

# Criptografía y seguridad informática

## **Entregable 2: Sistema de votación online**

### **Grupo 84 - id grupo de prácticas 12**

Repositorio de la práctica:

(<https://github.com/UC3MLP/criptoprojecto>)

*Hecho por :*

- 100522167@alumnos.uc3m.es - Laura Parrilla Ruiz
- 100522170@alumnos.uc3m.es - Julio Ruiz Sánchez

# ÍNDICE:

a. ¿Cuál es el propósito de su aplicación? ¿Cuál es su estructura interna?	3
b. ¿Para qué utiliza la firma digital? ¿Qué algoritmos ha utilizado y por qué? ¿Cómo se gestionan y almacenan las claves y las firmas?	4
c. ¿Cómo se generan los certificados de clave pública? ¿Qué jerarquía de autoridades de certificación se ha desplegado? ¿Por qué ha escogido esta configuración y no otra? ¿Cómo se ha implementado? ¿En qué momento se utilizan los certificados y para qué?	4
d. Indique las pruebas realizadas para garantizar la calidad del código	5
e. Anexo 1: Claves estándar	5
f. Anexo 2: README.md	6

a. ¿Cuál es el propósito de su aplicación? ¿Cuál es su estructura interna?

El propósito de la aplicación es crear un sistema de votación online seguro que simule una situación de democracia directa. Este sistema permite a los ciudadanos votar por leyes en línea, asegurando:

1. Elegibilidad: solo permite votar una vez.
2. Confidencialidad: el voto es secreto, el DNI no se guarda en claro, las claves no están en el código.
3. Integridad: el voto no puede ser alterado o negado, además de todas las firmas para garantizar esto.
4. Autenticación: solo votan usuarios autenticados, el voto sólo es válido si AuthServer lo firma.
5. No-repudio: el que firma no puede negar que firmó.

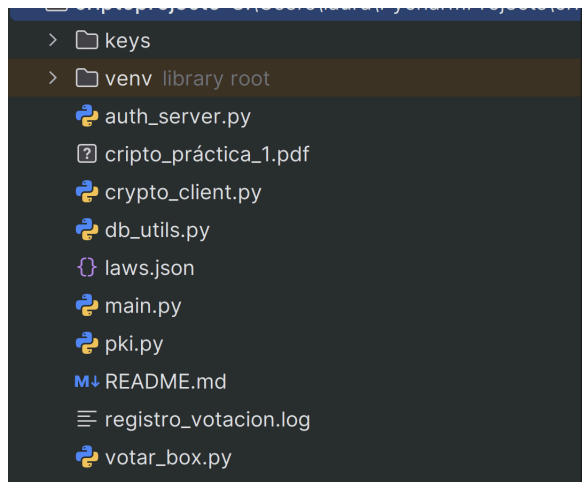
Nuestra aplicación funciona así:

- Autenticación de votantes: Cada votante se registra previamente con DNI, gmail y contraseña, para luego iniciar sesión.
- Votación anónima: Una vez haya iniciado sesión, el votante podrá elegir una ley a la que votar. Este voto se realizará de forma anónima, al solo verificarse la validez del token.
- Generación de más leyes: En una aplicación real, habría una nueva ley publicada cada domingo. Para imitar este sistema, se ha creado la posibilidad de “avanzar” en el tiempo en la propia aplicación, forzando que aparezcan más leyes. Para esta aplicación, solo hemos creado 10 leyes, pero en un programa real las leyes deberían estar en constante creación (no habría un final). Además, se asume de forma artificial que comienza la plataforma el primer domingo de 2025 (5-1-2025).
- Caducación de leyes & Recuento de votos: Al haber pasado 4 domingos, las leyes previas a esos 4 domingos ya no estarán disponibles para ser votadas. Independientemente de esto, al lado de cada ley habrá un botón para observar cuántos votos ha recibido esta ley (incluyendo si es abstención, a favor o en contra).

