heylogin

Security
Whitepaper

of **heylogin GmbH**, Sophienstraße 40, 38118 Braunschweig
for the product heylogin

Date: 2022-04-06
Version: 1.4

Table of Contents

1 | Introduction 3
1.1 Security and Usability of Master Passwords 3
1.2 Swipe to Login instead of Master Password 5
2 | Fundamentals 6
2.1 Cryptographic Algorithms and Key Notation 6
2.2 Architecture 7
3 | How Personal Logins Are Secured 8
3.1 The Smartphone as an Authenticator 8
3.2 Saving and Reading Logins 10
3.3 Pairing Another Device 11
3.4 Locking and Unlocking a Device 13
3.5 Account Recovery 14
4 | Summary 15

1 | Introduction

Remembering only one password, the *Master Password* instead of many is the main selling point of traditional password managers. With heylogin, a Master Password is no longer necessary. Instead, it uses the secure element present in modern smartphones and replaces the Master Password with *Swipe to Login*. Secure elements are security chips that protect secrets against unauthorized access and brute force attacks. When authorization is required, the user is asked to swipe on their smartphone instead of entering a Master Password. This dependency also makes heylogin two-factor secure (2FA) by design because login requests have to be authorized on a second device: the smartphone. While heylogin's Swipe to Login technique feels like a login method, it actually uses end-to-end encryption from the smartphone to the browser to make passwords available.

1.1 Security and Usability of Master Passwords

Legacy password managers require users to remember and regularly enter a Master Password. This Master Password is used to encrypt and decrypt all stored private information, such as usernames and passwords. A Master Password must be complex and kept private, as it is the single secret to all information. There are several problems associated with this cryptographic design:

- **1-factor Security:** While many password managers allow the setup of another factor, such as TOTP, U2F or FIDO2/WebAuthn, this is not done by most users. Furthermore, this second factor is not used for end-to-end encryption, but only an additional authentication via the provider's infrastructure. Exceptions are password managers with native smartcards that implement actual encryption using OpenPGP, PIV or FIDO2 hmac-secret.

- **Offline Attacks:** The Master Password, as a factor of knowledge, cannot be protected against brute force attacks as soon as they are performed offline. When a password vault is stolen or a data leak occurs at the large commercial password managers, the encrypted vaults can be attacked “offline”, i.e., there is no interactive protocol involved that rate-limits retries. A brute force attack or dictionary attack is only slowed down by the vault's Password-based Key Derivation Function (PBKDF). However, this never achieves the protection of a Hardware Security Module (HSM) since PBKDFs only slow down the brute force attack, but can never limit the number of tries like a HSM could.

- **Usability:** Studies show that not all users are able to generate and remember a sufficiently secure Master Password. In a study by Pearman et al [1], participants reused a different password as their Master Password or had it generated on a website. The participants involved had no technological training. So, especially for people who are not IT experts, using a password manager with a Master Password can actually reduce their security to a single point of failure.

- **Time Required:** Depending on the implementation and the security policies used, the Master Password must be retyped regularly by the user in order to temporarily decrypt the vault. We assume about 3 hours / month / user, which are spent for the regular typing of the Master Password and the password management.

The use of a legacy password manager is thus mainly associated with annoyances that go beyond the normal conflict between security and user-friendliness. Existing solutions cannot easily change their security architecture because basic user flows and user expectations go hand in hand with the Master Password.

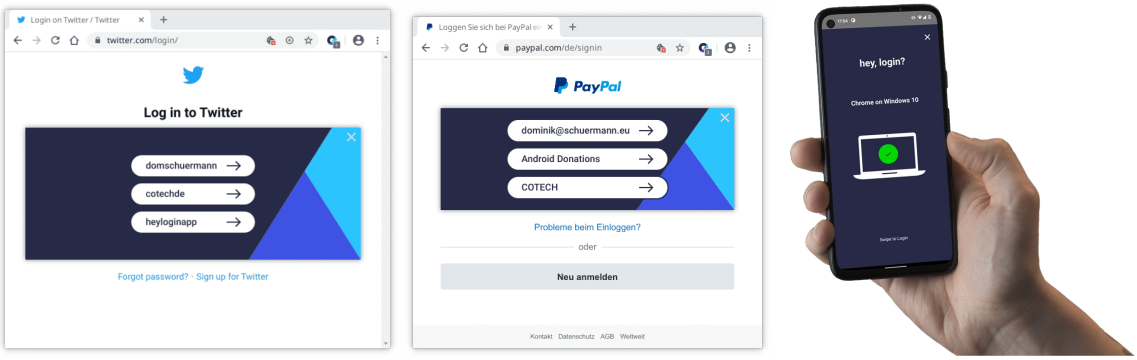

Figure 1: heylogin works with all websites in the user's browser. In this example, instead of typing in passwords for Twitter or PayPal, the user clicks the heylogin button, gets a notification on the phone and confirms the login using the Swipe to Login technology.

