partnr

Build Secure PKI-Based 2FA Microservice with Docker

Back

Mandatory Task

Domain

Backend Development

DevOps

Skills

API Development | Artificial Intelligence | Deep Learning | Email Security | Management | Public Relations | Time Management | Digital Signatures | Linux System Administration | Multi-Stage Docker Builds | Persistent Data Storage | Public Key Infrastructure (PKI) | RSA Encryption/Decryption | TOTP Authentication

Difficulty

Intermediate

Tools

Docker | Express.Js | FastAPI | Gin | Git | Go | Java | Linux | Node.Js | Python | Rust | Spring Boot | Actix | cron

Pending Submission

Deadline: 13 Dec 2025, 04:59 pm. Please submit your work before the deadline.

TIME REMAINING

4d

Overview | Instructions | Resources | Submit

Description

Objective

Report Issue

Implement a secure, containerized microservice that demonstrates enterprise-

partnr

cryptographic operations, REST API development, Docker containerization, and persistent storage management to create a production-ready authentication system.

You will build a complete authentication microservice that uses RSA 4096-bit encryption for secure seed transmission, implements TOTP-based 2FA for user verification, and runs as a containerized application with automated cron jobs. The system must handle cryptographic key management, secure data persistence across container restarts, and provide three REST API endpoints for seed decryption, 2FA code generation, and code verification.

This project will teach you critical security concepts including asymmetric encryption (RSA/OAEP), digital signatures (RSA-PSS), TOTP authentication protocols, Docker multi-stage builds, volume management for data persistence, and cron job scheduling in containerized environments. You'll gain hands-on experience with cryptographic best practices, secure API design, and production deployment patterns that are essential for building secure modern applications.

Core Requirements

Your microservice must implement the following capabilities:

Cryptographic Operations

- Generate RSA 4096-bit key pair with public exponent 65537
- Decrypt encrypted seed using RSA/OAEP with SHA-256 hash algorithm and MGF1
- Sign commit hash using RSA-PSS with SHA-256 and maximum salt length
- Encrypt signature with instructor's public key using RSA/OAEP with SHA-256
- Validate all cryptographic parameters match specifications exactly

API Endpoints

- **POST /decrypt-seed**: Accept base64-encoded encrypted seed, decrypt using student private key, store persistently at `/data/seed.txt`, return `{"status": "ok"}` on success or `{"error": "Decryption failed"}` on failure with HTTP 500
- **GET /generate-2fa**: Read seed from persistent storage, generate current TOTP code, calculate remaining validity seconds, return `{"code": "123456", "valid_for": 30}` or error with HTTP 500 if seed unavailable

- **POST /verify-2fa**: Accept `{"code": "123456"}`, verify against stored seed with ±1

partnr

TOTP Implementation

- Use SHA-1 algorithm (standard for TOTP)
- 30-second time period
- 6-digit codes
- Convert 64-character hex seed to base32 encoding before TOTP generation
- Implement time window tolerance of ±1 period for verification

Docker Implementation

- Multi-stage Dockerfile with builder and runtime stages
- Install and configure cron daemon
- Set timezone to UTC throughout container
- Expose port 8080 for API server
- Create volume mount points at `/data` and `/cron`
- Copy application code and scripts
- Install cron job that executes every minute
- Start both cron service and API server on container launch

Persistent Storage

- Store decrypted seed at `/data/seed.txt` in Docker volume
- Ensure seed survives container restarts
- Store cron output at `/cron/last_code.txt` in Docker volume
- Maintain proper file permissions for volume access

Cron Job

- Execute every minute using cron daemon
- Read seed from `/data/seed.txt`
- Generate current TOTP code
- Log to `/cron/last_code.txt` with format: `YYYY-MM-DD HH:MM:SS - 2FA Code: XXXXXX`
- Use UTC timezone for timestamps
- Handle errors gracefully with stderr output
- Cron configuration file must use LF (Unix) line endings

Security Requirements

partnr

- Use environment variables for configuration
- Validate all inputs before processing
- Handle cryptographic errors appropriately
- Store keys with proper file permissions

Implementation Guidelines

Technology Stack Flexibility

You may use any programming language and framework. The reference implementation uses Python with FastAPI, but Node.js/Express, Java/Spring Boot, Go/Gin, Rust/Actix, or any other stack is acceptable. Choose based on your expertise and the libraries available for RSA cryptography and TOTP generation.

