



Security White Paper

Passbolt Pro Edition v5.10 (rev10)

March 2026

Table of content

Introduction	7
Guiding principles	8
Core concepts	10
Resources & secrets	10
Resource permissions	11
Groups, Folders & Tags	11
Roles	12
System-wide roles	12
Group level roles	12
Logical roles	12
Crypto Overview	13
Encryption Across the Data Lifecycle	13
Data in use	13
Data in motion	13
Data at rest	14
Pseudo random number generator (PRNG)	14
Passphrase/Password Generator	15
OpenPGP	15
Core mechanics	15
Server-Side Implementation	16
Client-Side Implementation	17
Browser extension	17
Mobile and CLI Support	17
Encryption keys	17
User keys	18
Secret Key Passphrase	19
Public key exchange	19
Metadata keys	20
Organization Recovery Keys	21
Server keys	21
Authentication	23
Principles	23

GpgAuth	23
Server key verification steps	24
Login steps	25
Token format	25
GpgJwtAuth	26
Login steps	27
Multiple Factor Authentication (MFA)	27
TOTP	27
Yubikey OTP	28
Duo	29
Single Sign On (SSO)	29
SSO Login Process	30
SSO keys	31
Application architecture	33
Application components	33
Server components	33
JSON API	34
Server-side JavaScript	35
Server side command line tools	35
Cron Jobs & email queue	35
Database	36
Browser Extension	37
Layers & logic separation	37
Type of storage	38
Browser Extension Permissions	39
Risks mitigation strategies	41
Security Logging and Monitoring	42
User action logs	42
Application error logs	42
SIEM Integration	42
Authentication	42
Oracle Attacks	42
JWT	43
MFA Rate Limiting	43
Server Side	43
Access Control and Authorization	43
Input Validation and Sanitization	44
Key Validation	44

PHP exec	44
Unsecure deserialization	44
SQL Injection	44
File upload	45
CSRF	45
Server-Side Request Forgery (SSRF) Protection	46
Transport security	46
Security headers	46
Cookie security	46
LDAP Security	46
Browser Extension	47
Passphrase Phishing	47
Clickjacking Protection	47
Memory Security	47
Cross Site Scripting (XSS)	47
Persistent XSS	47
Reflected XSS	48
Content Security Policy	48
URL Validation and Suggestion Security	49
Use of unsafe methods	49
Javascript eval	49
Trusted domain	49
Autofill	49
Background / Content Script message spoofing	49
Manifest Security	50
Secure development best practices	50
Continuous integration and testing	50
Developer authentication	50
Signed releases	50
Code review and publication	51
Dependency monitoring	51
Static code analysis	51
Responsible Disclosure	51
Annual Security Audits	51
Source Code Audit	51
SOC 2 Type II Audit	52
Packaging	52
CIS Benchmarks	52

Non-root Docker Image	52
Residual risks	53
Compromised cryptographic primitives	53
Quantum resistance	53
Weak server side random number generation	53
Compromised Network	54
Man in the middle	54
Exposed metadata (applicable to v4 resource types)	55
Other Exposed metadata	55
Misconfigured server	56
Unverified Email in SSO Mapping	56
Compromised server	56
Server Access in Non-Zero Knowledge Mode	56
Metadata & secret integrity issues	56
User key integrity / group membership integrity issues	57
Compromised client	57
Memory access	57
Filesystem / keylogger access	57
Clipboard access	57
Misconfigured client	58
Weak secret key passphrase	58
Weaker key size / algorithms	58
Anti-phishing token	58
Malicious visitor	58
User enumeration	58
Homoglyph-based User Impersonation	58
Malicious logged in user	59
Data modifications / invalid encrypted content	59
Malicious public keys	59
Unsafe resource export	59
Malicious extension	60
Rogue vendor employee	60
Malicious third party website	60
Exposed plugin info	60
Included iframe	60
Best Practice Checklist	61
Metadata Encryption Configuration	61
HTTPS and TLS	62

Database	62
Backup	62
OS Hardening	63
Network Segmentation	63
Reverse Proxy	63
Identity Provider and MFA	64
Endpoint and Extension Control	64
Acknowledgements	65
Document Revision History	66

Introduction

Passbolt is a platform designed for organizations to centralize the management of credentials and related shared sensitive information.

This security white paper is our commitment to transparency. It provides a thorough technical overview of the design, architecture, security mechanisms, and residual risks associated with Passbolt, so you can evaluate the solution with confidence and clarity.

