LUMENSHINE

WALLET SECURITY WHITE PAPER

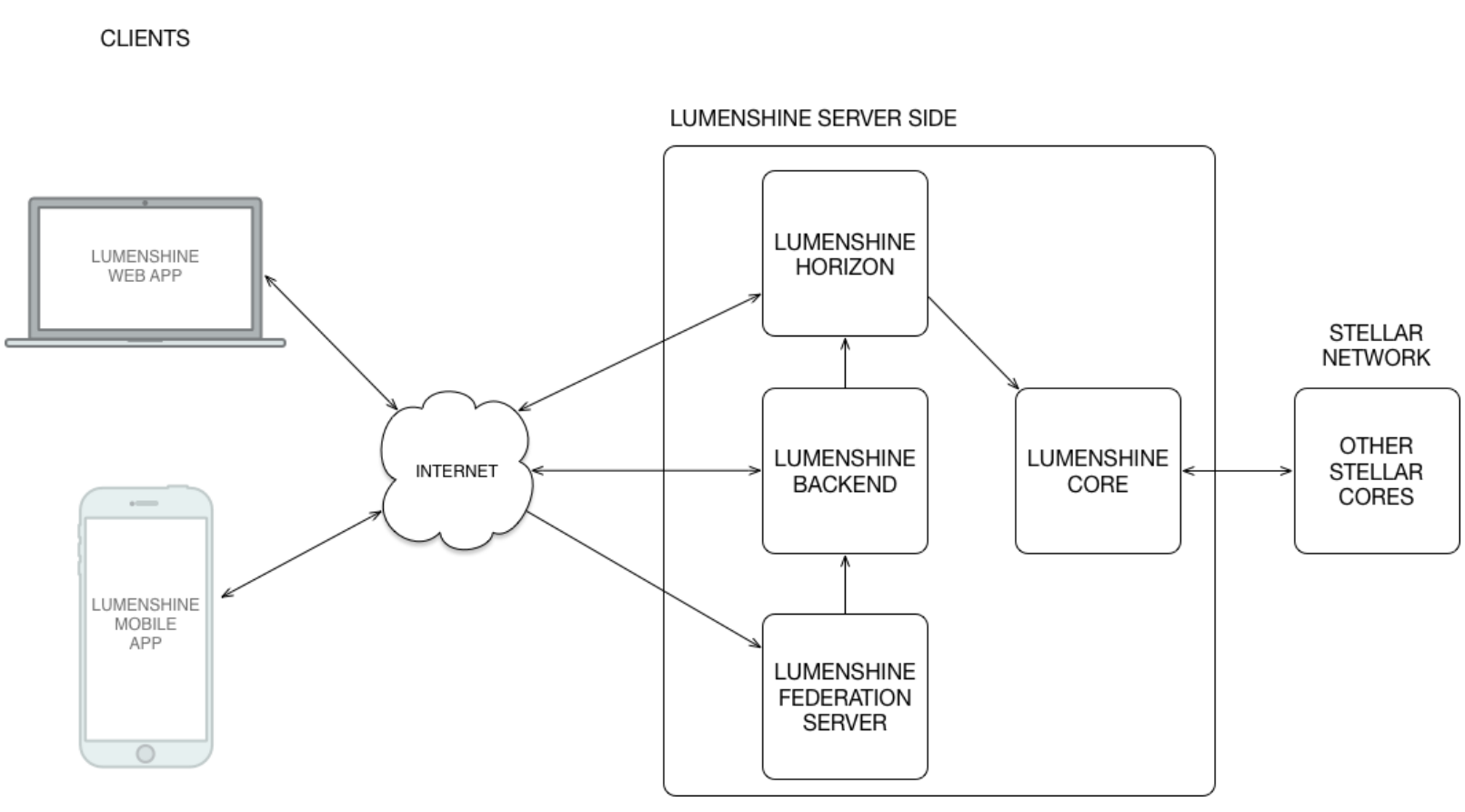
V 1.0

JANUARY 2019

INTRODUCTION

This document describes how the Lumenshine wallet manages security topics. It describes the security measurements that the Lumenshine wallet applies at registration, setup, login, and how it stores and uses the encrypted sensitive data of its users.

