passbolt

Passbolt Security v3

UX and Security Evolutions

Abstract: This document is aimed at security advisors to help guide the design of future evolutions of the passbolt product, as part of the work scheduled on the v3 roadmap. It focuses on items that have the biggest impact on the security model. It tries to describe the possible approaches from a high level perspective and how they will affect / integrate with the current security model.

Status: DRAFT
Diffusion: PUBLIC CC BY-SA 3

Change history

Date	Author	Changes
28/09/2020	Remy	Initial version
11/12/2020	Thomas / Remy	Revised threat model & passphrase schemes & authentication
27/01/2021	Wojciech / Michał	Mobile recommendations
05/02/2021	Remy	Escrow & cleanup

passbolt

Table of index

Introduction

Business goals

Here are some common comments from our current users:

“As a user I want to access my secrets on a mobile device.”

“As a user I don’t want to have to retype a password every time I need to access a secret in passbolt.”

“As an admin I want to be able to help users get access to data even if they lost their secret key or passphrase.”

“As an admin I don’t want my users to have to remember another password for passbolt. If they have to enter a password it should be the same as the one provided by our organization's main authentication provider.”

“As an admin I want to be able to enforce authentication using the provider selected by my organization e.g. Active Directory or Google, etc.”

Passbolt is currently doing an okay job at hiding key management for authentication, encryption and decryption. However because of end to end encryption design constraints it does fall short in providing an experience closer to a regular web application.

From a user (or even administrator) perspective it should be enough to be able to authenticate in order to decrypt the content stored inside passbolt. An administrator should be able to reset the means of authentication, they should be able to access the data when someone leaves.

All of these requirements are obviously not playing well with the targeted level of security. In this document we try to propose some evolutions to improve the usability of the solution without drastically lowering the security of passbolt.

Current situation

In this section we summarize the current state so that the proposed solutions can be evaluated from that starting point. More information can be found in the security whitepaper.

Type of data

Here is a quick summary of the current type of data and where they sit in which form.

Keys

Type of keys	Client Memory	Client storage	Server Side
OpenPGP Secret Key	Decrypted	Encrypted	No
OpenPGP Public Keys	Yes	Yes	Yes
OpenPGP Secret Key Passphrase	Decrypted	No	No

Passbolt relies on public key cryptography, and OpenPGP in particular, to encrypt data. The user secret key is generated (or imported) on the device and never leaves the device. The secret key is encrypted with the user passphrase and persisted in that form in the client storage. Similarly the passphrase never leaves the device, and is not persisted in the client storage / only kept in memory. Public keys are synchronized with the server and authentication is required to fetch or change keys.

Data

Type of data	Client Memory	Client storage	Server Side
Resource (metadata, ex. “name”)	Yes	Cached	Yes
Resource types & schemas	Yes	Cached	Yes
Secret (ex. “password”)	Decrypted	No	Encrypted

Data in passbolt is divided into two parts: the searchable non encrypted metadata called “resource”, and the encrypted part containing for example the passwords called “secret”. The secret is never stored on the client and downloaded from the server when needed, allowing to track access (provided the user doesn't make a local copy of course). Secrets are encrypted once per user that has access, allowing the delete access server side (e.g. without having to re-encrypt the secret).

The schema that describes what is included in the resource and secret is defined using “resource types”, which take the form of two JSON schemas. These schemas can be used to control the data validation process when (de-)serializing data. These schemas can be downloaded from the server by the client, but the default ones are generally hard coded directly in the client.

Threat model

Here is a quick list of the current threats and type of attacks:

1. Data leak. An attacker is able to read the content of the passbolt server database, for example via a leak.
2. Web page compromise (XSS). An attacker is able to execute JavaScript on the passbolt server instance domain.
3. 3rd Party Authentication server compromise. An attacker is able to login in passbolt via an issue in a third party system.
4. Network compromise. An attacker is able to perform a Man in the Middle (Miim) attack and intercept/modify network traffic.
5. Passbolt Server compromise. An attacker is able to edit the content of the passbolt server database and/or server side logic.
6. Content script compromise (XSS). An attacker is able to execute JavaScript in the context of a content script (or iframe inserted by content script) and therefore have access to encrypted browser extension storage (local, IndexedDB) and decrypted information inserted on the page (for example decrypted secret in an edit dialog).
7. Background page compromise (XSS). An attacker is able to execute code in browser extension and access decrypted secret keys in memory.
8. Browser exploit. An attacker is able to read browser extension memory or browser memory, from a regular web page or another malicious extension installed by the user.
9. Extension / Application marketplace exploits. An attacker is able to forge developer signatures and publish a malicious version of the extension / application.
10. Client System compromise. An attacker has partial or full system access.
11. Phishing. An attacker is able to convince the user to upload both the secret key and passphrase and optionally 2FA token.

Residual risks

As a quick summary the following risks are deemed acceptable:

- Availability: the client is not able to access plaintext secret if the network or passbolt server or optional 3rd party 2FA authentication provider is down.
- Integrity: there is no expectation of data integrity if the passbolt server or network layer is compromised.
- Confidentiality: An attacker is able to access plaintext secrets if the client machine or browser extension background page is compromised, e.g. attacker has access to the browser extension / browser / OS memory.