This document is for security professionals, IT administrators, auditors, and decision-makers who recognize the importance of choosing the right foundation for protecting their organization's secrets. We assume readers have a basic understanding of cryptography, web application security, modern authentication methods, and the high-level functionalities of Passbolt as a collaborative credential management solution.

Thank you for investing your time in reviewing this white paper. We hope it answers your questions and helps you make informed, sound decisions.

Guiding principles

When the Passbolt team embarked on the ambitious journey of creating the best possible credential manager designed for collaboration, we started by defining a few guiding principles. Here are some of these principles that helped shape the product in its current form:

Client-side end to end encryption. The server does not have access to secrets in cleartext form. The client is responsible for generating user keys, as well as encrypting and decrypting secrets. Moreover the code responsible for sensitive operations is distributed through another trusted channel and thus can never be altered by gaining access to the server.

Granular encryption and secrecy. Each secret is encrypted once for each user, and only when they need to have access. Only a user with access to a secret (and the permission to do so) can share content with another user. Removing an access means removing the ability to decrypt past and future versions of the secret.

No secret key, including derivatives, server side without consent. By default an encrypted or derived version of the user secret key should never be sent or stored server side. Encrypted copies of the secret key can be stored on the server to facilitate account recovery, but clear approval or warnings should be presented to the users.

Strong non-phishable authentication. Passbolt must rely on a cryptographic challenge mechanism tied to the current domain, instead of a password based one. Authentication must be multi-factor by default, e.g. something the user has, and something they know (or are).

Interoperable cryptography. A user must be able to take away any encrypted content and decrypt it with the tools of their choice, not just the one provided by us. Advanced users should have the right to select which algorithm to use and which system they trust to generate cryptographic keys.

Free and open source software. Both the client and the server should be fully available in an open source license. The underlying software stack required to run the server should also be open source.

No mandatory internet access. The application should not rely on scripts hosted on a 3rd party domain or require you to create a user account elsewhere. While some functionalities such as third party integrations will always depend on internet access, the software must remain usable in a closed network with no internet access.

Privacy by design. No tracking. The software should not store unnecessary personal information. The software should not report back on user behavior or aggregated analytics unless the administrator or the user explicitly opt-in.

Security by default. The solution should propose sane security settings by default. Administrators can however still adjust the settings to match their security requirements and risk appetite.

Trust starts with honesty. No security solution is perfect, and passbolt is no exception. We are committed to clearly communicating known limitations, design trade-offs, and residual risks so users can make informed decisions. When vulnerabilities are discovered, we strive to respond promptly, communicate clearly, and take responsibility.

If these principles are also important to you, there is a good chance that passbolt will be the right fit. The rest of the document will provide you with information about the implementation details and associated risks, so that you can make that call.

Core concepts

If you are new to passbolt, this chapter will help you understand how credentials are structured and managed. It explains the core concepts such as resources, secrets, as well as user groups and roles.

Resources & secrets

In passbolt, the different types of credentials (e.g. a password, a TOTP, etc.) are called resource types. The credentials are processed and stored as two distinct parts, the resource and the secret.

For example, in the case of a credential of type password, the name, username, url and other metadata will be included in the “resource” part. The password itself will be in the “secret” part. This split allows fine grained access control and logging.

The schema of which fields are included in which section is described using [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 in practice they are generally hard coded client side.

```
{
  "resource":{
    "type":"object",
    "required":["name"],
    "properties":{
      "name":{
        "type":"string"
      }, //etc.
    }
  },
  "secret": {
    "type":"object",
    "required":["password"],
    "properties":{
      "password":{
        "type":"string"
      }, //etc.
    }
  }
}
```

```
}  
}
```

Fig. example of a simplified credential using JSON schema format

Resource permissions

Passbolt offers three permissions at the resource level.

- **Owner:** can manage permissions (i.e. share), delete, update, read/use (i.e. autofill, copy, etc.).
- **Update:** can update, delete, and read/use.
- **Read:** can only read/use.

Groups, Folders & Tags

Resources can be organized in multiple dimensions.

Folders serve as a template for permissions, applying them to resources, but also to other folders, during operations like item creation or relocation.

Groups allow administrators to define sets of people, such as departments, project teams, or external collaborators. Groups can be used to assign permissions to specific resources or folders.

Tags do not carry permissions. They are used to complement the other two dimensions by offering a non-hierarchical method to classify resources. Tags can for example allow users to label credentials by attributes such as environment type (e.g., staging, production). Tags can be personal or shared. Shared tags starts with the character # (e.g. #our-tag).

