

Assignment 2

Scenario Implementation

Communications and Cybersecurity • 2025/2026

Report

Group

Daniel Coelho Pereira • 2021237092

Luís Miguel Ferreira Vaz • 2022214707

Index

Assignment 2.....	1
1. Plan the Services and Diagram	2
1.1. Final Plan to Instantiate Services	2
1.2. Diagram with the scenario and dataflows.....	4
1.3. Data flows and Port Mappings	5
1.3.1. External Communications	5
1.3.2. Internal Communications.....	5
2. Policies Implementation	6
3. Webpage Service	8
3.1 Tests	9
4.SSH Server	16
Approach	16
4.1 Tests	17
5. Database Service.....	19
5.1 Tests	19
6. PKI Service	22
6.1 PKI Container	22
6.2 Root CA	23
6.3 Certificate Lifecycle Management.....	24
6.4 OCSP Responder.....	24
6.5 Certificate Database.....	25
6.6 PKI Tests	26
7. Shared Volume Architecture	29
8. References	30

1. Plan the Services and Diagram

1.1. Final Plan to Instantiate Services

This project implements a secure web application infrastructure with four core services, each serving a specific security function. All the services are deployed as isolated Docker containers to ensure separation of concerns and minimize the attack surface.

Services Overview

Service	Technology	Principal Purpose	Security Role
PKI	OpenSSL 3.5.4 on Alpine Linux	Issues and manages X.509 digital certificates	Trust anchor for all TLS communications; OCSP responder for revocation checking
Web application	Flask 2.3.2 + Nginx 1.29.4 (reverse proxy)	Provides user-facing web interface with authentication	Implements 2FA, session management, CSRF protection, and serves as application entry point
Database	PostgreSQL 15	Persistent data storage	Stores user credentials (hashed), 2FA secrets, and application data with encrypted connections
SSH Server	OpenSSH 8.9p1	Administrative access	Provides secure remote management using certificate-based authentication

Deployment Architecture Rationale

Containerization Strategy

- Each service runs in an isolated Docker container.
- Containers communicate via dedicated Docker networks (`backend_net`, `frontend_net`).
- Volume mounts provides controlled file sharing between containers and host.
- No system has direct access to another service's filesystem or memory.

Network Segmentation

- **frontend_net:** Public-facing network connecting external users to Nginx proxy
- **backend_net:** Internal network for service-to-service communication

Why this Architecture?

- **Defense in depth:** Multiple security layers (network isolation, TLS, application-level controls)

- **Fail-Secure Design:** Compromise of one container does not automatically grant access to others
- **Operational Flexibility:** Services can be updated/restarted independently

1.2. Diagram with the scenario and dataflows

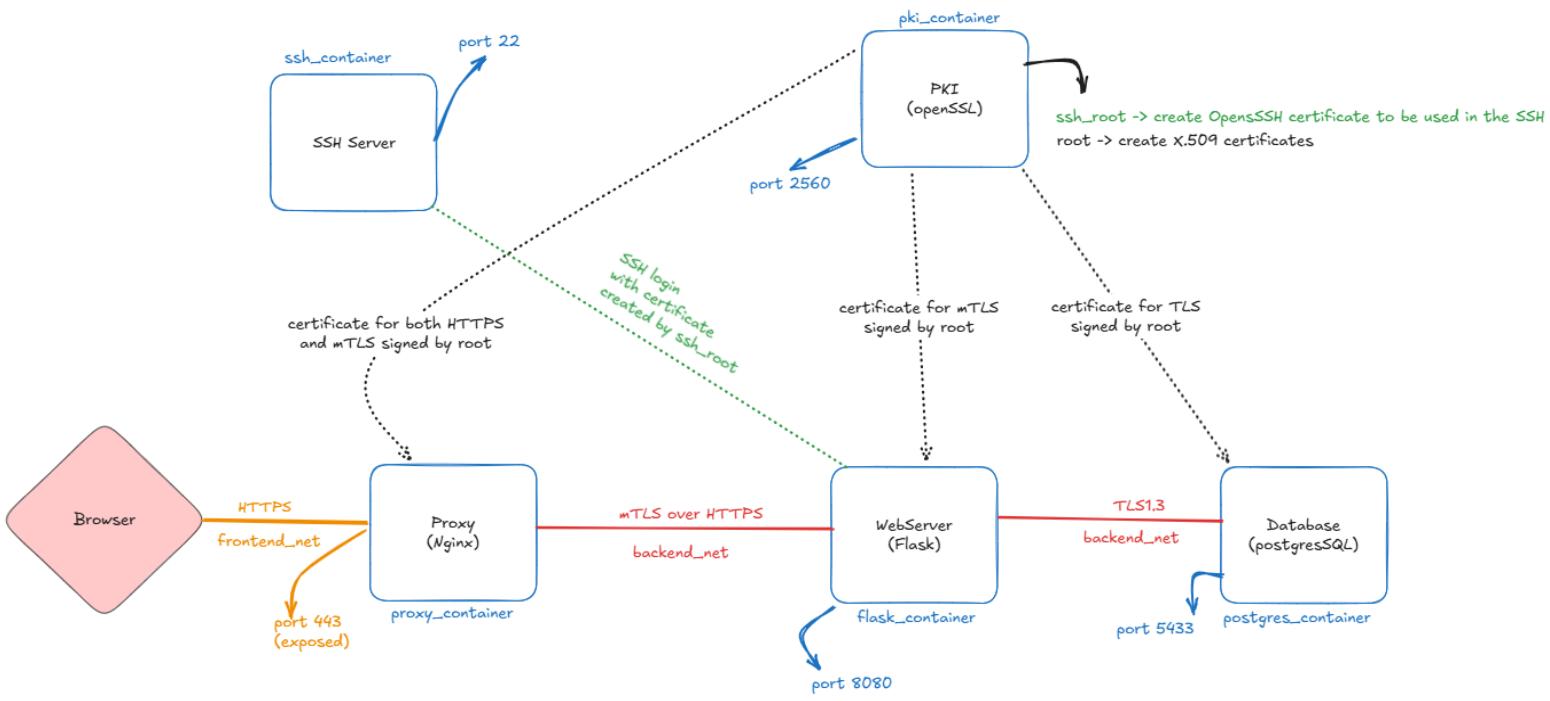


Figure 1- Project Architecture

Diagram Explanation

External Layer

- Users connects via HTTPS on port 443.
- Browser validates server certificate against Root CA. As Root CA is self-signed, it must be added to the system trust store.

Proxy Layer

- Acts as TLS termination point and reverse proxy.
- Forwards authenticated requests to Flask backend.

Application Layer

- Receives mTLS-encrypted requests from Nginx.
- Implements application logic, 2FA verification, session management.
- Connects to PostgreSQL over TLS for data persistence.

Data Layer

- Stores hashed passwords, 2FA secrets, application data.
- Enforces TLS for all connections.
- Uses parameterized queries to prevent SQL injection.

PKI Layer

- Issues certificates for all services.
- Provides OCSP responder on port 2560 for real-time revocation checking.
- Generates CRL.

SSH Layer

- Provides secure administrate access
- Uses certificate-based authentication (no passwords)

1.3. Data flows and Port Mappings

1.3.1. External Communications

Source	Destination	Protocol	Port	Encryption
Browser	Nginx	HTTPS	443	TLS 1.3
Admin	SSH	SSH	2222	openSSH cert

Browser

When the browser connects to https://localhost:443, a security process based on TLS 1.3 begins. During the handshake, the server presents the proxy_https.crt certificate, which was issued by an internal Certificate Authority. The client then verifies the certificate chain against the Root CA, ensuring its authenticity. Next, an ECDHE-ECDSA key exchange is performed to securely establish the session keys. From that point on, all communication between the browser and the server is protected, with application data being encrypted using AES-256-GCM, ensuring both confidentiality and integrity of the transmitted information.

SSH

In the SSH flow, the client connects to the server on port 2222 using a certificate for authentication. The server then validates this certificate, confirming that it was signed by a trusted SSH Certificate Authority (CA). Once the client's identity is verified, an encrypted session is established, securing all subsequent communication using strong encryption algorithms such as AES-256-GCM.

1.3.2. Internal Communications

Source	Destination	Protocol	Port	Encryption
Nginx	Flask	mTLS	8080	TLS 1.3 + Client Cert

Flask	PostgreSQL	TLS	5432	TLS 1.3
All services	PKI OCSP	HTTP	2560	Signed OCSP Response

Nginx

In the mTLS flow between Nginx and Flask, mutual authentication is enforced, meaning that both parties present and verify digital certificates. Nginx presents the **proxy_client.crt** certificate, while Flask presents the **flask_server.crt** certificate. Each side validates the other's certificate against the trusted RootCA, ensuring authenticity and trust. Once verification is complete, a secure and encrypted channel is established, allowing requests to be safely forwarded between Nginx and Flask.

Flask

In the PostgreSQL TLS flow, Flask connects to the database with TLS enforcement enabled using `sslmode='require'`. During the connection setup, PostgreSQL presents the **postgres_server.crt** certificate to the client. Once the secure connection is established, all SQL queries and responses are transmitted over an encrypted channel, ensuring the confidentiality and integrity of the database communication.

OCSP

In the OCSP flow, services query the PKI on port 2560 to verify the status of a certificate. The OCSP responder processes the request and returns a digitally signed response indicating whether the certificate is good, revoked, or unknown. These responses are signed with a dedicated OCSP certificate, ensuring their authenticity and preventing forgery or tampering.