Cryptographic Library Selection

Use your language's standard cryptography library. Ensure it supports:

- RSA key generation (4096-bit)
- RSA/OAEP encryption/decryption with SHA-256 and MGF1
- RSA-PSS signature generation with SHA-256 and maximum salt length
- PEM format key import/export

TOTP Library Selection

Use a well-maintained TOTP library for your language. Popular options include pyotp (Python), otplib (Node.js), Google Authenticator library (Java), github.com/pquerna/otp (Go), or totp-rs (Rust). Ensure the library supports base32-encoded seeds and standard TOTP parameters.

Docker Best Practices

- Use multi-stage builds to minimize image size
- Copy dependencies before source code to optimize layer caching
- Use minimal base images (alpine, slim variants)
- Clean up package manager caches
- Set proper working directory

- Use EXPOSE directive for documentation

partnr

- Return appropriate HTTP status codes (200, 400, 500)
- Provide meaningful error messages in JSON format
- Handle file I/O errors gracefully
- Validate inputs before processing
- Log errors appropriately for debugging

Testing Strategy

- Test each endpoint individually
- Verify seed persistence across container restarts
- Wait 70+ seconds to verify cron job execution
- Test with valid and invalid TOTP codes
- Verify timezone is UTC for both API and cron

Implementation Details

Step 1: Create GitHub Repository

⚠️ **IMPORTANT: Create your GitHub repository BEFORE starting the task!**

The instructor API requires your GitHub repository URL to generate the encrypted seed. You cannot get the seed without a valid repository URL.

1. Create a new public repository on GitHub
2. Note the exact URL (e.g., `https://github.com/yourusername/your-repo-name`)
3. Clone it locally: `git clone <your-repo-url>`
4. Initialize with a basic README if needed

Critical: Use the EXACT same repository URL when calling the instructor API and when submitting your work. URL mismatch will cause seed generation to fail or produce incorrect TOTP codes.

Step 2: Generate Student Key Pair

Generate an RSA 4096-bit key pair for your student identity:

Key Requirements:

- Key size: **4096 bits** (required)

partnr

Implementation signature:

```
def generate_rsa_keypair(key_size: int = 4096):
    """
    Generate RSA key pair

    Returns:
        Tuple of (private_key, public_key) objects

    Implementation:
    - Use your language's crypto library to generate 4096-bit RSA key
    - Set public exponent to 65537
    - Serialize to PEM format
    - Return key objects for further use
    """
    pass
```

Save files:

- `student_private.pem` - **MUST commit to Git** (required for Docker build)
- `student_public.pem` - **MUST commit to Git** (required for submission)

⚠️ **Security Note:** These keys will be public in your repository and should NOT be reused for any other purpose.

Step 3: Download Instructor Public Key

Download the instructor's public key from the course resources and save as instructor_public.pem. **Commit this to Git.**

The instructor's public key is used to encrypt your commit signature for proof of work verification.