La aplicación está estructurada en 5 módulos lógicos con una interfaz gráfica en python/Tkinter:

1. *main.py*: proporciona la interfaz de usuario para el registro, login y votación.
2. *crypto\_client.py*: Se ejecuta en el cliente. Se encarga de:
  - 2.1. Generar nonce y timestamp
  - 2.2. Empaquetar y codificar el voto
  - 2.3. Cifrar información sensible
  - 2.4. Formatear el paquete para enviar a la Urna
3. *auth\_server.py*: Gestiona toda la lógica de acceso:
  - 3.1. Registro y Login
  - 3.2. Generación del token de elegibilidad
  - 3.3. Firma digital del token
  - 3.4. Gestión de claves privadas y certificados
  - 3.5. Verificación del certificado contra la PKI
4. *votar\_box.py* (con *BB-BallotBox*): El encargado de procesar y aceptar o rechazar votos:
  - 4.1. Verificación de firma del token recibido
  - 4.2. Extracción campos paquete de voto
  - 4.3. Detecta tokens reutilizados

- 4.4. Registra el voto en la base de datos
- 5.
6. *db\_utils.py*: Crea la base de datos junto con las tablas necesarias para guardar la información de los usuarios, votos, etc.
7. *pki.py*: Implementa toda la lógica de certificados:
  - 7.1. Carga de claves privadas cifradas
  - 7.2. Validación de certificados con OpenSSL
  - 7.3. Verificación firma RSA
8. *laws.json*: contiene todas las leyes (10).



Finalmente, toda la utilización de PKI, certificados, firma del token y verificación del token se ha realizado de manera **asimétrica**.

b. ¿Para qué utiliza la firma digital? ¿Qué algoritmos ha utilizado y por qué? ¿Cómo se gestionan y almacenan las claves y las firmas?

La firma digital se utiliza para garantizar la *autenticidad*, *integridad* y *no repudio* en dos momentos clave:

1. Emisión de Tokens de Voto (AuthServer): El servidor de autenticación firma los tokens de elegibilidad. Esto permite a la Urna (*BallotBox*) verificar que el token fue emitido por una autoridad legítima y que no ha sido modificado, sin necesidad de conocer la identidad del votante (preservando el anonimato).
2. Registro de Votos (BallotBox): La Urna firma cada voto registrado en la base de datos (tallies). Esto asegura que los votos almacenados no puedan ser alterados posteriormente sin romper la verificación de la firma (detectable en la auditoría).

Los algoritmos usados son:

- Algoritmo de Firma: **RSA** con padding **PSS**.
- Algoritmo de Hashing: **SHA-256**.

Por qué: **RSA** es un estándar robusto y ampliamente soportado. El esquema de padding **PSS** es seguro ya que introduce aleatoriedad y tiene una prueba de seguridad más fuerte contra ciertos ataques criptográficos que su versión anterior. **SHA-256** proporciona un nivel de seguridad adecuado (128 bits de resistencia a colisiones) para las necesidades actuales.

Hemos abordado el tema de gestión de claves y firmas de esta manera:

- Claves Privadas: Se almacenan en disco en la carpeta *keys/* como archivos **PEM** (.key.pem). Están cifradas (protegidas por contraseña) utilizando **AES-256-CBC** para evitar su uso no autorizado si el archivo es robado.
- Claves Públicas: Se extraen de los certificados **X.509** (.crt.pem) almacenados también en *keys/*.
- Firmas: Las firmas generadas (por ejemplo, en los votos) se almacenan en la base de datos SQLite (tabla *tallies*, columna *signature*) como datos binarios o codificados, vinculados al registro que protegen.

c. ¿Cómo se generan los certificados de clave pública? ¿Qué jerarquía de autoridades de certificación se ha desplegado? ¿Por qué ha escogido esta configuración y no otra? ¿Cómo se ha implementado? ¿En qué momento se utilizan los certificados y para qué?