LUMENSHINE COMPONENTS



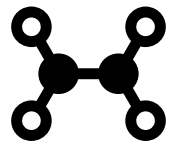
The Lumenshine wallet consists of clients (currently Web-App and iOS App) and server side components (Lumenshine horizon server, Lumenshine backend server, Lumenshine federation server and Lumenshine Core).

The Lumenshine clients communicate with the servers via encrypted HTTP/2 HTTPS. They use the Lumenshine backend server to register users, to login users and to request specific user-data such as wallets, contacts, personal data, encrypted mnemonic, etc. The clients use the Lumenshine horizon server to connect and communicate with the Stellar network. They query horizon to get data connected to the users Stellar accounts from the Stellar network. Clients also use horizon to submit transactions (such as payments) to the Stellar network. The clients never communicate directly with the Lumenshine Core.

The Lumenshine horizon server is connected to the Lumenshine Core. The Lumenshine Core is a Stellar validator node. By running a validator node, Lumenshine participates in consensus, adding more security to the decentralized Stellar network. Lumenshine sends transactions to the network without depending on a third party - providing its users a trusted node to submit transactions to - and see the state of their Stellar accounts.

The Lumenshine backend server is responsible for storing user-specific data and for providing util data to the clients such as charts data. It does not send transactions to the Stellar network, but it connects to the horizon database to set-up customisations, that will automatically trigger actions on events that occur in the Stellar network (such as push notifications on payment received or server sent events, SSE, to the clients via web-sockets).

The Lumenshine federation server connects to the Lumenshine backend database to be able to map Stellar addresses that have “lumenshine.com" as a domain name (ex. john*lumenshine.com) to corresponding Stellar account ids. It is a separate server that conforms to the Stellar federation protocol defined in SEP002. It is also used by other, external wallets, that do not need to authenticate to the Lumenshine backend to receive the needed mapping.



REGISTRATION AND SETUP

This chapter describes the security measurements applied during the registration and setup process.

SIGN-UP

A new user must first sign-up to be able to use the Lumenshine wallet app. On registration, the client requests the first name, last name, email-address and password from the new user. The user must provide a strong password, that must have at least 9 characters, must contain at least one number, one small and one capital character. As soon as the user provides that mandatory data, the client generates a 24 words backup passphrase, also called “mnemonic” for the new user. To generate the mnemonic the client selects 24 random words form the industry standard word list that can be found in BIP0039. The mnemonic is later used by the Lumenshine clients to create new Stellar accounts and to sign Stellar transactions on behalf of the user. To do so, Lumenshine uses the key derivation methods for Stellar accounts defined in SEP005.

The users mnemonic is very sensitive information because it indirectly contains the master keys (secret seeds) associated with the user’s Stellar accounts. Therefore this information must be protected very well.

On registration, the generated mnemonic is encrypted and then sent (encrypted) to the server along with the users first name, last name and email-address. To encrypt the users mnemonic, the client uses the user’s password. The password of the user is never sent to the server, meaning that only the user (who knows her password) can later decrypt her mnemonic.

The Lumenshine server has no access to the user's mnemonic since it can not decrypt it. Also a potential hacker, who may manage to steal the user's encrypted mnemonic, has no access to the user's sensitive data because she can not decrypt it without having the user's password. Because the password is never sent to the server a potential man in the middle attack during data transfer also makes no sense.

The process of encrypting the mnemonic is described in following chapter. It also describes why it is very hard for a potential hacker to "guess" the user's password to be able to decrypt a stolen, encrypted mnemonic.

ENCRYPTION OF THE MNEMONIC

This chapter describes how the user's mnemonic is encrypted before sending it to the server in the registration process.

To encrypt the mnemonic the client generates a random, 256-bit encryption master key. It uses the generated master key to encrypt the user's mnemonic. Next, the encryption master key itself is encrypted.

Good encryption implementations incorporate the use of user input as a meant to unlock the encryption. This ensures that the encryption depends on both "something we have" (the encrypted mnemonic and encrypted master key) and "something the user knows" (a passphrase or password). Key derivation functions (or KDFs) derive one or more keys from a secret value, such as a user's password. KDFs are capable of accepting a secret input value and then crunch it through a series of permutations to derive an encryption key of our desired size. This, from the user's password derived key with the size of 256-bit is then be used by the client to encrypt the master encryption key.

