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

Document Control

Version	Description	Date	Editor
1.0	Initial Version	25/07/2024	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

4. IMPLEMENTATION OF SMS ONE-TIME PASSWORD 6

4.1. Sending OTP code via Twilio API 6

4.2. Verifying the OTP code with the code stored in the OTP store 7

4.3. Example of usage of functions send_sms() and verify_otp() 7

5. CONCLUSION 8

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)

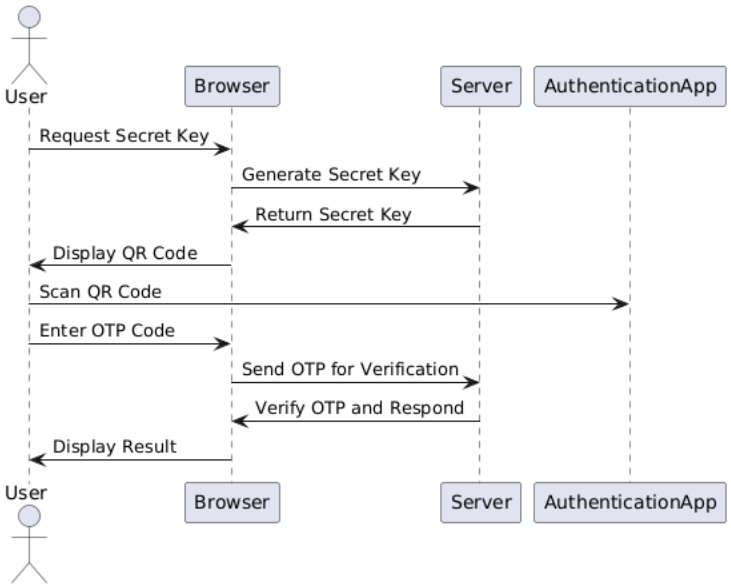
- **Description:** OATH (Initiative for Open Authentication) is a set of standards for strong authentication. Authenticator apps like Google Authenticator, Microsoft Authenticator, and Authy implement OATH standards to generate OTPs.
- **How It Works:**
  - OATH apps can generate TOTP or HOTP (HMAC-based 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.




```
@startuml
actor User
participant "Browser" as Client
participant "Server" as Server

User -> Client: Request Secret Key
Client -> Server: Generate Secret Key
Server -> Client: Return Secret Key
Client -> User: Display QR Code

User -> AuthenticationApp: Scan QR Code
User -> Client: Enter OTP Code
Client -> Server: Send OTP for Verification
Server -> Client: Verify OTP and Respond
Client -> User: Display Result
@enduml
```

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 return 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

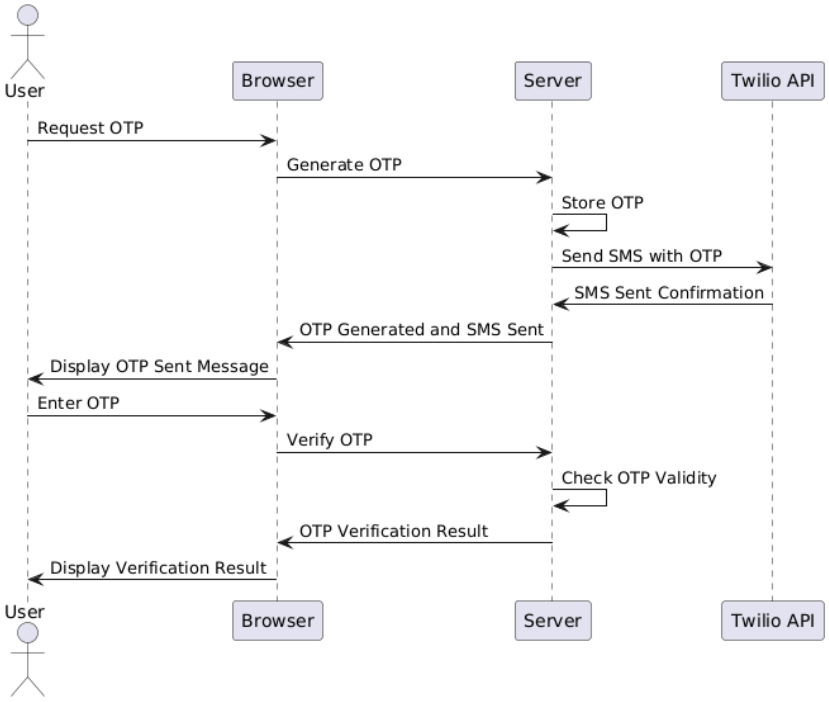
When the user enters the OTP code generated by the authentication app, we need to verify this code on the server for example with this **verify_otp()** function.

```
[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.

4. Implementation of SMS One-Time Password

The method is comparable to OTP verified via OATH, except that the OTP state is maintained on the server (OTP store) and the transport media is the GSM network. In this specification document, we will assume that the third-party service used will be Twilio.




