try_catch.asp [20]

Java. Exception Handling:
https://www.baeldung.com/java-exceptions [21]

Error Handling Strategies:
https://dzone.com/articles/error-handling-strategies [22]

Python. Errors and Exceptions:
https://docs.python.org/es/3/tutorial/errors.html#errors-and-exceptions [23]

8. Secure Registration

It consists of recording the activity of applications and system events related to security, in terms of authentication, authorisation, integrity or confidentiality, such as failed connection attempts, authorisation acquired by a user, accesses to sensitive data, etc., as well as possible threats detected that can be controlled from the application: injections, user enumeration, brute force attacks, etc. The latter is especially important when forensic analysis of security incidents that have occurred is desired.

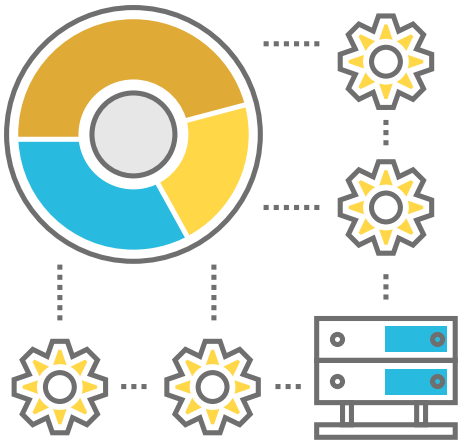
Security records must be specially protected at the level of authentication, integrity, confidentiality, availability and traceability:

- **Authentication/Authorisation:** only identified and authorised persons can have access to the registry.
- **Integrity:** the registry maintains an integrity signature that ensures that it has not been tampered with at the registry level or at the record level. This signature is updated with each new entry in the registry.
- **Confidentiality:** sensitive registry data is tokenised, anonymised or encrypted to prevent access to the registry by methods other than those implemented for authorised access.
- **Availability:** records are stored with redundancy and backup copies.
- **Traceability/Auditability:** they must be stored securely for a retention time for audit purposes.

It consists of recording the activity of applications and system events related to security, in terms of authentication, authorisation, integrity or confidentiality

8.1. Potential risks

- **Information leakage:** logs may contain sensitive information about the application or system and may not have been adequately protected against loopholes in access authorisation.
- **Log forgery:** an unauthorised user could modify log files, which can lead to loss of traceability in the application or even allow the malicious user to execute code on the system.
- **Log deletion:** a malicious user could delete logs to avoid leaving traces of his crimes.



8.2. Security recommendations

- Do not record sensitive information such as passwords, financial information, credit cards, personal details, etc. In these cases, use *tokens*, anonymisation of information or encryption.
- Validate the parameters of the variable components that will make up the log entry to avoid injections and unexpected behaviour.
- Make sufficiently accurate notes for security:
  - Authentication attempts, especially failures.
  - The accesses granted with the roles associated with the user.
  - Access to and actions taken on sensitive data, which role was used and by which user.
  - Input validation errors.
  - Exceptions to the system.

- Possible detected threats or threat attempts that can be controlled by the application: injections, brute force attacks, user enumeration, business logic failures, unprivileged operation attempts, path traversal attempts, etc.

- Use the *hash* functions to ensure data integrity of records.

- The security register must be independent of other registers and have its own security protections in the system.

8.3. Example

Secure logging in Java can be performed using the **java.util.logging** library.

```
import java.util.logging.Logger;

public class MyClass {
  private static final Logger log = Logger.getLogger(MyClass.class.getName());

  public void myMethod() {
    // algunas operaciones que pueden lanzar una excepción
    try {
      // código que puede lanzar una excepción
    } catch (Exception e) {
      log.severe("Ocurrió un error grave: " + e.getMessage());
    }
  }
}
```

8.4. References

Security Log. Best Practices for Logging and Management:
https://www.dnsstuff.com/security-log-best-practices [24]

Java. Logging:
https://docs.oracle.com/javase/7/docs/technotes/guides/logging/index.html [25]

9. Cryptography

Cryptography is a technique used to protect information and communications using secret codes or keys and is an essential tool for protecting information and communications.

It is used to ensure the confidentiality, integrity and authenticity of information and communications: confidentiality by protecting information so that it can only be accessed by authorised persons, integrity by protecting information against unauthorised alteration and authentication by verifying the identity of persons or systems accessing information.

Cryptography can be used in different ways: as encryption of sensitive data to protect information or as a digital signature to ensure data integrity.

Cryptography is a technique used to protect information and communications using secret codes

9.1. Use of encryption

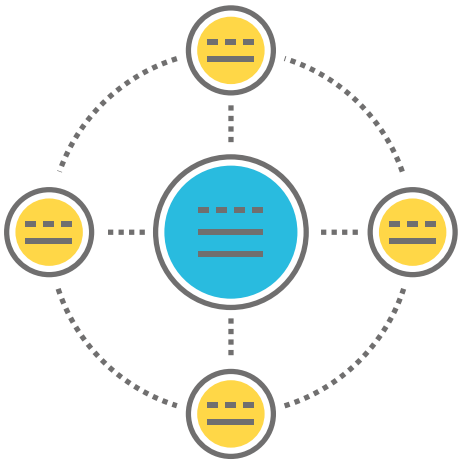
9.1.1. *Hash* functions

A cryptographic *hash* is a mathematical function that, from a set of data, produces a fixed amount of data called a "digest" or "hash". Depending on the function used, the number of data obtained may vary, but it will always generate the same amount of data for the same function, regardless of the number of data used in the input.

The power of cryptographic hashing is that the probability of generating the same output data, in the same order, from a different input is very low, very unlikely. If this were to happen, a so-called "collision" would occur, and this effect, discovered in one of these mathematical functions, could be exploited by malicious users for some security attacks.

A *hash* function is commonly used to verify the integrity of data, since any change in the original input will be reflected in a significant change in the resulting *hash*. When the *hash* function is fed with the data we want to secure, the output data from the function provides us with the verification *hash*. This *hash* could only be retrieved again with the same function, with the same input data and in the same order.

Hash functions are used for the storage of passwords or to check the integrity of data to ensure that it has not been modified.



9.1.2. Symmetric encryption

Symmetric encryption is a type of encryption in which the same key is used for both encrypting and decrypting information. It is used to protect the confidentiality of information during transmission or as storage on a device.

The main advantage of symmetric encryption is that it is fast and easy to implement. Its main disadvantage is that both parties must share the secret key in order to communicate securely. This can be a problem in distributed environments, as it requires a secure mechanism to share the key reliably.

9.1.3. Asymmetric encryption

Asymmetric encryption is a type of encryption in which two (2) different keys, known as public key and private key, are used to encrypt and decrypt information. The public key is used to encrypt the information and can be shared without problems, while the private key is used to decrypt the information and must be kept secret.

The advantages of asymmetric encryption are that the private key does not need to be shared to establish secure communication. The public key can be shared seamlessly and is used to encrypt the information, while the private key is used to decrypt it. However, asymmetric encryption is slower than symmetric encryption and can be more difficult to implement.

9.2. Potential risks

Information that is not encrypted or not properly encrypted is exposed to the following risks:

- **Exposure of sensitive information:** sensitive data would be clear to any unauthorised person.
- **Credential theft and spoofing:** sensitive password information stolen could be used to impersonate another user and to carry out other attacks such as spoofing.
- **Leakage of personal data:** this is data protected by country regulations whose privacy violation could result in financial penalties.
- **Loss of reputation of the company:** as a consequence of all of the above.
- **MitM (Man-in-the-Middle) attacks:** if communications are not well secured, communications could be intercepted, and the entire data stream decrypted to obtain valuable information that could be used for other attacks.

9.3. Security recommendations

- All sensitive information of an organisation such as passwords, personal data, log repositories or any other information labelled as confidential or superior to the company must be stored in an unreadable (encrypted) form to ensure the confidentiality of the company.
- Check that all random numbers, random file names, random GUIDs and random strings are generated by a random number generator approved by the cryptographic module.
- Check that there is an explicit policy on the handling of cryptographic keys (such as generation, distribution, revocation and deprecation).
- Check that the lifecycle of cryptographic keys is correctly implemented.
- Ensure that sensitive passwords or critical information residing in memory are overwritten with zeros as soon as they are no longer used to mitigate memory dump attacks.
- Check that random numbers are created with an appropriate level of entropy, even when the application is under high load.
- Check that obsolete or weak cryptographic algorithms such as the symmetric key DES algorithm or *hash* functions such as MD5 or SHA-1 are not used due to the impossibility to guarantee confidentiality.
- Use existing cryptographic libraries and in any case use custom or user-created cryptographic algorithms or implementations.