2. Policies Implementation

Service	Description of Policy
---------	-----------------------

Database	<p>The database server will be connected only with the internal network (backend_net), ensuring complete isolation from the public network and preventing unauthorized access.</p> <p>Communication between the web application and the database will occur through this internal Docker network, which already provides a protected communication channel.</p> <p>TLS will be configured to encrypt data in transit.</p> <p>Different user roles will be defined with specific permissions for each table, following the principle of least privilege.</p> <p>SQL injection prevention measures will be applied, using parameterized queries and input validation in the Flask application, ensuring data integrity and the overall security of the database.</p> <p>The configuration files of the database should have mechanisms to check integrity control</p>
WebServer	<p>The web server will support user authentication, requiring two-factor authentication, password and One Time Password. The second factor will be implemented using the Google Authenticator.</p> <p>Communication will be secured through HTTPS, using certificates issued by PKI infrastructure based on OpenSSL.</p> <p>User passwords will be hashed in the webapp, using Argon2 algorithm, before being stored in the database, this can be achieved by importing the argon2-cffi library in python.</p> <p>The application will use Flask sessions with strong secret keys and short expiration times.</p> <p>The server will integrate Flask-limiter, which monitors the rate of client requests and blocks excessive attempts. This helps to prevent brute-force and Denial-of-Service attacks.</p> <p>To protect the browser layer, the Flask-Talisman extension will be used to apply security headers and Content Security Policies (CSP).</p> <p>The Flask-WTF library will be implemented to protect against Cross-Site Request Forgery (CSRF) attacks.</p>
SSH Server	<p>The SSH Server will be deployed inside a Docker container to demonstrate secure application design and deployment.</p>

	<p>It will only be accessible through the internal Docker network, ensuring isolation from the public environment and limiting access exclusively to administrative purposes.</p> <p>Administrative access will use public key cryptography managed by the SSH server. Each administrator receives a unique key pair, and the server validates identities via public keys. This mechanism operates independently of the OpenSSL-based PKI used for other services, while still ensuring secure authentication.</p> <p>The configuration of the service should only allow a small set of simultaneous sessions (e.g., 5 users at the same time).</p> <p>The configuration files of the SSH service should have mechanisms to check integrity control.</p> <p>This setup allows secure remote access without relying on passwords, ensuring both confidentiality and integrity during authentication and data exchange through SSH protocol.</p>
PKI	<p>Use of Hardware Security Modules (HSMs) to enhance security. (won't be implemented)</p> <p>Support full lifecycle of certificates: Issuance, verification, revocation.</p> <p>Regular backups of the CA structure must be performed and stored securely, encrypted with strong symmetric encryption (e.g., AES-256-GCM) and kept offline to prevent unauthorized access or tampering. Backup integrity should be verified periodically to ensure recoverability in case of systems failure or compromise.</p> <p>The CA must verify the integrity of all configuration and certificate management files before execution. This can be achieved by maintaining digitally signed SHA-256 manifests of the files, signed by the CA's private key and verified during each startup. This ensures that no unauthorized modification has occurred, protecting the trust chain of the entire PKI.</p>

3. Webpage Service

The Flask web application provides secure user authentication and account management through HTTPS, implementing multiple security layers including certificate-based mutual TLS (mTLS) between the proxy and application server, input validation and sanitization, rate limiting on authentication endpoints (5 login attempts per minute), and CSRF protection. User passwords are hashed using Argon2, a GPU-resistant algorithm, and stored encrypted in PostgreSQL over TLS. The application supports two-factor authentication (2FA) via Time-based One-Time Passwords (TOTP) and includes file integrity verification at startup, refusing to launch if critical application

files are tampered with. All traffic is encrypted end-to-end: browser to proxy (TLS 1.3), proxy to Flask (mTLS), and Flask to database (TLS with certificate validation).

3.1 Tests

Requirement: The web server will support user authentication, requiring two-factor authentication (password and One Time Password).

In this test we created an account in the web app, activating the 2 factors authentication with the google authenticator app. After the 2 factors authentication is enabled in the account, every time the user tries to login it will have to confirm its 2FA code to complete the login.

The image consists of four screenshots arranged in a 2x2 grid, illustrating the 2FA setup process:

- Screenshot 1 (Left Top):** A QR code for Google Authenticator with instructions to "Digitaliza este QR Code no Google Authenticator". Below it is a text field labeled "Depois insere o código gerado:" and a button "Ativar 2FA".
- Screenshot 2 (Right Top):** A success message "Sucesso!" with a checkmark icon, stating "2FA ativado com sucesso!". A "Ir para o Dashboard" button is at the bottom.
- Screenshot 3 (Left Bottom):** A "Confirmar Código 2FA" screen with a text input field "Código do Authenticator" and a "Confirmar" button.
- Screenshot 4 (Right Bottom):** A welcome message "Bem-vindo, LinoxVaz" with the subtext "2FA está ativo na tua conta." and a "Terminar Sessão" button.

Requirement: Communication will be ensured through HTTPS, using certificates issued by PKI infrastructure based on OpenSSL. This configures ensures the confidentiality and authenticity of communication between client and the Webserver.

For the test we used the following command in the proxy container in the certs directory to be able to access the root.crt: `openssl s_client -connect localhost:443 -showcerts -CAfile ./root.crt -verify_hostname localhost`

```

----  

Certificate chain  

0 s:CN=proxy.cyber.local  

i:CN=RootCA.cyber.local  

a:PKEY: EC, (prime256v1); sigalg: ecdsa-with-SHA256  

v:NotBefore: Dec 12 17:53:24 2025 GMT; NotAfter: Mar 16 17:53:24 2028 GMT  

-----BEGIN CERTIFICATE-----  

MIIB4zCCAYmgAwIBAgICEBAwCgYIKoZIzj0EAwIwHTEbMBkGA1UEAwwSUm9vdENB  

LnN5YnVyLmxvY2FsMB4XDTI1MTIxMjE3NTMyNFoXDTI4MDMxNjE3NTMyNFowHDEa  

MBgGA1UEAwwRchJveHkuY3liZXIubG9jYWwwTATBgcqhkjOPQIBBggqhkJOPQMB  

BwNCAARw/o7SgxCKuXVu1DvPv3oy7m4SxoLaTmeX3pb/7Ugeu9sev1ylehfavGE  

7KKlZrK/d81PO8rh3Ud0bgmylUgHo4G5MIG2MAkGA1UdEwQCMAAwDgYDVROPAQH/  

BAQDAgWgMB0GA1UDJQQWMBQGCCsGAQUFbwMBBgrBgEFBQcDAjA6BgNVHREEMzAx  

ghFwcm94eS5jeWJlc15sb2NhBIJbG9jYWxbob3N0ggVmBFza4cEfwaAYcErBIA  

AjAdBgNVHQEFgQUjlwm0fy0xPGjKKFKIqx/yPS72MwHwYDVR0jBByFoAUgvnd  

1HxL/rfOp2SFeJzllDcE0yUwCgYIKoZIzj0EAwIDSAAwRQIhANpEgnfUAVK4RA1a  

5oQ/IJEk99yosDt+mAdZm023Nb/YAiBVd5hKvD7QSS+wdqDTN2lbMVVGwtRHfkBZ  

zcZJMHa4Zg==  

-----END CERTIFICATE-----  

---  

Server certificate  

subject=CN=proxy.cyber.local  

issuer=CN=RootCA.cyber.local  

---  

Acceptable client certificate CA names  

CN=RootCA.cyber.local  

Requested Signature Algorithms: id-ml-dsa-65:id-ml-dsa-87:id-ml-dsa-44:ECDSA+SHA256  

inpool_P512r1_sha512:rsa_pss_pss_sha256:rsa_pss_pss_sha384:rsa_pss_pss_sha512:RSA-384  

SSL-Session:  

Protocol : TLSv1.3  

Cipher   : TLS_AES_256_GCM_SHA384  

Session-ID: 87202CD8AD29E73A83B67EFB3483C282556445ECF003C028FEDF78A145157352  

Session-ID-ctx:  

Resumption PSK: F512FF40D02160B6F8AEDEFF4AB86A431BEB29BC032ADF5CFD377CCFB888E0FFC5264390C70B731CF8DAAD27E88EC4  

PSK identity: None  

PSK identity hint: None  

SRP username: None  

TLS session ticket lifetime hint: 600 (seconds)  

TLS session ticket:  

0000 - 65 7f 2d 0d 35 1a 42 50-ba ba e6 1d 61 12 98 1c e...5.BP....a...  

0010 - f9 7e 75 c3 76 43 8f 09-3c 19 dd 38 f0 a8 82 82 ..~u.vC..<..8....  

0020 - 9b a7 db 72 9e f8 98 05-68 18 8d 05 98 f2 0c 90 ...r....h.....  

0030 - 10 bd 6e 40 22 8b 77 c4-28 ed cf b3 ea 69 f5 83 ..n@".w.(....i...  

0040 - a8 d7 65 93 09 04 e2 83-db 83 18 b0 b7 51 a9 21 ..e.....Q.!  

0050 - cb b6 61 50 ab 25 7d cc-4f 22 3c cc 5f 58 c3 7c ..ap.%).0"<_X.|  

0060 - 77 94 81 5b 12 e6 d6 35-66 a7 6b b5 23 0c f2 eb w...[...5f.k.#...  

0070 - 4b b1 9a fd 9a 4c ce b9-4e 07 27 08 74 86 f4 d8 K....L..N.'t...  

0080 - 06 23 8d 2e 4e f7 d6 f8-6f e0 20 54 bf 43 17 65 .#..N...o.T.C.e  

0090 - 37 18 f3 02 85 c1 ea 7f-3f 75 e7 a3 24 50 49 21 7.....?u.$PI!  

00a0 - 6c 07 06 bc d7 2d 7f ff-46 81 39 ce de 2d 0c 33 l.....F.9...3  

00b0 - 56 bc 97 cb 68 84 c2 36-d7 0c c2 a3 b6 33 51 f7 V....h..6.....3Q.  

00c0 - 38 65 b7 05 95 f5 d8 62-97 54 98 99 45 cc ef 95 8e.....b.T..E...  

00d0 - 60 b2 29 99 10 35 f0 ea-b1 2c 4d 1d 7b 37 4c 79 `.).5...,M.{7Ly  

Start Time: 1765565621  

Timeout  : 7200 (sec)  

Verify return code: 0 (ok)  

Extended master secret: no  

Max Early Data: 0  

---  

read R BLOCK  

closed