1.2 Swipe to Login instead of Master Password

heylogin implements a cryptographic architecture that is two-factor-secure by design, protects against brute force attacks and requires less time in daily usage. As shown in Figure 1, one swipe on the user's phone decrypts the passwords and allows automatic login on websites in the user's browser.

2 | Fundamentals

2.1 Cryptographic Algorithms and Key Notation

This document presents the architecture and algorithms used by heylogin to store data. For information on the infrastructure and availability, please take a look at the Compliance Whitepaper [2]. heylogin only uses state-of-the-art cryptography algorithms. On all platforms we utilize libraries that implement the NaCl interface and therefore use Curve25519 [3] and the XSalsa20 [4] stream cipher with Poly1305 [5] authentication for asymmetric cryptography and the XSalsa20 stream cipher with Poly1305 authentication for symmetric cryptography. At time of writing, the library used on all platforms is TweetNaCl.js [6]. As it does not implement Argon2 [7], which the original TweetNaCl does, we use a separate implementation of Argon2 when needed.

Within heylogin, keys are named to indicate whether they are allowed to be saved on the device or not. In this document we will use the term *storable* to indicate that a key is allowed to be stored on the device and the term *high security* to indicate that the key may only exist in volatile memory and must not be persisted. Such keys must be derived or acquired before using them. Asymmetric key pairs consist of two keys, the secret and public key, which will use the following notation: $usage_{type}^{prefix}$ where *prefix* is either *s (storable)* or *hs (high security)*, *type* is either *sec (secret)* or *pub (public)* and *usage* is a unique name indicating what this key is used for. Symmetric keys use a similar notation, just without the key type: $usage^{prefix}$.

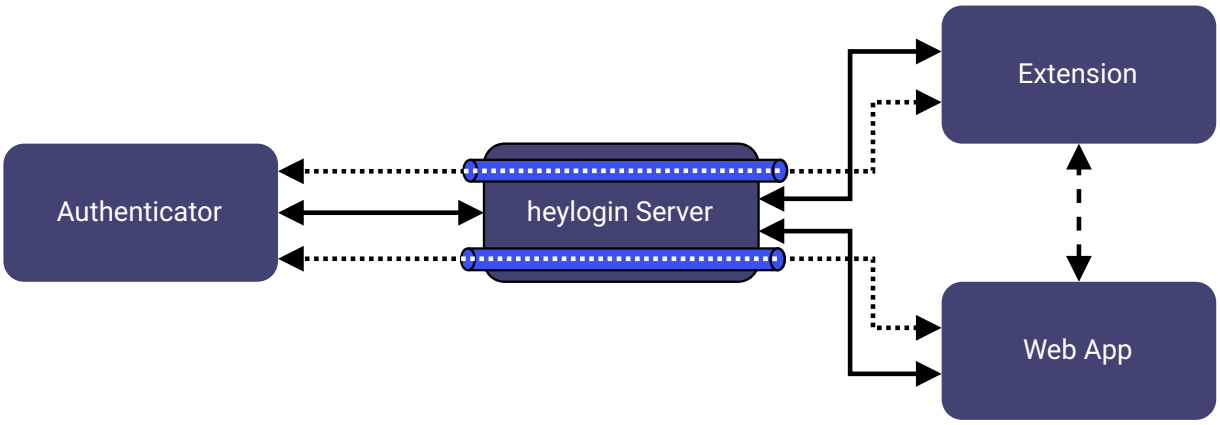

Figure 2: Overview of the heylogin architecture. Solid arrows indicate TLS-encrypted connections over the internet, dotted arrows indicate E2E-encrypted connections inside the TLS-encrypted connections and dashed arrows indicate a local, unencrypted connection on the same device.