All sensitive information of an organisation such as passwords, personal data, log repositories or any other information labelled as confidential or superior to the company must be stored in an unreadable (encrypted) form to ensure the confidentiality of the company.

9.4. Example

In this example, symmetric cryptography is used to encrypt a message using a secret key and an initialisation vector (IV) to protect against "key reuse" attacks.

```
import java.security.SecureRandom;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;

public class AdvancedCryptographyExample {
    public static void main(String[] args) throws Exception {
        // We generate a secret key for symmetric cryptography.
        KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
        keyGenerator.init(256); // we use a 256-bit key
        SecretKey secretKey = keyGenerator.generateKey();

        // We generate an initialisation vector (IV) for symmetric cryptography
        byte[] iv = new byte[16]; // los IV típicamente tienen un tamaño de 16 bytes
        SecureRandom secureRandom = new SecureRandom();
        secureRandom.nextBytes(iv);
        IvParameterSpec ivParameterSpec = new IvParameterSpec(iv);

        // Encrypt a message using the secret key and the IV
        String message = "Este es un mensaje secreto";
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, secretKey, ivParameterSpec);
        byte[] encryptedMessage = cipher.doFinal(message.getBytes());

        // We store the secret key in a secure way (e.g. using an encryption key)
        saveSecretKey(secretKey);

        // Save the IV and the encrypted message somehow (e.g. in a database)
        saveIvAndEncryptedMessage(iv, encryptedMessage);
    }

    private static void saveSecretKey(SecretKey secretKey) {
        // We store the secret key in some way (e.g. using an encryption key)
    }

    private static void saveIvAndEncryptedMessage(byte[] iv, byte[] encryptedMessage) {
        // Save the IV and the encrypted message somehow (e.g. in a database)
    }
}
```

9.5. References

OWASP. Secure cryptography in Java:
https://www.owasp.org/index.php/Using_the_Java_Cryptographic_Extensions [26]

Scrypt key derivation function:
https://www.tarsnap.com/scrypt.html [27]

Python. bcrypt cryptographic functions:
https://pypi.org/project/bcrypt/ [28]

10. Secure file management

It is a process that involves the protection of data stored in files to ensure their integrity, confidentiality and availability, including measures such as cryptography, authentication, authorisation, access control and backup.

This process should be considered during the design phase of an application and then implemented during its development. Most applications rely on internal files in order to function and, in addition, if the file uploading by users are allowed, appropriate security controls must be put in place.

10.1.Potential risks

Without good file management, the following risks could exist:

- **Unauthorised access to files, which would result in the:**
  - Disclosure of data.
  - Loss of sensitive information.
  - Data manipulation.
  - Deletion of data.
- **Loading of malicious files, which would result in the:**
  - Remote file execution.
  - Denial of service attack.
  - Malware infection.
- **Lack of backups:**
  - Unavailability of service (DoS).
  - Loss of user data.
  - Loss of valuable data to the company.

10.2. Security recommendations

- Authenticate and authorise the user before uploading or downloading any files, especially if the data is sensitive.
- Do not use user-supplied input to name files or directories.
- Validate content types not only by extension, but also check MIME types to verify files.
- Do not allow executable files to be uploaded to the application.
- Limit file sizes to the minimum that the server can handle without causing availability problems and impacting application functionality.
- Before processing the files to the server, an anti-virus scanner checks the files for malware or viruses.
- Disable execution privileges on directories where users can upload files.
- Do not use absolute paths when providing a download link to the user.
- Do not store files with their names sequentially.
- Do not use sensitive information for file naming.
- Ensure that the access control is set to read-only.
- Limit the number of files uploaded by the user.
- Store hashes of uploaded files to ensure their integrity.

10.3. Example

The following example validates the file size and checks that the *mimetype* matches what is expected before processing the file.

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.List;

public class FileManager {
   private static final long MAX_FILE_SIZE = 10485760; // 10MB
   private static final List<String> ALLOWED_MIME_TYPES = List.of
      ("text/plain", "image/jpeg", "image/png");

   public static void processFile(String filePath) throws IOException {
      // Check that the file exists and is readable
      File file = new File(filePath);
      if (!file.exists() || !file.canRead()) {
         throw new IOException("File does not exist or cannot be read");
      }
      // Check file size
      long fileSize = file.length();
      if (fileSize > MAX_FILE_SIZE) {
         throw new IOException(("File is too large");
      }
      // Check the file's mimetype
      Path path = Paths.get(filePath);
      String mimeType = Files.probeContentType(path);
      if (!ALLOWED_MIME_TYPES.contains(mimeType)) {
         throw new IOException("File type is not allowed");
      }
                  // Process the file
      // ...
   }
}
```

Another solution is to configure the Apache server to limit this type of request. This is optimal in terms of performance and more secure because the request stays on the web server and does not even enter the application.

```
LimitRequestBody 10485760
SetEnvIf Request_URI "^.*$" ALLOWED_MIME_TYPE=1
SetEnvIf Request_URI "^.*\.(txt|jpe?g|png)$" ALLOWED_MIME_TYPE=1
```

10.4. References

Apache Server. Configuration Directives:
https://httpd.apache.org/docs/2.4/mod/core.html#limitrequestbody [29]

Java. Canonisation of Path:
https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/io/File.html#getCanonicalPath() [30]

PHP. Path canonicalisation:
https://www.php.net/manual/es/function.realpath.php [31]

Python: Mapping files with their MimeTypes:
https://docs.python.org/3/library/mimetypes.html [32]

11. Transaction security

It is a set of measures designed to protect financial and payment transactions from possible fraud or attacks. These measures include:

- **User authentication:** only authorised persons have access to financial transactions and accounts. User authentication typically includes strong passwords and, ideally, two-factor authentication.
- **Cryptography:** helps to protect confidential information during transactions.
- **Validation of transactions:** to ensure that they are legitimate and not part of a fraud or cyber-attack.
- **Transaction monitoring:** to help detect and prevent potential fraud or attacks in real time.
- **Back-up:** A data recovery plan in case of an attack or accidental loss of data ensures data availability.

Any interaction with a complex data structure consisting of several sequentially applied processes must be carried out at once and in a secure manner.

They must meet the following properties:

Atomicity	Consistency	Isolation	Durability
Transactions must have a start point and an end point, always and in any case. They must be uniquely identified. And all operations performed by the transaction must execute successfully or, in case of error, rollback the operations to the initial state.	Transaction data must be validated and consistent in terms of its integrity and, in case of causing a change of state, a valid and expected state.	Transaction operations can be performed independently of the operations of other transactions without affecting each other's data or their particular states.	Transactions have a maximum lifetime, and once completed, their information is persistent.

11.1. Potential risks

There are several potential risks associated with transactions:

- **Exploitation of the Payment Bypass vulnerability:** due to an inadequate configuration of the payment system. It allows an attacker to manipulate the parameters exchanged between the client and the server by processing the response before it is sent to the payment gateway and bypassing the payment system in general.
- **Fraud:** making use of false information or manipulating transactions to obtain illicit benefits.
- **Theft of confidential information:** to be used to do commercial damage or for future attacks
- **Disruption of transaction processing:** through DoS/DDoS attacks.
- **Loss of data:** due to system failures, attacks or natural disasters with serious consequences for transactions, which may affect the confidentiality and integrity of information.

11.2. Security recommendations

- **To avoid Payment Bypass vulnerability:**
  - The authorisation and confirmation of a purchase must be done on the server side.
  - It is necessary to validate that the signatures used are correct during the communication process with the payment gateway.
  - Validate that the price is correctly set on the server side.
  - Validate that payments are not reused.
  - The payment server must always check at which stage of the transaction you are in.
- **Include a relatively short authorisation expiry time for each transaction.**

- **Ensure traceability of transactions.**
- **Encrypt communications with robust asymmetric algorithms.**
- **Detailed recording of all transactions with anonymisation of sensitive information.**

11.3. Example

This example could serve as a guide to avoid the Payment ByPass vulnerability:

