Product Specification

MULTI-FACTOR AUTHENTICATION (MFA)

Current Version	1.0
File Name	Multi-factor authentication (MFA)
Requirement unique ID	Sec_Req 1
Responsible / Approver	Richard Ben Aleya
Classification	Public

Vauban project

Document Control

Version	Description	Date	Editor
1.0	Initial Version	24/07/2024	Richard Ben Aleya

Table of contents

1. Introduction 3

1.1. Introduction 3

2. Authentication methods overview 3

2.1. Authenticator Apps (OATH)..... 3

2.2. SMS 3

3. Implementation of OATH (Open Authentication) 4

3.1. Generating the Secret Key..... 4

3.2. Creating the QR Code 5

3.3. Verifying the OTP code 5

1. Introduction

1.1. Introduction

The Vauban project wants the product to implement multi-factor authentication (MFA).

2. Authentication methods overview

Hereby are the details on the MFA methods that will be supported by the product:

2.1. Authenticator Apps (OATH)

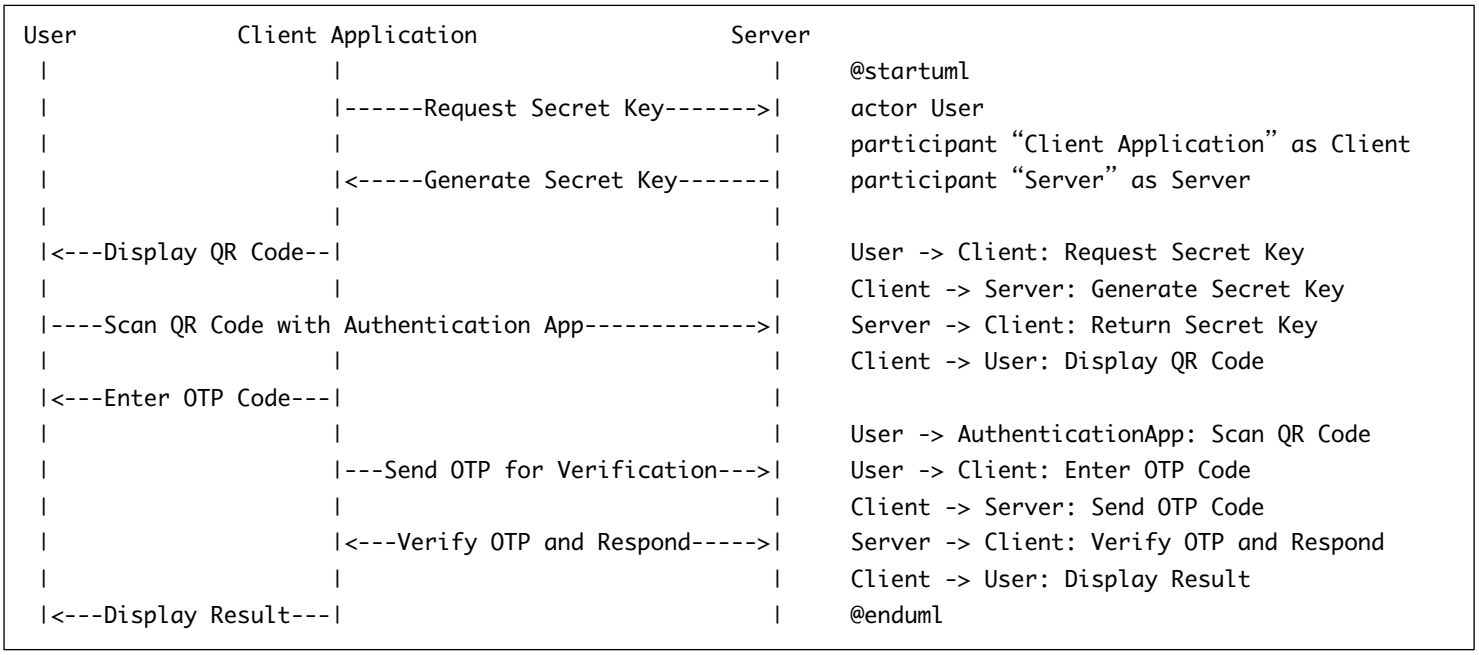
- **Description:** OATH (Initiative for Open Authentication) is a set of standards for strong authentication. Authenticator apps like Google Authenticator, Microsoft Authenticator, and Authy implement OATH standards to generate OTPs.
- **How It Works:**
  - OATH apps can generate TOTP or HOTP (HMAC-based One-Time Password) codes.
  - The user scans a QR code or enters a code provided by the service to link the app to their account.
  - After setup, the app generates temporary codes that the user must enter to log in.
- **Usage:** Used as a secondary authentication method (2FA) by many online services.
- **Advantages:**
  - Works without a network connection after initial setup.
  - More secure than SMS since codes are generated locally on the user’s device.
- **Disadvantages:**
  - The user must have a smartphone or compatible device.
  - Loss or replacement of the device requires reconfiguration.
- **Security:** Authenticator apps based on OATH are considered very secure, especially when using TOTP, as they are not vulnerable to network interceptions.

2.2. SMS

- **Description:** An authentication code is sent via SMS to the user's phone number. The user must enter this code to complete the authentication process.
- **Usage:** Often used as a second factor of authentication (2FA) in addition to a password.
- **Advantages:**
  - Easy to use and deploy.
  - Any mobile phone can receive SMS, eliminating the need for additional installations.
- **Disadvantages:**
  - Vulnerable to interception attacks (such as SIM swap attacks).
  - Dependent on network coverage and mobile service providers.
- **Security:** Although convenient, SMS is considered less secure compared to other methods due to the vulnerabilities mentioned.

3. Implementation of OATH (Open Authentication)

To implement OATH (Initiative for Open Authentication) in the application and enable authentication applications such as Google Authenticator, Microsoft Authenticator, or Authy to generate OTP (One-Time Passwords), here's how to proceed. The chosen algorithm is the TOTP (Time-Based One-Time Password) algorithm, which is the most widely used.



3.1. Generating the Secret Key

The first step is to generate a secret key for each user. This secret key will be shared between the server and the user's authentication app. The **generate_secret_key()** function could generate the secret key.

```
[dependencies]
rand = "0.8"
base32 = "0.4"

use rand::RngCore;
use base32::Alphabet;
use base32::encode;

fn generate_secret_key() -> String {
    // Create a buffer to hold 20 bytes of random data
    let mut secret = [0u8; 20];

    // Fill the buffer with random bytes
    rand::thread_rng().fill_bytes(&mut secret);

    // Encode the random bytes in Base32
    let encoded_secret = encode(Alphabet::RFC4648 { padding: false }, &secret);

    encoded_secret
}
```

3.2. Creating the QR Code

Provide a QR code that the user can scan with the authentication app. This can be done using for example the 'qrcode' crate. The **generate_qr_svg()** function will output a SVG content.

```
[dependencies]
qrcode = "0.8"

extern crate qrcode;
use qrcode::QrCode;
use qrcode::render::svg;

fn generate_qr_svg(secret: &str) -> String {
    // Create the data for the QR code (e.g., TOTP URL)
    let data = format!("otpauth://totp/Vauban:username@company.com?secret={}&issuer=YourApp", secret);

    // Create the QR code
    let code = QrCode::new(data).unwrap();

    // Render the QR code as an SVG string
    let svg_string = code.render::<svg::Color>().min_dimensions(200, 200).max_dimensions(200, 200).build();

    svg_string
}
```

3.3. Verifying the OTP code

When the user enters the OTP code generated by the authentication app, we need to verify this code on the server for example with this **verify_otp()** function.

```
[dependencies]
oath = "0.16.1"

extern crate oath;
use oath::totp_raw_now;
use oath::HashType;

fn verify_otp(secret: &str, otp: &str) -> bool {
    // Decode the base32 secret key
    let secret_bytes = base32::decode(base32::Alphabet::RFC4648 { padding: false }, secret)
        .expect("Invalid Base32 string");

    // Generate the current TOTP code
    let current_otp = totp_raw_now(&secret_bytes, 6, 0, 30, &HashType::SHA1).to_string();

    // Compare the provided OTP with the generated OTP
    current_otp == otp
}
```

The result of the **verify_otp()** function will determine the success or failure of the second-factor authentication. This function compares the provided OTP with the generated OTP using the secret key, ensuring that only the correct OTP will validate the authentication process.

Vauban project

TBC