You may be wondering what the purpose of using a master encryption key is, rather than simply using a derived key as an encryption key. A master encryption key, which is usually randomly generated as described before, never needs to change if it is protected at all times. If the user should change his password, the master key can simply be re-encrypted with the new derived key, whereas we'd have to re-encrypt all of the user's mnemonic if the password were tied directly to the encrypted mnemonic.

Not all applications use a key derivation function to encrypt master encryption keys, and this makes them more susceptible to certain types of attacks. The most common misuse of a user password is to simply create a cryptographic hash of it, and use that as an encryption

key. When designing encryption capable of withstanding brute force attacks, key derivation functions serve a critical role. Simply using a cryptographic hashing mechanism such as MD5 or SHA1 make a key susceptible to brute force or dictionary attacks with small computing clusters, or in some cases even a powerful desktop machine. Certain governments also have the ability to design and fabricate custom circuitry specifically geared at performing brute force attacks, which can also greatly increase an attack's efficiency. Simply hashing a passphrase is weak for basic computing today, and even weaker for applications that may be attacked by foreign governments.

The benefits to using a KDF over cryptographic hashing, or other methods, are many. First, KDFs run the input through a number of cryptographic iterations, which consume a given number of CPU cycles in order to derive a key. Instead of a single cryptographic hash, a KDF may iterate 10,000 or even 100,000 times. This causes a certain amount of computing power to be consumed every time a key is calculated, frustrating brute force attempts. Consider a derived key that takes approximately one second of real time to derive a key on the device. This delay would be virtually unnoticeable when logging into an application with the correct password, but a brute force attack would take considerably longer than if just a simple hash were used; each guess at the passphrase would take approximately two seconds to calculate on the device.

Another benefit to using a key derivation function is that it can stretch or shrink the passphrase to provide the key size desired. This means that even a simple four-digit passphrase (as unsafe as it is) could be used to generate a 128-bit, 256-bit, or other size key as needed.

To derive our key from the users password we use the PKCS5_PBKDF2_HMAC_SHA1 derivation function with 65.000 permutations to derive a 256-bit key that we use to encrypt the master key.

PBKDF2 (also known as password-based key derivation function) is a key derivation function included in RSA's PKCS specification to derive an encryption key from a password. It is used in many popular encryption implementations, including Apple's File Vault, TrueCrypt, and WPA/WPA2 to secure WiFi networks. PBKDF2 accepts a user passphrase as input and can generate an encryption key by performing the requested number of iterations on the input data. The PKCS5_PBKDF2_HMAC_SHA1 function is a popular PBKDF2 function also included with OpenSSL.

In cryptography, a salt is a series of bits used to complicate certain types of cryptanalytic attacks, such as dictionary attacks using rainbow tables. When a passphrase is combined

with a salt, the same passphrase used elsewhere will yield a different key. When the salt is unique to a device, encrypted data would no longer be readable if the user restored a copy of it to any other device. Depending on your encryption needs, this may be just what you're looking for, or entirely unacceptable. In our case, where data must be readable regardless of what device it's on, a salt can be generated randomly when the passphrase is initially set, and stored along with the (encrypted) master key on the server.

Regardless of how the salt is generated, the salt, passphrase, and number of iterations can then be used to generate an encryption key that can then be used to encrypt the master key.

Before encrypting the master key, we use it to encrypt the user's mnemonic. To do so, we are going to encrypt following data:

- shuffled word list that we used to generate the mnemonic
- list of 24 indexes (int) representing the positions of the words of the mnemonic within the shuffled word list

We do not encrypt the 24 words of the user's mnemonic, instead we encrypt a list of indexes of type UInt8 (unsigned integer with a length of 8 bits), each index having the same length, resulting in a list that always has the same encrypted size for each user. By doing this, a potential hacker can not draw conclusions from the length of the encrypted data such as it would be the case if we would just encrypt the words of the mnemonic (ex. the hacker could conclude that in a short encrypted mnemonic many short words have been used).