2.2 Architecture

The architecture of heylogin consists of two parts, the server side which stores encrypted customer data and the client applications talking to the server side. There are two kinds of clients available, the *Authenticator,* which is the mobile app, and the *App,* which are the web app and the extension.

Figure 2 shows how the parts communicate with each other. All client applications talk to the server side using TLS 1.3, depicted by solid arrows. The web app and extension can exchange data on the local machine directly inside the browser depicted by a dashed arrow. The authenticator cannot talk directly to the web app or extension and therefore the server side acts as a proxy. All messages between the apps and the authenticator are always end-to-end encrypted and sent over the TLS connections, the virtual communication channels are depicted as dotted arrows.

On the server side, all user data is saved encrypted with keys that are only available on the user's devices, see Section 3. The only accessible data is the user's email address and structural metadata, such as which data belongs to which user. The apps never send unencrypted user data to the server-side. The heylogin server-side merely act as data storage and communication proxy.

3 | How Personal Logins Are Secured

3.1 The Smartphone as an Authenticator

One of the core concepts of heylogin is the *Authenticator*. With heylogin, every user has at least one authenticator: their mobile phone. Modern mobile phones and their operating systems allow applications to store secrets and sensitive data securely by offering support for cryptographic operations with key material only available to the hardware. This hardware is known as a *secure element* and can encrypt and decrypt user data on request. The heylogin app utilizes the secure element to create an authenticator in the following way:

First, a random 32-byte seed is generated on the mobile phone. This seed is then given to the secure element to be encrypted and the encrypted form is saved on the device. The seed is then used to derive the five key pairs

$$(login^{hs}_{sec}, login^{hs}_{pub}),\ (id^{hs}_{sec}, id^{hs}_{pub}),\ (vaultKeyEnc^{hs}_{sec}, vaultKeyEnc^{hs}_{pub}),$$
$$(vaultKeyEnc^{s}_{sec}, vaultKeyEnc^{s}_{pub}) \text{ and } (sig^{s}_{sec}, sig^{s}_{pub}).$$

All public keys are saved on the server side. To derive all key pairs except the login key pair, a unique salt is also randomly generated and saved on the server side, see Figure 3. The login key pair is used for authenticating this authenticator against the server. The id key pair represents the identity of this authenticator. Both vaultKeyEnc key pairs are used to encrypt two types of vault keys which then encrypt actual data,