Here is the list of accepted residual risks:

Threats	Scope	Residual Risks (An attacker can…)
Data leak	In-scope	Access encrypted messages & resource metadata
Web page compromise	In-scope	Idem + Affect integrity / availability of data
3rd party auth compromise		Idem
Network compromise	In-scope	Idem + Inject malicious public keys
Passbolt Server compromise	In-scope	Idem
Content script / Iframe content code compromise	In-scope	Idem + Access decrypted content in workspace Access encrypted secret key
Background page compromise	Out of scope	Full access
Browser exploit	Out of scope	Full access
Extension marketplace exploits	Out of scope	Full access
System compromise	Out of scope	Full access
Phishing	Out of scope	Full access

New requirements & assumptions

The new requirements and associated assumptions are as follow:

- The user should not have to enter another passbolt-specific passphrase to decrypt the secret key and ultimately decrypt / sign content, unless they (or their administrators) explicitly want this.
- The user must be able to transfer and use their secret key on other devices.
- The solution must not require the encrypted secret key to be stored server side to work. This is to protect the end to end encryption scheme and prevent an attacker from being able to perform a bruteforce attack on the secret key. It is also driven by the end user expectations, especially the ones who for example reuse their secret key for email encryption.
- The user must be able to authenticate using another system, through passbolt directly or using redirect / alternative flows such as the ones supported by OpenID connect. These 3rd party credentials must not be stored in passbolt.
- Having access to the 3rd party credentials (or breaking the authentication) must not be enough to decrypt the data.
- Having read-only access to file system data on the client, for the same user, shouldn't be sufficient to decrypt the data.
- Having access to the chrome.storage API must not be enough to decrypt the data. In other words a XSS in a web extension content script must not be enough to decrypt the data. Therefore the user secret key and the passphrase should not be stored unencrypted in the local storage.
- No additional hardware should be required.

Proposed solutions

Remove the need for the user to enter an OpenPGP key passphrase

In a nutshell the idea is to remove the need for a secret key passphrase to be entered by the user if that configuration is desired.

In practice, if the administrator chooses to enable and/or enforce this optional passphrase-less mode, the OpenPGP secret key passphrase will still exist, but it will be randomized for the device during the setup (after the creation of a recovery kit with a user selected passphrase), and will be stored encrypted. To encrypt this passphrase the application will use a “non-extractable” key stored on the browser or device (in case of mobile phones). By non extractable we mean that if the extension background page or mobile app process is compromised, an attacker may be able to use the app's keys but cannot extract their key material (for example, to be used outside of the client / device context).

In the case of mobile apps, the encrypted passphrase will be stored in a secure location. In the case of the webextension, since the non-extractable key usage does not require another device password or biometric authentication, then we propose to store the encrypted passphrase server side, so that we can still perform another form of authentication (if needed).

In practice the new key landscape will look like:

Type of keys	Client Memory	Client storage	Server Side
OpenPGP Secret Key	Decrypted	Encrypted	No
OpenPGP Public Keys	Yes	Yes	Yes
OpenPGP Secret Key Passphrase	Decrypted	No	Encrypted *
Non extractable device/client specific keys, to encrypt passphrase.*	Available	Not extractable	No

* optional, and only in for webextension

In the browser

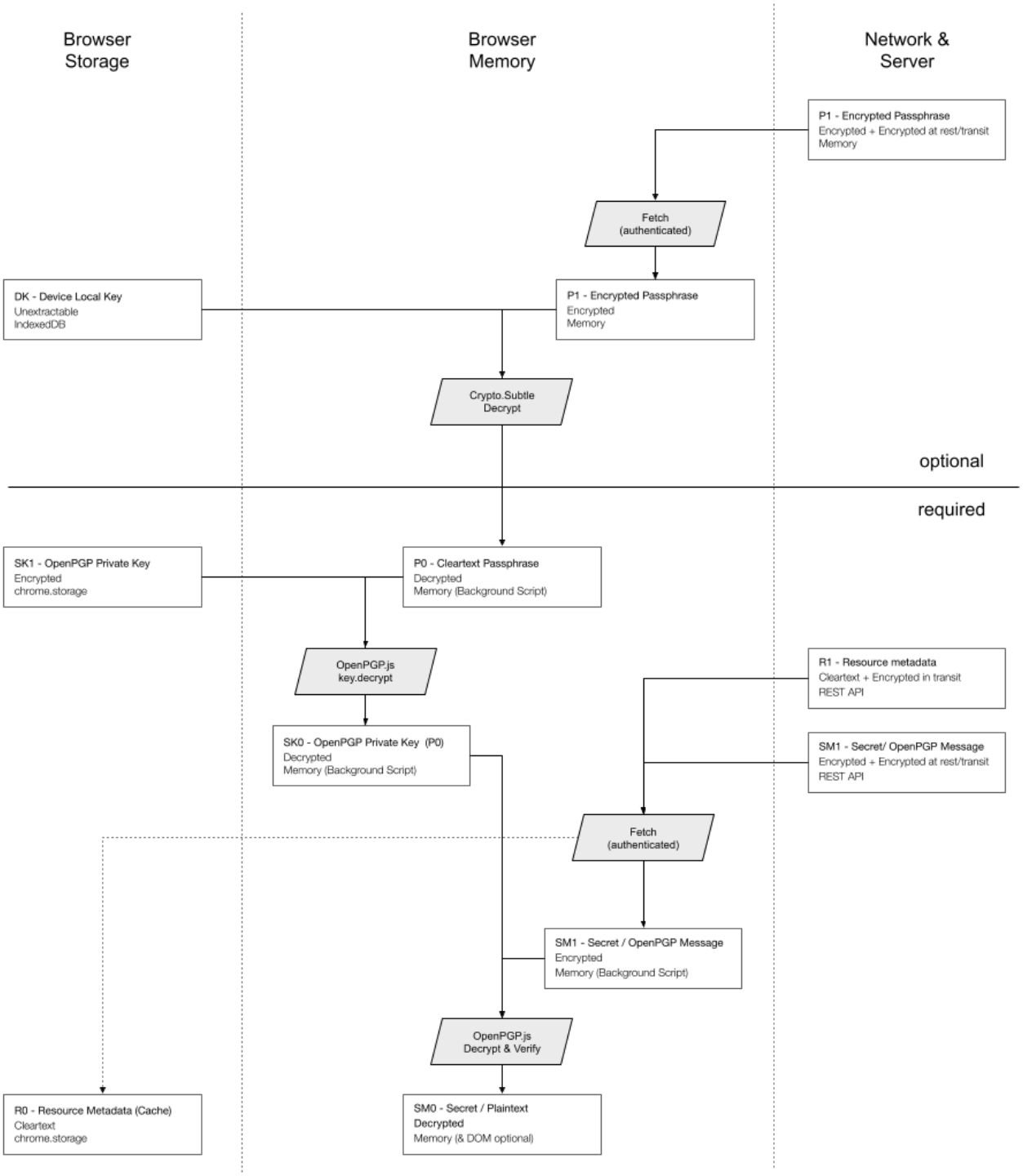


Fig. steps and data by location for decryption in the browser

During the setup the extension generates an asymmetric key with the Web Crypto API `crypto.subtle.generateKey`. The key will be generated with the parameter `extractable` set to false, which means the key can be used but the subtle key material cannot be exported. IndexDB is used to store these keys.