```
import java.math.BigDecimal;

public class PaymentProcessor {

   private static final BigDecimal MIN_PAYMENT_AMOUNT = new BigDecimal("0.01");

   public void processPayment(String tId, BigDecimal amount, String paymentMethod) {
      // Check if the transaction ID is valid
      if (tId == null || !isValidTransactionId(tId)) {
      throw new IllegalArgumentException(("Invalid transaction ID");
      }
      // Check if the amount and method of payment is valid
      if (amount == null || amount.compareTo(MIN_PAYMENT_AMOUNT) < 0) {
      throw new IllegalArgumentException("Invalid payment amount");
      }
      if (paymentMethod == null || !isValidPaymentMethod(paymentMethod)) {
         throw new IllegalArgumentException("Invalid payment method");
      }
      // Process payment
   }
   private boolean isValidTransactionId(String transactionId) {
      // Check if the transaction ID is in the list of allowed IDs
   }
   private boolean isValidPaymentMethod(String paymentMethod) {
      // Check if the payment method is in the list of allowed payment methods
   }
}
```

11.4. References

WordPress. Plugin to avoid the Payment ByPass vulnerability:
https://www.acunetix.com/vulnerabilities/web/wordpress-plugin-nab-transact-security-bypass-2-1-0/

12. Communications security

Security in application communications is essential to protect the confidentiality, integrity and availability of information transmitted through applications. This is especially important in the context of mobile applications, where information may be transmitted over insecure or public networks.

Security in application communications is essential to protect the confidentiality, integrity and availability of information transmitted through applications

12.1. Potential risks

An application without communication security measures is exposed to several potential threats:

- **Theft of confidential information.**
- **Service interruptiono.**
- **Identity theft.**
- **Modification or destruction of data.**
- **Spread of malicious software.**

12.2. Security recommendations

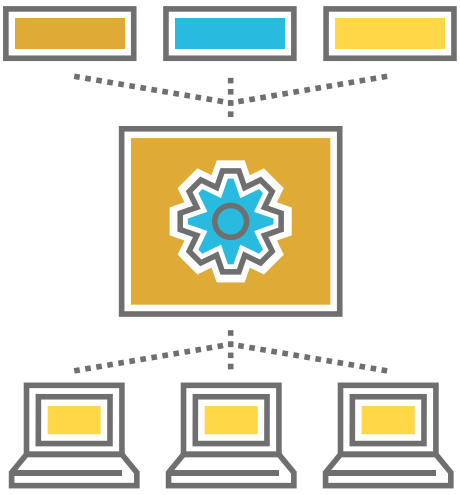
- **Always encrypt communication channels by means of:**
  - **TLS:** is a cryptographic protocol that allows communication channels to be encrypted. This protocol applies privacy, authentication and data integrity as properties of the communication channel.
  - **WebSocket:** is a technology that provides a bidirectional (two-way, send and receive) and full-duplex (simultaneous) communication channel over the same TCP socket.
- **It uses strong cryptography:** strong enough to make any attempt at decryption futile.
- **Use security protocols:** such as HTTPS or FTPS.

13. Data protection

The components on which security is focused to protect data are the following::

- **Authentication:** the user accessing the data is who he/she claims to be and is given roles or access privileges
- **Authorisation:** access privileges or user roles allow only certain types of operations on only certain data.
- **Confidentiality:** data must be protected from unauthorised observation or disclosure in transit and when stored.
- **Integrity:** data must be protected in case it is maliciously created, modified or deleted by unauthorised attackers.
- **Availability:** data must be available to authorised users whenever necessary (backup policies).

This standard assumes that data protection is implemented in a trusted system that has been built with sufficient security safeguards.



13.1. Potential risks

Any security vulnerability that is exploited in an application's data could lead to:

- **Failure to comply with the regulations** and legislation on the processing of personal data may result in financial penalties or interruption of the service.
- Compromise or **loss of** sensitive company **information**.
- Compromise or **loss of** sensitive third-party **information** which could lead to litigation and financial loss.
- **Loss of** corporate **image**.
- **Loss of certifications** as a consequence of non-compliance regarding data protection.

13.2. Security recommendations

- Check that all sensitive or personal information to be handled by the application is identified, and that there is an explicit policy specifying how access to it is to be controlled, processed and, when stored, properly encrypted in accordance with the correct data protection guidelines, in compliance with local laws and regulations.
- Ensure that all sensitive data is sent to the server in the HTTP message body or headers, avoiding sending sensitive data via URL parameters.
- Check that the communication channels used for sending confidential data are secure using strong encryption algorithms.

- Check that stored sensitive data is encrypted with strong encryption algorithms.
- Check that the application sets sufficient headers against caching, so that any sensitive information is not stored in the cache of modern browsers (e.g. visit over cache to check the disk cache).
- Ensure that sensitive information stored in memory is overwritten with zeros as soon as it is not needed, to mitigate memory dump attacks.
- Verify that a secure data deletion policy is in place when assets have reached the end of their life cycle.

13.3. Example

An example configuration of the Apache Tomcat server to secure communications with TLS would be, in the server.xml file:

```
<Connector
   protocol="org.apache.coyote.http11.Http11NioProtocol"
   port="8443" maxThreads="200"
   scheme="https" secure="true" SSLEnabled="true"
   keystoreFile="mykeystore" keystorePass="<password>"
   clientAuth="false" sslProtocol="TLS"/>
```

An example of using SSL sockets in Java:

```
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSocketFactory;

// Create an SSL socket factory
SSLSocketFactory sslSocketFactory = (SSLSocketFactory) SSLSocketFactory.getDefault();

// Create an SSL socket and establish the connection
SSLSocket sslSocket = (SSLSocket) sslSocketFactory.createSocket("www.example.com", 443);
```

An example of using HTTPS connections in Java:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.URL;
import java.net.URLConnection;

// Create an HTTPS connection
URL url = new URL("https://www.example.com");
URLConnection connection = url.openConnection();

// Send a POST request to the HTTPS connection
connection.setDoOutput(true);
OutputStreamWriter out = new OutputStreamWriter(connection.getOutputStream());
out.write("param1=value1&param2=value2");
out.close();

// Read HTTPS connection response
BufferedReader in = new BufferedReader(new InputStreamReader(connection.
getInputStream()));
String inputLine;
while ((inputLine = in.readLine()) != null) {
   System.out.println(inputLine);
}
in.close();
```

An example of using secure connections with WebSockets in Javascript, on the client side

```
// Create an HTTPS connection with WebSockets
const socket = new WebSocket("wss://www.example.com/ws");

// Send a message over HTTPS connection
socket.send("Hello, world!");

// Receive messages from HTTPS connection
socket.onmessage = function(event) {
   console.log("Mensaje recibido:", event.data);
};
```

13.4. References

SSL/TLS algorithms:
https://docs.oracle.com/en/java/javase/15/docs/specs/security/standard-names.html#sslcontext-algorithms [34]

Java. HttpClient with SSL:
https://www.baeldung.com/java-httpclient-ssl [35]

Python. SSL/TLS:
https://docs.python.org/3/library/ssl.html [36]

Python. WebSockets:
https://pypi.org/project/websockets/ [37]

14. Python: Complementary indications

14.1. Architecture

14.1.1. Virtual Environment

It is advisable to use a virtual environment in any Python project, which is equipped to separate application development in virtual environments.

A virtual environment isolates the Python interpreter, libraries and scripts installed in it. This means that instead of using a global version of Python and global Python dependencies for all the projects you want to develop, you can have specific virtual environments for each project and in each one you can have your own versions of Python, as well as its dependencies.

Virtual environments facilitate the development, packaging and delivery of secure Python applications. Most IDEs have built-in functions for switching between virtual environments.

A link to the Python library for creating **venv** virtual environments can be found below

https://docs.python.org/3/library/venv.html [38]

14.1.2. Importing packages

When working with external or internal Python modules, always make sure that you are importing in the correct way and using the correct paths. There are two types of import paths in Python, absolute and relative.

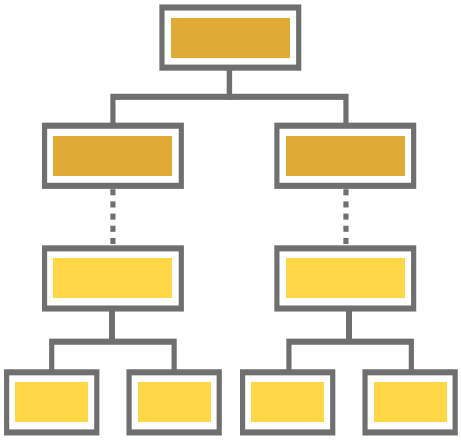
Absolute import specifies the path of the resource to import using its full path from the root folder of the project, while relative import specifies the resource to import relative to the current location in the project where the import statement is located.

```
/* Absolute Import */

from package1 import module1
from package1.module2 import function1


/* Relative Import */

from .some_module import some_class
from ..some_package import some_function
```



There are two (2) types of relative imports:

- **Implicit:** Implicit imports do not specify the path of the resource relative to the current module. Implicit import has been removed from Python 3, because if the specified module is in the system path, it will be imported and that could be very dangerous. It is possible that a malicious module with the same name is in a popular open-source library and finds its way into the system path. If the malicious module is found before the real module, it will be imported and could be used to exploit applications that have it in their dependency tree.