Optional verification:

```
openssl rsa -pubin -in instructor_public.pem -outform DER | \
  openssl dgst -sha256 -binary | \
  openssl enc -base64
```

Step 4: Request Encrypted Seed from Instructor API

⚠️ **CRITICAL: Use the EXACT same GitHub repository URL that you'll submit!**

partnr

Endpoint: POST https://eajeyq4r3zljoq4rpovy2nthda0vtjqf.lambda-url.ap-south-1.on.aws

Request Body:

```
{
  "student_id": "YOUR_STUDENT_ID",
  "github_repo_url": "https://github.com/yourusername/your-repo-name",
  "public_key": "-----BEGIN PUBLIC KEY-----\n...\n-----END PUBLIC KEY-----\n"
}
```

⚠️ **IMPORTANT:**

- The `public_key` must be a **single line** with `\n` characters for line breaks
- Use the **exact same** `github_repo_url` you'll use in your submission
- The API generates a deterministic seed based on your student_id and repo URL

Response:

```
{
  "status": "success",
  "encrypted_seed": "BASE64_ENCODED_CIPHERTEXT..."
}
```

Save the `encrypted_seed` to `encrypted_seed.txt` (do NOT commit this file).

Implementation steps:

```
def request_seed(student_id: str, github_repo_url: str, api_url: str):
    """
    Request encrypted seed from instructor API

    Steps:
    1. Read student public key from PEM file
       - Open and read the public key file
       - Keep the PEM format with BEGIN/END markers

    2. Prepare HTTP POST request payload
       - Create JSON with student_id, github_repo_url, public_key
       - Most HTTP libraries handle newlines in JSON automatically
```

partnr

```
        - Include timeout handling

    4. Parse JSON response
       - Extract 'encrypted_seed' field
       - Handle error responses appropriately

    5. Save encrypted seed to file
       - Write to encrypted_seed.txt as plain text
    """
    pass
```

Step 5: Implement Decryption Function

Decryption Algorithm: RSA/OAEP with SHA-256

Critical Parameters:

- Padding: **OAEP** (Optimal Asymmetric Encryption Padding)
- MGF: **MGF1 with SHA-256**
- Hash Algorithm: **SHA-256**
- Label: **None**

Implementation signature:

```
def decrypt_seed(encrypted_seed_b64: str, private_key) -> str:
    """
    Decrypt base64-encoded encrypted seed using RSA/OAEP

    Args:
        encrypted_seed_b64: Base64-encoded ciphertext
        private_key: RSA private key object

    Returns:
        Decrypted hex seed (64-character string)

    Implementation:
    1. Base64 decode the encrypted seed string

    2. RSA/OAEP decrypt with SHA-256
       - Padding: OAEP
       - MGF: MGF1(SHA-256)
       - Hash: SHA-256
       - Label: None
```

partnr

```
    - Check length is 64
    - Check all characters are in '0123456789abcdef'

5. Return hex seed
"""
pass
```

Key Points:

- The decrypted seed is a **64-character hexadecimal string**
- The seed is generated deterministically based on student_id and github_repo_url
- Store it at `/data/seed.txt` in your container

Step 6: Implement TOTP Generation

TOTP Configuration:

- Algorithm: **SHA-1** (default for most TOTP libraries)
- Period: **30 seconds**
- Digits: **6**
- Seed format: Convert hex seed to base32, then use with TOTP library

Implementation steps:

```
def generate_totp_code(hex_seed: str) -> str:
    """
    Generate current TOTP code from hex seed

    Args:
        hex_seed: 64-character hex string

    Returns:
        6-digit TOTP code as string

    Implementation:
    1. Convert hex seed to bytes
       - Parse 64-character hex string to bytes
       - Use your language's hex-to-bytes conversion

    2. Convert bytes to base32 encoding
       - Encode the bytes using base32 encoding
```

```
        - Decode to string (required by most TOTP libraries)
```

partnr

```
        - Use default settings: SHA-1, 30s period, 6 digits

    4. Generate current TOTP code
        - Call library method to get current code
        - Returns 6-digit string (e.g., "123456")

    5. Return the code
    """
    pass
```