The Web Crypto API ensures that a CryptoKey with extractable set to false cannot be exported. How such a key is stored in IndexedDB is not defined, and as IndexedDB does not have an encrypted storage on the file system we can assume that the secret key material is not protected on the file system level.

Reference:

- https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/generateKey

Sample code:

- https://gist.github.com/saulshanabrook/b74984677bccd08b028b30d9968623f5

Additional risks (Browser)

Risk type	Counter measure	Residual Risk
“Subtle key” does not support biometric authentication	Move the encrypted passphrase server side to perform authentication there.	TBD
Background page “subtle key” can be extracted.	TBD	TBD
“Subtle key” does not persists	Manually passphrase entry required. “Original passphrase” is made part of a recovery kit.	TBD

On Mobile

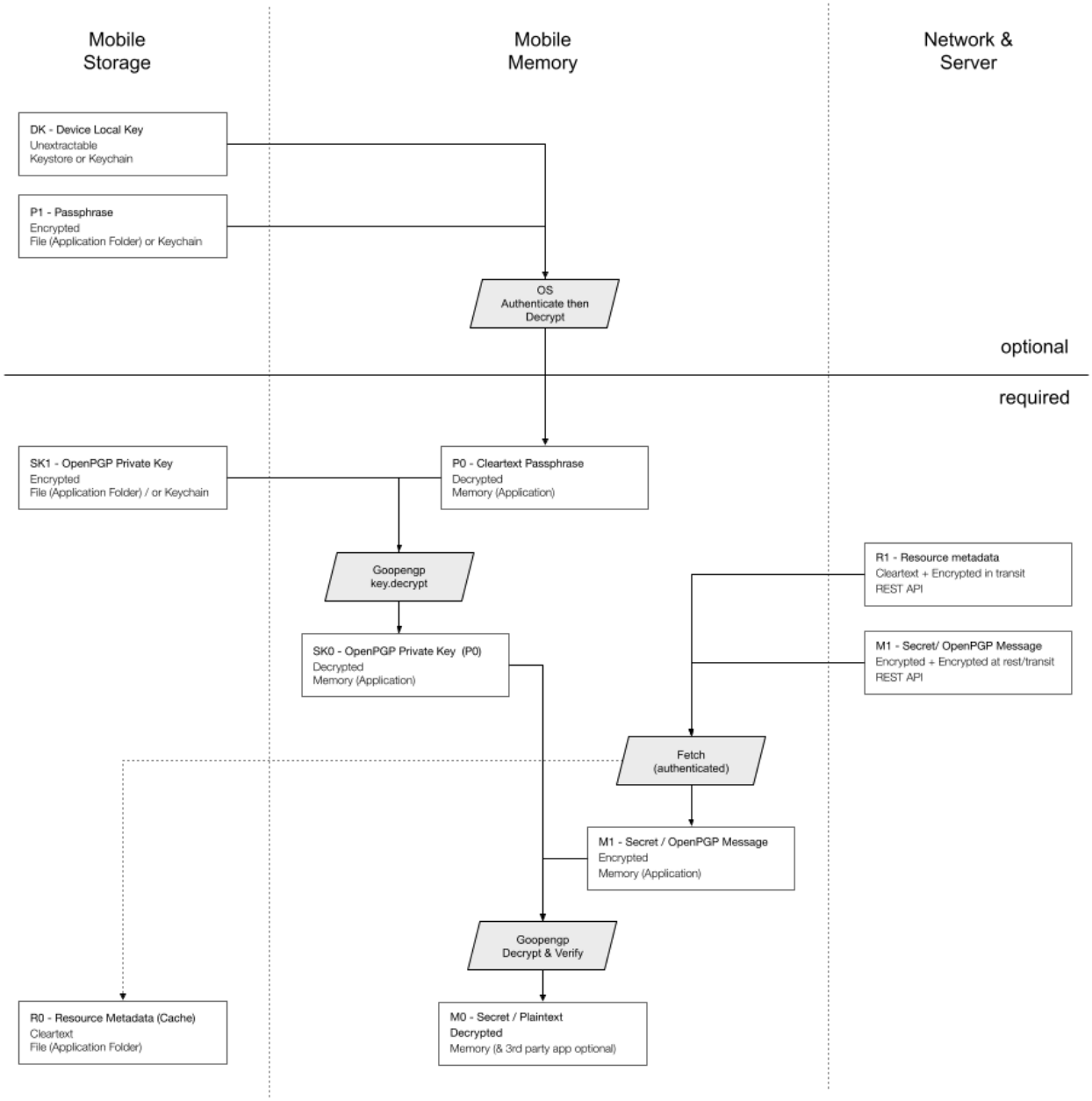


Fig. steps and data by location for decryption on mobile app

On Android

Android security guidelines suggest storing data in the device's internal storage. Internal storage is sandboxed per app on OS level so other apps can't access the files (unless apps belong to the same developer (are signed with the same certificate) or the device is rooted or an attacker uses specialized tools). Using internal storage doesn't require asking the user for a permission at runtime. When the user uninstalls an app, the device deletes all files that the app saved within internal storage. We can additionally use encryption on a file level (this is important especially for rooted devices).