An initialization vector (IV) is a random number used in combination with a secret key as a means to encrypt data. This number is sometimes referred to as a nonce, or "number occuring once," as an encryption program uses it only once per session. An initialization vector is used to avoid repetition during the data encryption process, making it impossible for hackers who use dictionary attack to decrypt the exchanged encrypted message by discovering a pattern. To encrypt the mnemonic we generate an IV with the length of 256 bit and use it in combination with the master key. The cypher algorithm that we use to encrypt the data is AES/CBC/NoPadding. The Advanced Encryption Standard (AES), also known by its original name Rijndael (Dutch pronunciation: ['rɛindaːl]), is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST). In June 2003, the U.S. Government announced that AES could be used to protect classified information:

“The design and strength of all key lengths of the AES algorithm (i.e., 128, 192 and 256) are sufficient to protect classified information up to the SECRET level. TOP SECRET information will require use of either the 192 or 256 key lengths.”

In combination with our 256-bit master key, AES is extremely secure. If you are interested, you can read more about the algorithm here and here.

To encrypt the shuffled word list we use a different random master key than the one used to encrypt the list of indexes.

On registration of the new user, after encrypting the mnemonic data as described above, the client encrypts the master keys and sends to the server following data via HTTPS:

- first name of the user
- last name of the user
- email-address of the user
- Salt used to derive the PKDF2 encryption key from the user’s passphrase
- IV used to encrypt the master key with the PKDF2 encryption key derived from the user’s passphrase
- encrypted shuffled mnemonic word list
- encrypted mnemonic index list
- Encrypted master key used for the encryption of the shuffled mnemonic word list
- IV used for encryption of the mnemonic word list
- Encrypted master key used for the encryption of the index list
- IV used for the encryption of the mnemonic index list
- public key at index 0 from the mnemonic (this is later used to validate that the user inserted the correct password, ex. at login)

The server checks if an user with the same email-address already exists and if not, it registers the user, associates the received data to the user’s email-address and stores it in the backend database.

SETUP PROCESS

After registration, the user is redirected by the client to the setup process. Here the user must complete 3 steps to finish the setup and gain access to the dashboard. The 3 steps are:

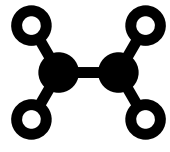
- Step1: confirm 2FA registration
- Step 2: confirm email-address
- Step 3: confirm and prove noting of the mnemonic

Step 1 is displayed immediately after the server successfully registered the user. The client receives a new 2FA secret from the server, and displays it to the user. The user must to add it to his 2FA authenticator app and insert the generated 2FA code to complete this step and move forward to step 2.

In step 2 of the setup, the user is required to confirm his email-address. Immediately after successful registration, the server sends an email to the user's email-address containing a confirmation link. After the user presses the confirmation link, step 2 is also finished and the user can continue with step 3.

In step 3 of the setup process, her 24 words mnemonic is displayed to the user. The user is requested to write it down and confirm the noting of her mnemonic. As soon as confirmed, the client shows a mnemonic quiz, asking the user to insert the positions of some random words of her mnemonic. By filling this quiz, the user "proves" that she noted her mnemonic.

The setup process can be interrupted at any time by the user. For example if the user closes the browser window with the web-app. In the case that it has been interrupted, the setup process will be continued on next login. This repeats until the user completed all steps of the setup process. After completing the setup process, the user is fully authenticated and can access the dashboard of the wallet app.



LOGIN

This chapter describes how the login of a user is secured. Since login on mobile devices using the native app is more user-friendly by providing features like Face-ID, we are going to describe the login process from the general perspective of the web client first and then describe the differences in the mobile app.

GENERAL / WEB CLIENT

From the system perspective the login process is always the same. It needs following data:

- the user's email-address
- the user's current 2FA code
- the user's password

The login process is split into 2 steps:

- Step 1: authenticate the user via email-address and 2FA code
- Step 2: on step 1 success, authenticate the user via his password.

