Authentication Project: 2FA Using Usernames, Passwords and Time-based One-time Passwords

Ward M. Zahran

Friday 10th May, 2024

Contents

1 Team Members

1. Ward M. Zahran.

2 Brief Description

This project aims to implement user two-factor authentication for a web application using usernames and password along with time-based one-time passwords as a second factor.
The project uses Go and SQlite3 and TOTP as defined in RFC 6238.

3 Primer on How TOTP Works

TOTP (Time-Based One-Time Password) is an algorithm defined in RFC 6238 which extends the HOTP (HMAC-Based One-Time Password) as defined in RFC 4226. We will explain the latter then explain the former.

3.1 HOTP

HOTP defines the following symbols:

- A symetric secret key (K).
- A hash function (H) the default being SHA-1.
- A counter value (C) in order to generate one time passwords.
- A HOTP value length (D) (6–10, default is 6, and 6–8 is recommended).

These values must be agreed upon by both parties, in our case the webapp and the user.
The algorithm uses the following formula in order to get a HOTP value:

$HOTPvalue = HOTP(K, C) mod 10^D$
$HOTP(K, C) = truncate(HMACH(K, C))$
$truncate(MAC) = extract31(MAC, MAC[(198 + 4) : (198 + 7)])$
$extract31(MAC, i) = MAC[(i8 + 1) : (i8 + 481)]$

1. We generate a HOTP value from the HTOP(K, C) modulo 10 to the power of the key length D.
2. The HOTP function first generates the HMAC using K and C and then truncates it.
3. The truncate function does some math on the hash and then produces the value.
4. Once the value generated both parties increment the counter C independently.

3.2 TOTP

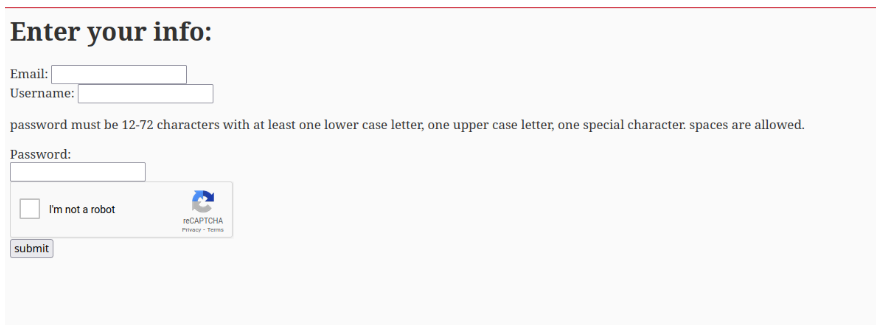
TOTP uses the same exact algorithm but instead of a counter we use a new value based on time which we will call T.
So our algorithm ends up being: $HOTPvalue = HOTP(K, T) mod 10^D$

Per RFC 6238:
T = (Current Unix time - T0) / X
where:

- T0 is is the Unix time to start counting time steps (default value is 0, i.e., the Unix epoch). Meaning an agreed upon time from which to start counting the time.
- X is the time step in seconds (default value X = 30 seconds). Meaning the amount of time before each code changes.

4 Implementation

My implementaiton was using Go with several libraries most important of which are the SQLite3, otp, http. As well as libraries we used various technologies in order to prevent automated attacks such as Captcha.
The user can register a new account through the following protal:



Enter your info:

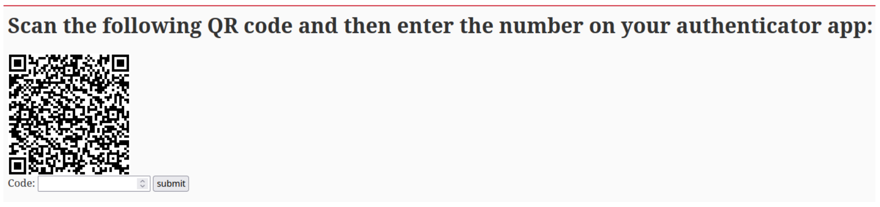
Email:
Username:

password must be 12-72 characters with at least one lower case letter, one upper case letter, one special character. spaces are allowed.

Password:

Figure 1: User registration.

Notice how we have implemented a captcha to disable autommated attacks from flodding the database with garbage information.
After the registration is done and user tries to login they will be faced with this page:

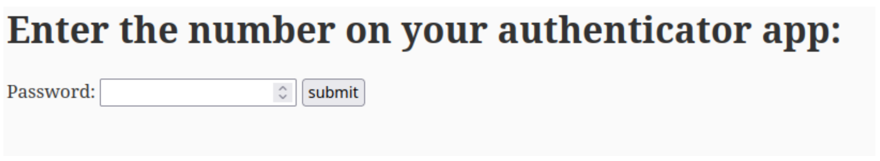


Scan the following QR code and then enter the number on your authenticator app:

Code: submit

Figure 2: 2FA setup page

The user cant access the rest of the website until he has scanned the QR code and used the OTP.
And on every subsequent login they will be faced with a different page without the QR code where they must enter the OTP. Notice the following figure:



Enter the number on your authenticator app:

Password: submit

Figure 3: Login after verification.