We propose to build the key management system on top of the Android Keystore system. Its 2 part system for key management consists of:

- Device local key: a primary AES256-GCM key that encrypts all keysets. This key is stored using the Android keystore system.
- A keyset that contains one or more keys to encrypt a file or shared preferences data. The keyset itself is stored in the mobile application SharedPreferences.

The local device key is always stored in a Trusted Execution Environment. It is protected from extraction using two security measures:

- Key material never enters the application process. When an application performs cryptographic operations, behind the scenes plaintext, ciphertext, and messages to be signed or verified are fed to a system process which carries out the cryptographic operations.
- Because key material is bound to the secure hardware (e.g., Trusted Execution Environment (TEE), Secure Element (SE)) of the Android device, it is never exposed outside of secure hardware. If the Android OS is compromised or an attacker can read the device's internal storage, the attacker may be able to use any app's Android Keystore keys on the Android device, but not extract them from the device.

When creating a Master Key, we can specify additional configuration parameters. The most important ones are:

- `userAuthenticationRequired` and `userAuthenticationValiditySeconds` can be used to create a time-bound key. Time-bound keys require authorization using BiometricPrompt for both encryption and decryption of symmetric keys.
- `unlockedDeviceRequired` sets a flag that helps ensure key access cannot happen if the device is not unlocked.

- Use `setIsStrongBoxBacked`, to run crypto operations on a stronger separate chip. This has a slight performance impact, but is more secure.
- `setInvalidatedByBiometricEnrollment` sets whether this key should be invalidated on fingerprint enrollment. By default, it is set to *true*, so keys that are valid for fingerprint authentication only are irreversibly invalidated when a new fingerprint is enrolled, or when all existing fingerprints are deleted. That may be changed by calling this method and passing *false* as an argument. Invalidating keys on enrollment of a new finger or un-enrollment of all fingers improves security by ensuring that an unauthorized person who obtains the password can't gain the use of fingerprint-authenticated keys by enrolling their own finger. However, invalidating keys makes key-dependent operations impossible, requiring some fallback procedure to authenticate the user and set up a new key.

For profile and resource metadata our preferred approach would be to use a SQL compatible database, for convenience. Google offers a library called Room which makes it easier and safer to deal with databases on Android. There are other popular solutions like SqlDelight. Both options could be used in combination with SQLCipher, with a passphrase randomly generated and stored and encrypted in the same fashion as the user passphrase.

Reference:

- https://developer.android.com/training/articles/keystore
- https://www.zetetic.net/sqlcipher/design/

Sample code

- https://github.com/passbolt/passbolt-poc-android/tree/master/app/src/main/kotlin/com/passbolt/poc/util

On IOS

Applications on iOS are sandboxed, so they don't have access to another's app storage, unless these applications are in the same App Group or the device is jailbroken. When the application is uninstalled, all the data associated with the app is deleted as well. Application files are encrypted by default when the device has set an active passcode. Application storage is limited only by left space on the device.

We propose to build the key management system on top of the iOS Keychain service. Keychain items are encrypted using two different AES-256-GCM keys: a table key (metadata) and a per-row key (secret key). Keychain metadata (all attributes other than *kSecValue*) is encrypted with the metadata key to improve search time, while secret value (*kSecValueData*) is encrypted with the secret key. The meta-data key is protected by Secure Enclave but cached in the application processor to speed-up keychain queries. The secret key always requires a roundtrip through the Secure Enclave.

Keychain allows to choose the protection level of data stored inside. By default, keychain data is accessible only when the device is unlocked. The list of all protection levels is described here, but here are the one we propose using:

- The most strict option is `kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly` that prevents the data to be backed up or synced with iCloud, what seems to be crucial in the Passbolt app.
- We can also demand user authentication to access Keychain data. `userPresence` flag lets the system choose appropriate authentication method (TouchID, FaceID or a passcode).
- There are more flags like `biometryCurrentSet` that constraints the keychain item for currently enrolled fingers or from Face ID with the currently enrolled user. More information about restricting access to the Keychain can be found here.

Our approach would be to store the user secret key and the passphrase as keychain items. For the application data, such as resource metadata, similarly our preference would be to use SQL compatible databases, such as SQLite (supported in iOS by default) or Realm or SQLCipher.

References:

- Apple "Keychain data protection overview"
- Apple "Restricting Keychain Item Accessibility"
- Apple "App security overview"
- Apple "Data Protection overview"
- Apple "Encryption and Data Protection overview"
- Apple App Group
- Realm encryption overview

Sample code

- https://github.com/passbolt/passbolt-poc-ios/tree/master/Passbolt%20POC/Milestones/SecureStorage

Additional risks (Mobile)

Risk type	Counter measure	Residual Risk
Rooted device	Root detection	TBD
App patching	Make sure that users download app only from genuine stores	Out of scope
Biometric authentication bypass (known fingerprint/FaceID vulnerabilities)	Optionally user can enter passphrase to the app	Out of scope
Physical attack on user	Introduce emergency passphrase, that locks/clears the app	Out of scope
Presence in RAM of unencrypted data	Make sure that device is not rooted, that prevents from memory dump?	TBD
Long “unlocked” time after successful authentication	Choose appropriate time-bound key for operations on secure storage	TBD

Enable cross-device secret key transfer using QR Codes

Requirements & constraints

As part of the setting up of a passbolt account on a new device, the user profile data and the associated OpenPGP secret key needs to be transferred easily and securely between a configured client and another non configured client, for example from the webextension to the mobile device.

Since an OpenPGP RSA key is quite large it cannot be typed directly by the user, or transferred using a single QR Code. Similarly it is not user friendly, and possibly insecure, to ask the user to download and handle the secret key transfer manually through another channel of their choice.