```

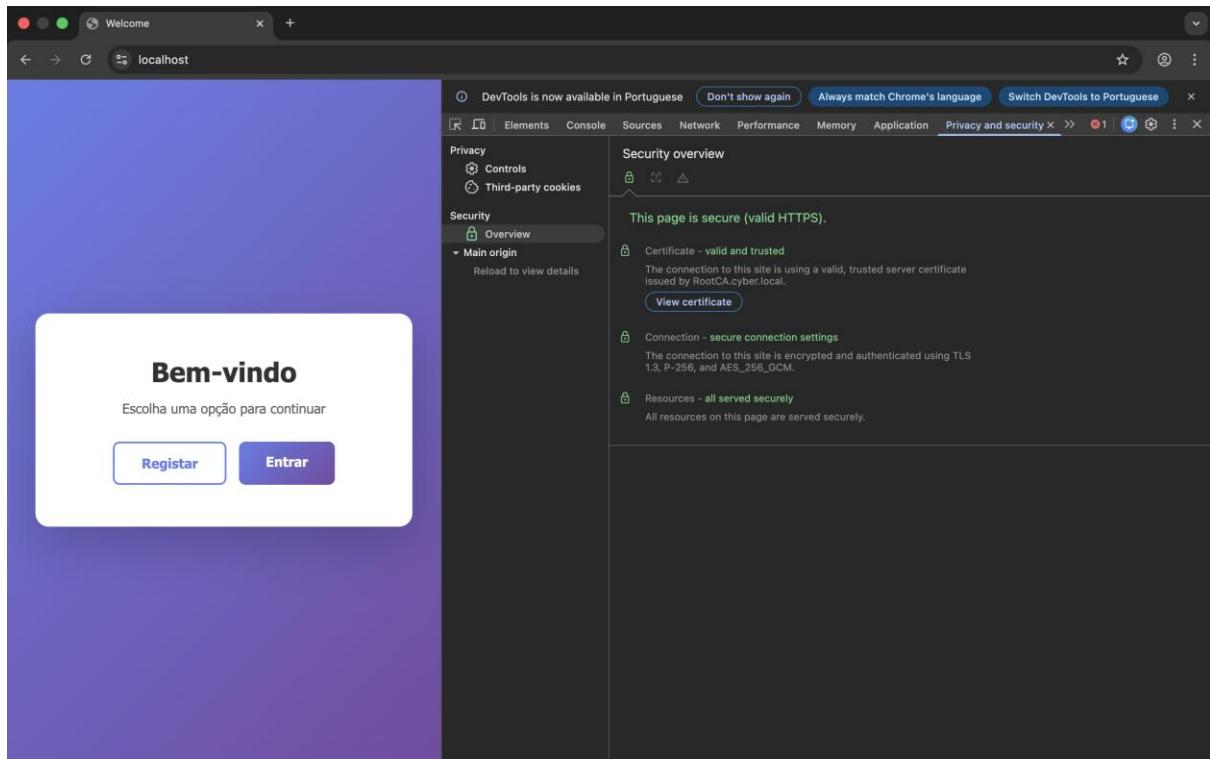


Figure 2 - DevTools output regarding the HTTPS

As shown in the image above, the browser DevTools confirm that the certificate is valid and trusted, the key exchange uses ECDHE with the P-256 curve, the connection is established using TLS 1.3, and the negotiated cipher suite is AES-256-GCM.

```
echo "Q" | docker compose exec -T proxy openssl s_client -connect flask:8080 -cert /etc/nginx/certs/proxy_https.crt -key /etc/nginx/private/proxy_https.key -CAfile /etc/nginx/certs/root.crt 2>&1 | Select-String "subject=","issuer=","Verification","Protocol","Cipher"
subject=CN=flask_server
issuer=CN=RootCA.cyber.local
Verification: OK
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Protocol: TLSv1.3
```

Requirement: *The server will integrate Flask-limiter, which monitors the rate of client requests and blocks excessive attempts. This helps to prevent brute-force and Denial-of-Service attacks.*

To execute this test, we opened our webserver in the browser and refreshed it more than 20 times to test if our rate limiting configuration was being correctly applied.

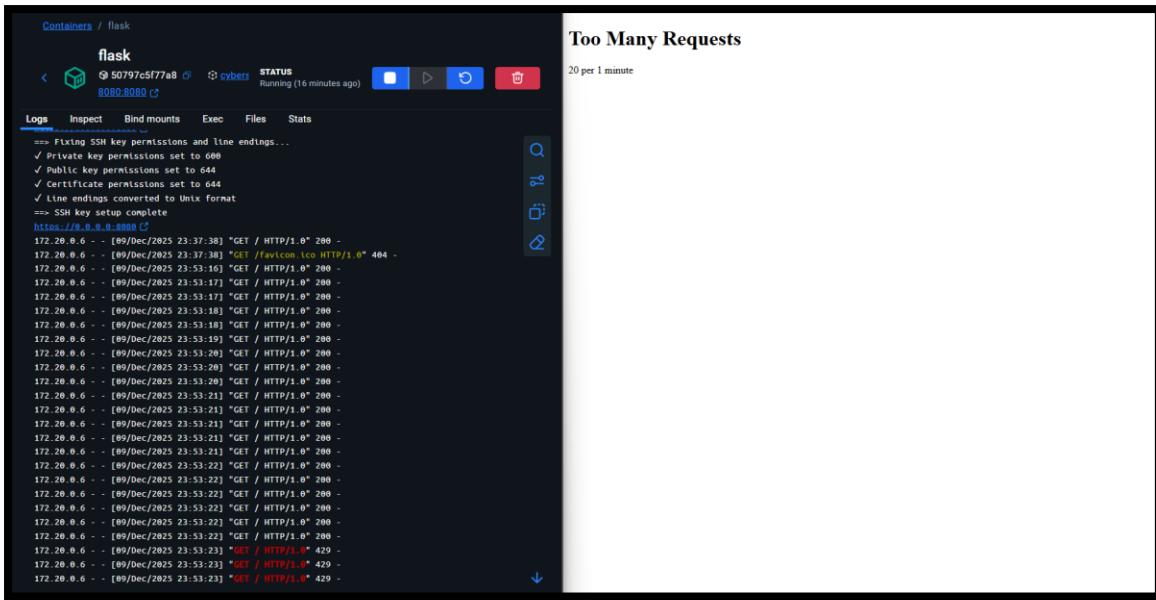


Figure 3 - Logs and Error Page after executing Rate-Limiting test

```
#-----#
# RATE-LIMITING
#-----#
limiter = Limiter(
    key_func=get_remote_address,
    default_limits=["100 per hour", "20 per minute"],
    storage_uri="memory://"
)
```

Figure 4- `__init__.py` code section responsible for webserver Rate Limiting

Requirement: The Flask-WTF library will be implemented to protect against Cross-Site Request Forgery (CSRF) attacks.

To validate CSRF protection, we performed two different tests against the /login endpoint. First, a POST request without a CSRF token was sent from the DevTools console: the server returned 400 Bad Request, confirming that Flask-WTF's CSRFProtect (initialized in `__init__.py`) correctly enforced token presence on POST.

In the second test, the CSRF token was extracted from the rendered login form and included in the POST body, the request was accepted (HTTP 200), proving that only valid, session-bound tokens pass verification.

The image shows a split-screen view. On the left is a screenshot of a browser displaying a 'Login' form with fields for 'Username' and 'Password' and a 'Entrar' button. On the right is a screenshot of the Chrome DevTools Network tab. It shows a failed POST request to 'https://localhost/login' with a status of 400 (Bad Request). The request payload is shown as 'Promise {<pending>}'.

Figure 5 - Request without CSRF token

The screenshot shows a 'Login' form on the left and a terminal window on the right. The form has fields for 'Username' and 'Password', a 'Entrar' button, and a link to register. The terminal window shows a Node.js script using the 'fetch' API to log in with a valid CSRF token, resulting in a 200 status code.

```

> // First, get the CSRF token from the login
form
fetch('/login')
  .then(r => r.text())
  .then(html => {
    // Extract CSRF token from HTML
    const parser = new DOMParser();
    const doc = parser.parseFromString(html,
      'text/html');
    const token =
      doc.querySelector('input[name="csrf_token"]').
      value;

    // Now POST with the valid token
    return fetch('/login', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/x-www-
        form-urlencoded'
      },
      body:
`username=test&password=test&csrf_token=${token}`
    });
  })
  .then(r => r.status)
  .then(status => console.log('Status with
  valid token:', status))
<- ▶ Promise {<pending>}
Status with valid token: 200 VM208:20

```

Figure 6 - Request with CSRF token

Requirement: SQL injection prevention measures will be applied, using parameterized queries and input validation in the Flask application, ensuring data integrity and the overall security of the database.