Verification implementation:

```
def verify_totp_code(hex_seed: str, code: str, valid_window: int = 1) -> bool:
    """
    Verify TOTP code with time window tolerance

    Args:
        hex_seed: 64-character hex string
        code: 6-digit code to verify
        valid_window: Number of periods before/after to accept (default 1 = ±30s)

    Returns:
        True if code is valid, False otherwise

    Implementation:
    1. Convert hex seed to base32 (same process as generation)

    2. Create TOTP object with base32 seed

    3. Verify code with time window tolerance
        - Use valid_window parameter (default 1 = ±30 seconds)
        - This accounts for clock skew and network delay
        - Library checks current period ± valid_window periods

    4. Return verification result
    """
    pass
```

Step 7: Implement API Endpoints

Framework Choice: Use any HTTP framework (FastAPI, Express, Spring Boot, etc.)

Endpoint 1: POST /decrypt-seed

Request:

partnr

```
  "encrypted_seed": "BASE64_STRING..."
}
```

Response (200 OK):

```
{
  "status": "ok"
}
```

Response (500 Internal Server Error):

```
{
  "error": "Decryption failed"
}
```

Implementation checklist:

- Load student private key from file
- Base64 decode the encrypted_seed
- Decrypt using RSA/OAEP-SHA256
- Validate decrypted seed is 64-character hex
- Save to `/data/seed.txt`
- Return `{"status": "ok"}`

Endpoint 2: GET /generate-2fa

Response (200 OK):

```
{
  "code": "123456",
  "valid_for": 30
}
```

Response (500 Internal Server Error):

```
{
  "error": "Seed not decrypted yet"
```

```
}
```

partnr

- Check if `/data/seed.txt` exists
- Read hex seed from file
- Generate TOTP code
- Calculate remaining seconds in current period (0-29)
- Return code and valid_for

Endpoint 3: POST /verify-2fa

Request:

```
{
  "code": "123456"
}
```

Response (200 OK):

```
{
  "valid": true
}
```

or

```
{
  "valid": false
}
```

Response (400 Bad Request):

```
{
  "error": "Missing code"
}
```

Response (500 Internal Server Error):

partnr

Implementation checklist:

- Validate code is provided
- Check if `/data/seed.txt` exists
- Read hex seed from file
- Verify TOTP code with ±1 period tolerance
- Return `{"valid": true/false}`

Step 8: Create Dockerfile

Requirements:

- Multi-stage build (builder + runtime)
- Base image appropriate for your language
- Install system dependencies (cron, timezone tools)
- Set timezone to **UTC**
- Copy application code
- Set up cron job
- Create volume mount points: `/data` and `/cron`
- Expose port 8080
- Start both cron service and API server

Structure template:

```
# Stage 1: Builder
# - Use base image for your language (python:3.11-slim, node:20-slim, etc.)
# - Set working directory
# - Copy dependency file (requirements.txt, package.json, etc.)
# - Install dependencies (optimize for caching)

# Stage 2: Runtime
# - Use minimal runtime base image
# - Set TZ=UTC environment variable (critical!)
# - Set working directory

# Install system dependencies
# - Update package manager
# - Install cron daemon
```

```
# - Install timezone data
```

partnr

```
# - Create symlink to UTC timezone
# - Set TZ environment variable

# Copy dependencies from builder
# - Copy installed packages
# - Update PATH if needed

# Copy application code
# - Copy source directory
# - Copy scripts directory
# - Copy cron configuration

# Setup cron job
# - Set permissions on cron file (0644)
# - Install cron file with crontab

# Create volume mount points
# - Create /data directory
# - Create /cron directory
# - Set permissions (755)

# EXPOSE 8080

# Start cron and application
# - Start cron daemon
# - Start HTTP server on 0.0.0.0:8080
```

Step 9: Create Docker Compose Configuration

Requirements:

- Define service with build context
- Map port 8080:8080
- Create named volumes: `seed-data` and `cron-output`
- Mount volumes to `/data` and `/cron`
- Set environment variables: `TZ=UTC`
- Mount key files (for local development)