Roles

System-wide roles

The system proposes by default two system roles “admin” and “user”. This system is at the core of the authorization mechanism for user actions.

In a nutshell, an administrator manages the instance. In practice it means that they can create or delete users and groups, manage organization-wide settings such as the content of the email notifications or which multiple factor authentication provider is enabled, or perform synchronization with a user directory, etc.

Group level roles

Each group must have at least one group manager in charge of adding and removing group members. The administrators can appoint themselves as group manager or appoint a regular user.

Only someone with access to the secrets of a given group can add a member to that group (as they need to be able to decrypt and encrypt the secret for the new member).

Logical roles

Finally another layer of logical access control can be enforced depending on the context, most notably when the user is the creator of a given item and no specific permission system applies to it. For example, a user can edit their own name.

Crypto Overview

This chapter provides a comprehensive overview of the solution cryptographic architecture, detailing how encryption is applied and how the OpenPGP standard is leveraged. It also covers the core cryptographic primitives, key management strategies, and implementation details across server and client environments.

Encryption Across the Data Lifecycle

We can consider three different stages of the data lifecycle: in use, in motion, and at rest. Multiple layers of encryption serve a distinct purpose in securing sensitive information in these different stages. While some protections are handled directly by Passbolt, others rely on external configuration and best practices set at the hosting environment level.

Data in use

Passbolt servers never see the plaintext secrets, making the system resilient against server-side data breaches. Therefore at the application layer, sensitive data is encrypted before being transmitted server side, a principle that helps protect against unauthorized access even if the underlying transport or storage layers are compromised.

In practice, this encryption is performed client-side, within the user's browser, mobile app, or CLI, using asymmetric cryptography. Each user possesses their own private key, which is securely stored and used locally to decrypt data, ensuring that only authorized users can access sensitive information.

Data in motion

For the data in motion, i.e. at the transport layer level, all the communications are encrypted using TLS. The strength of the security at that level is not controlled by the passbolt solution itself but rather a combination of other factors such as the level of security of the

organization issuing the certificate and the web server configuration chosen by the hosting provider.

Passbolt makes reasonable efforts to help enforce encryption for the data in motion. By default passbolt installation scripts will help the administrator set up certificates on the server. Similarly, the default configuration of passbolt server makes sure a non encrypted request is redirected to its HTTPS counterpart. While it is possible to disable that behavior, it will trigger some warnings in the application health checks.

Data at rest

Passbolt can be deployed across a wide range of hosting environments and supports multiple database engines, offering flexibility to meet different infrastructure needs. However, this diversity also means that the underlying system-level operations, such as file system or database encryption, vary significantly depending on the chosen setup.

As a result, Passbolt does not provide an out-of-the-box solution for configuring encryption at the storage layer. Instead, these additional security measures must be implemented by administrators based on their specific operating system, database engine, and hosting provider.

Pseudo random number generator (PRNG)

Cryptographically secure pseudo-random bytes generation is a critical component of any secure system.

On the server side, Passbolt relies on GnuPG, which uses its own cryptographic random number [generator](#) built on top of the operating system's entropy sources. In virtualized environments administrators may consider enabling additional entropy sources, such as hardware RNGs, to ensure secure key generation and cryptographic operations.

The PHP application uses the [random_bytes](#) function, which generates cryptographically secure random values using the Linux kernel's [getrandom](#) system call. This PRNG is used in sensitive components such as generating random UUIDs for authentication challenges and creating proofs for two-factor authentication.

On the web extension side, the application uses the browser's native [crypto.getRandomValues](#) function from the Web Crypto API as its source of entropy. This cryptographically secure PRNG provides the randomness seed consumed by the OpenPGP library for operations such as key generation and secret encryption, and also powers the password generator functionality. It is important to note that the output of `crypto.getRandomValues` is not used directly as key material. As the Web Crypto API documentation advises, raw values from this function should not be used as cryptographic keys without further derivation. In Passbolt's case, the entropy is passed to the OpenPGP layer, which handles proper key construction internally, and to the password generator, which uses it to produce random character sequences rather than cryptographic keys.

Passphrase/Password Generator

Passbolt itself offers a cryptographically secure password generator. With the default settings this generator creates high-entropy passwords of 18 char in length with upper/lower case letters, numbers and special characters. Passbolt does also offer options to further customize/configure these defaults.