Los certificados se generan utilizando la herramienta de línea de comandos **OpenSSL**, que se usa a través de scripts de Python (*subprocess* en *pki.py*). Funciona así:

1. Generación de la clave privada (RSA 4096/3072 bits).
  - a. RSA 4096 es usado para las CA con mayor nivel de autoridad (*root* y *sub*).
  - b. RSA 3072 es usado para las CA con menor nivel de autoridad (*auth* y *ballot*).
2. Creación de una Solicitud de Firma de Certificado (CSR).
3. Firma de la CSR por la Autoridad de Certificación (CA) superior para emitir el certificado final.

La jerarquía de autoridades de certificados en nuestro proyecto es:

1. Root CA (Autoridad Raíz, *root*): La entidad de máxima confianza. Se autofirma.
2. Sub CA (Autoridad Intermedia, *subroot*): Firmada por la Root CA. Es la encargada de emitir los certificados para las entidades finales.
3. Entidades Finales: *AuthServer* y *BallotBox* (*auth* & *ballot*), cuyos certificados son firmados por la Sub CA.

Hemos decidido hacer el proyecto con esta configuración porque esta estructura es un estándar en la industria:

- Permite mantener la Root CA offline o altamente protegida, usándola solo para firmar a la Sub CA.
- Si la clave de la Sub CA se ve comprometida, se puede revocar y generar una nueva sin necesidad de redistribuir el certificado raíz en todos los clientes/sistemas.
- Es más escalable y segura que tener una única CA que firme todo directamente.

La implementación está automatizada en *pki.py*. Este script comprueba si las claves existen y, si no, ejecuta una secuencia de comandos **OpenSSL** para: Crear y autofirmar la Root CA, crear y firmar la Sub CA con Root CA, crear y firmar los certificados de *AuthServer* y *BallotBox* con Sub CA y generar la cadena de confianza (*ca\_chain.pem*).

Los certificados se ven usados en:

- Al inicio (Startup): Tanto *AuthServer* como *BallotBox* verifican la validez de sus propios certificados contra la cadena de confianza (*ca\_chain.pem*) al arrancar para asegurarse de que están correctamente configurados y son legítimos.
- Durante la votación: *AuthServer* usa su clave privada (asociada a su certificado) para firmar tokens. *BallotBox* usa la clave pública del *AuthServer* (extraída de su certificado) para verificar que el token de votación es válido. El cliente (*crypto\_client.py*) usa el certificado de la *BallotBox* para obtener su clave pública y cifrar el voto (sobre digital), asegurando que solo la Urna pueda leerlo.

#### d. Indique las pruebas realizadas para garantizar la calidad del código

(Todas las pruebas han sido realizadas y verificadas con logs en el código)

##### 1. Prueba que muestra la generación de claves para cada CA, además de su lectura y aceptación

```
Password para root:
Password para subroot:
Password para AuthServer:
Password para BallotBox:
[PKI] Generando Root CA...
.....++++
[PKI] Generando Sub CA...
.....++++
.....++++
Signature ok
subject=/C=ES/O=VotePKI/OU=DefaultUnit/CN=DefaultCN
Getting CA Private Key
[PKI] Generando clave y certificado de AuthServer...
.....++++
.....++++
Signature ok
subject=/C=ES/O=VotePKI/OU=DefaultUnit/CN=DefaultCN
Getting CA Private Key
[PKI] Generando clave y certificado de BallotBox...
.....++++
.....++++
.....++++
Signature ok
subject=/C=ES/O=VotePKI/OU=DefaultUnit/CN=DefaultCN
Getting CA Private Key
[PKI] Generando cadena de certificados (ca_chain.pem)...
[PKI] PKI generada correctamente.
[AuthServer] Verificando certificado propio contra la PKI.
[PKI] verify cert_path: /Users/julio/Documents/visual studio/Cripto/criptoprojecto/keys/auth.crt.pem: OK
[AuthServer] Cargando clave privada RSA desde disco.
[AuthServer] Clave RSA cargada.
[BallotBox] Verificando certificado propio contra la PKI.
[PKI] verify cert_path: /Users/julio/Documents/visual studio/Cripto/criptoprojecto/keys/ballot.crt.pem: OK
[BallotBox] Cargando clave privada RSA de BallotBox.
[BallotBox] Clave RSA cargada.
[BallotBox] Iniciando Urna
```

