Distributed Software Architecture

Lab 5

Security Lab

Prof. Dr. Dmitry Kachan

EMW, Hochschule Anhalt
July 1, 2025

Lab 5: End-to-End Protection with TLS, mTLS and OAuth 2 / OpenID Connect

1 Why this lab?

Until now every request inside your stack travelled in *plain text* and anyone on the wire could eavesdrop or tamper with it. In production we need at least:

1. **Encryption in transit** – HTTPS/TLS.
2. **Strong service identity** – mutual TLS.
3. **User authentication and authorisation** – OAuth 2 + OIDC with short-lived JWT access tokens.

You will implement all three.

⚠ Learning outcomes

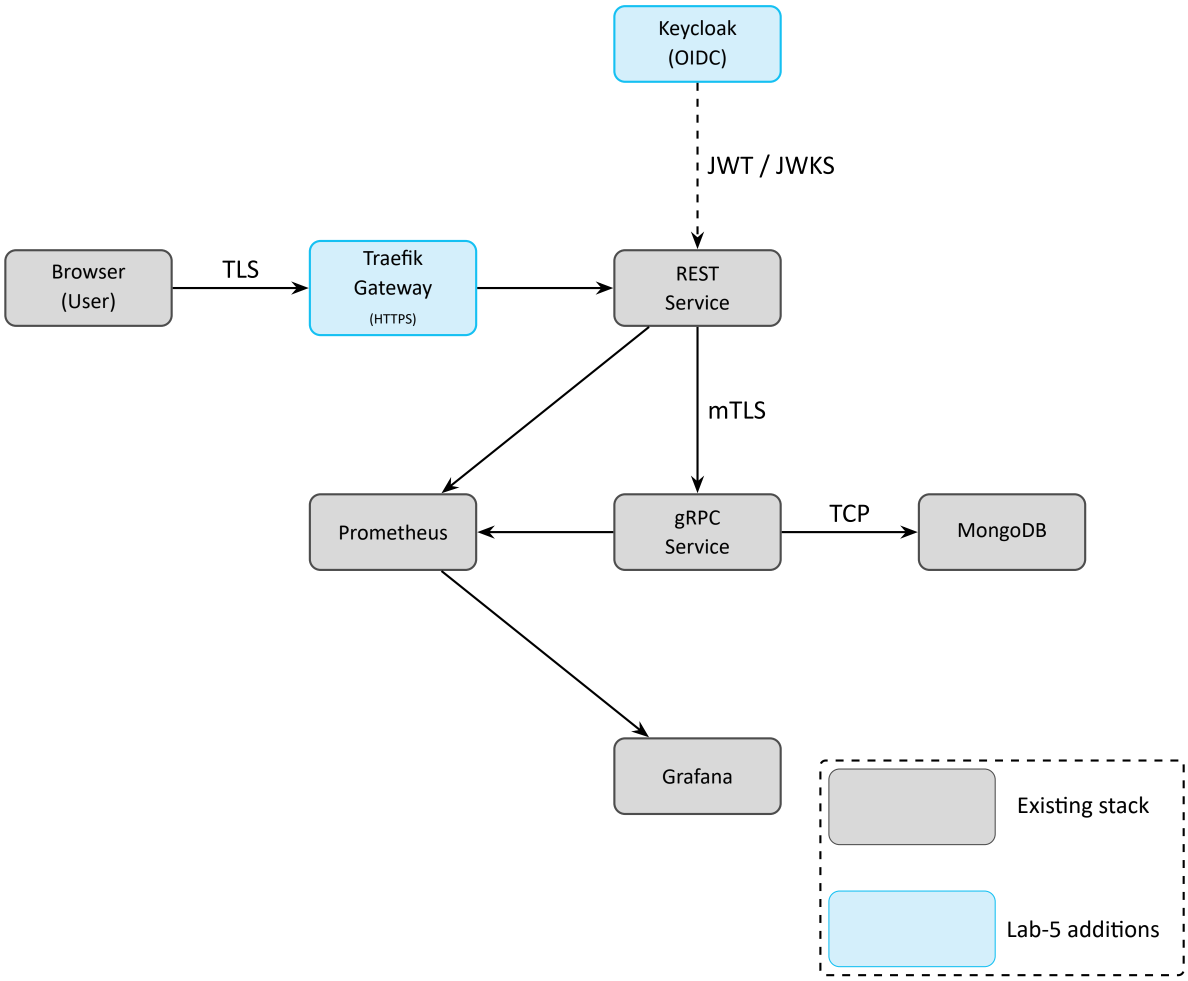
- Understand and create X.509 certificates (root CA, server cert, client cert). *Take-away:* you no longer need to fear OpenSSL spells.
- Deploy Keycloak, the most widely used open-source OAuth 2 / OIDC provider, and protect a REST API with bearer tokens.
- Upgrade gRPC traffic to **mutual TLS**, achieving a zero-trust, service-to-service posture.