In step 1 the client sends the email-address and inserted 2FA code to the server. The server validates the code and if correct, the server returns a JWT representing partial authentication. It also returns a transaction challenge as defined in SEP0010 and the encrypted mnemonic data associated with the user's email address:

- Salt used to derive the PKDF2 encryption key from the user's passphrase

- IV used to encrypt the master key with the PKDF2 encryption key derived from the user's passphrase
- encrypted shuffled mnemonic word list
- encrypted mnemonic index list
- Encrypted master key used for the encryption of the shuffled mnemonic word list
- IV used for encryption of the mnemonic word list
- Encrypted master key used for the encryption of the index list
- IV used for the encryption of the mnemonic index list
- public key at index 0 from the mnemonic

Next, in step 2, the client tries to decrypt the data with the inserted password and applies the key derivation methods for Stellar accounts defined in SEP0005 to get the public key at index 0 from the decrypted mnemonic and the secret seed associated with that public key. If the public key derived from the mnemonic matches the public key received from the server, we can be sure that the user inserted the correct password. Next, the client must prove to the server that the user inserted the correct password without sending the password to the server. To do this, the client must sign the received SEP0010 transaction challenge and send it back to the server. Of course we must first verify if the received transaction challenge is valid, ex. sequence number is 0, it only contains a mange data operation, it is signed by our server, etc. If valid, the client signs the transaction challenge with the user's secret seed that is associated with the user's public key at index 0 from the mnemonic. After signing the transaction challenge, the client sends it along with the partial auth JWT to the server to prove that the user inserted the correct password. The server can now verify that the transaction was signed by the secret seed associated with the public key at index 0 (the server has the public key at index 0 in it's database). The server first verifies that the transaction challenge is valid (contains the server's own signature, has the correct data, still valid regarding it's time bounds, etc.). If the transaction is valid, the server next verifies that it was correctly signed with the user's seed associated with the stored public key at index 0. On success, the server returns a new JWT that represents the full authentication status to the client. The client can now use that full-auth JWT to access the dashboard server interfaces. The user is now logged in. The full-auth JWT must be refreshed every couple of minutes by the client because otherwise it expires. If there is no user action in the client for more the 10 minutes, the client automatically loges out the user.

MOBILE APP

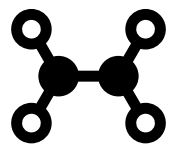
The mobile app must of course also pass the two step login process described above to login the user and to receive the full-auth JWT from the server. On first login the user must insert email-address, 2FA code and password like in the web app. The mobile app logs in the user and receives the full-auth JWT from the server.

But if the app goes to background or is closed, the user is not logged out from the app. If more that 10 seconds pass before the user brings the app back to front, the app requests the user to unlock the app. The user can now sign out or unlock the app by inserting her password or, if activated, by using biometric authentication (Face-ID/Touch-ID). The current 2FA code must not be inserted by the user. This is because of usability reasons.

Of course the app must pass the server login process again at unlock, because it needs a new valid full-auth JWT from the server. As we know from before, the login process consist of two steps. In the first step, the user's email-address must be passed together with the current 2FA code. In the second step, the password must be validated and the client must prove to the server that the user inserted the correct password without sending the password to the server. To be able pass step 1, the app must generate the current 2FA code for the user (who is still logged in the app but not in the server). To do so, the app must have the 2FA secret of the user. Therefore the 2FA secret of the user is requested after each successful login and stored in the device keychain on behalf of the user until the user signs out from the app. This means that the app now already has the 2FA secret of the user because it stored it in the keychain at first login. Not requesting the current 2FA code from the user but generating it from the stored 2FA secret is not problematic from a security point of view for the mobile app, because the user's 2FA authenticator app is on the same device.

After passing step 1 of the login process, the app validates the inserted password and proves to the server that the user inserted the correct password by signing the transaction challenge as described above. The server returns the full auth JWT and the user can access the dashboard. But if the user activated biometric authentication (Face-ID/Touch-ID) she must not even insert her password to unlock the app. From a usability point of view this is very good, because the user can authenticate with her face or finger to unlock the app instead of inserting 2FA code + password. But this usability feature also has it's backside. This is because if the user activates biometric authentication, the app must store the user password into the keychain of the user's device until the user signs out from the app. It

needs to do so because the app needs the user's password to be able to decrypt the mnemonic and login the user in the server. Of course this is a standard process for biometric authentication on mobile devices and the device keychain is a very secure location on the user's device, but Lumenshine can not take any responsibility for the device keychain as it is implemented by Apple (for iOS) or Google (for android) and not by Lumenshine. Lumenshine can not guarantee that the device keychain can not be hacked or otherwise compromised. This should be considered by users when activating biometric authentication. Lumenshine however tries to find out if the user's device is jailbroken before starting. It only starts if it can not find any hints that the device might be jailbroken.