\* con la función getpass, no es posible ver lo que se escribe, para más seguridad en las claves

##### 2. Prueba que rechaza claves erróneas

```
Password para root:
Password para subroot:
Password para AuthServer:
Password para BallotBox:
[PKI] Claves ya existentes, no se regenera la PKI.
[AuthServer] Verificando certificado propio contra la PKI.
[PKI] verify cert_path: /Users/julio/Documents/visual studio/Cripto/criptoprojecto/keys/auth.crt.pem: OK
[AuthServer] Cargando clave privada RSA desde disco.

ERROR inicializando AuthServer: Incorrect password, could not decrypt key
```

##### 3. Prueba de que el voto funciona

```
[AuthServer] Token firmado emitido para ...670T
Tipo dni_hash_bytes: <class 'bytes'>
Tipo election_id_len: <class 'bytes'>
Tipo election_id_bytes: <class 'bytes'>
Tipo nonce: <class 'bytes'>
Tipo ts: <class 'bytes'>
[BallotBox] Firma valida
[BallotBox] Recalculando cadena de hashes desde 2063b611...
[BallotBox] Cadena recalculada (1 tokens actualizados).
```



**Su voto ha sido registrado. Gracias por participar.**

OK

4. Estas son un ejemplo de pruebas entre otras que incluyen:
  - Autoverificación de PKI al inicio: Tanto *AuthServer* como *Ballotbox* verifican la validez de sus propios certificados x.509 contra la cadena de confianza (ca\_chain.pem) utilizando OpenSSL. Si esta falla, el sistema se detiene por seguridad.
  - Control de doble voto: Se verifica que el token no haya sido marcado como usado antes de aceptar un voto.
  - Verificación del token en tiempo real: Cuando la Urna recibe un voto, se verifica la firma digital del token de elegibilidad utilizando la clave pública del *AuthServer*.

\* imagen de prueba para mostrar la generación y verificación de PKI, certificados, claves privadas y firmas así como la cadena de bloques que se crea para evitar que se cambie algún valor en los votos.

```
Getting CA Private Key
[PKI] Generando cadena de certificados (ca_chain.pem)...
[PKI] PKI generada correctamente.
[AuthServer] Verificando certificado propio contra la PKI.
[PKI] verify cert_path: /Users/julio/Documents/visual studio/Cripto/criptoprojecto/keys/auth.crt.pem: OK
[AuthServer] Cargando clave privada RSA desde disco.
[AuthServer] Clave RSA cargada.
[BallotBox] Verificando certificado propio contra la PKI.
[PKI] verify cert_path: /Users/julio/Documents/visual studio/Cripto/criptoprojecto/keys/ballot.crt.pem: OK
[BallotBox] Cargando clave privada RSA de BallotBox.
[BallotBox] Clave RSA cargada.
[BallotBox] Iniciando Urna
Auditoría: No hay votos registrados
Integridad de votos (firmas) verificada.
[Auditoría] Verificando cadena de bloques de tokens...
[Auditoría] Cadena de tokens verificada correctamente (0 bloques).
```

## e. Anexo 1: Claves estándar

Al realizar nuestro proyecto, hemos llegado a un consenso sobre las claves que se utilizarían tanto para la generación de Root CA, Sub CA, *AuthServer* y *BallotBox* como para la clave de cifrado del DNI. En este pequeño anexo se detallarán.