2 Lab road-map (at a glance)

1. **Generate a private PKI** – create one root certificate and two leaf certificates (`rest-service`, `grpc-service`).
2. **Put an HTTPS gateway in front of the stack** – Traefik terminates TLS, injects the public user identity (JWT) and talks only HTTPS to the REST service.
3. **Add Keycloak for login & tokens**. Learn the three building blocks: *realm*, *client*, *user/role*.
4. **Modify the REST service** – validate JWTs issued by Keycloak on every call.
5. **Enable mutual TLS between REST ↔ gRPC**. No internal request is accepted unless the caller presents a valid client certificate.
6. **Threat simulation**. Show how missing tokens or wrong certificates are rejected.

3 Step by step implementation

Baseline architecture



3.1 Step 1 – build your own Certificate Authority

What is a CA?

A Certificate Authority signs other certificates. Any certificate that chains back ("is signed by") a trusted CA is accepted by clients.

> ⓘ **Files we are going to produce**
>
> `ca.key` The CA's private key (keep secret!).
>
> `ca.crt` The CA's self-signed certificate (public).
>
> `rest-service.key` Private key for the REST container.
>
> `rest-service.crt` Public certificate for REST, signed by our CA. It contains a Subject Alternative Name (SAN) `DNS:rest-service`.
>
> `grpc-service.{key,crt}` Same idea for the gRPC container.

Generate the root key & certificate

```
#  run ONCE, in security/certs/
```

```
openssl req -x509 -newkey rsa:4096 -days 365 \
  -keyout ca.key -out ca.crt -nodes \
  -subj "/C=DE/O=FH-Anhalt/CN=DSA-Lab-CA"
```

What happened?

- `ca.key` – a 4096-bit RSA private key without a pass-phrase (`-nodes` means "no DES encryption" → convenient in labs).
- `ca.crt` – a public X.509 certificate whose *issuer* equals its *subject*. That makes it a *root*.

Issue leaf certificates

Repeat for both internal services. Example for REST:

```
# create a Certificate Signing Request (CSR)
openssl req -new -newkey rsa:2048 -nodes \
  -keyout rest-service.key -out rest-service.csr \
  -subj "/C=DE/O=FH-Anhalt/CN=rest-service"

# sign it with our CA ("-CAcreateserial" generates ca.srl once)
openssl x509 -req -in rest-service.csr \
  -CA ca.crt -CAkey ca.key -CAcreateserial \
  -out rest-service.crt -days 180
```

Where to store the files?

- Create `security/certs/` in your repo:

```
security/certs/
     ca.crt
     rest-service.{key,crt}
     grpc-service.{key,crt}
```

- Keep `*.key` out of GitHub (!) – add the directory to `.gitignore`. In the lab PCs it is fine to leave them on disk; on personal laptops consider encrypting the folder.

ⓘ Troubleshooting tips

- Use `openssl x509 -text -noout -in rest-service.crt` to inspect the certificate and verify the SAN and validity dates.
- If Chrome/Firefox warns "self-signed", you forgot to import `ca.crt` into your OS/browser trust store.