OpenPGP

The application level encryption of passbolt is built upon multiple layers of [OpenPGP](#) support, using different implementations and integration strategies for both the server and client components. The current standard is formally specified in [RFC 9580](#).

In this section we review briefly how it is used and implemented in the context of passbolt.

Core mechanics

OpenPGP is based on a combination of cryptographic techniques:

- **Public-Key Cryptography:** Each user has a key pair (public + private). The public key is used to encrypt messages or verify signatures; the private key is used to decrypt or sign.
- **Symmetric Encryption:** A random session key is generated for each message. This key encrypts the message content and is itself encrypted with the recipient's public key.
- **Digital Signatures:** The sender signs the hash of the message using their private key. The signature can be verified by the recipient using the sender's public key.
- **Key Derivation:** Private keys are typically protected by a passphrase, which is converted into an encryption key using a String-to-Key (S2K) mechanism to derive a symmetric key.

Passbolt clients uses by default a subset of OpenPGP cryptographic primitives:

- **Asymmetric Algorithms:** such as RSA (with 3072-bit, 4096-bit key size), or modern ECC-based algorithms (e.g., EdDSA with Curve25519).
- **Symmetric Ciphers:** such as AES-256 in CFB mode.
- **Hash Functions:** such as SHA-256 or SHA-512.

Server-Side Implementation

On the server side, Passbolt uses [openpgp-php](#), a pure PHP implementation of the OpenPGP message format. This component is responsible for pre-validation of keys and messages. This helps to parse and inspect data before passing it on for cryptographic operations.

The actual signing, encryption, verification, and decryption operations are performed by GnuPG, accessed through the native [gnupg-php](#) PHP extension, maintained by the PHP Foundation. This extension acts as a bridge to [libgpgme](#), the official high-level C API for GnuPG. In 2019, Passbolt introduced an abstraction layer to decouple the application from GnuPG, enabling support for alternative backends.

Client-Side Implementation

Browser extension

On the browser extension side, Passbolt uses [OpenPGP.js](#), a JavaScript library for OpenPGP operations in the browser.

This library is also used by well-known secure communication tools such as ProtonMail and Mailvelope. Its security has been externally validated through multiple security audits by Cure53, notably in [2015](#) and [2018](#).

Mobile and CLI Support

For mobile clients and command-line interface (CLI) tools, Passbolt uses [GopenPGP](#), a Go-based wrapper built on top of a fork of the golang crypto library¹. OpenPGP library maintained by the ProtonMail team. GopenPGP provides a secure, memory-safe, and cross-platform abstraction over OpenPGP operations. The library underwent an audit by SEC Consult in [2019](#).

Encryption keys

One of the main differences with other password managers is that there is no symmetrically encrypted vault acting as a collection of credentials, shared with multiple users. Passbolt instead treats each secret individually. Each secret is encrypted once per user that requires access to that credential.

¹ Ref. [x/crypto/openpgp](#)