LOST PASSWORD

If the user lost her password, we can not recover it, because we never store the user's password. Therefore if lost, the user must enter his 24 words mnemonic (backup passphrase) to be able to reset his password. This chapter describes the process of resetting the password from the security point of view.

If the user lost her password, she must pass following process to be able to reset her password:

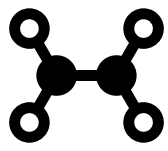
- Step 1: insert email-address and current 2FA code (if valid, the server sends an email)
- Step 2: click link in the "password lost" email received from the server
- Step 3: insert mnemonic into the Lumenshine form
- Step 4: insert new password

In step 1, the client requests the user to insert her email-address and current 2FA code. If the user also lost her 2FA secret, it must contact Lumenshine support. Otherwise, the client sends the email-address and inserted 2FA code to the server. If the inserted 2FA code is valid, the server sends a "password lost" email to the user that contains a link to be pressed by the user. As soon as the user presses the link, a new browser tab with the client is

opened having the token from the email-link. With this email-token, the client requests a partial auth JWT and the users public key at index 0 from the server. Next, the user must insert her 24 mnemonic words. After inserting, the client validates the input by deriving the public key at index 0 from the inserted mnemonic and comparing it to the one received from the server. If they match, we can be sure that the user inserted the correct mnemonic. Next, the user can insert her new password. The client than encrypts the inserted mnemonic (same process as at registration) by using the new password. Next, the encrypted data has to be replaced in the server backend database, meaning that the client must send it to the server. But to be accepted by the server, the client must first prove to the server that the user inserted the correct mnemonic. To do so, the client requests a SEP010 transaction challenge from the server, validates and signs it with the derived secret seed that is associated with the user's public key at index 0. Then it sends the new encrypted data and the signed transaction challenge to the server. The server validates the signed SEP010 transaction challenge first and if its valid and correctly signed by the user seed associated with the public key at index 0, that the server updates the received data in his backend database. The users password has been reset.

As mentioned above, if the user not only lost her password but also lost her 2FA secret, the user can not reset her password, even if she has the correct mnemonic. This is because she needs her password to reset her 2FA secret. In this case she has to call support and prove that she is the user holding the account with the specified email-address. Since we have first name and last name from the registration process, the user can send us a photo of her ID and we can compare the data. If the names match, we can trigger reset of the users 2FA secret from our administration client and the Lumenshine server will setup a new 2FA secret and send an email to the user so that she can confirm the new 2FA secret first by inserting the current 2FA code. As soon as this is done, she can continue with the password lost process because now she owns her 2FA secret again and can insert her current 2FA code in the reset password process.

There is also another relevant case that we want to be mention here. In case that the user lost her password after registration but has not yet finished the setup process, meaning she has not confirmed and proved writing down of her mnemonic, the user can not be requested to insert her mnemonic for resetting her password. In this case, the user must just insert a new password, and the client will generate a new mnemonic for her.

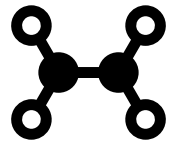


CHANGE PASSWORD

The user must be logged in to be able to change her password. She can change her password in the settings of the app. Here, the client already has a full auth JWT. To be able to change the password, the user must insert her old password, new password and repetition of the new password.

After the user inserted her passwords, the client must first validate the input. To validate if the old password is correct, it needs the encrypted mnemonic from the server. Therefore the client requests the encrypted mnemonic and related data (Salt, IVs, encrypted master keys, public key at index 0, etc.) from the server by using the full auth JWT that it received from the server at login. The server returns the encrypted data and the client can validate the old password. If correct, the client next encrypts the master keys by using new IVs and new derived keys from the user's new password (also a new salt). The updated salt, encrypted master keys and IVs must next be updated in the server's backend database.