3.2 Step 2 – add an HTTPS gateway (Traefik)

Why a gateway?

One edge component terminates TLS, enforces rate limits, writes access logs, and forwards traffic to the internal REST service.

What is Traefik? *Traefik* is an open-source edge router / reverse proxy that automatically discovers running containers (or Kubernetes pods, Consul services, …) and routes external HTTP/HTTPS traffic to them. Key features:

- **Dynamic discovery.** When a container starts or stops, Traefik watches the Docker socket, updates its in-memory routing table, and reloads in milliseconds—no manual `nginx -s reload`.

- **Built-in TLS.** Traefik can present your own certificates (our lab) or obtain Let's Encrypt certs automatically (ACME) in production.
- **"Batteries included".** Rate limiting, circuit-breakers, access logs, Prometheus metrics, Web UI dashboard—all first-class options, no extra modules.

Why is it perfect for a Docker-Compose stack? Compose does not offer an ingress controller; each service would otherwise have to publish its own port. Traefik gives us a **single HTTPS entry-point (:443)** and a declarative way to expose only those containers that *opt in* with a few labels—great for local labs and CI pipelines.

What do the Docker labels mean?

traefik.enable=true Opt-in switch; Traefik ignores all other containers by default because we set exposedByDefault=fa

traefik.http.routers.rest.rule A *router* matches incoming requests. The rule PathPrefix(`/api`) means "anything that starts with /api".

traefik.http.routers.rest.entrypoints Which listener to use—in our case the TLS listener named websecure.

traefik.http.services.rest.loadbalancer.server.port Tells Traefik that, once a request matches the router, it should forward the traffic to port 5000 inside the container. (If you later scale to multiple replicas, Traefik load-balances between them automatically.)

With these four labels the REST service becomes reachable at

https://<lab-host>/api/*

while every other container remains internal.

Add Traefik to docker-compose.yml

Minimal Traefik service

```
services:
  gateway:
    image: traefik:v2.11             # stable tag (works with v3-rc too)
    restart: always
    command:
      #     static flags
      - "--entryPoints.websecure.address=:443"
      - "--providers.docker=true"
      - "--providers.docker.exposedByDefault=false"
      - "--log.level=INFO"
      # (optional) quick dashboard at :8080   remove in production
      - "--api.dashboard=true"
    ports:
      - "443:443"                    # HTTPS for users
      - "8080:8080"                  # Traefik dashboard (optional)
    volumes:
      - ./security/traefik.yml:/etc/traefik/traefik.yml:ro   # static file
      - ./security/certs/:/certs/:ro                         # X.509 bundle
      - /var/run/docker.sock:/var/run/docker.sock:ro         # service discovery
```

Tell Traefik which certificate to present

Create `security/traefik.yml`:

```yaml
#   Static configuration
entryPoints:
  websecure:
    address: ":443"

# Explicit self-signed certificate bundle
tls:
  certificates:
    - certFile: "/certs/rest-service.crt"
      keyFile:  "/certs/rest-service.key"

providers:
  docker:
    exposedByDefault: false          # enforce "opt-in with "labels
```

Explanation

- Traefik presents `rest-service.crt` to browsers. That certificate is trusted because their trust store now contains `ca.crt`. (You imported it in Step 1.)
- `exposedByDefault: false` means a container is reachable *only* when it opts in via a Docker label.

Expose the REST container via the gateway

Add labels to `rest-service` in the compose file:

```yaml
labels:
  - "traefik.enable=true"
  - "traefik.http.routers.rest.rule=PathPrefix(`/api`)"
  - "traefik.http.routers.rest.entrypoints=websecure"
  - "traefik.http.services.rest.loadbalancer.server.port=5000"
```

Start the stack:

```
docker compose up -d gateway rest-service
curl -kI https://localhost/api/healthz
# → HTTP/1.1 200 OK
```

`-k` ignores unknown CAs; once `ca.crt` is trusted you can omit `-k`.

3.3 Step 3 – Keycloak 101

What students need to know

Realm An isolated security domain (≈ tenant).