Protection regarding SQL injections we used placeholders in the queries used in the webserver POST methods.

```
cur.execute("SELECT id, password_hash, twofa_enabled FROM users WHERE username=%s", (username,))
row = cur.fetchone()
```

Figure 7 - Example of query using placeholder

Requirement: User passwords will be hashed in the webapp, using Argon2 algorithm, before being stored in the database, this can be achieved by importing the argon2-cffi library in python.

To test this requirement, we can simply do a query in the postgres container to see the hash of the password that is being stored in the database.

```
docker compose exec postgres psql -U app_read -d cyber -c "SELECT id, username,
LENGTH(password_hash) as hash_length, SUBSTRING(password_hash, 1, 50) as
hash_preview FROM users;"
```

id	username	hash_length	hash_preview
1	LinoxVaz	97	\$argon2id\$v=19\$m=65536,t=3,p=4\$HQMFQLciSWnkIJ3Bsgr

id	username ↑	password_hash	otp_secret	twofa_enabled
7	123	\$argon2id\$v=19\$m=65536,t=3,p=4\$shwBioy404vvulaqWC3LQ\$EJG7HYyxTyajO4M3bC0dfD...	NULL	FALSE
6	daniel	\$argon2id\$v=19\$m=65536,t=3,p=4\$frEK08aRirkIcuFxxN1qgg\$LhK5hzll2aAdUDg3QzBpEaeN... L5ZFXOLF5GPVQEQA04N4ZQ5N...	L5ZFXOLF5GPVQEQA04N4ZQ5N...	TRUE

Figure 8 - Query output confirming the argon password hash

Requirement: All users and inputs will be validated and sanitized.

In the following test we tried to create an account with only 1 character as password and got an error message explaining the requirements for the password. In this case we got an error related to the length of the password but as we can see from the following image the password has to meet certain safety requirements to prevent brute force attacks.

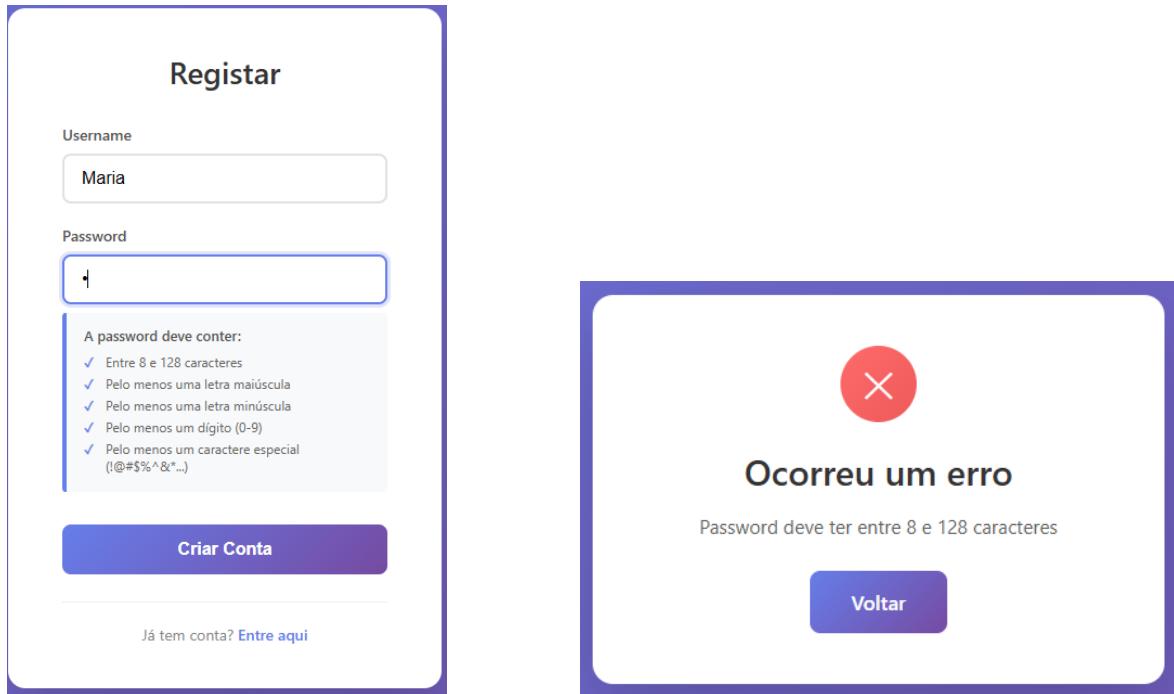


Figure 10- Register account page

Figure 9 - Error message after trying register with weak password

Requirement: Integrity Checks for Nginx

In this test we started the container for the flask and got the log in the docker compose confirming the success in the configuration integrity of the proxy. After this, we edited the nginx.conf adding a comment in aiming to change the hash of the file and consequently when we started the container again, the integrity test would fail, which was successfully accomplished with the message “CRITICAL: nginx config has been tampered with:” and the comparison of the old config hash and the new hash after we’ve changed the nginx configuration file.

```

==> Verifying Nginx configuration integrity...
/etc/nginx/nginx.conf: OK
✓ Nginx configuration integrity verified
==> Verifying Nginx configuration integrity...
/etc/nginx/nginx.conf: OK
✓ Nginx configuration integrity verified
==> Verifying Nginx configuration integrity...
/etc/nginx/nginx.conf: FAILED
✗ CRITICAL: Nginx configuration has been tampered with!
Expected hash:
6711ecbca41f5189377fc816852aba63564dd96d468f76708f5e0edd454d5df
/etc/nginx/nginx.conf
Current hash:
694a4390412c36175bebb6c0c2d8528224c0aa153f92f7911492e71645aee89c
/etc/nginx/nginx.conf

```

Requirement: Integrity Checks for Nginx

A similar test was performed for the flask integrity control but in this case the changed file was wsgi_mtls.py

```

==> Verifying Flask application integrity...
✓ Flask application integrity verified
=====
Flask with mTLS ENABLED
Client cert: REQUIRED
check_hostname: DISABLED
=====
==> Fixing SSH key permissions and line endings...
chmod: changing permissions of '/flask/ssh/sshuser': Read-only file system
✓ Private key permissions set to 600
chmod: changing permissions of '/flask/ssh/sshuser.pub': Read-only file system
✓ Public key permissions set to 644
chmod: changing permissions of '/flask/ssh/sshuser-cert.pub': Read-only file system
✓ Certificate permissions set to 644
✓ Line endings converted to Unix format
==> SSH key setup complete
==> Verifying Flask application integrity...
Traceback (most recent call last):
  File "/app/app/wsgi_mtls.py", line 3, in <module>
    from app import app
    ^^^^^^^^^^^^^^^^^^
  File "/app/app/__init__.py", line 151, in <module>
    app = create_app()
    ^^^^^^^^^^^^^^
  File "/app/app/__init__.py", line 114, in create_app
    verify_integrity()
  File "/app/app/__init__.py", line 75, in verify_integrity
    raise RuntimeError(f"✗ File integrity check failed for: {filepath}")
RuntimeError: ✗ File integrity check failed for: /app/app/wsgi_mtls.py

```

4.SSH Server

Approach

The SSH server implementation simulates secure remote access within the organization, where a computer tries to access a SSH server. To satisfy the certificate-based authentication requirement, a dedicated SSH Certificate Authority (CA) was established within the PKI infrastructure, separate from the X.509 TLS CA. This separation follows best practices by isolating authentication domains and utilizing native OpenSSH certificate formats rather than X.509 structures.

The authentication workflow begins with key generation on both the client and PKI sides using ssh-keygen. The client generates an ED25519 key pair (sshuser private key and sshuser.pub public key). Simultaneously, the PKI generates its own SSH CA key pair (ssh_user_ca private key and ssh_user_ca.pub public key). The client's public key (sshuser.pub) is then transmitted to the PKI container via the shared Docker volume (ssh).

Upon receiving the client's public key, the PKI's signs it using the CA's private key. This process generates an OpenSSH certificate (sshuser-cert.pub) containing the original public key with cryptographic proof of CA endorsement. The certificate is returned to the client through the shared volume, completing the cycle without exposing private keys outside their respective containers.