  Therefore, either absolute import or explicit relative import should be used, ensuring the import of the actual and the intended module.

```
from safe_module import package, function, class
o
from  ..relative_module import package, function, class
```

- **Explicit:** Explicit imports specify the exact path of the module to be imported in relation to the current module

14.2. Authentication

The Django *framework*, based on the Python language, has, in its default configuration, an **authentication system** and supports an extension and customisation of authentication. It provides both authentication and authorisation together. It handles user accounts, groups, permissions and user sessions based on *cookies*. This system consists of:

- **"Users" objects:** are the core of the authentication system. They typically represent the people who interact with the application and are used to allow things like restricting access, registering user profiles, associating content with creators, etc. There is only one kind of user in Django's authentication *framework*, i.e. "superusers" or "staff" admin users are just user's objects with special attributes set.

- **Permissions and Authorisation:** Django comes with a built-in permissions system. It provides a way to assign permissions to specific users and user groups.

- **Authentication on web requests:** Django uses sessions and middleware to hook the authentication system into request objects. These provide a **request.user** attribute on each request that represents the current user. If the current user is not logged in, this attribute will be an instance of **AnonymousUser**, otherwise it will be an instance of User.

- **User management in the admin panel:** when both **django.contrib.admin** and **django.contrib.auth** are installed, the admin panel provides a mode.

The Django *framework*, based on the Python language, has, in its default configuration, an authentication system and supports an extension and customisation of authentication

14.3. Session management

It is recommended use the default implementations of CSRF protections that exist in the vast majority of *frameworks*, such as the **Django framework**, which has CSRF middleware enabled by default, as well as the template tag and the **Flask framework**.

Python example with Django *framework*.

- **settings.py file:** leave the default middleware CSRF enabled in the settings file.

```
…
MIDDLEWARE = [
…
'django.middleware.csrf.CsrfViewMiddleware',
…
]
```

- **File template_example_csrf.html:** add the **csrf_token** tag inside the <form> element with POST method for an internal URL:

```
…
<form method="post">{% csrf_token %}
…
```

- **Views.py file:** **RequestContext** must be used in the rendering of the response for the **{% csrf %}** tag to work correctly. Note that if you use the **render()** function, generic views apps contrib, this is already covered as they all make use of **RequestContext:**.

```
…
def ejemplo_uso_request_context(request):
    …
return render(request, 'plantilla_ejemplo_csrf.html', {form: form, 'username': username'}))
```

14.4. Validation of input parameters

14.4.1. Parameterised database queries

The use of parameterised queries is recommended for any application access to the database.

Example of safe Python code with safe query parameter passing. The following code fragment defines the is_admin function that receives a string as input parameter and returns a boolean. If the user does not exist it returns False and if the user exists, it returns the value of the admin column which can be True if it is an administrator or False if it is not an administrator.

By parameterising the query and validating whether the query result is **None**, a SQL Injection vulnerability attack is prevented:

```
def is_admin(username: str) -> bool:
    with connection.cursor() as cursor:
        cursor.execute("SELECT admin FROM users WHERE username = %(username)s",{'username': username});
        result = cursor.fetchone()

    if result is None:
        # User does not exist
        return False

    admin = result
    return admin
```

```
>>> is_admin('leo')
False
>>> is_admin('irati')
True
>>> is_admin('lemon')
False
>>> is_admin("'; select true; --")
False
```

14.4.2. Protection of Forms

It is recommended to protect forms where there are parameters whose value can be modified by users. This is achieved with the following measures:

- **Properly encoding data output:** this mainly consists of applying HTML encoding to any output that reproduces data entered by the user at the input, so that it cannot be interpreted as code by the browser:
  - Indicate the type of the response content in its Content-type header (see **IANA media-types for supported types: application/json, text/html...), so that the client side knows how to interpret it.**[1] [39] for supported types: **application/json, text/html...**), so that the client side knows how to interpret it.
  - Include X-Content-Type-Options=nosniff header to avoid MIME-sniffing by some browsers, and therefore rendering content differently than declared in the Content-type.

- **Filter potentially dangerous meta-characters:** in vulnerable input. For example, the characters "<" ">" ";" "/" "/" "and all non-printable characters should be properly filtered out from the input in the application.

- **Apply form data filtering policies.**

  Example of secure Python code making use of the functions **html.escape** (converts the &, < and > characters of the string received as input parameter into HTML-safe sequences) and **html.unescape** (converts the) from the html library:

```
import html

bp1 = html.unescape('<html><head></head><body><h1>Best Practices</h1></body></html>')
bp2 = html.unescape('x > 2 && x < 7 single quote: \' double quote: "')
bp3 = html.escape('<html><head></head><body><h1>Best Practices</h1></body></html>')
bp4 = html.escape('x > 2 && x < 7 single quote: \' double quote: "')
print("UNESCAPE")
print(bp1)
print(bp2)
print("ESCAPE")
print(bp3)
print(bp4)
```

```
UNESCAPE
<html><head></head><body><h1>Best Practices</h1></body></html>
x > 2 && x < 7 single quote: ' double quote: "
ESCAPE
&lt;html&gt;&lt;head&gt;&lt;/head&gt;&lt;body&gt;&lt;h1&gt;Best Practices&lt;/h1&gt;&lt;/body&gt;&lt;/html&gt;
x &gt; 2 &amp;&amp; x &lt; 7 single quote: &#x27; double quote: &quot;
```

1 Media Types https://www.iana.org/assignments/media-types/media-types.xhtml

- **Session *cookies*** marked with the **HttpOnly** *flag* to prevent session theft in case of exploiting an XSS vulnerability.

- **Implement all recommendations on the server side.**

- **Validation of Inputs.** It is recommended to always use a whitelist or blacklist approach, with the whitelist being the most recommended approach where anything that does not fit the specifications is rejected.

  These checks must be done on both the client-side and server-side, at a minimum on the server-side to ensure that the business logic treats the data securely **(CWE-602)**[2] [40].

  On the other hand, it is also necessary in any typed language to transform the data types into the expected type, for example, if a **String** is received and an Int is expected, to perform a casting or transformation.

Example of secure Python whitelisted code:

```
if tipo_seguro == "opciónA" or tipo_seguro == "opciónB":
    print("Sí soportado")
else:
    print("No soportado")
```

Example of safe Python code regular expression checking:

```
import re

pat = re.compile(r"[A-Za-z0-9]+")
test = input("Introduzca cadena: ")

if re.fullmatch(pat, test):
    print(f"'(test)' Sí es cadena alfanumérica")
else:
    print(f"'(test)' No es cadena alfanumérica")
```

```
>Introduzca cadena : Alfanumerica1
  ' Ejemplo cadena alfanumérica ' Sí es cadena alfanumérica.
>Introduzca cadena : Alfanumerica 2
  ' Ejemplo cadena alfanumérica ' No es cadena alfanumérica.
```

```
#Integer
x = int(1)   # x es 1
y = int(2.8) # y es 2


z = int("3") # z es 3
v = "6"
if not isinstance(v,int):
    print("No es tipo int")
    u = int(v)     # u es 6

#Float
x = float(1)     # x es 1.0
y = float(2.8)   # y es 2.8
z = float("3")   # z es 3.0
w = float("4.2") # w es 4.2
v = "7"
if not isinstance(v,float):
    print("No es tipo float")
    u = float(v)     # u es 7.0

#String
x = str("s1") # x es 's1'
y = str(2)    # y es '2'
z = str(3.0)  # z es '3.0'
v = 2
if not isinstance(v,str):
    print("No es tipo string")
    u = str(v)     # u es '2'
```

Example of Python safe code type checking and casting.

2 CWE-602 https://cwe.mitre.org/data/definitions/602.html

- **Avoid misuse of user input.** The use of user input for system calls or to provide file parts should be avoided. Here is a Python example of how you can restrict access to files within a specific directory:

```
import os
import sys

def is_safe_path(basedir, path, follow_symlinks=True):
    # resolves symbolic links
    if follow_symlinks:
        matchpath = os.path.realpath(path)
    else:
        matchpath = os.path.abspath(path)
    return basedir == os.path.commonpath((basedir, matchpath))

def main(args):
    for arg in args:
        if is_safe_path(os.getcwd(), arg):
            print("safe: {}".format(arg))
        else:
            print("unsafe: {}".format(arg))

if __name__ == "__main__":
    main(sys.argv[1:])
```

- **Prevent XEE.** Always disable the resolution of external DTDs, so that only statically defined local DTDs are used. It is also recommended to always validate the structure of the XML document provided by the user, using the server's statically defined format definition DTD file as a basis.