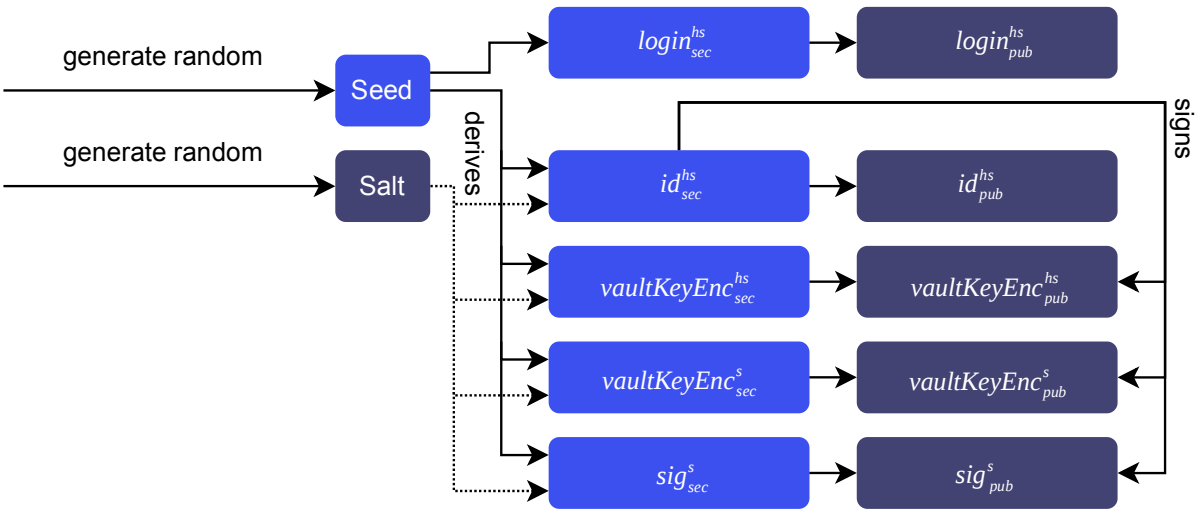

Figure 3: Key generation for authenticators

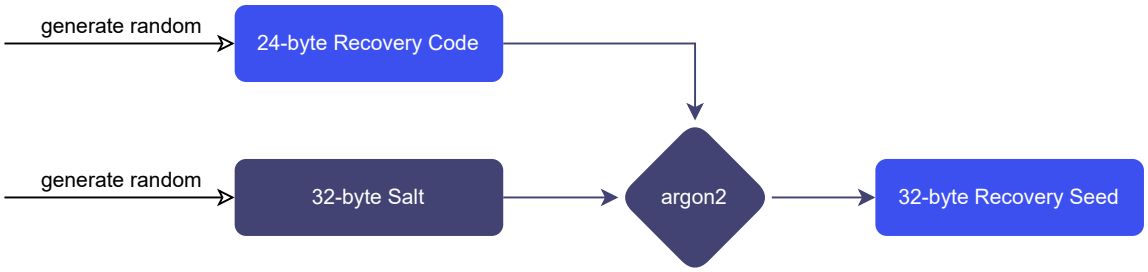

Figure 4: Generation of backup code

see Section 2.2. The $vaultKeyEnc^{hs}_{pub}$, $vaultKeyEnc^{s}_{pub}$, sig^{s}_{pub} public keys are signed using id^{hs}_{sec} and the signatures are saved on the server in addition to the public keys. This prevents the server from changing any of the public keys without also changing id^{hs}_{pub} and therefore changing the identity of the authenticator.

The user now has a so called *Push Authenticator*. It is called a push authenticator as it can receive push notifications that prompt the user to unlock secrets for a client. To prevent loosing access to heylogin should the push authenticator be unavailable, e.g. because the mobile phone is broken, a *Backup Authenticator* is generated in the same way, except that its seed is not encrypted by the secure element. It is stored unencrypted inside the cloud backup of the app. On both iOS [8] and Android [9] the cloud backup is encrypted which makes this a safe way to store the seed.

Another type of recovery exists using the backup code displayed inside the app. This code represents an optional third authenticator that is created when the code is first retrieved. As shown in Figure Figure 4, to create this authenticator, the 24-byte random backup code is generated first alongside a 32-byte salt. Then, the seed is generated by password hashing the backup code using Argon2 [7]. At the time of writing, the used Argon2 parameters are: 6 iterations with a parallelism of 8 and a memory cost of $48 \cdot 1024 = 49152\,KiB$.

The salt is saved on the server side and is retrieved when needed. For more information on the account recovery, please see Section 2.5.