Structure template:

```
services:
  app:
```

```
      # Build configuration
```

partnr

```
      # Container name (optional)

      # Port mapping
      # - Map 8080:8080

      # Volume mounts
      # - seed-data -> /data
      # - cron-output -> /cron
      # - student_private.pem -> /app/student_private.pem:ro
      # - student_public.pem -> /app/student_public.pem:ro
      # - instructor_public.pem -> /app/instructor_public.pem:ro

      # Environment variables
      # - TZ=UTC

      # Restart policy
      # - unless-stopped or on-failure

  volumes:
    seed-data:
    cron-output:
```

Step 10: Implement Cron Job

Cron Configuration File: `cron/2fa-cron`

⚠️ CRITICAL: This file MUST use LF line endings!

Content:

```
* * * * * cd /app && /usr/local/bin/python3 scripts/log_2fa_cron.py >> /cron/last_code.t
```

(Adjust path to match your language's interpreter)

Cron Script: `scripts/log_2fa_cron.py`

Implementation steps:

```
#!/usr/bin/env python3
# Cron script to log 2FA codes every minute

# 1. Read hex seed from persistent storage
```

```
#    - Open /data/seed.txt file
```

partnr

```
# 2. Generate current TOTP code
#    - Use the same TOTP generation function
#    - Convert hex seed to base32
#    - Generate code using TOTP library

# 3. Get current UTC timestamp
#    - Use UTC timezone (critical!)
#    - Format: YYYY-MM-DD HH:MM:SS

# 4. Output formatted line
#    - Format: "{timestamp} - 2FA Code: {code}"
#    - Print to stdout (appended by cron)
```

⚠️ CRITICAL: `.gitattributes` Configuration

Create `.gitattributes` file in repository root:

```
cron/2fa-cron text eol=lf
```

This ensures the cron file uses LF line endings, required for cron to work on Linux.

Step 11: Set Up `.gitignore`

⚠️ IMPORTANT: Key Files Must Be Committed

For this project, you **MUST commit** `student_private.pem` and `student_public.pem` to your repository. This is required because:

- Your Docker container needs the private key to decrypt the seed
- The evaluator needs to verify your implementation
- The keys are included in the Docker image build

⚠️ SECURITY WARNING:

- These keys will be **PUBLIC** in your GitHub repository
- **DO NOT reuse these keys** for any other purpose
- These keys are **ONLY for this assignment**
- Generate new keys for any production use

Required exclusions:

partnr

```
# Python cache
__pycache__/
*.pyc

# Environment files
.env

# IDE files
.vscode/
.idea/
```

Files to COMMIT:

- ✅ `student_private.pem` - Required (will be public)
- ✅ `student_public.pem` - Required
- ✅ `instructor_public.pem` - Required
- ✅ All source code
- ✅ `Dockerfile`
- ✅ `docker-compose.yml`
- ✅ `requirements.txt` or equivalent
- ✅ `.gitattributes`
- ✅ `.gitignore`
- ✅ `cron/2fa-cron`
- ✅ All scripts

Step 12: Test Locally

Build and run:

```
docker-compose build
docker-compose up -d
```

Test endpoints:

```
# 1. Decrypt seed
curl -X POST http://localhost:8080/decrypt-seed \
  -H "Content-Type: application/json" \
  -d "{\"encrypted_seed\": \"$(cat encrypted_seed.txt)\"}"
```

```
# 2. Generate 2FA code
```

partnr

```
CODE=$(curl -s http://localhost:8080/generate-2fa | jq -r '.code')
curl -X POST http://localhost:8080/verify-2fa \
  -H "Content-Type: application/json" \
  -d "{\"code\": \"$CODE\"}"

# 4. Verify invalid code
curl -X POST http://localhost:8080/verify-2fa \
  -H "Content-Type: application/json" \
  -d '{"code": "000000"}'

# 5. Check cron output (wait 70+ seconds)
sleep 70
docker exec <container-name> cat /cron/last_code.txt
```