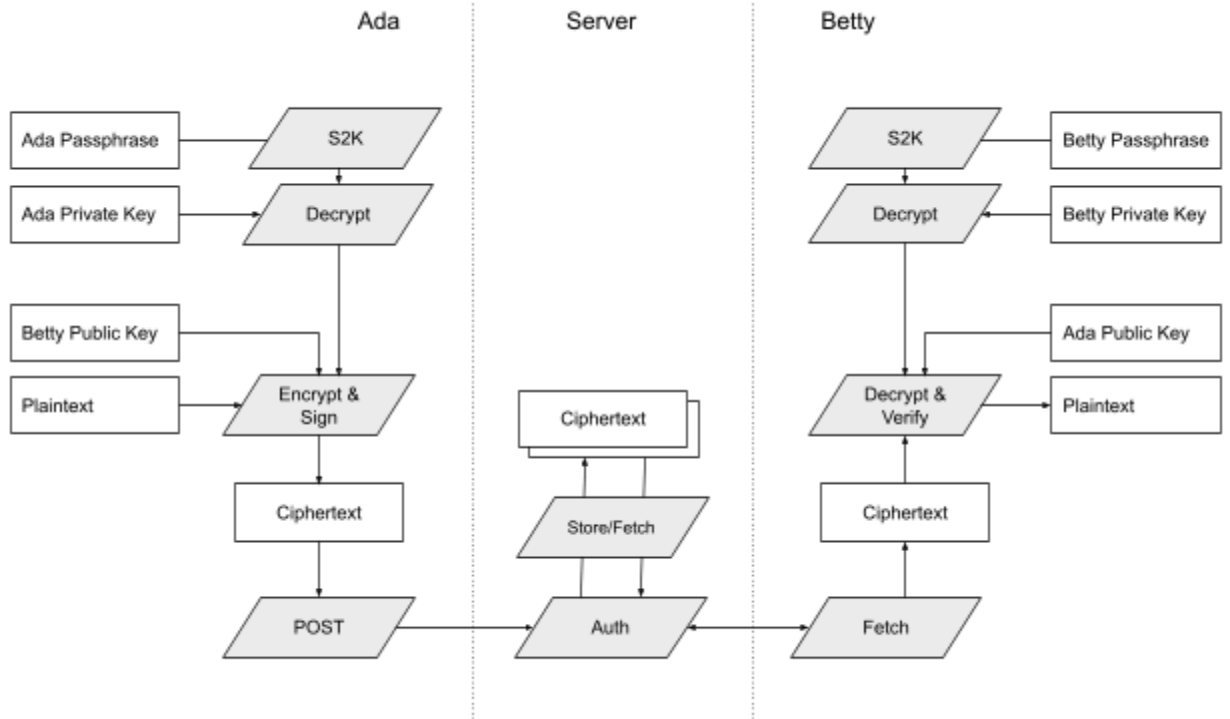


Fig. Encryption and decryption cycle in passbolt

Another difference is that the private key is not derived from the user selected passphrase, the server doesn't know anything about the passphrase. Additionally the private key, if stored server side, is not encrypted with a key derived from a user selected passphrase.

User keys

The primary OpenPGP key pairs are used by individual users to sign, encrypt and decrypt secrets and personal resources metadata. They are also used during the authentication process. If lost, non shared secrets encrypted with the user key are inaccessible unless account recovery is enabled.

Type of keys	Presence Type		
	Client Memory	Client storage	Server Side
OpenPGP Secret Key Passphrase	Cleartext	N/A	N/A

OpenPGP Secret Key	Cleartext	Encrypted	N/A** or Encrypted
OpenPGP Public Keys	Cleartext	Cleartext	Cleartext

** The OpenPGP private key is not stored server-side by default. It is only present server-side in encrypted form if the account recovery feature is explicitly enabled by the administrator.

The client secret key is generated during the initial setup. By default it uses elliptic curve cryptography keys instead (curve25519 / ed25519). Additionally it is possible for a system administrator to change the default server side RSA keys. Historically it used 3072-bit RSA keys, but it was/is possible to also use a larger RSA key by importing one.

Secret Key Passphrase

The secret key is encrypted with the user passphrase and persisted in that form in the client storage. The passphrase never leaves the device, and is not persisted in the client storage / only kept in memory.

When generating a key via passbolt by default the webextension only enforces the secret key passphrase to be at least 80 bits of entropy by default. It provides additional prompt for complexity and information such as if the passphrase has been compromised in a breach before (using [haveibeenpwned](#) service).

Passbolt proposes a “remember me” option for the passphrase in order to allow the user to reduce typing while performing multiple operations involving the secret keys. In the browser, unlike on mobile, for the lack of a better option, the passphrase is stored in cleartext in memory, along with the timeout function to reset it. Additionally, if the browser is closed the passphrase is cleared.

Public key exchange

Public key exchange, and the need for the user to manage the keyring, is often a pain point for the users of systems relying on OpenPGP.

Passbolt does not require manual keyring management or for a web of trust. Public keys are synchronized with the server. Authentication is required to fetch or add keys.

Therefore the client relies on the server to provide the valid user keys and hides the key exchange mechanism from the user. It is our opinion that this approach improves the ease of use of the system and is worth the risks that it introduces.

In practice during the setup, after a validation of the email address using a link sent by email, the public key of the user is sent to the server. It is validated then added to the database in a record associated with the user.

This public key is then distributed to the other users using an API endpoint that returns the OpenPGP keys when needed. This key synchronization takes place prior to all encryption involving multiple users.

The public key fingerprint is presented to users that want to verify it, in high risk scenarios. Possible future improvements are discussed in the residual risks section of this document.

Metadata keys

This type of key allows encryption of shared resource metadata, allowing multiple users (and optionally the server) to decrypt the metadata.

Type of keys	Presence Type		
	Client Memory	Client storage	Server Side
OpenPGP Secret Key Passphrase	N/A	N/A	N/A
OpenPGP Secret Key	Cleartext	Encrypted	Encrypted
OpenPGP Public Keys	Cleartext	Cleartext	Cleartext

The metadata keys are stored locally and server side using the user keys. Metadata keys are trusted on first use. They subsequently encrypted and

signed using the user key for future use. Any changes are displayed and require user manual approval.

The API supports the rotation of metadata keys, for example to ensure shared metadata forward secrecy when people leave the organization.

Organization Recovery Keys

These are optional organization level keys that are used to encrypt a backup of the user keys. This type of keys enable administrators to assist users in account recovery.

Type of keys	Presence Type		
	Client Memory	Client storage	Server Side
OpenPGP Secret Key Passphrase	Cleartext	N/A	N/A
OpenPGP Secret Key	Cleartext	N/A	N/A
OpenPGP Public Keys	Cleartext	Cleartext	Cleartext

Encrypting the user private key using the organization recovery key private key is a sensitive operation that requires manual user approval. Even though they can be stored in passbolt, it is recommended to store such organization recovery keys offline. Moreover they can be rotated if needed, for example after each use, so that they can be discarded after, allowing for a break glass type of workflow.

Server keys

This server key is used in the server verification part of the authentication (see Authentication section) as well as to encrypt / decrypt some sensitive settings stored in the database such as the LDAP credentials.

Type of keys	Presence Type		
	Client Memory	Client storage	Server Side
OpenPGP Secret Key Passphrase	N/A	N/A	N/A

Type of keys	Presence Type		
	Client Memory	Client storage	Server Side
OpenPGP Secret Key	N/A	N/A	Cleartext
OpenPGP Public Keys	Cleartext	Cleartext	Cleartext

The OpenPGP server key is set during the setup. It can be generated via `openpgp.js` in the web installation wizard (via a script in a page served by the server), or manually using the tool of choice of the administrator.

By default passbolt does not encourage using passphrases on the server secret key to facilitate the deployment. In practice the benefits of using a passphrase server side are quite limited, since it will need to either be stored unencrypted on file, or using environment variables or cached in memory using `gpg-agent`.

The client downloads the server public key during the setup (or account recovery) and ties it in the configuration of the client to the domain. If the key changes an error will be displayed, and the user is offered an option to accept the new key.

Authentication

Principles

Instead of a classic form based authentication, Passbolt performs a challenge based authentication called GpgAuth. This authentication mechanism uses Public/Private keys to authenticate users to a web application. The process works by the two-way exchange of encrypted and signed tokens between the user and the service.

On top of the usability benefit of not having to remember an additional password there are several additional benefits:

Phishing: this risk is mitigated because the client does not enter a password, i.e. getting the secret key passphrase alone would not allow an attacker to login. Since the client can verify the server identity based on the server key (pre-validated when added to the keyring), it is not enough for an attacker to fake a form and domain.

Authentication strength: the random part of the authentication token is 122 bits long (i.e. a random UUID) and is therefore stronger than a user selected password. Moreover a different secret is used for every authentication attempt.

Passphrase crackability: the secret used by the challenge is not related to the user passphrase. So in the event of a leak it provides no additional data that can help crack the user's password.

GpgAuth

This is the historical login method for passbolt. Once successful the server sets a session cookie that is used to identify the user. The session is deleted once the user logs out, or the session times out due to inactivity.