During SSH authentication, the client presents both the certificate and proves possession of the corresponding private key by signing a server-generated challenge. The SSH server validates this authentication attempt through multiple checks configured in sshd_config: first, it verifies the certificate's cryptographic signature against the trusted CA public key specified in TrustedUserCAKeys, second, it confirms that the certificate's principal name matches an entry in the AuthorizedPrincipalsFile, and third, it validates the client's proof-of-possession signature without ever receiving the private key itself. Only when all three conditions are met does the server grant access. Notably, password authentication is explicitly disabled (PasswordAuthentication no) and root login is

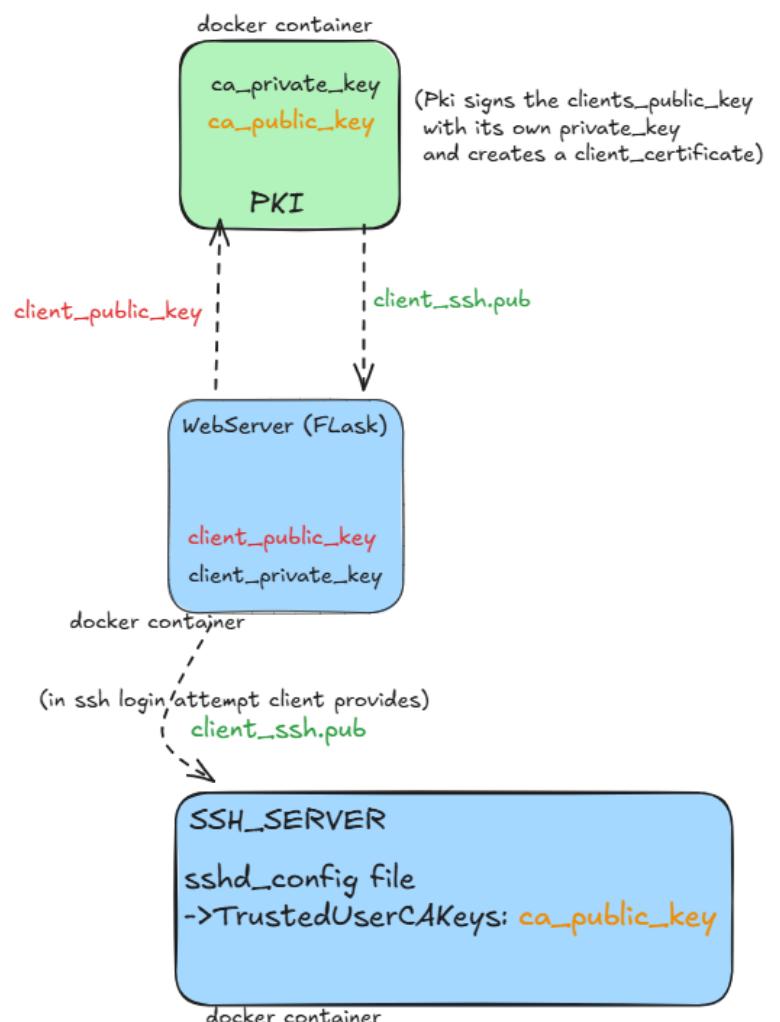


Figure 11- SSH_Server certificate authentication approach architecture

prohibited (PermitRootLogin no), eliminating brute-forcing attacks and password reuse vulnerabilities.

4.1 Tests

Requirement: Supports authentication using certificates.

```
PS C:\Users\Luis Vaz\Documents\Universidade\MEI\CC\Projeto\Cybersecurity-and-Communications> docker compose exec flask ssh -i /flask/ssh/sshuser -o CertificateFile=/flask/ssh/sshuser-cert.pub sshuser@ssh
Last login: Sat Dec 13 02:19:18 2025 from 172.21.0.4
sshuser@dcdd4c2716c:>$
```

Figure 12 - Successful login from the flask docker to the ssh_server after presenting the certificate

```
PS C:\Users\Luis Vaz\Documents\Universidade\MEI\CC\Projeto\Cybersecurity-and-Communications> docker compose exec flask ssh sshuser@ssh
sshuser@ssh: Permission denied (publickey,keyboard-interactive).
```

Figure 13 - Verify that the password authentication is disabled

```
PS C:\Users\Luis Vaz\Documents\Universidade\MEI\CC\Projeto\Cybersecurity-and-Communications> docker compose exec flask ssh-keygen -L -f /flask/ssh/sshuser-cert.pub
Type: ssh-ed25519-cert-v01@openssh.com user certificate
Public key: ED25519-CERT SHA256:+G+HwXyOBxS4nnB1Qc5C34UPyy/EZMotQeURBGzHdV8
Signing CA: ED25519 SHA256:rX7CLvLmXAvhSLhx:JLURc3vpHbd4BXqLNEzw9IUyyE (using ssh-ed25519)
Key ID: "sshuser-cert"
Serial: 0
Valid: from 2025-12-07T03:28:00 to 2026-12-06T03:29:02
Principals:
    sshuser
Critical Options: (none)
Extensions:
    permit-X11-forwarding
    permit-agent-forwarding
    permit-port-forwarding
    permit-pty
    permit-user-rc
```

Figure 14 - Output of the Metadata of the OpenSSH certificate file

Requirement: The configuration of the service should only allow a small set of simultaneous sessions (e.g., 5 users at the same time).

To validate the MaxStartups 5 configuration, we create a python script test_ssh_max_connections to simulate concurrent connection attempts. The test opens 8 simultaneous TCP connections to the SSH server using Python's ThreadPoolExecutor, holding each connection for 10 seconds to occupy authentication slots. Each connection attempt verifies receipt of the SSH protocol banner to confirm the server accepted the handshake before rejecting it. The test measures success by expecting exactly 5 connections to receive SSH banners while connections 6-8 are dropped with errors such as Connection reset by peer or timeout during banner reception. resource exhaustion from excessive concurrent authentication attempts.

```

# python test_ssh_max_connections.py
=====
SSH MaxStartups Test
=====
Target: ssh:22
Expected limit: 5 simultaneous connections
Testing with: 8 concurrent connection attempts
=====

[20:07:23] Starting 8 concurrent connections...

XConnection 5: No SSH banner received - rejected by MaxStartups
XConnection 7: No SSH banner received - rejected by MaxStartups
XConnection 8: No SSH banner received - rejected by MaxStartups
✓Connection 2: Connected in 0.00s (banner: SSH-2.0-OpenSSH_8.9p...)
✓Connection 3: Connected in 0.00s (banner: SSH-2.0-OpenSSH_8.9p...)
✓Connection 6: Connected in 0.00s (banner: SSH-2.0-OpenSSH_8.9p...)
✓Connection 4: Connected in 0.00s (banner: SSH-2.0-OpenSSH_8.9p...)
✓Connection 1: Connected in 0.00s (banner: SSH-2.0-OpenSSH_8.9p...)

=====
Test Results
=====
Total attempts: 8
Successful: 5
Failed: 3

Detailed Results:
Connection 1: ✓PASS - Success (held for 10s)
Connection 2: ✓PASS - Success (held for 10s)
Connection 3: ✓PASS - Success (held for 10s)
Connection 4: ✓PASS - Success (held for 10s)
Connection 5: XFAIL - No SSH banner received - rejected by MaxStartups
Connection 6: ✓PASS - Success (held for 10s)

```

Figure 16 - Max connections Test Output

```

# Session limits
MaxSessions 5
MaxStartups 5
LoginGraceTime 30
MaxAuthTries 10

```

Figure 15 - SSHD config that limits the number of sessions

Requirement: The configuration files of the SSH service should have mechanisms to check integrity control

In this step we first started the ssh server with no modifications and the server started and we got the message “SSH configuration integrity verified”. After that we stopped the container, added a comment to the sshd_config to simulate a change in the configurations and started the container for the ssh server again and our implementation

of integrity checks warn us about the failure when checking the integrity of the files giving us two messages alerting for the error and avoiding the startup of the container.

```
==> Starting SSH server...
Server listening on 0.0.0.0 port 22.
Received signal 15; terminating.
==> Setting up auth_principals with correct permissions...
==> Using existing baseline from /integrity
==> Running SSH configuration integrity check...
/etc/ssh/sshd_config: OK
✓ SSH configuration integrity verified
==> Starting SSH server...
Server listening on 0.0.0.0 port 22.
Received signal 15; terminating.
==> Setting up auth_principals with correct permissions...
==> Using existing baseline from /integrity
==> Running SSH configuration integrity check...
/etc/ssh/sshd_config: FAILED
sha256sum: WARNING: 1 computed checksum did NOT match
cat /run/sshd_config.sha256
✗ FATAL: Configuration integrity check failed!
✗ Aborting SSH startup, maybe an hacker changed the config -_-
```

Figure 17 - Log for the SSH server integrity test

5. Database Service

The PostgreSQL service provides secure storage for application data with role-based access control and encrypted connections. Three application roles enforce least privilege: `app_read` (SELECT-only), `app_write` (SELECT/INSERT/UPDATE), and `app_admin` (full CRUD), verified via `information_schema.grants` and `role_tests`. All traffic to the database is protected with TLS (configured via `postgresql.conf`), using a certificate issued by the project's PKI Root CA, and connections are restricted to SSL-only through `pg_hba` rules. This setup ensures authenticated, authorized access and confidentiality of data in transit.