Verification checklist:

- [ ] All endpoints return correct responses
- [ ] Seed persists after container restart
- [ ] Cron job logs codes every minute
- [ ] TOTP codes match expected values
- [ ] Timezone is UTC

Step 13: Generate Commit Proof

Process:

1. Commit all code to Git
2. Get commit hash: `git log -1 --format=%H`
3. Sign commit hash with student private key (RSA-PSS-SHA256)
4. Encrypt signature with instructor public key (RSA/OAEP-SHA256)
5. Base64 encode the encrypted signature

Signature Algorithm: RSA-PSS with SHA-256

Critical Parameters:

- Padding: **PSS** (Probabilistic Signature Scheme)
- MGF: **MGF1 with SHA-256**
- Hash Algorithm: **SHA-256**
- Salt Length: **Maximum** (PSS.MAX_LENGTH)

- Message: **ASCII bytes of commit hash** (40-character hex string)

partnr

```
def sign_message(message: str, private_key) -> bytes:
    """
    Sign a message using RSA-PSS with SHA-256

    Implementation:
    1. Encode commit hash as ASCII/UTF-8 bytes
       - CRITICAL: Sign the ASCII string, NOT binary hex!
       - Use message.encode('utf-8')

    2. Sign using RSA-PSS with SHA-256
       - Padding: PSS
       - MGF: MGF1 with SHA-256
       - Hash Algorithm: SHA-256
       - Salt Length: Maximum

    3. Return signature bytes
    """
    pass
```

Encryption implementation:

```
def encrypt_with_public_key(data: bytes, public_key) -> bytes:
    """
    Encrypt data using RSA/OAEP with public key

    Implementation:
    1. Encrypt signature bytes using RSA/OAEP with SHA-256
       - Padding: OAEP
       - MGF: MGF1 with SHA-256
       - Hash Algorithm: SHA-256
       - Label: None

    2. Return encrypted ciphertext bytes
    """
    pass
```

Complete proof generation:

```
# 1. Get current commit hash
#    - Run: git log -1 --format=%H
#    - Extract 40-character hex string
```

```
# 2. Load student private key
```

partnr

```
# 3. Sign commit hash with student private key
#    - Use sign_message with RSA-PSS-SHA256
#    - Sign ASCII bytes of commit hash

# 4. Load instructor public key
#    - Read instructor_public.pem
#    - Parse PEM format

# 5. Encrypt signature with instructor public key
#    - Use encrypt_with_public_key with RSA/OAEP-SHA256

# 6. Base64 encode encrypted signature
#    - Encode to base64
#    - Decode to string (single line)

# Output:
# - Commit Hash: 40-character hex
# - Encrypted Signature: Base64 string (single line)
```

Step 14: Prepare Submission

Required Information:

1. **GitHub Repository URL**
   - Exact URL used when calling instructor API
   - Example: `https://github.com/yourusername/your-repo-name`
2. **Commit Hash**
   - 40-character hex string from `git log -1 --format=%H`
3. **Encrypted Commit Signature**
   - Base64-encoded encrypted signature
   - **MUST be single line** (no line breaks!)
4. **Student Public Key**
   - Contents of `student_public.pem`
   - Include BEGIN/END markers
   - For API: single line with `\n` characters

- For form: can be multi-line

partnr

- Contents of `encrypted_seed.txt`
- **MUST be single line** (no line breaks!)

6. **Docker Image URL** (optional)

- If pushed to registry
- Example: `docker.io/yourusername/pki-2fa:latest`

Common Mistakes to Avoid

1. Repository URL Mismatch

Mistake: Using different GitHub repository URLs when calling instructor API vs. submission form.

Impact: The evaluator will generate a different seed, causing all TOTP codes to be incorrect.