  Example of a secure Python XML parser library: **defusedxml**[3] [41] is a pure Python package with modified subclasses of all stdlib XML parsers that prevent any potentially malicious operation. The use of this package is recommended for any server code that parses untrusted XML data. The package also includes example attacks and extended documentation on more XML vulnerabilities, such as XPath injection..

  The **defusedxml** library prevents XEE attacks because it does not allow the use of XML with **<!ENTITY>** declarations inside the DTD and throws the **EntitiesForbidden** exception when an entity is declared. On the other hand, it does not allow any remote or local resource access in external entities or DTD and raises the External **ReferenceForbidden** exception when a DTD or entity references an external resource.

  Python example with the defusedxml library:

```
from defusedxml import pulldom
data = parse('ejemplo.xml')
```

- **Standardisation.** Use Python's native functions for the standardisation of the charset of all the information processed by the system.

  Example: **codecs** library[4] [42]:

```
codecs.encode(obj, encoding='utf-8', errors='strict')
codecs.decode(obj, encoding='utf-8', errors='strict')
```

3 XML parser https://pypi.org/project/defusedxml/

4 Standardisation Codecs https://docs.python.org/3/library/codecs.html

- **Data sanitisation.** Some of the most common special character set languages and recommended packages for sanitisation are:
  - HTML/URL: **html-sanitizer**[5] [43]
  - XML: **EscapingXML**[6] [44]
  - JSON: **JsonSchema**[7] [45] implementation of the JSON Schema specification for Python.

  Most *frameworks* come with sanitization functions: **Flask**[8] [46] and **Django**[9] [47]

- **String formatting.** Python has one of the most powerful and flexible methods for formatting strings and if not used properly, it could end up opening up a security vulnerability in the code. Python3 introduced **f-strings**[10] [48] and **str.format()**[11] [49] as a flexible way of formatting strings and it is really very interesting.

  However, this opens a loophole for data exploitation when it comes to user input. If the application built in Python allows users to control the format string, they can be misused to leak sensitive data. For example:

```
CONFIG = {
    "API_KEY": "secret_key"
}
class User:
    name = ""
    email = ""
    def __init__(self, name, email):
        self.name = name
        self.email = email
    def __str__(self):
        return self.name

name = "Nombre"
email = "micorreo@gmail.com"
user = User(name, email)
print(f"{user.__init__.__globals__['CONFIG']['API_KEY']}")
/* secret_key */
```

  With this, sensitive global data of a CONFIG dictionary can be accessed via the argument.

  However, Python has a built-in string module that can be used to fix and prevent this. Using the Template class of the string module:

```
from string import Template

nombre_plantilla = Plantilla("Hola, mi nombre es $nombre.")
greeting = name_template.substitute(name="Python")
/* Hola, me llamo Python */
```

5 HTML Sanitisation https://pypi.org/project/html-sanitizer/
6 XML Sanitisation https://wiki.python.org/moin/EscapingXml
7 JSON Schema Sanitisation https://python-jsonschema.readthedocs.io/en/latest/
8 Flask Sanitisation https://flask.palletsprojects.com/en/2.0.x/api/#flask.escape
9 Django Sanitisation https://docs.djangoproject.com/en/4.0/_modules/django/utils/html/
10 Formatting Chains I https://docs.python.org/3/tutorial/inputoutput.html#formatted-string-literals
11 Chain Formatting II https://docs.python.org/3/library/stdtypes.html#str.format

15. Checklist security controls

ARCHITECTURE	ID	SECURITY CONTROL	DESCRIPTION
	ARQ-01	**Identify components**	Components that have not been correctly identified are potential security risks
	ARQ-02	**Bastioning components**	Ensure that the configurations are as secure as possible: there are not debugging options enabled, nor default users and passwords, etc. Ensure that only those communication ports that are strictly necessary are open. Status of the latest system update. Identification of all system components: Libraries, Modules, Frameworks, Services, etc. For each system component, perform the same baseline review of configurations and update status.
	ARQ-03	**Analysing risks**	Obtain a report of the components in which there are vulnerabilities detected for which there is currently no security patch and analyse their level of risk within the application.
	ARQ-04	**Monitoring updates**	Keep a close watch on vulnerable components so that they are updated as soon as possible.
	ARQ-05	**Alternative mitigations**	Conduct a study of how the security problems created by these risk vulnerabilities could be avoided or mitigated by alternative security systems.

[ARCHITECTURE]	ARQ-06	**Logical perimeter security**	By installing firewalls, IDS or similar devices, or by segmenting the network.
	ARQ-07	**Securing sensitive data**	Ensure that data are protected by authorisation mechanisms between environments through physical or logical segregation, and by backups to ensure their availability.
	ARQ-08	**Programming language security**	Use the most recent version of the programming language. Use a Virtual Environment as a project workspace if applicable according to the programming language. Correct import of packages according to programming language, thoroughly checking the security of the packages to be installed.
	ARQ-09	**Disable debugging options**	Especially in Production to avoid information leakage in the detailed error messages.
	ARQ-10	**Development environment**	Use tools in the IDE that perform basic semantic and security analysis.

AUTHENTICATION	ID	SECURITY CONTROL	DESCRIPTION
	AUT-01	***Hash* for passwords**	Ensure that passwords are not stored in a readable format, so that if the system or resource containing the passwords is compromised, the malicious user is still unable to use them.
	AUT-02	**Disconnect button**	Each page of the application has a logout link, that the session expires when the user logs out, and that the session expires when a reasonable amount of time of non-activity has passed.
	AUT-03	**Do not expose credentials**	Never expose credentials in the URL.
	AUT-04	**Use POST**	When using forms, use POST methods for sending information between the client and the server.

[AUTHENTICATION]	AUT-05	**Use multi-factor authentication**	Use multi-factor authentication to implement multiple layers of security for sensitive applications. Use two-factor authentication to the user for critical functions in the application, such as changing passwords or accessing particularly sensitive resources.
	AUT-06	**Blocking of accounts**	Implement account locking after 3 failed *login* attempts and a way to contact the administrator to unlock the account.
	AUT-07	**Use of CAPTCHA**	Implement CAPTCHA to mitigate brute force attacks on applications exposed on the internet.
	AUT-08	**Prevent user enumerations**	Providing generic error messages in case of authentication failure, such as "The user and/or password provided are not correct". For providing different response times in case of non-existent user or incorrect password. In password recovery procedures that require entering the username.
	AUT-09	**Disable "autocomplete".**	Disable the "auto-complete" attribute of the password field in the application, as it is enabled by default.
	AUT-10	**Password complexity requirements**	It must have a minimum of eight characters and a sensible maximum. It must contain at least three of the following characters: one upper case letter, one lower case letter, one number, one special character.
	AUT-11	**Do not store *Cookie***	Not to persistently store the authentication *cookie* on the customer's computer, and not to use it for other purposes such as personalisation.
	AUT-12	**Register access**	Record all accesses in a specific security log stating the outcome of the access attempt (whether it was successful or unsuccessful, cancelled or blocked).

AUTORIZACIÓN	ID	SECURITY CONTROL	DESCRIPTION
	ATZ-01	**Minimal privilege**	Ensure the implementation of the principle of least privilege: Users have restricted access only to the functions and data they really need to perform their work normally.
	ATZ-02	**Roles and privileges**	Assign permissions and privileges to application roles, never directly to users. Users must have roles and their privileges are taken from these roles.
	ATZ-03	**Protection by authorisation**	Check that access to confidential data is protected, so that only authorised objects or data accessible to each user can be reached.
	ATZ-04	**Directory browsing disabled**	Verify that directory browsing is disabled, unless it is deliberately enabled.
	ATZ-05	**Access control on the server**	Ensure that access control rules are applied on the server side.
	ATZ-06	**Safe Handling of User Information**	Verify that all user attributes, data and policy information used by access controls cannot be manipulated by end-users unless specifically authorised.
	ATZ-07	**Safe handling of resources**	Verify that there is a centralised mechanism (including libraries that call external authorisation services) to protect access to each type of protected resource.
	ATZ-08	**Anti-CSRF *tokens***	The application uses strong anti-CSRF random *tokens* or implements another transaction protection mechanism.
	ATZ-09	**Register access control**	Record all operations on sensitive data in a specific security log even if they have been denied.