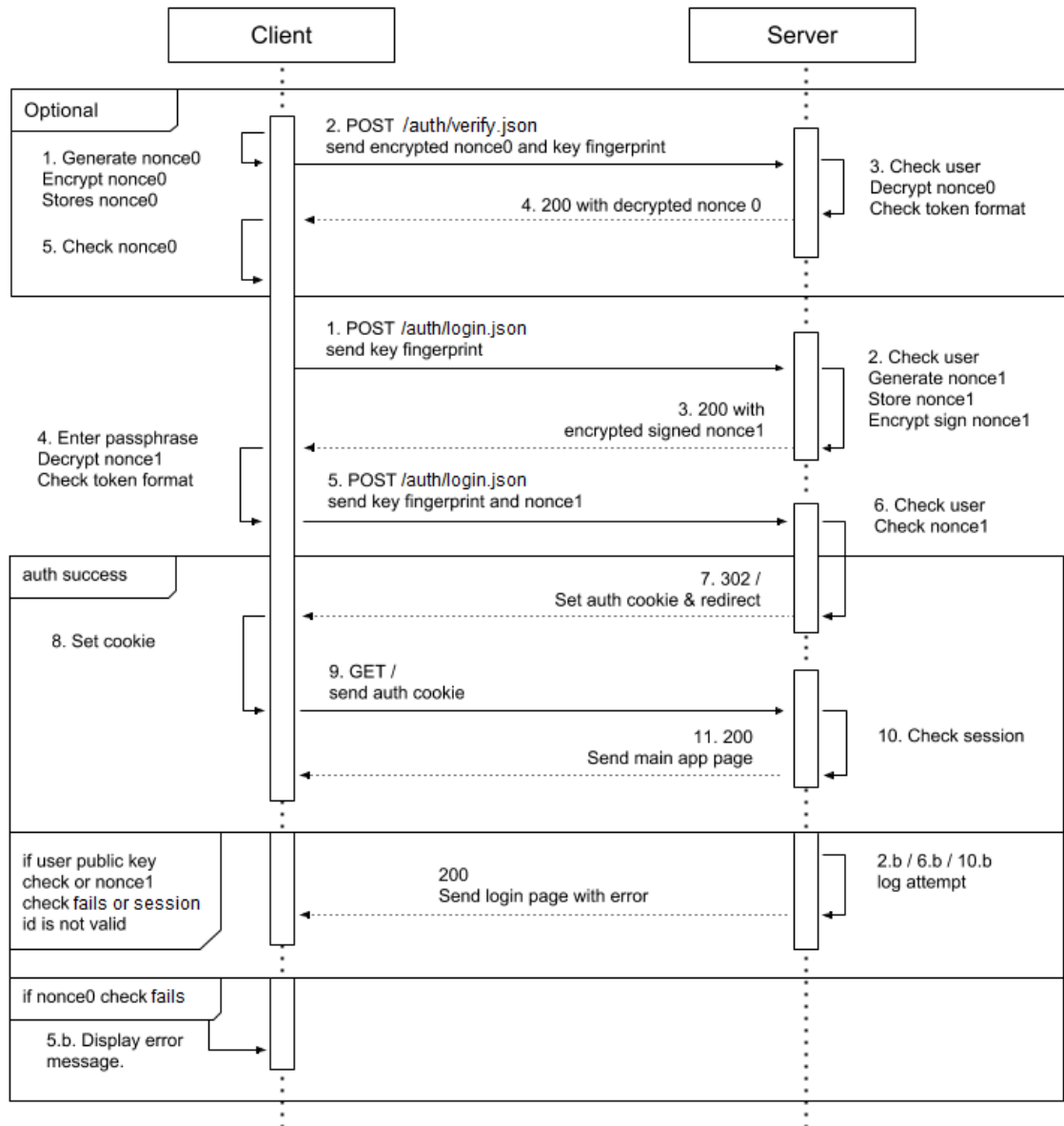


Fig. Sequence diagram of a GPGAuth based authentication

Server key verification steps

Passbolt performs server key verification as part of the authentication process. Although optional in theory, it is always enforced. This step does not protect against man-in-the-middle attacks but can help detect server identity issues, such as domain name seizure.

Login steps

1. The user sends their key fingerprint.
2. The server checks to see if the fingerprint and user associated with it are valid. It then generates an encrypted token of random data, and stores the unencrypted version locally.
3. The server sends the unencrypted signed user token, and the encrypted server token to the user.
4. The user enters their private key passphrase, the client decrypts the nonce and checks the token format.
5. The client sends back the decrypted nonce along with the user key fingerprint.
6. The server compares the un-encrypted signed token sent from the client to make sure it matches. If the server is satisfied, the authentication is completed as with a normal form based login: session is started using a cookie.
7. The session cookie is sent by the client to authenticated endpoints.

Token format

This challenge token is expected by the server and client to be in a specific format in order to prevent the authentication mechanism as a way to leak other content encrypted for the same keys.

A valid header consists of a 4 pipe delimited sections:

- version
- the length of the token (36 in our case)
- The token (a UUID v4)
- version

Example:

```
gpgauthv1.3.0|36|8661be60-23df-11e5-b16c-0002a5d5c51b|gpgauthv1.3.0
```

GpgJwtAuth

This updated authentication scheme uses a less verbose but similar challenge concept. Login requires a single call instead of three.