Solution: Create repository FIRST, use the EXACT same URL everywhere, double-check before calling API and before submission.

2. Public Key Formatting in API Request

Mistake: Submitting public key with actual newlines instead of `\n` escape sequences.

Impact: API may reject the request or fail to parse the key.

Solution: Most HTTP libraries handle JSON newlines automatically. If manual escaping is needed, replace newlines with `\n`.

3. Encrypted Signature Line Breaks

Mistake: Copying encrypted signature with line breaks or extra whitespace.

Impact: Base64 decoding will fail during evaluation.

Solution: Copy the ENTIRE base64 string as a SINGLE line, remove any line breaks, spaces, or newlines.

4. Wrong Encryption/Decryption Parameters

Mistake: Using wrong padding scheme, hash algorithm, or MGF.

partnr

Solution:

- Decryption: RSA/OAEP with MGF1(SHA-256), SHA-256, label=None
- Signature: RSA-PSS with MGF1(SHA-256), SHA-256, max salt length
- Encryption: RSA/OAEP with MGF1(SHA-256), SHA-256, label=None

5. TOTP Seed Conversion Error

Mistake: Using hex seed directly instead of converting to base32.

Impact: TOTP codes will be incorrect.

Solution:

```
# WRONG:
totp = TOTP(hex_seed)  # ❌

# CORRECT:
seed_bytes = bytes.fromhex(hex_seed)
base32_seed = base64.b32encode(seed_bytes).decode('utf-8')
totp = TOTP(base32_seed)  # ✅
```

6. Cron File Line Endings

Mistake: Cron file using CRLF (Windows) line endings instead of LF (Unix).

Impact: Cron job will not execute, causing 0 points for cron test.

Solution: Create `.gitattributes` with `cron/2fa-cron text eol=lf`, verify with `file cron/2fa-cron`.

7. Timezone Not UTC

Mistake: Using local timezone instead of UTC for TOTP and cron logs.

Impact: TOTP codes may be off by timezone offset, causing verification failures.

Solution: Set `TZ=UTC` in Dockerfile and docker-compose.yml, use UTC functions for timestamps.

8. Seed Not Persisting

Mistake: Storing seed in container filesystem instead of Docker volume.

partnr

Solution: Use Docker volume mounted at `/data`, save to `/data/seed.txt`, verify volume in docker-compose.yml.

9. Wrong TOTP Verification Window

Mistake: Not allowing time window tolerance for TOTP verification.

Impact: Valid codes may be rejected due to clock skew.

Solution: Use `valid_window=1` (accepts current period ±1 period = ±30 seconds).

10. Missing Error Handling

Mistake: Not handling cases where seed file doesn't exist.

Impact: API endpoints crash instead of returning proper error responses.

Solution: Check if `/data/seed.txt` exists before reading, return appropriate HTTP status codes and JSON errors.

11. Signing Binary Instead of ASCII

Mistake: Signing commit hash as binary hex instead of ASCII string.

Impact: Signature verification will fail.

Solution:

```
# WRONG:
commit_hash_bytes = bytes.fromhex(commit_hash)  # ❌ Binary
signature = private_key.sign(commit_hash_bytes, ...)

# CORRECT:
signature = private_key.sign(commit_hash.encode('utf-8'), ...)  # ✅ ASCII
```

12. Not Testing After Changes

Mistake: Making changes without rebuilding and testing Docker container.

Impact: Submission may have bugs that could have been caught.

Solution: Always rebuild with `docker-compose build --no-cache`, test all endpoints, wait

partnr

FAQ

Q1: Can I use a different programming language?

A: Yes! You can use any language and framework. The reference uses Python, but Node.js, Java, Go, Rust, etc. are all acceptable.

Q2: Do I need to push my Docker image to a registry?

A: No, it's optional. The evaluator will build your Docker image from your GitHub repository. However, pushing to a registry (Docker Hub, GHCR) is recommended for faster evaluation.