Client An application that wants tokens, e.g. our `rest-client`.

User Human identity performing login.

Role/Scope Permissions encoded into the issued token.

Flow We use the *Resource-Owner-Password* flow to keep Curl examples simple; in production you would use the *Authorization-Code* flow with PKCE.

Add Keycloak to `docker-compose.yml`

Keycloak service

```
keycloak:
  image: quay.io/keycloak/keycloak:26.2.5        # current LTS tag
  command: start-dev --import-realm              # loads realm-export.json
  environment:
    KC_DB: dev-mem                               # in-memory DB (fine for lab)
    KC_BOOTSTRAP_ADMIN_USERNAME: admin           # ← new names in 25+
    KC_BOOTSTRAP_ADMIN_PASSWORD: admin
  volumes:
    - ./security/keycloak/realm-export.json:/opt/keycloak/data/import/realm.json:ro
  ports:
    - "8080:8080"
```

Prepare a realm export (one-time)

A ready-made JSON file is provided (`lab-resources/realm-export.json`). It creates:

- `dsa-lab` realm.
- Client `rest-client` (public type, `http://localhost` as redirect URI).
- User `alice@example.com` with password `secret`.
- Role `ROLE_USER`.

Students can still open http://localhost:8080/admin (`admin/admin`) and click around to see the objects.

Get a token from Keycloak

```
export KC=http://localhost:8080/realms/dsa-lab/protocol/openid-connect
curl -s \
  -d "client_id=rest-client" \
  -d "grant_type=password" \
  -d "username=alice@example.com" \
  -d "password=secret" \
  ${KC}/token | jq -r .access_token > token.jwt

cat token.jwt | cut -d'.' -f2 | base64 -d | jq   # take a look!
```

Important claims you will later verify:

- `iss` Issuer → `http://keycloak:8080/realms/dsa-lab`
- `aud` Audience → `rest-client`
- `exp` Expiry (Unix epoch seconds)
- `preferred_username` or `sub`
- `realm_access.roles[]` contains `ROLE_USER`

3.4 Step 4 – add JWT validation to the REST service

Install a JWT library (or add into requirements.txt file)

```
pip install "python-jose[cryptography]:==3.3.0" requests==2.32.1
```

Do not forget to rebuild docker.

Fetch Keycloak's JWKS once at startup

```python
import requests, jose.jwt
from jose import jwk
JWKS_URL = "http://keycloak:8080/realms/dsa-lab/protocol/openid-connect/certs"
KEYS     = {k["kid"]: k for k in requests.get(JWKS_URL).json()["keys"]}
ISS      = "http://keycloak:8080/realms/dsa-lab"
AUD      = "rest-client"
```

Middleware for every request

```python
from flask import request, abort
from jose import jwt, jwk, JWTError   # ← explicit imports keep linters happy

# global constants set at startup  (see Step 4.2)
# KEYS: dict(kid -> JWK dict)
# ISS : issuer string       ("http://keycloak:8080/realms/dsa-lab")
# AUD : audience string     ("rest-client")

def verify_token() -> None:
    """Flask before_request hook - stops the request with 401 if the
    bearer token is missing / expired / has wrong audience / bad sig."""
    auth = request.headers.get("Authorization", "")
    if not auth.startswith("Bearer "):
        abort(401, description="missing Bearer token")

    token = auth.removeprefix("Bearer ").strip()

    # 1) read unverified header to find the key ID (kid)
    try:
        header = jwt.get_unverified_header(token)
    except JWTError as exc:                       # malformed JWT
        abort(401, description=f"invalid header: {exc}")

    kid = header.get("kid")
    if kid not in KEYS:
        abort(401, description="unknown kid")

    # 2) construct a JWK object for python-jose
    public_key = jwk.construct(KEYS[kid])         # alg inferred from kty

    # 3) verify signature + standard claims
    try:
        claims = jwt.decode(
            token,
            public_key,
            algorithms=[header["alg"]],
            audience=AUD,
            issuer=ISS,
        )
    except JWTError as exc:
        abort(401, description=f"invalid token: {exc}")

    # 4) make the user identity available to downstream code
    request.user = claims.get("preferred_username", claims.get("sub"))
```

Attach `verify_token` as `@app.before_request`. Every endpoint now fails with **401** if the token is missing, expired or has a wrong audience.