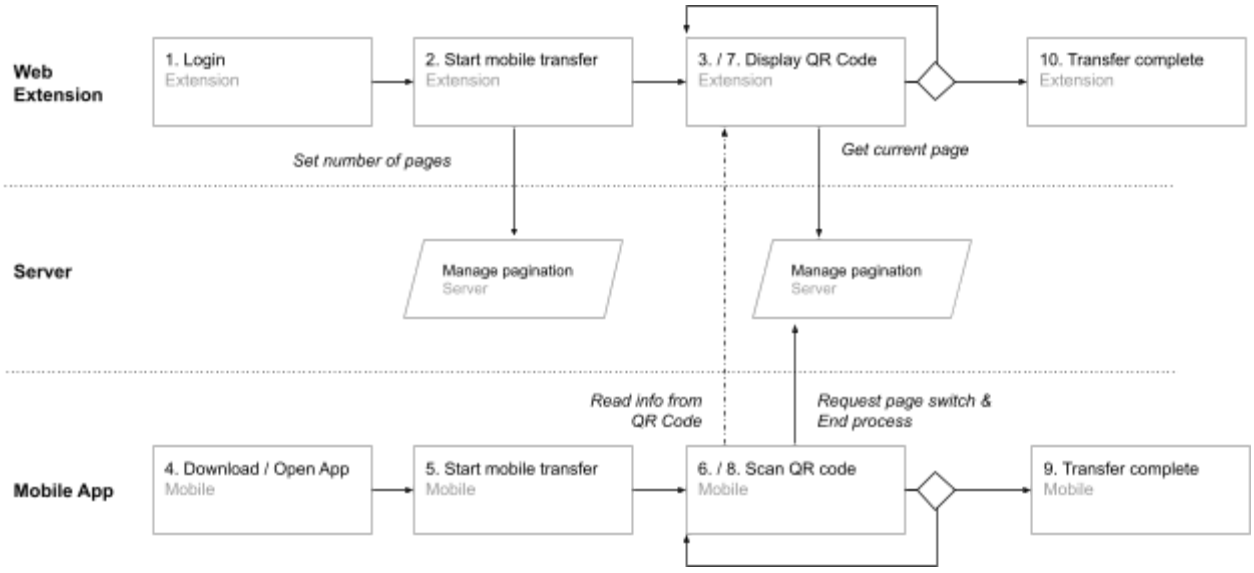
In order to limit residual security risks we also want to avoid transferring the secret key to the same server where the encrypted messages are present. This might be counterintuitive but we do not want to store the secret key server side even if it is encrypted with the passphrase or through another one-off layer of symmetric encryption. One of the reasons is that there is a strong expectation from users that the key will only stay on the device, especially for more advanced users where an existing key, for example used also for email encryption, is imported in passbolt.

Moreover to simplify the hosting (and therefore support) of the solution, we cannot introduce additional server services and/or additional networking requirements. Therefore we cannot rely on additional protocols, such as WebSocket or WebRTC. Finally that protocol should not be device specific, e.g. it should be possible to reuse this method across similarly capable clients: a mobile app, an extension, a native desktop app.

Selected solution

In a nutshell the solution relies on multiple QR Codes and an API on the server to synchronize the pagination between the two clients. The minimum requirements for two clients to exchange data are as follow:

Requirements	Configured device	Non configured device
Ability to generate / display QR Codes	Required	
Access to API server via HTTPs	Required	Required
Access to a camera		Required
Ability to parse QR Codes		Required



In a nutshell the high level process from a user perspective is as follow:

1. The user sets up their account using the webextension. The user login using the web extension, goes to their settings, under the mobile section.
2. The user sees a page in the webextension explaining how the process works. They click on a button to start the process, they are requested to enter their passphrase, in the background the extension registers a transfer, and receives from the server an authentication token to be used by the mobile phone to connect to the server later.
3. The users sees a QR code displayed in the webextension as well as a progress bar set to the beginning.
4. The user downloads the mobile application from the marketplace and opens it.
5. By default on mobile there is no user profile, the user is showns a page explaining the process and a button to get started. If there is already another profile configured for another user account, the user can go to the mobile application settings and add another profile.
6. The user starts the process, provides some authorization to use the mobile camera if needed, and using the mobile phone scans the first QR code. The mobile uses the URL contained in the QR code to request the next data page, it returns it gets information about the total number of pages and a hash to assert the integrity of the data at the end of the process.

7. The webextension reads the page change request from the server and displays the next QR code.
8. The mobile application scans the QR and turns the next page. The process continues until all the pages are read.
9. Once the last page is read the mobile application sets the transfer status to complete and display a success screen. The user can then proceed to login using the private key.
10. Similarly a success message is displayed on the webextension side.

In the future we could easily imagine reversing the process with a desktop application or web extension not configured that could, using the camera, read the profile information from the mobile.