```
@startuml
actor User
participant "Browser" as Client
participant "Server" as Server
participant "Twilio API" as Twilio

User -> Client: Request OTP
Client -> Server: Generate OTP
Server -> Server: Store OTP
Server -> Twilio: Send SMS with OTP
Twilio -> Server: SMS Sent Confirmation
Server -> Client: OTP Generated and SMS Sent
Client -> User: Display OTP Sent Message

User -> Client: Enter OTP
Client -> Server: Verify OTP
Server -> Server: Check OTP Validity
Server -> Client: OTP Verification Result
Client -> User: Display Verification Result
@enduml
```

4.1. Sending OTP code via Twilio API

The asynchronous **send_sms()** function will send the SMS containing the OTP code via Twilio API.

```
[dependencies]
reqwest = { version = "0.11", features = ["json"] }
tokio = { version = "1", features = ["full"] }

use reqwest::Client;
use std::env;

pub async fn send_sms(to_phone: &str, otp: &str) -> Result<(), Box<dyn std::error::Error>> {
    let account_sid = env::var("TWILIO_ACCOUNT_SID").expect("TWILIO_ACCOUNT_SID not set");
    let auth_token = env::var("TWILIO_AUTH_TOKEN").expect("TWILIO_AUTH_TOKEN not set");
    let from_phone = env::var("TWILIO_PHONE_NUMBER").expect("TWILIO_PHONE_NUMBER not set");

    let message = format!("Your verification code is: {}", otp);
    let url = format!("https://api.twilio.com/2010-04-01/Accounts/{}/Messages.json", account_sid);

    let client = Client::new();
    let params = [("To", to_phone), ("From", &from_phone), ("Body", &message)];

    let response = client.post(&url)
        .basic_auth(account_sid, Some(auth_token))
        .form(&params)
        .send()
        .await?;

    if response.status().is_success() { Ok(()) } else {
        Err(format!("Failed to send SMS: {}", response.text().await?).into())
    }
}
```

4.2. Verifying the OTP code with the code stored in the OTP store

The **verify_otp()** function will compare the user's OTP code with the code stored in the OTP store.

```rust
use std::collections::HashMap;
use std::time::{SystemTime, Duration};
use std::sync::Mutex;

struct OtpStore {
    otps: Mutex<HashMap<String, (String, SystemTime)>>,
}

impl OtpStore {
    fn new() -> Self {
        OtpStore {
            otps: Mutex::new(HashMap::new()),
        }
    }

    fn store_otp(&self, user_id: &str, otp: &str) {
        let mut otps = self.otps.lock().unwrap();
        otps.insert(user_id.to_string(), (otp.to_string(), SystemTime::now()));
    }

    fn verify_otp(&self, user_id: &str, otp: &str) -> bool {
        let otps = self.otps.lock().unwrap();
        if let Some((stored_otp, timestamp)) = otps.get(user_id) {
            if stored_otp == otp && timestamp.elapsed().unwrap() < Duration::from_secs(300) {
                return true;
            }
        }
        false
    }
}
```

4.3. Example of usage of functions send_sms() and verify_otp()

Here's an example how we can use these functions together.

```rust
// Initialize OTP store
let otp_store = OtpStore::new();

// Generate an example OTP (in a real scenario, generate a random OTP)
let user_id = "user123";
let otp = "123456";

// Store OTP
otp_store.store_otp(user_id, otp);

// Send SMS
match send_sms("+1234567890", otp).await {
    Ok(_) => println!("SMS sent successfully"),
    Err(e) => eprintln!("Error sending SMS: {}", e),
}

// Simulate user input for OTP verification
let user_input_otp = "123456";
if otp_store.verify_otp(user_id, user_input_otp) {
    println!("OTP verified successfully");
} else {
    println!("Invalid or expired OTP");
}
```

5. Conclusion

In conclusion, implementing multi-factor authentication (MFA) via OATH (Initiative for Open Authentication) and SMS is not only feasible but also reliable, secure, and well within our technical capabilities. Utilizing OATH standards for generating one-time passwords (OTPs) ensures a robust and universally recognized approach to enhancing security. Applications like Google Authenticator and Microsoft Authenticator can seamlessly integrate with the product, providing users with an additional layer of protection against unauthorized access.

Furthermore, incorporating SMS-based verification adds another dimension to our authentication process. By leveraging reputable SMS service providers like Twilio, we can ensure timely and secure delivery of OTPs to users' mobile devices. This method is particularly advantageous for users who may not have access to or prefer not to use authenticator apps.

Both methods—OATH and SMS—are designed to be user-friendly while maintaining high security standards. The integration of these technologies will significantly reduce the risk of security breaches by requiring multiple forms of verification, thus ensuring that only authorized users gain access to sensitive access.

By implementing MFA through these dual approaches, we are taking a proactive stance in safeguarding the information system. The combination of OATH for app-based authentication and SMS for additional verification provides a comprehensive, flexible, and highly secure solution that can be confidently deployed in any infrastructure.

*** End of specification document ***