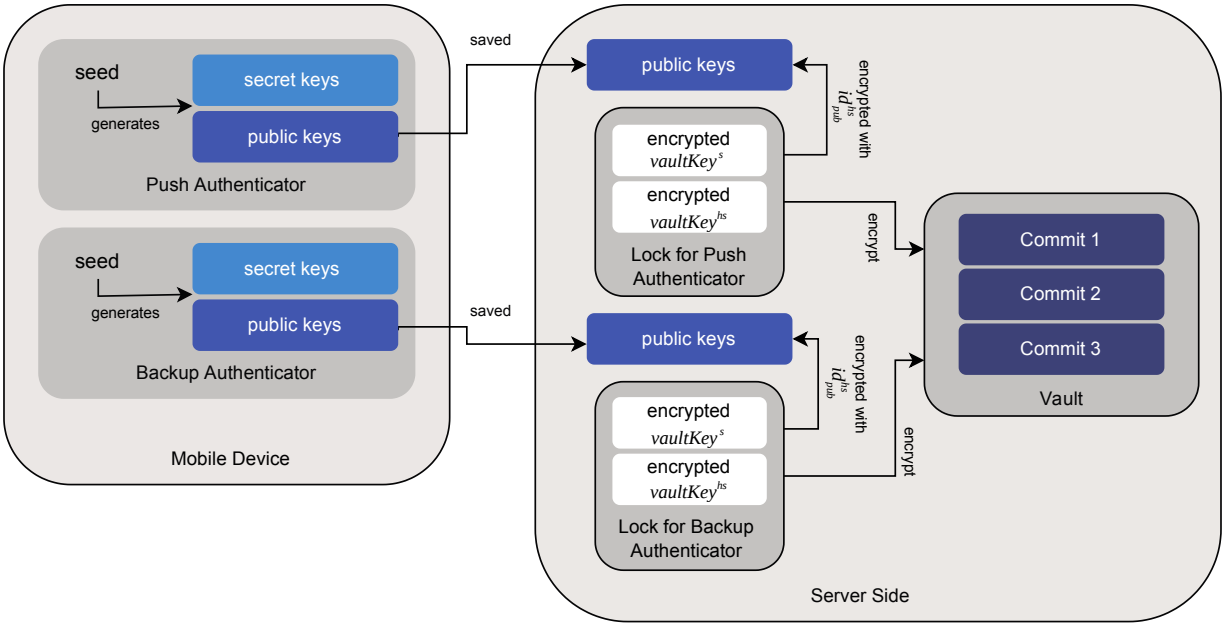

Figure 5: Overview for saving and reading logins. Logins are stored inside commits which are encrypted with the symmetric vault keys. These keys are encrypted with the public key of an authenticator to form a Lock.

3.2 Saving and Reading Logins

Figure Figure 5 depicts the architecture for saving and reading logins. Logins are saved inside a *Vault* as part of a *Commit*. A login consists of at least one associated website, a username or email address and a password. The website and username can be empty but a password must be set. Additionally, a login can contain custom fields with a freely choosable name and value. Optionally, the value of a custom field can be marked as protected which causes it to be treated as if it were a password. Also optionally, a TOTP secret can be saved which is also treated as if it were a password.

Each user has at least two vaults, their *Personal Vault* (in the web called *My Logins*) which stores their personal logins and a *Meta Vault* which stores metadata such as the names of *Sessions*. Each vault consist of a series of *Commits* that, when applied in order, represent the current state of the vault. Commits are not cryptographically linked but are ordered by their creation time on the server side. Commits are encrypted with a symmetric key called $vaultKey^{s}$. Additionally, sensitive parts inside the commit, i.e., the password, custom fields marked as protected and the TOTP secret, are additionally encrypted with another symmetric key called $vaultKey^{hs}$.

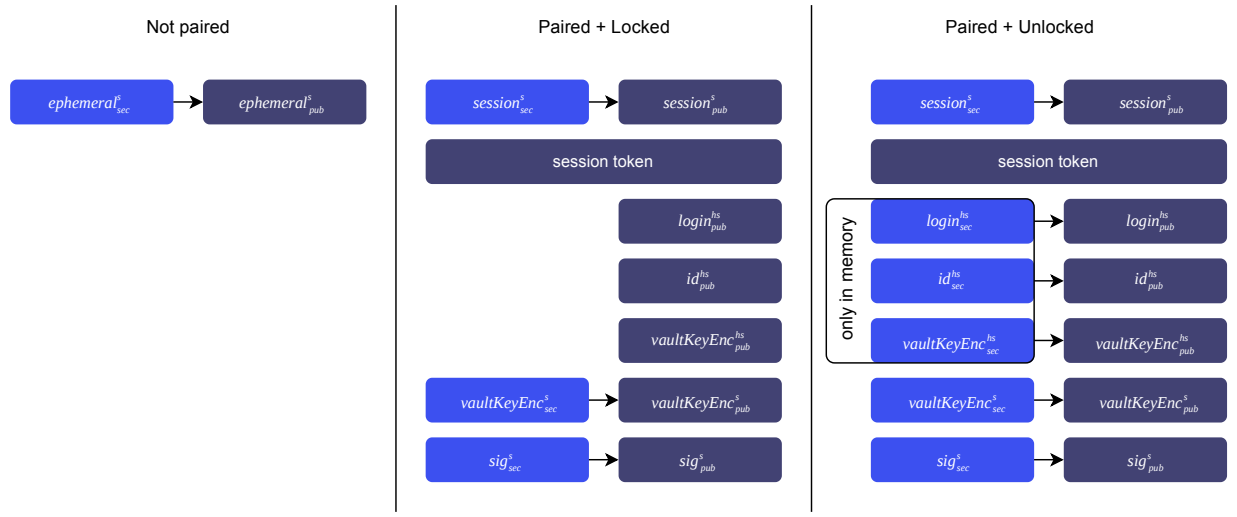

Figure 6: Different states of a client device and which keys are available to it.