Protocol definition

Sequence

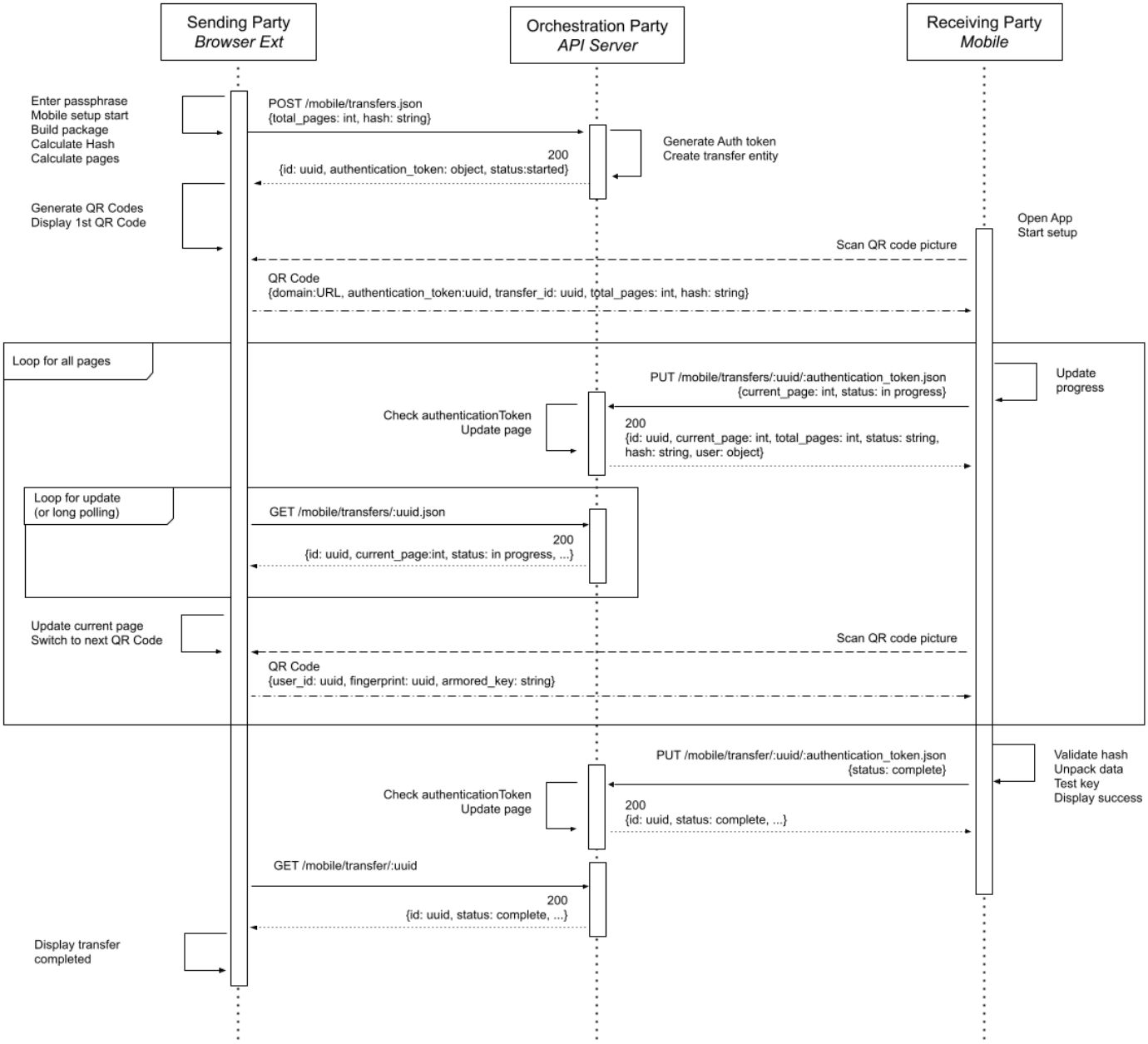


Fig. QR Code Exchange Sequence Diagram (Success Scenario)

Possible Transfer Statuses

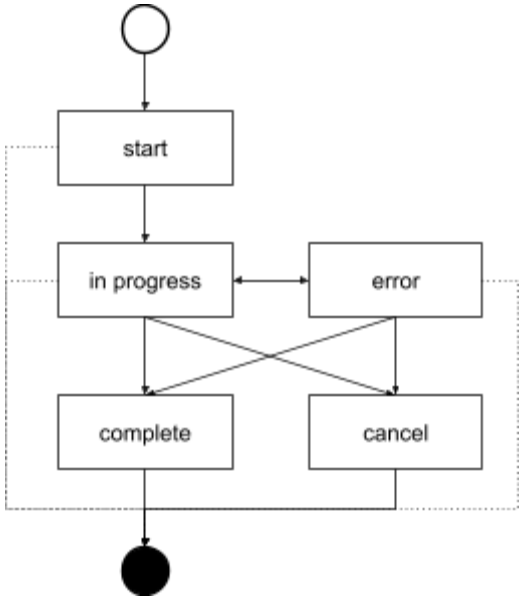


Fig. QR Code Exchange State Diagram

As shown in the sequence diagram the ‘start’ state is created by the sending party and the rest of the status updates are controlled by the receiving party. Generally the flow will be ‘start’, ‘in progress’, ‘complete’.

However there is also the possibility for the post party to ‘cancel’, or set an ‘error’. The ‘error’ status can be used for example by the receiving party to retry a previous page for example if a scanning failed.

The ‘cancel’ status can be used by both parties if the user requests to cancel the operation, for whatever reason. Unlike the error status, and much like a complete status, the cancel status is definitive.

Additional risks

Risk type	Counter measure	Residual Risk
Secret key capture using client/server MiiM	No key is sent over the network.	None?
XSS in Web Extension shouldn’t	Display QR code in a separate	TBD

provide access to secret keys.	iframe to prevent access.	
Unwarranted key export.	Request passphrase before starting transfer.	TBD
CSRF Attack	Unique authentication token and transfer ID.	Out of scope
DDos attack on server endpoint affecting availability of service	CDN / WAP Firewall (Not provided)	Out of scope
3rd party applications snoop on camera / screen and record.	OS Level + Secret key is encrypted with passphrase.	Out of scope
3rd party camera / people record screens from afar.	OpSec + Secret key is encrypted with passphrase.	Out of scope

No sensitive data is transmitted over the network, but only client to client, the risks with regards to the key confidentiality or integrity are reduced.

To protect the secret key further on the extension side, the process will only start after the user enters their passphrase. The secret key QR code will be displayed in a separate iframe, separate from the main application context, to protect the secret key in the context of XSS in the main application.

Considering the transfer ID is random and unique and tied to a random and unique user authentication token, there is no need to provide a CSRF token in the header like with other API endpoints, for the mobile application.

Residual risks remain if an attacker has the ability to access the client memory and/or the ability to view/record the browser screen, and/or has access to the camera. These types of attacks are considered out of scope.

Similarly an attacker could affect the availability of the transfer if they are able to mess with the pagination. By providing authentication for both clients we consider that this risk is limited sufficiently. DDOS attacks are considered out of scope.