The workflow of how the login process works are as follows:

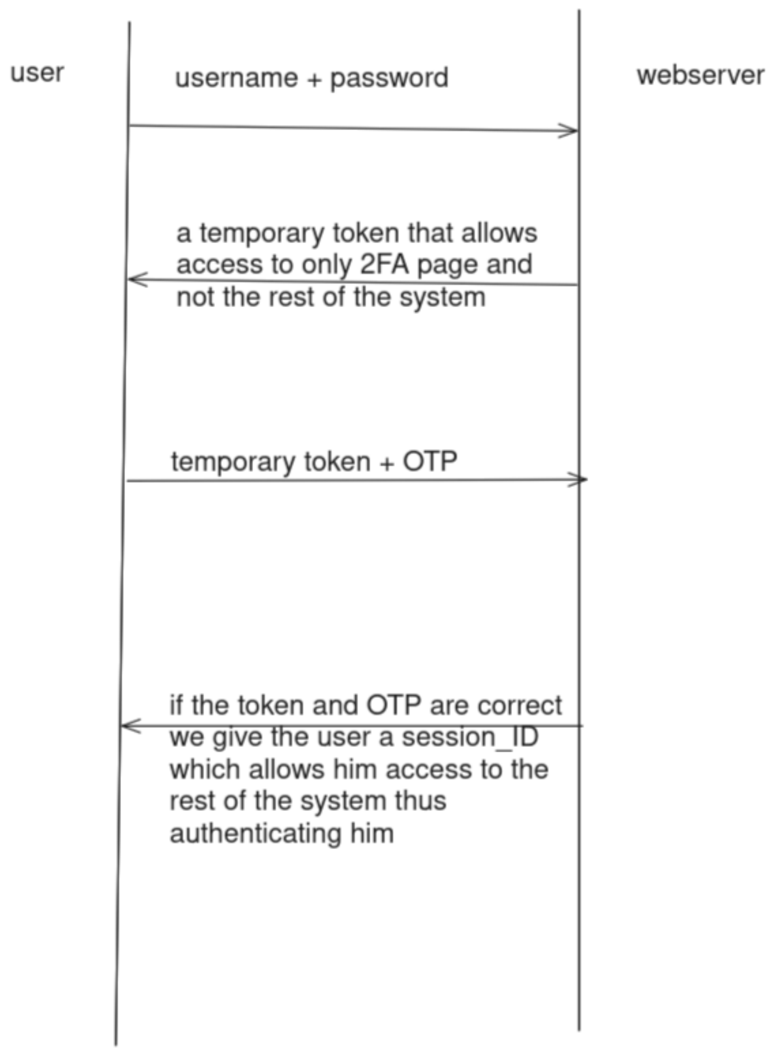
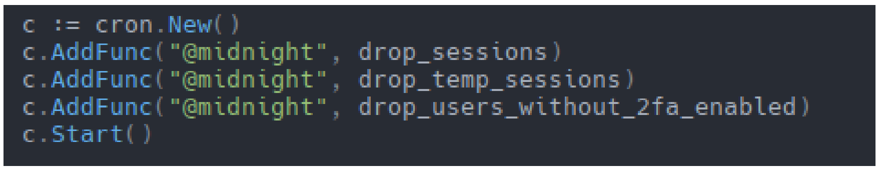

user
username + password
webserver
a temporary token that allows access to only 2FA page and not the rest of the system
temporary token + OTP
if the token and OTP are correct we give the user a session_ID which allows him access to the rest of the system thus authenticating him

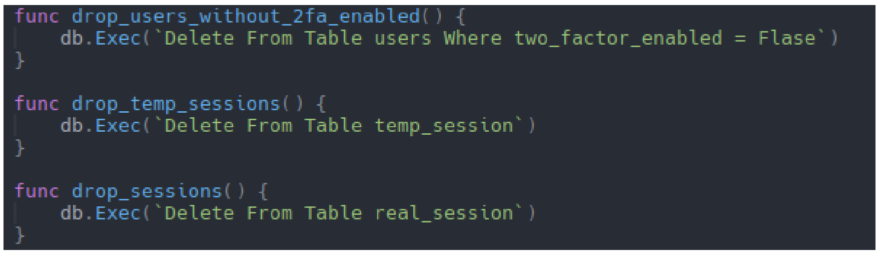
Figure 4: Control flow

After 3 invalid submissions the temporary token is invalidated thus protecting the system from brute force attacks. And aquiring a new temporary token is to automate because of the Captcha at the login page.
Other protections include Cron Jobs which automatically remove any unverified users at every midnight as well as purgin expired tokens and sessions.



```
c := cron.New()
c.AddFunc("@midnight", drop_sessions)
c.AddFunc("@midnight", drop_temp_sessions)
c.AddFunc("@midnight", drop_users_without_2fa_enabled)
c.Start()
```

Figure 5: Cron Jobs.



```
func drop_users_without_2fa_enabled() {
    db.Exec(`Delete From Table users Where two_factor_enabled = Flase`)
}

func drop_temp_sessions() {
    db.Exec(`Delete From Table temp_session`)
}

func drop_sessions() {
    db.Exec(`Delete From Table real_session`)
}
```

Figure 6: The functions run in those Cron jobs.

As well as that we use envoriment variables rather than hard coding credentails into the server itself which protects them incase the software itself gets leaked.