5.1 Tests

Requirement: *Supports authentication and authorization of users, where different users should have distinct access to the information. For instance, userA can write on table news, while userB can only read.*

```
docker compose exec postgres psql -U myuser -d cyber -c "SELECT grantee, privilege_type FROM information_schema.role_table_grants WHERE table_name='users' ORDER BY grantee, privilege_type;"
```

```

PS C:\Users\Luis Vaz\Documents\Universidade\MEI\CC\Projeto\Cybersecurity-and-Communications> docker compose exec postgres psql -U myuser -d cyber -c "SELECT grantee, privilege_type FROM information_schema.role_table_grants WHERE table_name = 'public' ORDER BY grantee, privilege_type;" 
+-----+-----+
| grantee | privilege_type |
+-----+-----+
| app_admin | DELETE |
| app_admin | INSERT |
| app_admin | REFERENCES |
| app_admin | SELECT |
| app_admin | TRIGGER |
| app_admin | TRUNCATE |
| app_admin | UPDATE |
| app_read | SELECT |
| app_write | INSERT |
| app_write | SELECT |
| app_write | UPDATE |
| myuser | DELETE |
| myuser | INSERT |
| myuser | REFERENCES |
| myuser | SELECT |
| myuser | TRIGGER |
| myuser | TRUNCATE |
| myuser | UPDATE |
+-----+-----+
(18 rows)

```

Figure 18 - Role Permissions on Tables in Database

```
docker compose exec postgres psql -U myuser -d cyber -c "\du"
```

```

PS C:\Users\Luis Vaz\Documents\Universidade\MEI\CC\Projeto\Cybersecurity-and-Communications> docker compose exec postgres psql -U myuser -d cyber -c "\du"
          List of roles
   Role name | Attributes           | Member of
+-----+-----+-----+
| app_admin |                   | {} |
| app_read |                   | {} |
| app_write |                  | {} |
| myuser   | Superuser, Create role, Create DB, Replication, Bypass RLS | {} |
+-----+-----+-----+

```

Figure 19 - Existent Users in Database

Requirement: The traffic to the database should be encrypted

```
docker compose exec -e PGPASSWORD=mypassword postgres psql -U myuser -d cyber -c "SELECT name, setting FROM pg_settings WHERE name IN ('ssl', 'ssl_min_protocol_version', 'ssl_max_protocol_version') ORDER BY name;"
```

```

PS C:\Users\Luis Vaz\Documents\Universidade\MEI\CC\Projeto\Cybersecurity-and-Communications> docker compose exec -e PGPASSWORD=mypassword postgres psql -U myuser -d cyber -c "SELECT name, setting FROM pg_settings WHERE name IN ('ssl', 'ssl_min_protocol_version', 'ssl_max_protocol_version') ORDER BY name;" 
          name | setting
+-----+-----+
| ssl      | on
| ssl_max_protocol_version | TLSv1.3
| ssl_min_protocol_version | TLSv1.3
+-----+-----+
(3 rows)

```

Figure 20 - Configurations used in PostgreSQL

```
docker compose exec postgres openssl x509 -in /shared/certs/postgres_server.crt -text -noout
```

```
PS C:\Users\Luis Vaz\Documents\Universidade\WEI\CC\Projeto\Cybersecurity-and-Communications> docker compose exec postgres openssl x509 -in /shared/certs/postgres_server.crt -text -noout
Certificate:
Data:
Version: 3 (0x2)
Serial Number: 4100 (0x1004)
Signature Algorithm: ecdsa-with-SHA256
Issuer: CN=RootCA.cyber.local
Validity
    Not Before: Dec 13 16:19:19 2025 GMT
    Not After : Mar 17 16:19:19 2028 GMT
Subject: CN=postgres_server
Subject Public Key Info:
    Public Key Algorithm: id-ecPublicKey
        Public-Key: (256 bit)
            pub:
                0E:83:ed:91:22:08:96:0a:c0:9a:61:48:68:53:
                36:99:e0:e1:fa:19:f0:97:8f:b1:65:ba:b3:11:
                ad:a0:ae:0d:24:47:4c:7a:23:c5:4a:29:11:7f:56:
                8d:39:d6:f6:31:78:3a:e6:10:8a:31:ce:16:c3:1b:
                fe:3c:83:78:61
            ASN1 OID: prime256v1
            NIST CURVE: P-256
X509v3 extensions:
    X509v3 Basic Constraints:
        CA:FALSE
    Netscape Cert Type:
        SSL Server
    X509v3 Key Usage: critical
        Digital Signature, Key Encipherment
    X509v3 Extended Key Usage:
        TLS Web Server Authentication
    X509v3 Subject Alternative Name:
        DNS:proxy.cyber.local, DNS:localhost, DNS:flask, IP Address:127.0.0.1, IP Address:172.18.0.2
    Authority Information Access:
        OCSP - URI:http://pki:2560
        CA Issuers - URI:http://pki/root.crt
    X509v3 CRL Distribution Points:
        Full Name:
            URI:http://pki/crl/crl.pem
    X509v3 Subject Key Identifier:
        91:F5:0F:1E:E2:CD:37:10:4F:AA:3D:69:3B:B7:AC:2C:B8:FA:58:20
    X509v3 Authority Key Identifier:
        83:51:88:08:68:7D:94:58:77:E0:07:11:5B:6E:2D:EC:FE:0C:43:E9
    Signature Algorithm: ecdsa-with-SHA256
Signature Value:
    30:45:02:21:00:e2:a0:bd:f1:99:f8:7a:14:02:9f:7c:87:69:
    4a:0b:ff:5a:8b:46:87:78:73:59:8e:c2:11:fa:a2:e0:24:5f:
    e9:02:20:39:47:9d:f5:3a:d1:2a:2f:7b:a6:6d:1b:bf:5b:a7:
    3a:86:bd:ca:b5:51:9c:6c:95:c2:7b:59:08:1a:82:9d:b5
```

Figure 21 - Check PostgreSQL Certificate issuer

Requirement: The configuration files of the database should have mechanisms to check integrity control

```
danielpereira@MacBook-Pro-de-Daniel-2 postgres % docker exec postgres cat /integrity/postgresql.conf.sha256
f6f59eace9c4c1628891210d9b98e4aff0a0799f7ade01d3bb0115a2b84807c4  /var/lib/postgresql/data/postgresql.conf
danielpereira@MacBook-Pro-de-Daniel-2 postgres % docker exec postgres /scripts/verify_config_integrity.sh
==> Verifying PostgreSQL configuration integrity...
/var/lib/postgresql/data/postgresql.conf: OK
✓ Configuration integrity verified
```

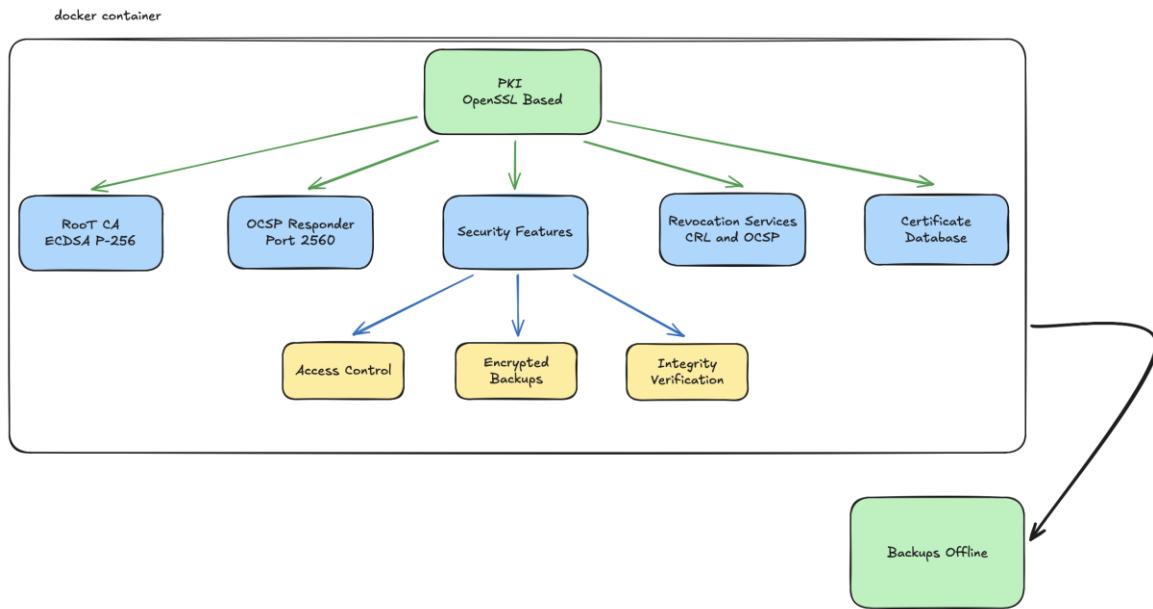
Figure 22 - Example of postgres container start with configuration integrity verified

```
danielpereira@MacBook-Pro-de-Daniel-2 postgres % docker exec postgres /scripts/verify_config_integrity.sh
==> Verifying PostgreSQL configuration integrity...
/var/lib/postgresql/data/postgresql.conf: FAILED
sha256sum: WARNING: 1 computed checksum did NOT match
✗ CRITICAL: Configuration integrity check FAILED!
✗ Aborting startup
```

Figure 23 - Example of postgres container with failed configuration integrity

6. PKI Service

The PKI serves as the trust anchor for the entire infrastructure, providing certificate management services for all secure communications. The design follows a centralized Certificate Authority model where a single Root CA issues and manages certificates for all services. Developed using OpenSSL 3.5.4, the PKI assumes responsibility for the complete lifecycle management of X.509 digital certificates, from issuance to revocation.



6.1 PKI Container

All PKI components were isolated in a dedicated Docker container without external network access, ensuring protection of Root CA and private keys against unauthorized access.

The PKI container operates exclusively on the internal backend_net network, with no port exposure to the host or Internet. This configuration ensures that only internal application services can communicate with the PKI, eliminating external attack vectors.