SESSION MANAGEMENT	ID	SECURITY CONTROL	DESCRIPTION
	SES-01	**Secure headers**	Implementing secure headers with directives such as **cache-control** or **strict-transport security**
	SES-02	**Session invalidation**	Check that sessions are invalidated when the user logs out.
	SES-03	**Session expiry**	Sessions must expire after a specified time of inactivity.
	SES-04	**Logging out**	Ensure that all pages requiring authentication have easy and user-friendly access to the logout functionality.
	SES-05	**Do not expose session ID**	Verify that the session ID never appears in URLs, error messages or logs.
	SES-06	**New session IDs**	Ensure that every successful authentication and re-authentication generates a new session and a new session ID, destroying the old one.
	SES-07	**Session *Cookie***	Check that the session ID stored in the *cookies* is defined using the **HttpOnly** and **Secure** attributes. Properly configure the **Path** attribute of session *cookies* to prevent access to other domains.
	SES-08	**Follow-up sessions**	Verify that the application keeps track of all active sessions and allows the user to terminate sessions selectively or globally from their account. In the case of high-value applications, ensure that the user is required to close all active sessions if the password has just been successfully changed.
	SES-09	**Record session activity**	Record in a security log all sessions that are created, manually closed or expired from the client or from the server.

VALIDATION OF INPUT AND OUTPUT DATA	ID	SECURITY CONTROL	DESCRIPTION
	VAL-01	Validate on the server side	Validations must always be performed on the server side.
	VAL-02	Validation schemes	Use data validation schemes as the best solution to validate data entry. Failing this, use standard input validation mechanisms provided by technology-specific libraries.
	VAL-03	Data typification	Check that structured data is strongly typed and validated according to a defined schema, including allowed characters, length and pattern.
	VAL-04	Data sanitisation	Verify that unstructured data is sanitised to impose generic security measures, such as allowed characters and length, and avoid potentially harmful characters. Ensure that all unreliable input is properly sanitised using a sanitisation library.
	VAL-05	Accept only expected data	Accept only the expected data at each entry point of the application. Any unexpected data must be rejected.
	VAL-06	SQL injection protection	Ensure that all database queries are protected using parameterised queries to prevent SQL injection.
	VAL-07	Correct encoding of HTML variables	To ensure that the application is not susceptible to Cross-Site Scripting (XSS) reflected, stored or DOM
	VAL-08	XML parser constraints	Verify that the application restricts XML parsers to use only the most restrictive settings possible and ensure that dangerous functions, such as external entity resolution, are disabled.
	VAL-09	Deserialisation of data	Verify that deserialisation of unreliable data is prevented or largely protected when deserialisation cannot be avoided.

ERROR MANAGEMENT	ID	SECURITY CONTROL	DESCRIPTION
	ERR-01	Exposure of information in errors	Use generic error messages that do not give clues to end-users about any sensitive aspects of the application.
	ERR-02	Centralisation of errors	Use centralised exception handling. The application must handle errors without relying on server error messages displayed to users.
	ERR-03	Default refusal in case of error	Any access control logic that leads to an error must deny access by default.
	ERR-04	Controlled errors	Analyse in detail all exceptions that may occur due to the use of system libraries or third-party libraries in the application, handle them appropriately and provide a safe exit to the application. Use **try/catch/finally** to ensure shutdown of all resources in case of error.
	ERR-05	Recording errors	Log all unexpected exceptions in a specific security log.

SECURE REGISTRATION	ID	SECURITY CONTROL	DESCRIPTION
	LOG-01	Do not record sensitive information	Such as passwords, financial information, credit cards, personal data, etc. In these cases, use *tokens*, anonymisation of information or encryption.
	LOG-02	Validate record variables	Validate the parameters of the variable components that will make up the log entry to avoid injections and unexpected behaviour.
	LOG-03	Precise notes	Make sufficiently precise notes that allow the user's operations and activities to be known for safety.
	LOG-04	Integrity of the register	Use *hash* functions to ensure data integrity of records.
	LOG-05	Isolation of the security log	The security register must be independent of other registers and have its own security protections in the system.

CRYPTOGRAPHY	ID	SECURITY CONTROL	DESCRIPTION
	CRT-01	**Encryption of sensitive information**	All sensitive information of an organisation must be stored in an unreadable (encrypted) form to ensure its confidentiality.
	CRT-02	**Secure random generator**	Check that all random numbers, random file names, random GUIDs and random strings are generated by a random number generator approved by the cryptographic module. Check that random numbers are created with an appropriate level of entropy, even when the application is under high load.
	CRT-03	**Key management policy**	Check that there is an explicit policy on the handling of cryptographic keys (such as generation, distribution, revocation and deprecation). Check that the lifecycle of cryptographic keys is correctly implemented.
	CRT-04	**Release of sensitive information**	Ensure that confidential passwords or critical information residing in memory are overwritten with zeros as soon as they are no longer used to mitigate memory dump attacks.
	CRT-05	**Use of robust algorithms**	Check that obsolete or weak cryptographic algorithms such as the symmetric key DES algorithm or *hash* functions such as MD5 or SHA-1 are not used due to the impossibility to guarantee confidentiality. Use existing cryptographic libraries and in any case use custom or user-created cryptographic algorithms or implementations.

SECURE FILES	ID	SECURITY CONTROL	DESCRIPTION
	FIL-01	Download authorisation	Authenticate and authorise the user before uploading or downloading any files, especially if the data is sensitive.
	FIL-02	Do not allow renaming	Do not use user-supplied input to name files or directories.
	FIL-03	Validate MIME types	Validate content types not only by extension, but also check MIME types to verify files. Do not allow executable files to be uploaded to the application.
	FIL-04	Limit file size	Limit file sizes to the minimum that the server can handle without causing availability problems and impacting application functionality.
	FIL-05	Scanning files	Before processing the files to the server, an anti-virus scanner checks the files for malware or viruses.
	FIL-06	Directory privileges	Disable execution privileges on directories where users can upload files. Do not use absolute paths when providing a download link to the user.
	FIL-07	Secure file names	Do not store files with their names sequentially. Do not use sensitive information for file naming.
	FIL-08	Read only	Ensure that the access control is set to read-only.
	FIL-09	File limit	Limit the number of files uploaded by the user.
	FIL-10	File integrity	Store hashes of uploaded files to ensure their integrity.

TRANSACTION SECURITY	ID	SECURITY CONTROL	DESCRIPTION
	TRN-01	**Payment Bypass Protection**	The authorisation and confirmation of a purchase must be done on the server side. It is necessary to validate that the signatures used are correct during the communication process with the payment gateway. Validate that the price is correctly set on the server side. Validate that payments are not reused. The payment server must always check at which stage of the transaction you are in.
	TRN-02	**Transaction expiry time**	Include a relatively short authorisation expiry time for each transaction.
	TRN-03	**Transaction traceability**	Ensure traceability of transactions.
	TRN-04	**Encrypt communication between transactions**	Encrypt communications with robust asymmetric algorithms.
	TRN-05	**Record transaction activity**	Detailed recording of all transactions with anonymisation of sensitive information.

COMMUNICATIONS SECURITY	ID	SECURITY CONTROL	DESCRIPTION
	COM-01	**Always encrypt communication channels**	TLS, a cryptographic protocol for encrypting communication channels. WebSocket, a technology that provides a bi-directional communication channel over the same TCP socket.
	COM-02	**Use strong cryptography**	Strong enough to make any attempt at deciphering futile.
	COM-03	**Use secure protocols**	Uses security protocols: such as HTTPS, or FTPS

DATA PROTECTION	ID	SECURITY CONTROL	DESCRIPTION
	DAT-01	Data protection policy	Check that an explicit data protection policy is in place.
	DAT-02	Sending sensitive data	Ensure that all sensitive data is sent to the server in the HTTP message body or headers. Check that the communication channels used for sending confidential data are secure. Check that the application sets sufficient headers against caching.
	DAT-03	Securely stored data	Check that stored sensitive data is encrypted with strong encryption algorithms. Check that backup policies are in place for data availability.
	DAT-04	Release of sensitive data	Ensure that sensitive information stored in memory is overwritten with zeros as soon as it is not needed, to mitigate memory dump attacks.
	DAT-05	Deletion of data	There is a policy of secure deletion of data when assets have reached the end of their life cycle.