Comenzando con el **DNI**: nuestro programa requerirá una clave, preferiblemente random, en base64 y AES. Recomendamos el uso de OpenSSL en la terminal, aplicando este comando:

```
openssl rand -base64 32
```

Después de ejecutar esto, obtendremos una clave adecuada para el DNI. Se añadirá a la terminal antes de ejecutar el archivo, de esta manera:

```
set DNI_KEY= {key} (windows en terminal)
```

```
DNI_KEY= {key} (linux en terminal)
python main.py
```

Finalmente, el código se ejecutará sin problemas. Si, por alguna razón, se nos olvida poner el DNI, hay una prompt que te preguntará por la clave del DNI para ejecutar el programa. Si, aún así, no se da una clave, ya no se ejecutará. Igualmente, proporcionamos una clave de ejemplo para ejecutar el código sin tener que crear ninguna clave:

**KeEn6FVMn26JTAPDBvR/mFm5kufFmnL2r3mZUsR5BIg=**

Ahora, en cuanto a las claves: Nos hemos decantado por unos nombres bastante comunes y fáciles de recordar – ahora bien, como está implementado puedes usar cualquier password que quieras, con tal de que te acuerdes. Para no tener problemas al recordarlas, hemos decidido siempre usar estas:

- Root CA/root = root
- Sub CA/subroot = subroot
- AuthServer = auth
- BallotBox = ballot

A parte de todo esto, y solo como una última opción si, por alguna razón, da error la creación de claves, hemos adjuntado unas claves dentro de una carpeta llamada “claves\_old” la que contiene claves listas para usar, con las mismas contraseñas que las dadas arriba, pero con un sufijo “\_old” que se debería de borrar.

## f. Anexo 2: README.md

Esto es una pequeña sección que copia lo escrito en el archivo *markdown* [README.md](#), con las pautas de instalación para que el programa funcione correctamente:

### Sistema de Votación Online Seguro

Este proyecto implementa un sistema de votación electrónica utilizando criptografía. La interfaz gráfica ha sido desarrollada con *CustomTkinter*.

#### Requisitos

Para ejecutar esta aplicación, necesitarás tener instalado:

1. *Python* (cualquier versión superior o igual a la 3.8).
2. Las dos librerías listadas a continuación.

#### Librerías Necesarias

Las dos librerías externas son

- A. *CustomTkinter*: para la interfaz.
- B. *Cryptography*: para los algoritmos criptográficos como AES-GCM, RSA-OAEP, PBKDF2 y HMAC.

En caso de no tener **python 3.8 o superior** instalado, en el terminal, ejecuta el siguiente comando:

```
```bash
pip install python3.14
```
```

O ve a la página principal de python [python.org/downloads](https://python.org/downloads) y sigue las instrucciones para instalarlo dependiendo de tu sistema operativo. Para verificar la instalación puedes usar este comando en la terminal:

```
```bash
pip python3 --version
```
```

Teniendo instalado python pondremos los siguientes comandos también en el terminal para instalar *CustomTkinter* y *Cryptography*.

→ *CustomTkinter*:

```
```bash
pip install customtkinter
```
```

Si ese comando no funciona, prueba usando este otro:

```
```bash
python3 -m pip install customtkinter
```
```

→ *Cryptography*:

```
```bash
pip install cryptography
```
```

Si ese comando no funciona, prueba usando este otro:

```
```bash
python3 -m pip install cryptography
```
```

### Contraseñas:

Para Root CA (root) (4096 bits - estándar): root

Para Sub CA (subroot) (4096 bits): subroot

Para AuthServer (3072 bits): auth

Para BallotBox (3072 bits): ballot

### Requisitos previos:

Es necesario tener instalado *OpenSSL* en el sistema operativo.

Para comprobarlo usa este comando:

```
```
openssl version
```
```

Si el comando funciona, el entorno está listo. En Windows puede ser necesario definir la variable de entorno, aunque con la existencia de *openssl.cnf* no debería ser necesario. Igualmente, por si acaso:

```
```
setx OPENSSL_CONF "C:\Program Files\OpenSSL-Win64\ssl\openssl.cnf"
```
```

Reiniciar la terminal después.