The implemented isolation architecture provides the following security guarantees:

- **Network Isolation:** The PKI container has no connectivity to the frontend_net network or external networks, preventing unauthorized access.
- **Volume Segregation:** Docker volumes are used to separate critical data (/pki/private/) from shared data (/shared/certs/), with granular permissions applied to each directory.
- **Principle of Least Privilege:** The container executes processes only with strictly necessary permissions, limiting the impact of potential vulnerabilities.
- **Private Key Protection:** Private keys reside exclusively within the container, in directories with 400 permissions, preventing unauthorized modifications.

- **Auditability:** All operations executed in the container are logged, enabling complete traceability of certificate-related actions.

6.2 Root CA

The Root CA serves as the root of trust for the whole infrastructure, using ECDSA with the NIST P-256 curve. This choice reflects modern cryptographic standards and provides several advantages over traditional RSA implementations. ECDSA P-256 delivers equivalent security to RSA-3072 while using smaller keys, resulting in faster cryptographic operations and reduced bandwidth during TLS handshakes.

The Root CA private key is stored within the PKI container at /pki/private/root.key with restrictive 400 permissions, ensuring protection against unauthorized access and modifications. The corresponding public certificate is distributed to all services in the shared folder. In real world applications, the Root CA private key would require enhanced protection through Hardware Security Modules (HSMs) such as Thales Luna HSM. Root CA would be kept offline (air-gapped) and accessed only for critical operations under multi-person authorization. For this academic project, Docker container isolation with restrictive file permissions provides adequate security within the controlled environment.

During the implementation, we opted for a simplified PKI architecture without an Intermediate CA, contrary to the initial planning. However, our simplified approach maintains all essential security properties while reducing operational overhead for the project scope.

6.3 Certificate Lifecycle Management

The PKI implements a comprehensive certificate lifecycle management system through automated scripts that handle issuance, revocation and verification of digital certificates.

Certificate Issuance (issue_cert.sh)

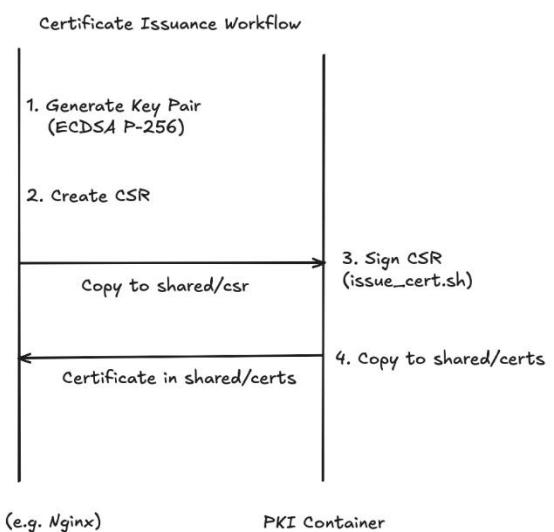
The certificate issuance accepts service name and certificate type (server/client) as parameters, locates the CSR in /shared/csr/, signs it with the Root CA private key applying appropriate extensions (serverAuth or clientAuth), and outputs the certificate to /shared/certs/ while updating the index.txt database.

Certificate Revocation (revoke_cert.sh)

Invalidates certificates by updating their status in index.txt from 'V' (valid) to 'R' (revoked), automatically regenerates CRL files in both PEM and DER formats and updates the OCSP responder in real-time without requiring service restarts.

Certificate Verification (verify_cert.sh)

Performs multi-layered validation including expiration checks, cryptographic signature verification against the Root CA, CRL_based revocation checking, and real time OCSP queries to http://pki:2560, providing complete visibility into certificate validity.



6.4 OCSP Responder

The Online Certificate Status Protocol (OCSP) responder provides real-time certificate revocation checking, complementing the offline Certificate Revocation List mechanism. The OCSP responder runs continuously on port 2560, providing the infrastructure for real-time certificate status verification. The responder correctly processes queries and returns immediate responses indicating whether certificates are valid, revoked, or unknown, demonstrating superior responsiveness compared to periodic CRL downloads.

The implementation demonstrates superior responsiveness compared to periodic CRL downloads, as status changes (particularly revocations) are immediately reflected in OCSP responses without requiring clients to download updated CRLs.

To enhance performance and privacy, OCSP stapling is configured in the Nginx proxy server. With stapling enabled, the server periodically queries the OCSP responder, caches the signed response, and includes it directly in the TLS handshake. This eliminates the need for clients to contact the OCSP responder directly, reducing latency, improving privacy (the OCSP responder doesn't see client IP addresses), and decreasing load on the OCSP infrastructure.

```
(base) MacBook-Pro-de-Daniel-2:Cybersecurity-and-Communications danielpereira$ docker exec pki openssl ocsp \
> -issuer /pki/certs/root.crt \
> -cert /shared/certs/proxy_https.crt \
> -url http://localhost:2560 \
> -CAfile /pki/certs/root.crt
Response verify OK
/shared/certs/proxy_https.crt: good
    This Update: Dec 11 21:22:22 2025 GMT
```

```
(bonequinho) danielpereira@MacBook-Pro-de-Daniel-2 Cybersecurity-and-Communications % openssl s_client \
-connect localhost:443 \
-status \
-servername localhost \
-showcerts
Connecting to ::1
CONNECTED(0x00000005)
depth=1 CN=RootCA.cyber.local
verify error:num=19:self-signed certificate in certificate chain
verify return:1
depth=1 CN=RootCA.cyber.local
verify return:1
depth=0 CN=proxy.cyber.local
verify return:1
OCSP response:
=====
OCSP Response Data:
  OCSP Response Status: successful (0x0)
  Response Type: Basic OCSP Response
  Version: 1 (0x0)
  Responder Id: CN = OCSP Responder
  Produced At: Dec 12 01:33:41 2025 GMT
  Responses:
    Certificate ID:
      Hash Algorithm: sha1
      Issuer Name Hash: 3904A18F63F4291D17486FB64EECE08A5D431E4D
      Issuer Key Hash: 066E6D133989677FF0EAA1EB4B3BD3BDE86A0B93
      Serial Number: 1001
      Cert Status: good
      This Update: Dec 12 01:33:41 2025 GMT

      Signature Algorithm: ecdsa-with-SHA256
      Signature Value:
        30:46:02:21:00:85:3f:74:72:a3:d5:aa:04:1c:a8:19:9f:27:
        42:9d:71:ca:58:88:fc:46:5f:13:94:57:1f:cb:03:d3:46:35:
        03:02:21:00:cc:1e:62:93:38:01:41:49:d9:4e:e8:c4:a9:80:
        d7:78:af:a1:14:fd:fd:1c:cf:ce:48:e2:0d:29:d4:c0:c4:9e
Certificate:
```

6.5 Certificate Database

The certificate database maintains all issued and revoked certificates through OpenSSL's text-based format. The index.txt file tracks certificate status (Valid, Revoked, or Expired), serial numbers, expiration dates, and subject distinguished names. Serial numbers are managed sequentially through the serial file, ensuring unique identifiers for each certificate. The crlnumber file tracks CRL version numbers, incrementing with each update. This database enables the OCSP responder to provide real-time certificate status queries and supports CRL generation. File permissions are set to 640, allowing PKI updates during certificate operations while preventing unauthorized modifications. The database is included in automated PKI backups to ensure complete recovery capability in case of system failure.

V	280316004904Z	1000	unknown /CN=OCSP Responder
R	280316004905Z	251212203601Z	1001 unknown /CN=proxy.cyber.local
V	280316004905Z	1002	unknown /CN=proxy_client
V	280316004906Z	1003	unknown /CN=flask_server
V	280316004906Z	1004	unknown /CN=postgres_server
V	280317003321Z	1005	unknown /CN=proxy.cyber.local

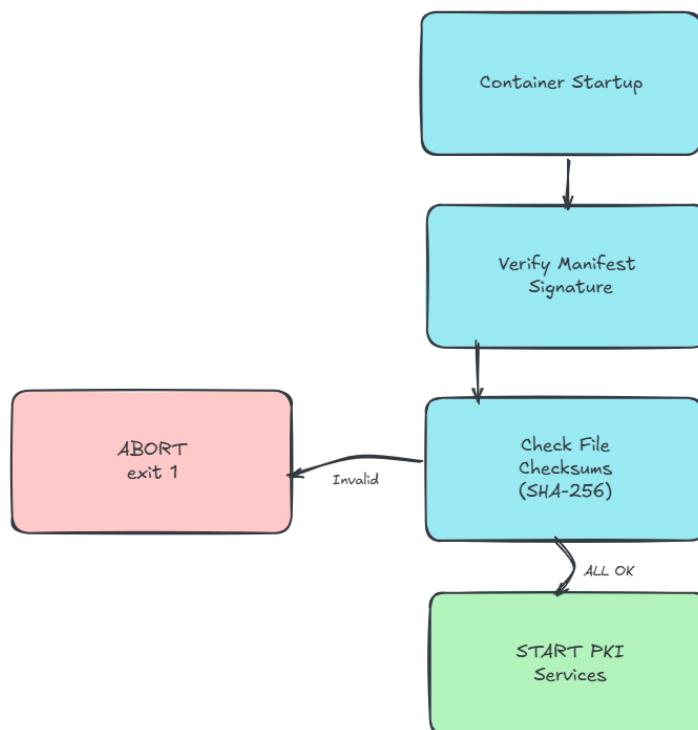
6.6 PKI Tests

Requirement: Regular backups of the CA structure must be performed and stored securely, encrypted with strong symmetric encryption (e.g., AES-256-GCM) and kept offline to prevent unauthorized access or tampering.