Add optional escrow key

Currently if a user loses their private key or passphrase, the secrets that have not been shared with other users are lost. This design is intentional. However in some regulated environments (banking, insurance, statecraft) administrators are legally required to have a way to access encrypted data, for example to transfer secrets when a user leaves the organization.

To solve this problem we propose to introduce the concept of escrow key, and leverage the existing “share” capabilities built in passbolt. In practice, the administrator configuring the instance will generate an escrow private key and upload the associated public key.

We propose administrators will be allowed to select one of the following escrow modes:

- No escrow (Default). As it is currently.
- Escrow for shared password with opt-out option. All passwords that are shared with more than one person will by default be subject to be encrypted with the escrow key. However the user can “remove” the escrow account when sharing.
- Mandatory escrow for shared secrets. Same than previous but without the ability to opt-out.
- Full escrow. All passwords require to be encrypted with the escrow key, including personal secrets.

Additional risks

Risk type	Counter measure	Residual Risk
Not all secrets are in escrow by default if activated on an existing instance.	Produce reports to show level of compliance with escrow policy. Trigger background process to add missing secrets to escrow after login for users.	TBD
Escrow key is misused by administrator or leaked	Recommend storing escrow key in 3rd party safe location with strong authentication (bank safe?).	TBD
Escrow key is used without users knowledge	Send email notifications to other administrators when the key is used. (TBD) Shamir secret sharing for secret key passphrase.	TBD

Attacker replaces the escrow key with his own to extract secrets.	Escrow key changes require the user manual approval. Escrow key requires administrator signature.	TBD

Add support for multiple authentication schemes

Unlike most password managers, passbolt is currently based on an authentication scheme called GpgAuth that is based on a challenge that requires the secret key and passphrase (and not, for example, a version of the passphrase that is hashed and sent to the server for validation).

If we can validate the "Removing the need for the user to enter an OpenPGP key passphrase" solution for the web browser, this system can not be used with this option enabled, as the secret key and its passphrase are required for this authentication mode. However it could be used after another method of authentication (say SSO), once the passphrase is recovered, for a strong authentication based on the secret key.

Supported schemes

	Authentication providers		
Authentication mode	GpgAuth	LDAP	OpenID Connect
Current mode	Yes	No	No
SSO without passphrase	Optional after	Yes	Yes
SSO with additional passphrase	After one other	Yes	Yes

GpgAuth

GpgAuth is the authentication provided by default by passbolt. It is a challenge based mechanism where the user (and/or server) needs to prove they can decrypt a secret encrypted with the user (and/or server) public key, and/or sign with the secret key.

More detailed information about each step can be found in the security white paper or the online documentation.

LDAP authentication

In this scheme the LDAP username and password are sent to the application. The application builds the distinguished name, for example "`uid=ada,ou=People,dc=passbolt,dc=com`" (see. DN or RDN) that is then sent with the password, performing an LDAP `bind` operation server side to verify the authenticity of the credentials.

It offers several advantages:

- **User friendly:** same password used everywhere from the user perspective.

It offers several disadvantages:

- **Security:** prone to phishing, replay, brute force attacks… Password is sent to a 3rd party service before being relayed.

- **Introduce a single point of failure:** if LDAP is unavailable the user cannot login. A commonly used technique consists of caching the passwords locally in the application, which in turn produces more issues (e.g. integration needs to be built to keep password in sync for example, hashing may be weak in certain applications, etc.).

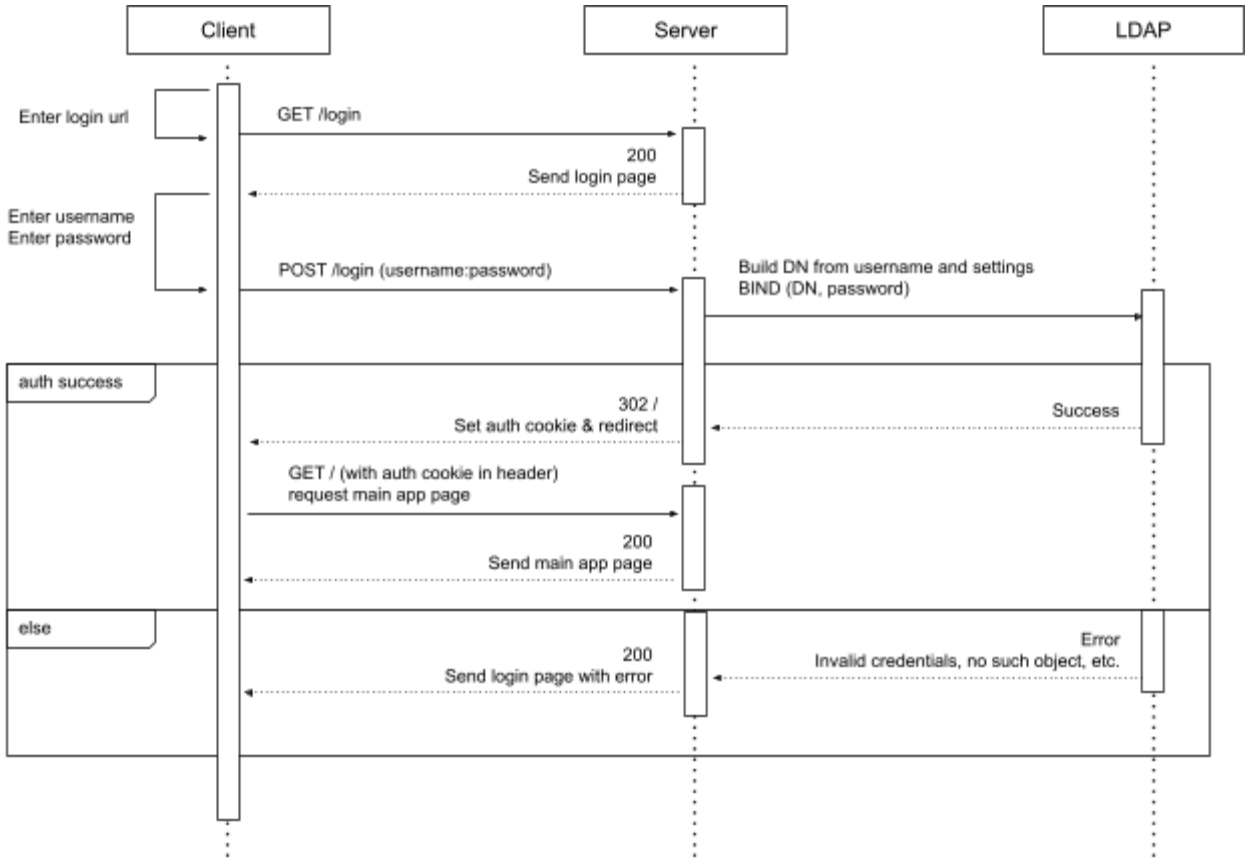


Fig. LDAP based authentication

OpenID Connect

In decentralized authentication schemes such as OpenID connect (OIDC), the client (in our case the browser) is redirected from the relying party (in our case the passbolt instance), to an authentication provider (a third party service), with the claim (user info) and the flows (authentication scheme) they want to use.

The user must then authenticate against the 3r party authentication provider. Once authenticated they will need to give trust to the relying party if it is not already trusted. Once trust is given, the user is then redirected to the relying party with the credentials, typically in the form of a token containing the user claim signed by the authentication provider. That claim is then checked by the relying party using a secret previously shared between the relying party and the authentication provider.

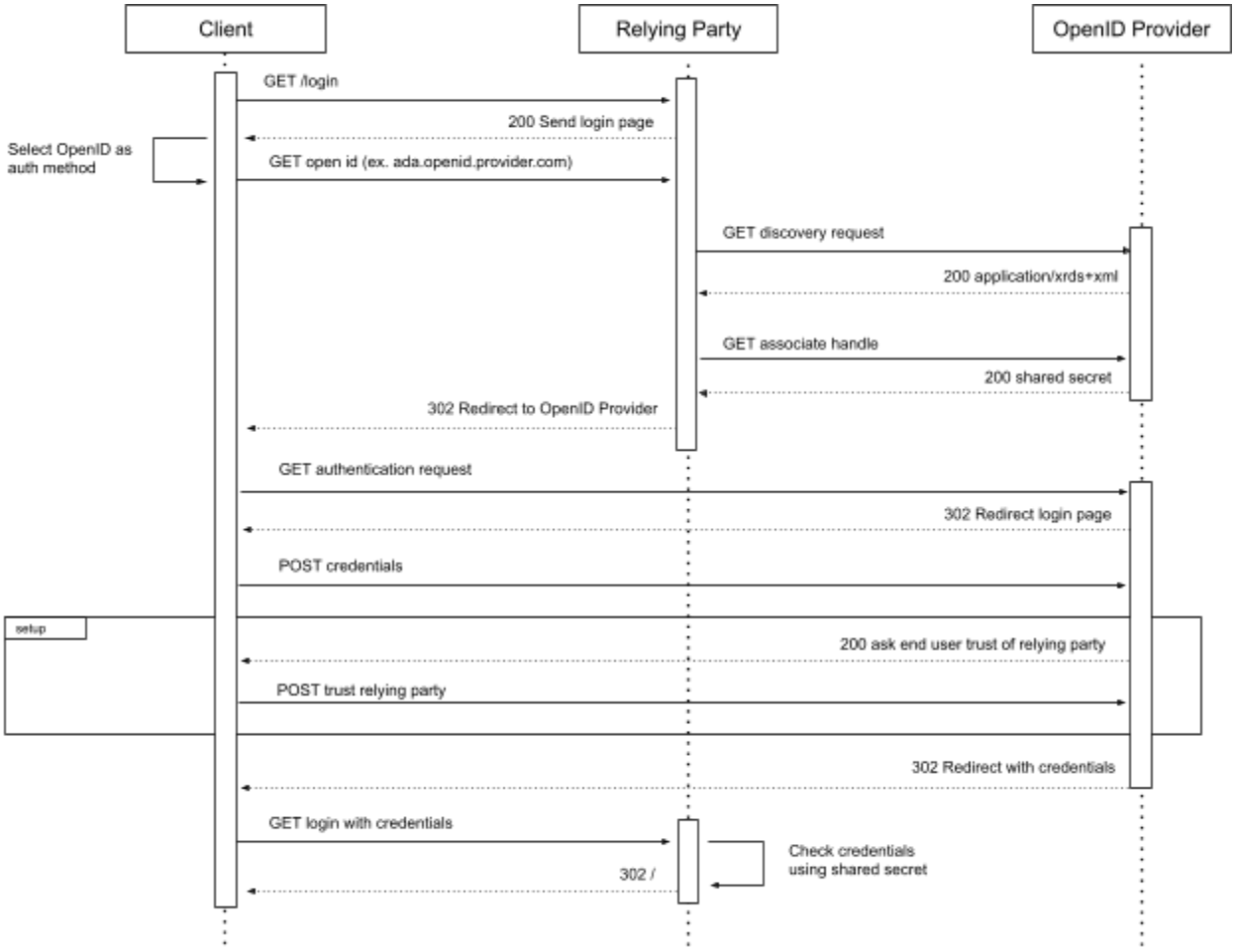


Fig. OpenID based authentication

There is a lot more to OpenID connect than what's the above. There are many variations of the flows as well as many options built-in the protocol, designed to support different usage. But in a nutshell we want to enable the passbolt server to act as a relying party.

Reference:

- https://developer.okta.com/blog/2017/07/25/oidc-primer-part-1
- https://auth0.com/docs/videos/learn-identity-series/introduction-to-identity