The server only accepts the changed data if the client proves that the user inserted the correct old password. To do so, the client next requests an SEP010 transaction challenge from the server. After validating the transaction challenge, the client signs it with the seed associated to the public key at index 0 and sends it back to the server along with the full auth JWT and changed data. Next the server validates the signed transaction and on success the server updates the received data in the backend database. The user's password is changed.

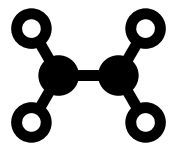


LOST 2FA SECRET

If the user lost her 2FA secret she can request a new one. To receive a new 2FA secret she must pass following process:

- Step 1: insert her email-address
- Step 2: press the link in email received from the server
- Step 3: provide her password
- Step 4: confirm the new 2FA secret by inserting the current 2FA code.

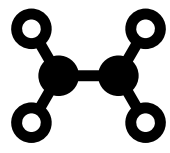
In step 1 the client requests the server to send an “lost 2FA secret” email to the user. Next, the user must press the link in the received email. This will open a new tab with a client that receives the token from the email. With the email-token, the client requests a partial auth JWT from the server. Next, the user must insert her password. After the user inserted her password, the client requests the encrypted mnemonic and related data from the server by passing the partial auth JWT received before. Next the client can validate the inserted password by decrypting the mnemonic and comparing the public key at index 0. If the inserted password is valid, the client must next request an SEP010 transaction challenge from the server. After validating and signing the received transaction challenge, the client requests a new 2FA secret from the server for the user by passing the partial auth JWT and signed transaction challenge to the server. If both are valid, the server sends a new 2FA secret to the client that can be displayed to the user. Next the user must confirm the new 2FA secret by inserting the current 2FA code. The client sends the inserted 2FA code to the server and if the code is valid, the server activates the new 2FA secret for the user.



CHANGE 2FA SECRET

The user can only change her 2FA secret if she is logged in. She can change her 2FA secret in the settings of the app. Here, the app already has a full auth JWT from login.

To change the 2FA secret, the user must insert her password first. The client than requests the encrypted mnemonic and related data from the server by providing the full auth JWT. Next, the client validates if the inserted password is correct. If so, it requests a SEP0010 transaction challenge from the server. Now the client can validate the received transaction, sign it and request a new 2FA secret from the server by sending the full auth JWT and signed SEP0010 transaction to the server. Next, the server responds with a new 2FA secret that can be displayed to the user. The user must now confirm the new 2FA secret by adding it to her authenticator app and inserting the currently generated 2FA code into the Lumenshine client. The client than sends the 2FA code to the server for validation. If valid, the server activates the new 2FA secret.



SIGN STELLAR TRANSACTIONS

To be able to sign Stellar transactions on behalf of the user, the Lumenshine client needs to have access to the secret seed associated with the public key of the Stellar source account used for the transaction. In Lumenshine each wallet of the user represents a Stellar account. As soon as a new wallet is created for a user, the next available public key is derived from the user's mnemonic and associated with the new wallet.

When the user submits transactions to the Stellar network by using Lumenshine, she always needs to select the wallet that she wants to use as a source account for the transaction.

Currently Lumenshine does not provide support for multi-sig. Therefore, normally, only the secret seed of the source account (selected wallet) is needed to sign the transaction. However, if the master signer has not enough weight to sign the transaction, Lumenshine gives the user the possibility to select another signer (if exists) that has sufficient weight to sign the transaction. In this case the user can insert the secret seed associated with that signer manually to sign the transaction.

But, as stated above, normally the master signer has enough weight to sign the transaction. Before submitting transactions, such as payments, the user must enter her password to approve/sign the transaction. Lumenshine can only obtain the needed secret seed by deriving it from the user's mnemonic. Since the server can not decrypt the mnemonic, only the client can sign transactions on behalf of the user.

To sign the transaction, the client requests the encrypted mnemonic and related data from the server and decrypts it by using the the users password (the "Login" chapter describes how the decryption of this sensitive data is done). If the password is valid, the client next derives the secret seed associated with the wallet's public key from the decrypted mnemonic. The derived secret seed is then used to sign the transaction on behalf of the user.