Automated backups capture the complete PKI state including index.txt, serial, crlnumber, issued certificates, and the root private key. For demonstration purposes, backups are configured to run every minute with cron, though production environments would typically use daily schedules. Backups are stored in /backups_offline with timestamped filenames, compressed with gzip, and encrypted using AES-256-CBC with SHA-256 HMAC for integrity protection. The encryption key is stored in /pki/private/backup_password.key with permission 400, ensuring only the PKI process can access it. The restore_pki_backup.sh script validates backup integrity, decrypts the archive, and restores all PKI components automatically. A retention policy maintains the last 10 backups to demonstrate the automated cleanup mechanism.

backups_offline	
> logs	•
≡ pki_backup_20251213_012001.info	•
≡ pki_backup_20251213_012101_manifest.txt.enc	U
≡ pki_backup_20251213_012101.info	U
≡ pki_backup_20251213_012101.sig	U
≡ pki_backup_20251213_012101.tar.gz.enc	U

Requirement: The configurations of the CA (database/file with the information of the certificates that are generated) should have mechanisms to check integrity control.



Integrity verification uses SHA-256 manifests signed by the Root CA private key to detect unauthorized modifications. The system checksums critical files and signs the manifest with ECDSA-SHA256. The verify_manifest.sh script validates signatures and checksums on every container startup, terminating with exit code 1 on any mismatch. A bootstrap mode permits first-run initialization without a manifest, addressing the first-use trust problem through isolated Docker deployment. After initial setup, the manifest is version-controlled, and all subsequent startups enforce mandatory integrity verification before PKI operations.

```
==[Checking CA integrity]==  
==[Checking manifest integrity]==  
Verified OK  
==[Valid signature. Checking file hashes]==  
==[Everything is OK. CA is reliable and secure]==  
==[Integrity OK]==
```

```
(bonequinho) danielpereira@MacBookPro Cybersecurity-and-Communications % docker exec pki /pki/manifests/generate_manifest.sh  
==[Generating SHA-256 manifest]==  
[WARN] Missing optional file: /pki/crl/crl.pem (skipped)  
[WARN] Missing optional file: /pki/crl/crl.der (skipped)  
==[Signing manifest using root.key]==  
==[DONE! Manifest and signature generated successfully]==  
(bonequinho) danielpereira@MacBookPro Cybersecurity-and-Communications % docker exec pki /pki/scripts/verify_manifest.sh  
==[Checking manifest integrity]==  
Verified OK  
==[Valid signature. Checking file hashes]==  
==[Everything is OK. CA is reliable and secure]==
```

If the integrity has been compromised, the PKI Docker service will fail to start.

```
==[FAILURE: Integrity compromised: /pki/openssl.cnf]==  
==[Expected: f0bcb077a53bb54570d8588374cf4cd3e512b0a2574b441ef13c721b2b99d9c2]==  
==[Obtained: fb5118fffcfc8c9e5dedf0f6411952bbfd4385d80d1ad097f1c48ab3ab7fe9e84]==  
==[Integrity check FAILED - aborting]==
```

Requirement: Support full lifecycle of certificates: Issuance, verification, revocation.

verify_cert with OCSP

```
# ./verify_cert.sh proxy_https
...
Response verify OK
/shared/certs/proxy_https_chain.crt: good
This Update: Dec 12 20:31:46 2025 GMT
=====
==[Verification complete.]==
```

revoke_cert

```
# ./revoke_cert.sh proxy_https
==[Revoking certificate:]==
==[CN: proxy_https] ==
==[Serial: 1001] ==
==[Path: /pki/newcerts/1001.pem] ==
Using configuration from /pki/openssl.cnf
Revoking Certificate 1001.
Database updated
==[Regenerating CRL...] ==
Using configuration from /pki/openssl.cnf
==[Done.] ==
==[CRL updated:] ==
==[/pki/crl/crl.pem] ==
==[/pki/crl/crl.der] ==
```

verify_cert with OCSP

```
# ./verify_cert.sh proxy_https
==[VALIDATING CERTIFICATE: proxy_https] ==
==[Certificate details:] ==
subject=CN=proxy.cyber.local
issuer=CN=RootCA.cyber.local
notBefore=Dec 12 00:49:05 2025 GMT
notAfter=Mar 16 00:49:05 2028 GMT
serial=1001
X509v3 Subject Alternative Name:
    DNS:proxy.cyber.local, DNS:localhost, DNS:fla
==[Checking certificate expiration:] ==Certificate
==[Certificate is valid (not expired)] ==
==[Verifying certificate chain (against Root CA).]
/shared/certs/proxy_https.crt: OK
==[Checking revocation status via CRL...] ==
CN=proxy.cyber.local
error 23 at 0 depth lookup: certificate revoked
```

Verify certification Test

- (bonequinho) danielpereira@MacBookPro Cybersecurity-and-Communications % docker exec pki openssl verify \
-Cfile /pki/certs/root.crt \
/shared/certs/proxy_https.crt \
/shared/certs/proxy_https.crt: OK

Requirement: Support secure access to the certificates and private keys of the CA. For instance, restricting access to the folder with the information of certificates, like private keys, where only a specific user has access.

Private Key Isolation: CA private keys reside only in private and are mounted exclusively into the pki container. No other service mounts this path, preventing lateral access to CA secrets.

Ownership and Permissions Enforcement: On pki startup, scripts set restrictive permissions:

- Private keys: 600, owned by the CA process user.
- Public certs: 644.

```
docker compose exec pki sh -c "ls -l /pki/private; stat -c '%n %U:%G %a' /pki/private/* >/dev/null || true"
```

```
-r----- 1 pki_admin pki_group 246 Dec 13 16:35 root.key
/pki/private/root.key pki_admin:pki_group 400
```

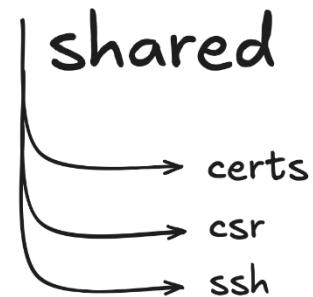
This pattern is mirrored in other services (e.g., PostgreSQL private/), ensuring keys are never readable.

7. Shared Volume Architecture

The /shared/directory provided secure file exchange between containers with segregated subdirectories for different certificate lifecycle stages. The PKI container has read-write access to all subdirectories, while client containers have restricted access based on operational needs.

Directory structure and permissions:

- **/shared/certs/**: Issued certificates
 - PKI: read-write (issues and manages certificates)
 - Services: read-only (retrieve signed certificates)
- **/shared/csr/**: Certificate Signing Requests
 - PKI: read-write (reads CSRs to sign)
 - Services: read-write (submit CSRs, verify submission)
- **/shared/ssh/**: SSH public keys for authentication
 - PKI: read-write (manages authentication keys)
 - Services: read-only (retrieve public keys)



This architecture ensures services can submit certificate requests and retrieve issued certificates but cannot modify existing certificates or access private keys. Each service generates its EC P-256 key pair locally in its own /private/ directory and only shares the public CSR through /shared/csr/, maintaining cryptographic separation between components.

8. References

- [1] Exercises from Practical Classes 1, 2, 3, 4, and 6, Cybersecurity and Communications course, University of Coimbra, Prof. Bruno Miguel de Oliveira Sousa, Academic Year 2025–2026, unpublished course materials.
- [2] Thales Group, *Network Hardware Security Modules (HSMs)*, Thales, 2024. [Online]. Available: <https://cpl.thalesgroup.com/pt-pt/encryption/hardware-security-modules/network-hsms>
- [3] National Institute of Standards and Technology (NIST), *NIST Special Publication 800-32: Cryptographic Key Management*, Gaithersburg, MD, USA, 1988. [Online]. Available: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-32.pdf>
- [4] D. Cooper et al., *RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, IETF, May 2008. [Online]. Available: <https://tools.ietf.org/html/rfc5280>
- [5] E. Rescorla, *RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3*, IETF, Aug. 2018. [Online]. Available: <https://tools.ietf.org/html/rfc8446>
- [6] OWASP Foundation, *Certificate Pinning Cheat Sheet*, OWASP Cheat Sheet Series, 2024. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Pinning_Cheat_Sheet.html
- [7] OpenSSL Project, *OpenSSL Documentation: Certificate Authority*, 2024. [Online]. Available: <https://www.openssl.org/docs/man1.1.1/man1/ca.html>
- [8] Center for Internet Security (CIS), *CIS Docker Benchmark: Security Best Practices*, CIS, 2024. [Online]. Available: <https://www.cisecurity.org/benchmark/docker>
- [9] PostgreSQL Global Development Group, *PostgreSQL SSL/TLS Support Documentation*, 2024. [Online]. Available: <https://www.postgresql.org/docs/current/ssl-tcp.html>
- [10] Mozilla Foundation, *Mozilla SSL Configuration Generator*, 2024. [Online]. Available: <https://ssl-config.mozilla.org/>
- [11] B. Laurie, A. Langley, and E. Kasper, *RFC 6962: Certificate Transparency*, IETF, June 2013. [Online]. Available: <https://tools.ietf.org/html/rfc6962>
- [12] National Institute of Standards and Technology (NIST), *NIST Special Publication 800-53 Revision 5: Security and Privacy Controls for Information Systems and Organizations*, Gaithersburg, MD, USA, 2020. [Online]. Available: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf>