Two-factor authentication overview

Objective

The objective is to implement two-factor authentication (hereafter referred to as 2FA). 2FA is a second layer of authentication after the user has logged in as they usually would using a username and password. This can take the form of a scratch-card, a security token, or in our case, an app.

The aim is to improve security by implementing this. It could allow us to later down the line allow users to “digitally sign” submitted documents.

Table of Contents

Terminology

- **2FA**: Two Factor Authentication
- **Shared secret**: data known only by certain parties. Can be used to verify the identity of a person. May be simply referred to as “secret” for brevity. Each secret is tied to a certain device AND user. The device may be in use by more than one user.
- **Public key**: An encryption key that, when used to encrypt data, renders encrypted data that can only be decrypted using a corresponding **private key**.
- **Volatile memory**: Memory that, when it loses power, loses all the information stored therein. Contrast non-volatile memory, such as hard drives and CDs. Volatile memory generally refers to RAM.

User stories

User that choses to enable 2FA

A user decides to enable 2FA. They open their app, and open the menu. They go to a screen labeled “Two factor authentication”. There, they find a toggle for enabling 2FA. They slide it to the right. It works briefly, then it turns green. Two buttons are now also available on this screen: “Authenticate for web”, and “Authenticate for mobile”. They then try to log into the webpage, to which they are not logged in. They put in their username and password. Instead of letting them in right away, the web page asks them to open one of their 2FA enabled devices and authenticate for web. Their cell phone buzzes, telling them that they have an authentication request. They open the app, and navigate through to where they can press the button labeled “Authenticate for web”. The holding screen goes away, and they’re now logged in.

Users forced to use 2FA

A user tries to use the website. They will find that they are not logged in. They type in their username and password, and hit enter. They’re presented with a screen that instructs them to go to their cell phone. They open their cellphone, and find that it too is not logged in. They log in, which may take longer than usual, and the app tells them that they now have two factor authentication, on request of their organization. They go to the website and attempt to log in again. This time, they’re presented with a holding screen, telling them to open their phone and press a button labeled “Authenticate for web”. Their phone buzzes, letting them know they have an authentication request. They navigate to the authentication screen, and press “Authenticate for web”. The holding screen goes away, and they’re now logged in.

User-initiated flows

All the flows described below are completely user-initiated, as opposed to ones that are a following step of another flow.

First logon for user of organization that forces 2FA

A user without a secret logs in after their organization has decided to force the use of 2FA. The web service prompts them to log in via the app, or contact an administrator if they cannot. The web service does not consider them logged in. The user provides correct login credentials using the app. The app, upon getting a certain response from the login resource, realizes that the organization they belong to requires 2FA to be enabled. From here, all actions in the “App enabling 2FA” flow take place. After this, the app logs in using the previously provided login, username, and also submits the second factor automatically, essentially following the “Login via app for 2FA enabled user” flow.

Login via app for 2FA enabled user

A user enters their (correct) username and password into the app. It makes a request to the server using these. The server checks its databases, finds that the user and password are correct, and finds one or more secrets for this user. It responds that it wants further authentication. The app checks to see if a locally stored secret for this username exists. The service stores the IP from which the login attempt originated and the time of attempted login, such that when/if the user submits their second factor by pressing the authenticate button in the app, it can be confirmed that it is a timely login attempt[1]. However, if a login attempt already exists, the user is asked to cancel it or wait until it expires, and the login fails.

If it does not, the user must authenticate using a second device, and tells the user this. It shows the user a holding screen until the user authenticates using a second device. This holding screen should have a cancel button. The flow for pressing this button is described in “Canceling authentication”. The service sends out a push notification to all devices (which are logged in) that the user has stored a secret on. The flow then continues and terminates according to “Providing second factor”, for the second device.

If, however, a locally stored secret for this username exists on the first device, the secret is decrypted using the users password. If it cannot be decrypted using a valid password, the users secret is deleted and the user is asked to authenticate using a different device, following the flow in the above paragraph[2]. Otherwise, the secret is then encrypted using the servers public key, turning it into a message readable only by the server. It sends this to the server. The server checks if the secret has been sent in a timely manner- if this is not the case, it effectively ignores it and sends a response saying that the authentication request was too slow. If the secret is of a valid format but not valid for this user and device, the server responds that the apps secret is out of date and should be deleted- the app should do this and prompt the user to authenticate using a different device. It should also be mentioned to the user that if they cannot do this, they should then reset their password. If the secret is valid and not out of date, the user is let in. Note that the users password will need to be kept in volatile memory for later authentication.

Login via webpage for 2FA enabled user

A user enters their (correct) username and password into the webpage. Their browser makes a request using these. The server checks its database, finds that the given credentials are correct, and finds one or more secrets for this username. It redirects the user to a holding page until the user authenticates using the app. This holding screen should have a cancel button. The flow for pressing this button is described in “Canceling authentication”. The service stores the IP from which the login attempt originated and the time of attempted login, such that when/if the user submits their second factor by pressing the authenticate button in the app, it can be confirmed that it is a timely

1 This authentication request should be stored in separately from login requests for the web service.
2 This is caused by the user changing their password. As such, the secret won’t be able to be decrypted.

login attempt[3]. It also sends out a push notification to all devices (which are logged in) that the user has stored a secret on. However, if a login attempt already exists, the user is asked to cancel it or wait until it expires, and the login fails. The flow continues in “Providing second factor”.

Reset password

A user prompts the reset of their password, either via app or website. A reset link is generated and sent to their email. After a certain period of time, the link is rendered invalid. If the user visits this link before this period of time has passed, they are prompted to input a new password. The password is checked for validity (not too long, not too short, not “bad”?). If it’s invalid, the procedure is repeated until the user finds a suitable password or the timeout is reached. If the user has shared secrets stored on the server, they are deleted. If the user has any login attempt timestamps, they are deleted. The users password is updated, and they are asked to log in again using their new password. The three previous sentences are the most important additions, as I suspect we already have a flow for password resets. Appending those three sentences to those flows may be sufficient.

User activating 2FA

In the app, a logged in user opens an options menu. A check is made to see if a secret is stored locally. As this is not the case, the toggle for 2FA is shown as “off”. The user presses a toggle in an options menu. From here, all actions in the “App enabling 2FA” flow take place.

User de-authenticating device

In the app, a logged in user opens an options menu. A check is made to see if a secret is stored locally. As this is the case, the toggle for 2FA is shown as “on”. A check to the server is made to see if the user is part of an organization that forces 2FA. If this is the case, the toggle is greyed out and no further action can be taken. Otherwise, the user presses a toggle in an options menu. The app sends a request to the server to delete its secret. The server receives the request and double-checks to see that the user is not a part of an organization that forces 2FA. If the user is allowed to remove 2FA, the server deletes the secret, and responds that the device should do so as well. Otherwise, no action is taken and the flow ends here.

Enabling forced 2FA for users from organization

An administrator, on request by an organization, changes the setting for forced 2FA for the organization via the ADM. The organization is updated in the database. All currently logged in users that haven’t already enabled 2FA, who belonging to that organization are (by neccessity) logged out, as to ensure that everyone in the organization has 2FA from this point on.

User following email link for automatic login

Nothing should happen, and the user should be told to login as they normally would. Preferably, change this kind of reminder upon enabling 2FA.

---

3 This authentication request should be stored in separately from login requests for the mobile service.

Non-user initiated flows

All the flows described below are part of some user-initated flow.

App enabling 2FA

A (secure) request is made towards the web service, which generates and stores a secret on its side. The app stores the servers response (the secret) locally, in a way that is tied to a user (in a file with the users username as the file name or in a database table where the primary key is the username), encrypted using the users password.

Disabling forced 2FA for users from organization

An administrator, on request by an organization, changes the setting for forced 2FA for the organization via the ADM. The organization is updated in the database. No further changes are needed.

Providing second factor

The user then opens their app on a device with one of their secrets on it. They log in if they are not already logged in, following the flow of “Login via app for 2FA enabled user”. They press a button corresponding to the service they want to authenticate for. From here, the flow closely follows that of the app auth flow.

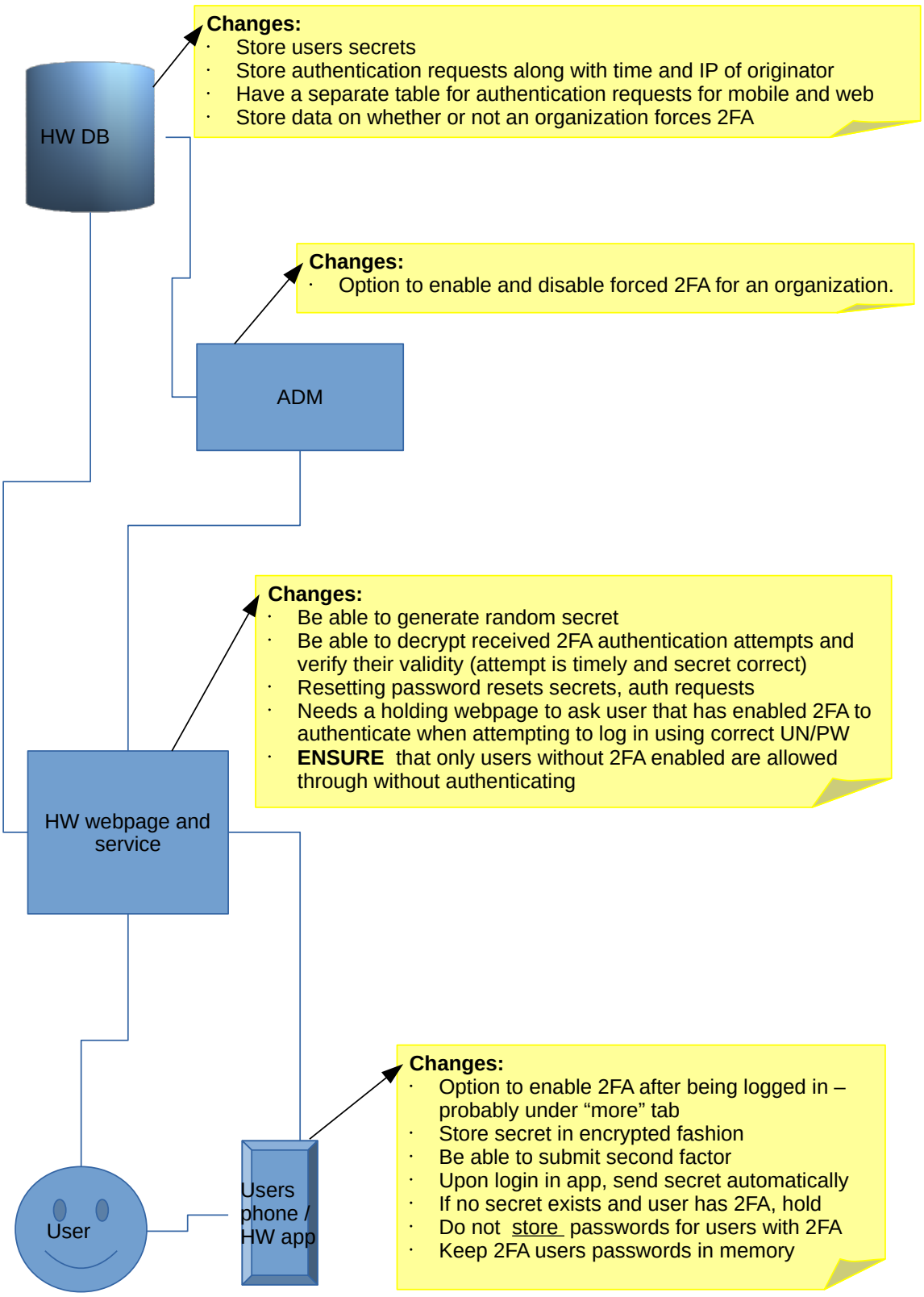
The users secret on the device is decrypted using the users password[4]. The secret is then encrypted using the servers public key, turning it into a message readable only by the server. It sends this to the server. The server checks if the secret has been sent in a timely manner- if this is not the case, it effectively ignores it and sends a response saying that the authentication request was too slow and the user will have been redirected away from the holding page. If the secret is of a valid format but not valid for this user and device, the server responds that the apps secret is invalid and should be deleted- the app should do this and prompt the user to authenticate using a different device. It should also be mentioned to the user that they cannot do this, they should then reset their password. If the secret is valid and not out of date, the user is let in. If no device can successfully log in and authenticate in time, the user is simply returned to the front page.

Canceling authentication

A request to the server containing the users username is sent. The server then checks if it originated from an IP that is on record in the authentication attempt database. If it is, it then checks if any of those authentication attempts are for this user. If this is the case, it removes those authentication requests. The server should respond that everything is okay if it found a corresponding record and removed it. The user is then redirected away from the holding page, and the login flow is stopped. If no record was found, the server should respond with a different response, and the user should not be

4 Retrieved from volatile memory where it is stored; see end of “Login via app for 2FA enabled user”.

directed away from the holding page. This is to avoid the user being unable to cancel an authentication request. If the users device somehow is completely out of sync with the service, it will eventually return away from the holding screen due to timeout, or the user will simply kill the app and be able to attempt login again.


HW DB
Changes:
· Store users secrets
· Store authentication requests along with time and IP of originator
· Have a separate table for authentication requests for mobile and web
· Store data on whether or not an organization forces 2FA
ADM
Changes:
· Option to enable and disable forced 2FA for an organization.
HW webpage and service
Changes:
· Be able to generate random secret
· Be able to decrypt received 2FA authentication attempts and verify their validity (attempt is timely and secret correct)
· Resetting password resets secrets, auth requests
· Needs a holding webpage to ask user that has enabled 2FA to authenticate when attempting to log in using correct UN/PW
· ENSURE that only users without 2FA enabled are allowed through without authenticating
User
Users phone / HW app
Changes:
· Option to enable 2FA after being logged in – probably under “more” tab
· Store secret in encrypted fashion
· Be able to submit second factor
· Upon login in app, send secret automatically
· If no secret exists and user has 2FA, hold
· Do not store passwords for users with 2FA
· Keep 2FA users passwords in memory