The $vaultKey^{s}$ and $vaultKey^{hs}$ are stored encrypted on the server side inside a *Lock*. A Lock binds the $vaultKey^{s}$ and $vaultKey^{hs}$ to a specific authenticator. The $vaultKey^{s}$ is encrypted with $vaultKeyEnc^{s}_{pub}$ and the $vaultKey^{hs}$ is encrypted with $vaultKeyEnc^{hs}_{pub}$ of the respective authenticator. For a personal vault, up to three locks exist that hold the encrypted $vaultKey^{s}$ and $vaultKey^{hs}$ for the push, backup and recovery authenticator.

3.3 Pairing Another Device

A client can be in one of three states with different key material available to it as seen in Figure 6.

After the mobile device has been set up, other devices, such as a browser, here called client, can be paired. Pairing is done by navigating to heylogin.app and then scanning the displayed QR code with the mobile device. The QR code embeds a public key of an ephemeral key pair $(ephemeral_{sec}, ephemeral_{pub})$ which is generated by the client. The SHA256 hash of that public key is send to the server side which proxies the reply of the mobile device to the web app. The content of the QR code is the URL

$$\text{"https://heylogin.app/qr/\#"} \parallel Base64(ephemeral_{pub})$$

Should the URL be scanned by another QR code scanner app that is not the heylogin mobile app and subsequently be opened in a web browser, then the server will simply issue a redirect to the web app. Furthermore, the heylogin mobile app registers itself as a handler for the https://heylogin.app/qr/ URL so that it is opened automatically. To

prevent a malicious actor from pairing a new session by simply invoking the heylogin app QR code URL handler, the app forces the user to scan the QR code again and ignores the passed URL.

Upon scanning the QR code with the heylogin mobile app, the mobile device encrypts its push authenticator seed to that public key, sends it to the server side which relays it to the client who then decrypts the seed to generate the login key pair. With this key pair, the client authenticates itself against the server side and obtains the salt to generate all other authenticator key pairs. It now generates a session key pair $(session^{s}_{sec}, session^{s}_{pub})$, signs $session^{s}_{pub}$ with the authenticator identity secret key id^{hs}_{sec} and saves $session^{s}_{pub}$ and the signature inside the personal meta vault. This mechanism creates a *Session* on the server side for this client. The authenticator seed is also encrypted with the $session^{s}_{pub}$ and saved alongside the session so that the client can retrieve it again later.

Each session also contains a JSON Web Token (JWT) generated by the server side that authorizes this client to send requests to the server side. This is also true for push authenticators which also have a session. All sessions except the own are displayed in the mobile app and can be deleted if needed. When a session is deleted, the respective token on the server side is deleted and the client can no longer perform a request to the server side. A client that gets its request rejected due to a deleted session will delete all key material it has.

So far, a client has been described as being the heylogin.app web app. The heylogin extension installed inside a browser with a paired web app will communicate with the web app and use the same session credentials and therefore will be identified as the same device as the web app. The extension will also save the session credentials to be able to restore the session for the web app even if the browser cookies and local storage is deleted.

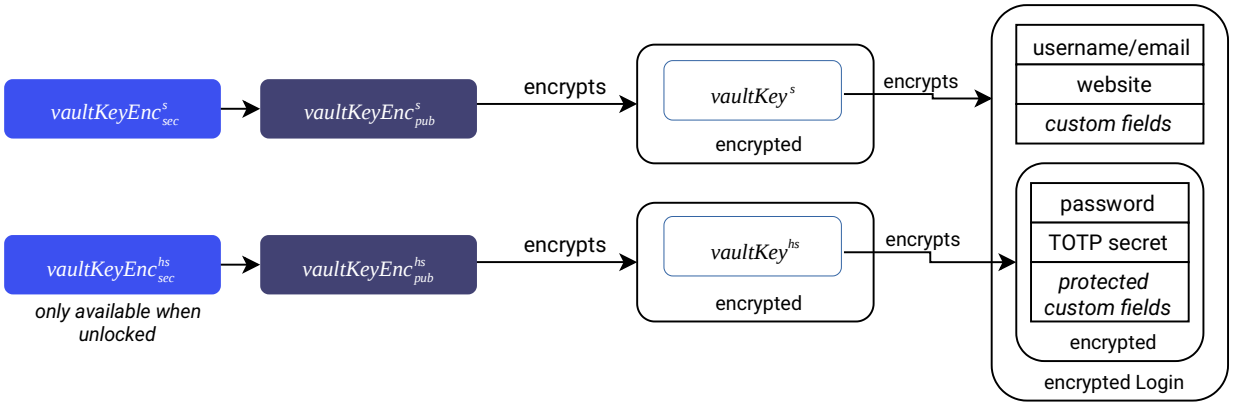

Figure 7: In the paired, but locked state, username, website and other custom fields are available. Passwords, TOTP secrets and other protected custom fields are only available in the unlocked state.