16. Security vulnerabilities and controls

It is important to be aware of the most common vulnerabilities located in the code in order to know how they should be addressed. These vulnerabilities have been cross-referenced with the security controls in the previous chapters

VULNERABILITY	SECURITY CONTROL
Insecure Communication	Authentication
Enumeration of Usernames	
Weak Password	
Cross-Site Request Forgery - Cross-Site Request Forgery (CSRF)	Authorisation
Identification and Authentication Failures	
Direct Access to Objects	
Access Control	
Cross-Site Scripting - Cross-Site Scripting (XSS)	Data Validation
SQL Injection	
Buffer overflow	
Falsification of Records	
Dynamic SQL	
Open Redirect Vulnerability	
Output Coding	
Disclosure of Information	
Weak Session Management	Session Management
Forms Cache	
Information Disclosure	Error Handling
Information Disclosure	Register
Registration does not exist for Critical Functions	
Storage of Sensitive Information in Unencrypted Text	Cryptography
Weak Cryptography	
Insecure File Upload	Secure File Management
Information Disclosure	

17. Security measures and security controls

SECURITY MEASURES		SAFE DEVELOPMENT GUIDE
op.exp	**Exploitation**	**Architecture**
op.exp.4.1, op.exp.4.2, op.exp.7.r4.1		SECURITY RECOMMENDATIONS
mp.com	**Protection of communications**	
mp.com.1.1, mp.com.4		
op.acc	**Access control**	**Authentication**
op.acc.5.r3.2, op.acc.5.8, op.acc.5.r1.2, op.acc.5.r6.1		SECURITY RECOMMENDATIONS
mp.sw	**Protection of applications**	**Authorisation**
mp.sw.1.r1.1		SECURITY RECOMMENDATIONS
mp.s	**Protection of services**	
mp.s.2.1		
op.pl	**Planning**	
op.pl.2.4		
op.mon	**System monitoring**	
op.mon.1.r2.1		
op.acc	**Access control**	
op.acc.4.1, op.acc.6.r5.1		

mp.eq	**Protection of equipment**	**Session Management**
mp.eq.2.r1.1		RECOMENDACIONES DE SEGURIDAD
op.pl	**Data validation**	**Validación de Parámetro**
op.pl.2.r3.1		RECOMENDACIONES DE SEGURIDAD
mp.s	**Protection of services**	
mp.s.2.3		
op.exp	**Exploitation**	**Registro**
op.exp.5.1, op.exp.7.r2.1, op.exp.8.2		RECOMENDACIONES DE SEGURIDAD
mp.s	**Protection of services**	
mp.s.3.r1.1		
op.acc	**Access control**	
op.acc.6.r9.2		
mp.si	**Protection of information media**	**Criptografía**
mp.si.2.1		RECOMENDACIONES DE SEGURIDAD
op.exp	**Exploitation**	
op.exp.10		
op.exp	**Exploitation**	**Secure File Management**
op.exp.6.3		RECOMENDACIONES DE SEGURIDAD
mp.com	**Protection of communications**	**Communications Security**
mp.com.2.r5.1		RECOMENDACIONES DE SEGURIDAD
mp.info	**Protection of information**	**Data Protection**
mp.info.1.1, mp.info.2		RECOMENDACIONES DE SEGURIDAD

18. Glossary

OWASP (Open Web Application Security Project): is an open source project dedicated to identifying and combating the causes that make software insecure. The OWASP Foundation is a non-profit organisation that supports and manages the OWASP projects and infrastructure.

The OWASP community is made up of companies, educational organisations and individuals from around the world. Together they form a computer security community that works to create articles, methodologies, documentation, tools and technologies that are released and can be used free of charge by anyone.

***Cookie*:** a small file sent by a website and stored in the user's browser, so that the website can consult the browser's previous activity. In this way, it is possible to identify the user visiting a website and to keep a record of their activity on the website.

CSRF (Cross Site Request Forgery): is a cross-site request forgery vulnerability. It involves tricking a legitimate user into executing requests or actions without their consent, without knowing what they are doing.

MFA (Multi-Factor Authentication): is a method of access control in which a user is granted access to the system only after he/she provides two or more different proofs that he/she is who he/she claims to be.

XEE/JEE (Xml/Json External Entity): is a code injection vulnerability in an application that parses XML/Json data.

DTD (Document Type Definition): is a definition in an SGML or XML document, which specifies restrictions on the structure and syntax of the document.

CWE: is a community-developed list of types of software and hardware weaknesses. It serves as a common language, a yardstick for security tools, and as a baseline for weakness identification, mitigation and prevention efforts.

19. References

[1] "Design Patterns," [Online]. Available: https://refactoring.guru/es/design-patterns

[2] "OWASP: Design Secure Web Applications," [Online]. Available: https://owasp.org/www-pdf-archive/APAC13_Ashish_Rao.pdf

[3] Ámbitos de la Seguridad Nacional: Protección de Infraestructuras Críticas," [Online]. Available: file:///Users/lagor/Downloads/BOE-400_Ambitos_de_la_Seguridad_Nacional_Proteccion_de_Infraestructuras_Criticas.pdf

[4] "Java Secure Authentication," [Online]. Available: https://docs.oracle.com/javaee/5/tutorial/doc/bncbe.html#bncbn

[5] "Python: Library for implementing OTP," [Online]. Available: https://pypi.org/project/pyotp/

[6] "Java: Secure Authorization," [Online]. Available: https://docs.oracle.com/en/java/javase/19/security/java-authentication-and-authorization-service-jaas1.html

[7] "Python: Simple authorization secure library," [Online]. Available: https://pypi.org/project/python-authorization/

[8] "OWASP: Secure Session Management," [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

[9] "Java: SpringSession Framework," [Online]. Available: https://www.baeldung.com/spring-session

[10] "OWASP: Parameter Validation," [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html

[11] "OWASP: Data Validation - ESAPI Library," [Online]. Available: https://owasp.org/www-project-enterprise-security-api/

[12] "Validating Forms in Javascript," [Online]. Available: https://www.tutorialspoint.com/javascript_form_validation_web_application/index.asp

[13] "Java: Spring Boot Validation," [Online]. Available: https://www.baeldung.com/spring-boot-bean-validation

[14] "XSS Prevention," [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

[15] "Colander Validation," [Online]. Available: https://pypi.org/project/colander/

[16] "Cerberus Validation," [Online]. Available: https://docs.python-cerberus.org/en/stable/

[17] "Validation Schematics," [Online]. Available: https://schematics.readthedocs.io/en/latest/

[18] "Schema Validation," [Online]. Available: https://pypi.org/project/schema/

[19] "JSON Schema Validation," [Online]. Available: https://pypi.org/project/jsonschema/

[20] "Java: Try - Catch - Finally," [Online]. Available: https://www.w3schools.com/java/java_try_catch.asp

[21] "Java: Exception Handling," [Online]. Available: https://www.baeldung.com/java-exceptions

[22] "Error Handling Strategies," [Online]. Available: https://dzone.com/articles/error-handling-strategies

[23] "Python: Errors and Exceptions," [Online]. Available: https://docs.python.org/es/3/tutorial/errors.html#errors-and-exceptions

[24] "Security Log: Best Practices for Logging and Management," [Online]. Available: https://www.dnsstuff.com/security-log-best-practices

[25] "java: Logging," [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/guides/logging/index.html

[26] "OWASP: Secure Cryptography in Java," [Online]. Available: https://www.owasp.org/index.php/Using_the_Java_Cryptographic_Extensions

[27] "Scrypt key derivation function," [Online]. Available: https://www.tarsnap.com/scrypt.html

[28] "Python: Cryptographic functions bcrypt," [Online]. Available: https://pypi.org/project/bcrypt/

[29] "Apache Server: Configuration Directives," [Online]. Available: https://httpd.apache.org/docs/2.4/mod/core.html#limitrequestbody

[30] "Java: Canonisation of Path," [Online]. Available: https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/io/File.html#getCanonicalPath()

[31] "PHP: Canonisation of Path," [Online]. Available: https://www.php.net/manual/es/function.realpath.php

[32] "Python: Mapping files to their MimeTypes," [Online]. Available: https://docs.python.org/3/library/mimetypes.html

[33] "WordPress: Plugin to avoid Payment ByPass vulnerability," [Online]. Available: https://www.acunetix.com/vulnerabilities/web/wordpress-plugin-nab-transact-security-bypass-2-1-0/

[34] "SSL/TLS Algorithms," [Online]. Available: https://docs.oracle.com/en/java/javase/15/docs/specs/security/standard-names.html#sslcontext-algorithms

[35] "Java: HttpClient with SSL," [Online]. Available: https://www.baeldung.com/java-httpclient-ssl

[36] "Python: SSL/TLS," [Online]. Available: https://docs.python.org/3/library/ssl.html

[37] "Python: WebSockets," [Online]. Available: https://pypi.org/project/websockets/

[38] "Virtual Environment," [Online]. Available: https://docs.python.org/3/library/venv.html

[39] "Media Types," [Online]. Available: https://www.iana.org/assignments/media-types/media-types.xhtml

[40] "CWE-602," [Online]. Available: https://cwe.mitre.org/data/definitions/602.html