Quick test

```
curl -k -H "Authorization: Bearer $(cat token.jwt)" \
  https://localhost/api/items  # → 200 OK
curl -k https://localhost/api/items                # → 401
```

3.5 Step 5 – mutual TLS between services

Why? With HTTPS at the edge only the browser authenticates the server, not vice-versa. Inside the cluster we want *both* ends to present certificates – otherwise any rogue pod could call gRPC.

Server side (gRPC)

Mount `grpc-service.key,crt` and `ca.crt` into the container, then:

```
creds = grpc.ssl_server_credentials(
  [(open("grpc-service.key","rb").read(),
    open("grpc-service.crt","rb").read())],
  root_certificates=open("ca.crt","rb").read(),
  require_client_auth=True)
server.add_secure_port("[::]:50051", creds)
```

Client side (REST)

```
channel = grpc.secure_channel(
  "grpc-service:50051",
  grpc.ssl_channel_credentials(
     root_certificates=open("ca.crt","rb").read(),
     private_key      =open("rest-service.key","rb").read(),
     certificate_chain=open("rest-service.crt","rb").read()))
stub = ItemServiceStub(channel)
```

Test mutual TLS

a) Stop REST ⇒ gRPC. Start again with an **empty** cert directory. Call `/api/items`. You should see `transport: authentication handshake failed` in the REST logs – perfect, the connection is rejected.

b) Restore the files ⇒ try again ⇒ 200 OK.

3.6 Step 6 – threat simulation (5 minutes)

Missing token no `Authorization` header → 401.

Expired token manually set exp in the past (use `jwt.io`) → 401.

Wrong certificate rename `rest-service.key` to test the handshake failure → gRPC refuses the call.

Packet sniff run `tcpdump -i lo port 443` – payload is unreadable.

Screenshot at least two failures for the report.

4 Deliverables (submit in GitHub)

- Updated `docker-compose.yml` with `gateway` and `keycloak`.
- Source code changes in `rest-service` (`verify_token()`, gRPC channel, cert mounts) and in `grpc-service` (server credentials).

- **Screenshots**
  a) Browser with green padlock on `https://localhost/api/healthz`.
  b) Keycloak login page.
  c) Failed request (401) when the token is removed.
  d) Failed request when the client cert is missing (`UNAVAILABLE: TLS handshake error`).
  e) (Bonus) Vault UI showing the generated Mongo credential.
- PDF reflection (<2 pages) – explain *how each layer contributes to zero trust and defence in depth*.

5 Timing guidance (3 × 60 min)

ⓘ Suggested flow

- **0–45 min** – PKI: generate CA + leaf certs, commit to repo, import CA into browser.
- **45–90 min** – Traefik gateway and HTTPS verification.
- **90–120 min** – Keycloak setup, obtain/inspect JWT, integrate token check in REST.
- **120–150 min** – mutual TLS wiring and tests.
- **150–180 min** – screenshots, write reflection, (optional) Vault experiment.

Appendix A – Common OpenSSL recipes

Check a cert	`openssl x509 -text -noout -in xxx.crt`
Decode a JWT quickly	`python -m jwt (pyjwt)`
Inspect TLS handshake	`openssl s_client -connect host:443`

Appendix B – Curl cheat-sheet

```
# 1) Obtain access token (--ResourceOwnerPassword flow)
curl -s -d "client_id=rest-client" -d "grant_type=password" \
     -d "username=alice@example.com" -d "password=secret" \
     http://localhost:8080/realms/dsa-lab/protocol/openid-connect/token \
     | jq -r .access_token > token.jwt

# 2) Call API over HTTPS with token
curl --cacert security/certs/ca.crt \
     -H "Authorization: Bearer $(cat token.jwt)" \
     https://localhost/api/items
```