3.4 Locking and Unlocking a Device

A device being locked or unlocked is an implicit state. The presence of an encrypted authenticator seed inside a session means that the session is unlocked. If the encrypted seed is not present, the session is locked. The mobile device can add and remove the encrypted seed to and from sessions at will. These are the toggles inside the mobile app beside each session. A client never persists the seed and only holds it in memory to derive the keys as shown in Section 2.1.

An unlocked session might persist any keypair that is *storable*. In practice, these are the $(vaultKeyEnc^s_{sec}, vaultKeyEnc^s_{pub})$ and $(sig^s_{sec}, sig^s_{pub})$ keypairs. This allows a locked client to decrypt the encrypted $vaultKey^s$ which in turn allows it to parse logins but *not* their password, private custom fields or the TOTP secret, see Figure 7. This property is used by the extension to show the overlay on websites with known logins. As soon as the user clicks on a specific login to log into the website, the extension will request an unlock so it can obtain the seed, derive all other high security keypairs, delete the local copy of the seed, decrypt the encrypted $vaultKey^{hs}$ and then decrypt the password to finish the login.

3.5 Account Recovery

Account recovery is needed when a user has lost access to their push authenticator, e.g., because their mobile device broke down. In such a case, access to either the backup or recovery authenticator is required. The backup authenticator seed it stored inside a cloud backup so switching to a new mobile device should make that authenticator automatically available as soon as the cloud backup has been restored. Alternatively, the recovery code has to be used.

When either the backup or recovery authenticator are used, the server side will remove the push authenticator and all its locks from the database. The new mobile device has to generate a new push authenticator seed and derive the necessary keys to register it as a new push authenticator with new locks. When using a backup or recovery authenticator, the server side only allows for replacing the primary authenticator with a new one. Other operations are denied.

Afterwards, the new push authenticator well regenerate the vault keys of the personal vaults by first squashing all current commits into a new, single commit that holds the most up-to-date state. This new commit marks the start of a new *Generation* in the vault. All previous commits are discarded from the server. The new commit is encrypted with a new $vaultKey^{s}$ and its sensitive contents with a new $vaultKey^{hs}$ which are encrypted and stored inside new locks for that vault. Old locks are discarded.

4 | Summary

We presented heylogin, a 2-factor secure password manager without master password. First, we have detailed heylogin's architecture and how keys are derived on authenticators. Logins are stored inside commits which are encrypted with the symmetric vault keys. These keys are encrypted with the public key of an authenticator to form a Lock. Usernames, website URLs and other metadata are decrypted after a browser has been paired to make them available in the overlay, even when the specific browser is locked in the authenticator. Only in the paired+unlocked state, the passwords are decrypted and made available to allow the automated login process on websites. These seperate encryptions together with heylogin's backup functionality forms a user experience previously only known from Single-Sign On solutions.

References

1: Sarah Perman, et al., Why people (don't) use password managers effectively, 2019
2: , heylogin Compliance Whitepaper, 2022, https://www.heylogin.com/de/compliance/
3: Internet Research Task Force (IRTF), Elliptic Curves for Security, 2016, https://datatracker.ietf.org/doc/html/rfc7748
4: Daniel J. Bernstein, Extensing the Salsa20 nonce, 2011, https://cr.yp.to/snuffle/xsalsa-20110204.pdf
5: Internet Research Task Force (IRTF), ChaCha20 and Poly1305 for IETF Protocols, 2015, https://datatracker.ietf.org/doc/html/rfc7539
6: , TweetNaCl.js, , https://tweetnacl.js.org/#/
7: Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich, Argon2: the memory-hard function for password hashing and otherapplications, 2017, https://www.cryptolux.org/images/0/0d/Argon2.pdf
8: , , , https://support.apple.com/en-us/HT202303
9: , , , https://security.googleblog.com/2018/10/google-and-android-have-your-back-by.html