[41] "XML parser," [Online]. Available: https://pypi.org/project/defusedxml/

[42] "Codecs Standardisation," [Online]. Available: https://docs.python.org/3/library/codecs.html

[43] "HTML Sanitization," [Online]. Available: https://pypi.org/project/html-sanitizer/

[44] "XML Sanitisation," [Online]. Available: https://wiki.python.org/moin/EscapingXml

[45] "Sanitisation JSON Schema," [Online]. Available: https://python-jsonschema.readthedocs.io/en/latest/

[46] "Flask Sanitization," [Online]. Available: https://flask.palletsprojects.com/en/2.0.x/api/#flask.escape

[47] "Django Sanitisation," [Online]. Available: https://docs.djangoproject.com/en/4.0/_modules/django/utils/html/

[48] "String Formatting I," [Online]. Available: https://docs.python.org/3/tutorial/inputoutput.html#formatted-string-literals

[49] "String Formatting II," [Online]. Available: https://docs.python.org/3/library/stdtypes.html#str.format

ANNEX A. Basic cheatsheet

PRINCIPIOS DE DESARROLLO SEGURO		*CHECKLIST* CONTROLES DE SEGURIDAD	
Minimum Privilege	An actor must have the minimum level of permissions on system resources for the minimum possible time. Access to resources should be denied by default.	**Authorisation System**	To split an application into an authorisation system from the outset, preventing any unauthorised agent from accessing the entire application. E.g., RBAC system.
Separation of Responsibilities	Any complex or sensitive task should require the involvement of more than one actor with different role levels. This prevents a single actor from compromising the whole system.	**Parameterised**	Use of parameterised queries for any application access to the database.
Defence in Depth	Establish multiple layered security mechanisms, with different levels of complexity and control factors. The aim is to avoid the "Single Point of Failure".	**Protection of Forms**	Protection of forms where there are parameters whose value can be modified by users. This protection consists of properly encrypting the data output, filtering potentially dangerous meta-characters in vulnerable entries and enforcing form data filtering policies.
Failure Sure	In the event of a failure, the system must be returned to a secure state, minimising compromise to the confidentiality, integrity or availability of the system.	**Validation of Inputs**	Validation of any input area and in an efficient way including data from the internet, customers, suppliers and regulators.
Economics of Mechanisms	The implementation of a system's functionalities and security controls should be as simple as possible. Simpler means that fewer things can go wrong.	**Secure Transaction Method**	• The authorisation and confirmation of a purchase must be done on the server side. • Validate that the signatures used are correct during the communication process with the payment gateway. • Validate that the price is correctly set on the server side. • Validate that payments are not reused. • The payment server must always check at which stage of the transaction you are in. • The authorisation of each transaction should have a relatively short expiry period.
Full Mediation	Any request for access to system resources must be validated so that authentication and authorisation controls cannot be bypassed.	**Use of MFA**	Implementation of a Multiple Factor Authentication for sensitive data handling applications exposed to the internet and privileged user access.

Open Design	Shared resources between different users of the system should be restricted to a minimum, for confidentiality and concurrency reasons.	**Protection of Sensitive Data**	Perform a series of actions from the beginning of the development phase of an application, such as the following: • Identify the data to be processed that are sensitive with respect to regulatory privacy requirements. • Apply controls such as encryption in transit or at rest. • Do not store sensitive data unnecessarily. • Disable caching of sensitive data. • Have encrypted data at rest. • All data traffic must be encrypted with methods such as TLS. • Store passwords using strong hashing algorithms with a duty factor that slows down possible password cracking.
Least Common Mechanism	The impact of security controls on the usability of the system should be considered. Ideally, security controls should be transparent to the user.	**CSRF protections**	Include default implementation of CSRF protections in the *framework* in use.
Psychological Acceptability	The main point of compromise of a system must be identified and the necessary security controls implemented to protect it. A system is only as secure as its weakest link.	**Prevent XEE/JEE**	Disable resolution of external DTDs and validate XML document structure.
Weakest Link	It is advisable to encourage the reuse of well-proven components and established solutions in the system architecture.	**Avoid use of certain inputs**	Avoid using user input for system calls or to provide file parts.
Leveraging Existing Components	Es recomendable fomentar la reutilización de componentes bien probados y soluciones consolidadas en la arquitectura del sistema.	**Use updated version of the language**	Use the most recent version of the
For more information on the Secure Development Principles see CCN-CERT - Workshop - Practical Approach to Secure Application Development v 1.0.		**Use Virtual Environment**	Use of a Virtual Environment as a project workspace if applicable according to the programming language.
		Correct import of packages	Perform import path according to programming language.
		Use Formatting Secure Chains	Formatting user input strings in a way that does not allow control of the format string and prevents filtering of sensitive data.
		Secure Use of HTTP Requests	Handling HTTP requests securely by avoiding requests to exploited sources that may return exploited code in the headers or in the body of the response.
		Installed and imported packages Secure	Thorough security check of the packages to be installed.
		Secure Data Deserialisation	Make use of deserialisation library functions that prevent attack vectors.
		Keeping Vulnerabilities Up to Date	Maintain up-to-date Open-Source Vulnerabilities in installed and imported Packages.
		Disable Debugging in Production	Set Production Debugging to False to avoid information leakage in the detailed error messages.
		Code scanning	Use tools in the IDE that perform semantic analysis.

ANNEX B. Advanced cheatsheet

PRINCIPLES OF SECURE DEVELOPMENT	CHECKLIST SECURITY CONTROLS	REFERENCE ENTITIES		SECURITY CONTROLS MAPPING	
Minimum Privilege	Authorisation System	**ISO**	Promotes safety, clarity and reliability of products and for various industries by publishing globally applicable standards.	**Insecure Communication**	Authentication
Separation of Responsibilities	Parameterised	**NIST**	Physical Sciences Laboratory and a non-regulatory agency of the US Department of Commerce.	**Enumeration of Usernames**	
Defence in Depth	Protection of Forms	**OWASP**	Online community that provides free articles, methodologies, documentation, tools and technologies in the field of web application security.	**Weak Password**	
Failure Sure	Validation of Inputs	**MITRE**	Organisation that provides systems engineering, research, development and IT support to the US government.	**Cross-Site Request Forgery Cross-Site Request Forgery (CSRF)**	Authorisation
Economics of Mechanisms	Secure Transaction Method	**PROACTIVE CONTROLS**	For more information on the Secure Development Principles see CCN-CERT - Workshop - Practical Approach to Secure Application Development v 1.0	**Identification and Authentication Failure**	
Full Mediation	Use of MFA	**Define security requirements**		**Direct Access to Objects**	
Open Design	Protection of Sensitive Data	**Leveraging security *frameworks* and libraries**		**Access Control**	
Least Common Mechanism	CSRF protections	**Secure database access**		**Cross Site Scripting (XSS) Cross Site Scripting (XSS)**	Data Validation
Psychological Acceptability	Prevent XEE/JEE	**Coding and escaping data**		**SQL Injection**	
Weakest Link	Avoid use of certain inputs	**Validate all entries**		**Buffer overflow**	
Leveraging Existing Components	Use updated version of the language	**Implementing digital identity**		**Falsification of Records**	
	Use Virtual Environment	**Strengthening access controls**		**Dynamic SQL**	

Correct import of packages	**Protect data continuously**		**Open Redirect Vulnerability**	
Use Formatting Secure Chains	**Implement security monitoring**		**Output Coding**	
Secure Use of HTTP Requests	**Handle all errors and exceptions**		**Disclosure of information**	
Installed and imported packages Secure			**Weak Session Management**	Session Management
Secure Data Deserialisation			**Forms Cache**	
Keeping Vulnerabilities Up-to-Date			**Information Disclosure**	Error Handling
Disable Debugging in Production			**Information Disclosure**	Register
Code scanning			**Registration does not exist for Critical Functions**	
			Sensitive Information Storage in Unencrypted Text	Cryptography
			Weak Cryptography	
			Insecure File Upload	Secure File Management
			Information Disclosure	

centro criptológico nacional

centro criptológico nacional

www.ccn.cni.es

www.ccn-cert.cni.es

oc.ccn.cni.es


ccn-cert
centro criptológico nacional
CCN
centro criptológico nacional


PUBLICACIONES
Pd
DE DEFENSA


GOBIERNO
DE ESPAÑA
MINISTERIO
DE DEFENSA
SUBSECRETARÍA DE DEFENSA
SECRETARÍA GENERAL TÉCNICA
SUBDIRECCIÓN GENERAL
DE PUBLICACIONES
Y PATRIMONIO CULTURAL