Q3: What if I regenerate my keys after getting the encrypted seed?

A: If you regenerate your key pair after receiving the encrypted seed, you MUST request a new encrypted seed from the instructor API with your new public key. The old encrypted seed cannot be decrypted with the new private key.

Q4: Can I test the endpoints before submitting?

A: Yes! You should test thoroughly: build and run Docker container, test all three endpoints, verify cron job logging, test container restart for persistence, verify TOTP codes.

Q5: What if my TOTP codes don't match the evaluator's?

A: Check: correct seed (must match student_id + github_repo_url), timezone is UTC, hex seed converted to base32 correctly, using SHA-1 algorithm for TOTP.

Q6: How do I verify my cron job is working?

A: Wait 70+ seconds after container start, then check: `docker exec <container-name> cat /cron/last_code.txt`. Should see multiple lines with timestamps and codes.

Q7: What files should I commit to Git?

A: Commit: all source code, `student_private.pem` (required for Docker), `student_public.pem` (required), `instructor_public.pem` (required), `Dockerfile`, `docker-compose.yml`, dependencies file, `.gitattributes`, `.gitignore`, `cron/2fa-cron`, all scripts. Do NOT commit: `encrypted_seed.txt`.

Q8: What if I get "Decryption failed" error?

partnr

student_id and github_repo_url when requesting seed.

Q9: How do I know if my signature is correct?

A: You can't verify it yourself (you don't have instructor's private key), but verify: commit hash is correct (40-char hex), signature created with RSA-PSS-SHA256, encrypted signature is valid base64, signature encrypted with instructor's public key using RSA/OAEP-SHA256.

Q10: What happens if I submit the wrong encrypted seed?

A: The evaluator will try to decrypt it, but it will either fail to decrypt (if not encrypted with your public key) or decrypt to wrong seed (if from different student_id/repo_url), causing all TOTP codes to be incorrect.

Q11: Can I use a different port than 8080?

A: No. The evaluator expects your API on port 8080. Ensure your Docker container exposes and maps port 8080 correctly.

Q12: What if my container takes too long to start?

A: The evaluator waits up to 30 seconds. Optimize your Dockerfile with multi-stage build, cache dependencies properly, ensure application starts quickly.

Q13: Do I need a health check endpoint?

A: Recommended but not strictly required. A `/health` endpoint that responds quickly helps the evaluator verify the API is ready.

Q14: What if I need to change my repository URL?

A: If you change your repository URL, you MUST request a new encrypted seed from the instructor API with the new URL and use the new seed for testing and submission. Changing URL changes your seed.

Q15: Why do I need to commit my private key? Is this secure?

A: You must commit your private key because your Docker container needs it to decrypt the seed, and the evaluator needs to verify your implementation. However, your keys will be PUBLIC in your repository. These keys are ONLY for this assignment -

do NOT reuse them. Consider them compromised once committed. Generate new

partnr

Outcomes

- Functional microservice with three working REST API endpoints accessible on port 8080
- Successful RSA/OAEP decryption of encrypted seed with proper parameter configuration
- Valid TOTP code generation matching expected values for the given seed
- TOTP code verification with time window tolerance functioning correctly
- Cron job executing every minute and logging valid 2FA codes with UTC timestamps
- Docker container persisting seed data across restarts using named volumes
- Multi-stage Dockerfile optimizing image size and build efficiency
- Docker Compose configuration with proper volume mounts and environment variables
- Commit proof generation with RSA-PSS signature and proper encryption
- All cryptographic operations using correct algorithms and parameters
- Proper error handling with appropriate HTTP status codes and JSON responses
- Working container that can be built, run, tested, and restarted successfully
- Repository containing all required files with proper .gitattributes and .gitignore
- Comprehensive testing demonstrating all features work as specified
- Clear documentation of implementation decisions and setup instructions

---

partnr