Labo 2 - Rust web server

SLH

Timothée Van Hove

6 janvier 2024

Table des matières

Introduction

The goal of this lab is to manage the authentication of a simple website. The authentication is managed with JWT access and refresh tokens. The only functionalities of the website are to register, login, and change password.

Question1

> *What is the purpose of a JWT access token? And what is the purpose of a JWT refresh token? Why do we have both?*

In a typical web application, our objectives are:

1. **Seamless User Experience**: We aim to keep users logged in for as long as possible without interruptions. Think of how you stay logged in on platforms like LinkedIn or Google Drive for months at a stretch.
2. **Robust Security**: While ensuring user convenience, we must not compromise on security.

Starting with the first goal, let's consider **persistent login**. When a user registers or logs in on our web app, our server responds with an **access token**. To facilitate a long-lasting session, we could set this token's expiry to, say, one year. Each time the user performs actions requiring authentication, this token is sent with the request.

This approach nails our first goal. But what about security? If an attacker gets hijack this access token, they can impersonate the user for a long duration. we have a huge security problem!

The solution? Short-lived access tokens. Ideally, these should expire quickly – In our application, 2 minutes. An attacker hijacking a short-lived token is less problematic, as their window of opportunity is short. But, if the user's access expires every 2 minutes and they have to login again, the UX is ugly.

Enter refresh tokens. When a user logs in, they receive two tokens: a short-lived access token and a long-lived refresh token. The refresh token doesn't grant access directly but can be used to obtain a new access token. As an access token nears expiry, our web app silently requests a new one using the refresh token, keeping the user experience smooth and uninterrupted.

By combining short-lived access tokens with securely managed refresh tokens, we strike a crucial balance between user convenience and robust security, ensuring our web application remains both user-friendly and secure.

But what if someone hijacks the refresh token?

While less likely, it's still a risk, particularly in less secure environments like browser-based applications. To mitigate this, we could implement a strategy known as refresh token rotation, recommended by the Internet Engineering Task Force. **This rotation is not implemented** in our web app, but here is how it would work:

- Each time a user logs in, new access and refresh tokens are generated and sent to the client. The new refresh token replaces the old one in the database.
- Each refresh token can be used only once. If a token is reused (indicating potential theft), the server immediately locks the user's account and alerts the support team.
- If an attacker uses the stolen refresh token before the legitimate user, the subsequent legitimate refresh attempt fails, triggering account lockout and alerting us to the breach.

This method would ensures that even if a refresh token is compromised, the damage is contained, and we're alerted to the breach promptly. Users can then re-authenticate after we've addressed any security concerns.

Question 2

Where are the access token stored? Where are the refresh token stored ?

Access tokens

Stored in the client's browser (cookie) and sent with each request that requires authentication.

Refresh tokens

Stored on the client side in the local storage. The local storage is a web storage object that allows JavaScript sites and apps to store right in the browser with no expiration date. This means the data stored in the browser will persist even after the browser window is closed.

Question 3

Is it a good idea to store them there? Is there a better solution?

Access tokens

Yes, it is a good idea to store access tokens in a cookie and are automatically sent with every request to the same domain, which is easy for maintaining a session. They are configured as **HttpOnly**, preventing JavaScript access and mitigating the risk of XSS attacks, and use the **SameSite** attribute to prevent them from being sent in cross-site requests. In addition with the usage of anti-CSRF tokens, it mitigates CSRF attacks.

Refresh tokens

No, it is not a good idea to store the refresh tokens on the client side. While, it is easier to use and persist across browser sessions, it is vulnerable to XSS attacks. If a script can inject malicious code into our application, it can steal tokens from local storage. We should store the refresh tokens in the database (server-side) and implement the refresh token rotation strategy as mentioned in the Question 1.

DB not saved after a user password change

In the given code base, the new password is never saved in the mocked database. I don't know if it was a deliberate implementation choice, but if we want to fix this behavior, we have to modify the `change_password` function:

```rust
pub fn change_password(email: &str, new_hash: &str) -> Result<bool> {
        info!("Change password of user");
        let mut db = DB.write().or(Err(anyhow!("DB poisoned")))?;

        let user = match db.get_mut(email) {
            None => {
                trace!("User not found");
                return Ok(false)
            },
            Some(u) => u,
        };

        user.hash = new_hash.to_string();

        trace!("Password changed");

        save(db).ok(); // Add this line
        Ok(true)
    }
```