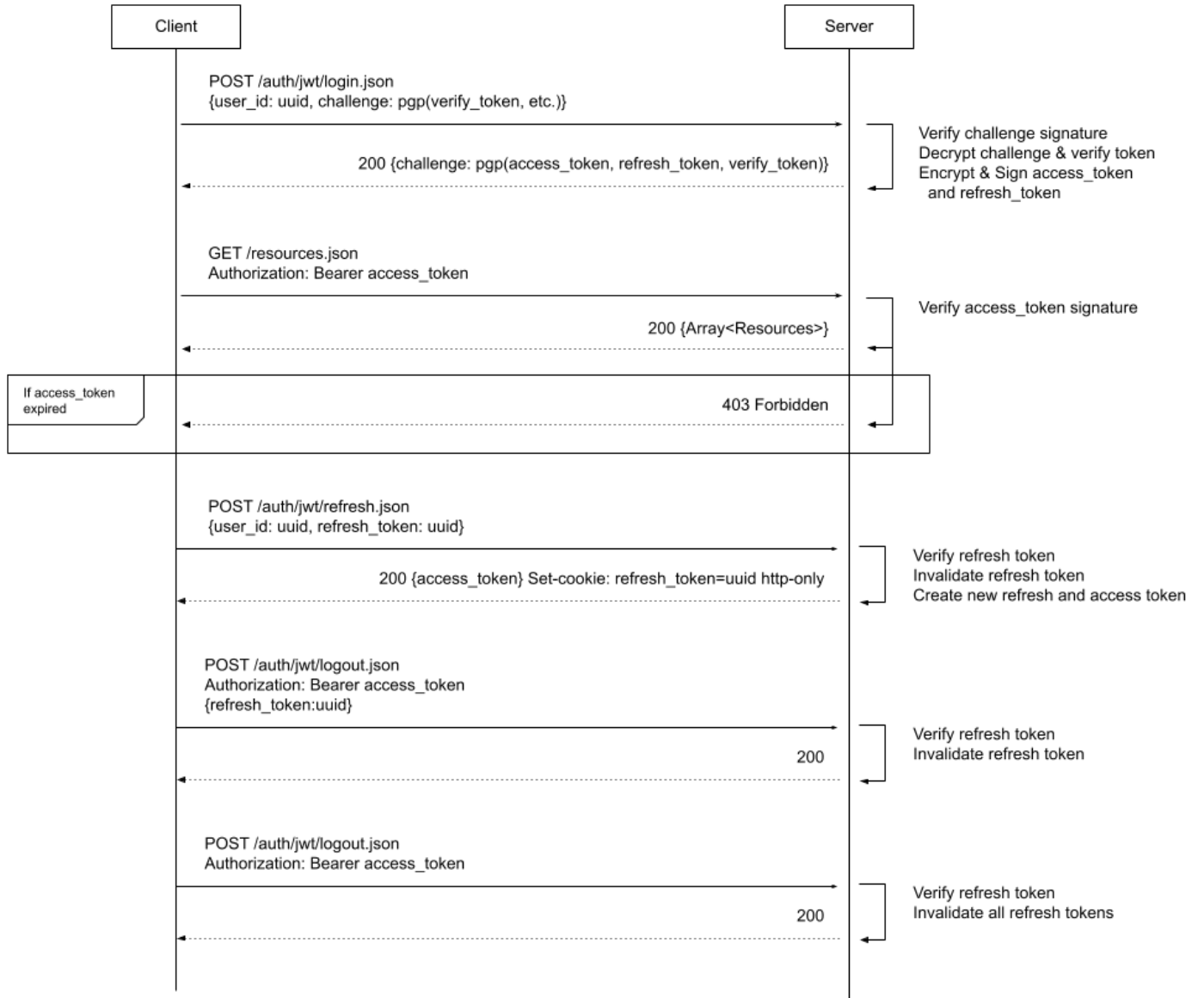


Fig. Sequence diagram of a GpgJwtAuth based authentication

Login steps

- The client sends a PGP-signed and encrypted challenge containing a nonce, expiry, and domain info.
- The server verifies the challenge and replies with encrypted access and refresh tokens.
- JWT access tokens are signed with the server's key; refresh tokens are returned either as HTTP-only cookies or encrypted payloads.
- The session is then managed via short lived JSON Web Token (JWT). The JWT is sent in the header as a Bearer authentication.
- Refresh token is sent / renewed to when the JWT expires.

Multiple Factor Authentication (MFA)

Passbolt also provides additional factor authentication in the form of TOTP (compatible Google Authenticator, Authy, FreeOTP, etc.), Yubikey OTP and Duo. Multiple authentication providers can be enabled at the same time to allow a fallback method for example in case a device is lost or one method is temporarily unavailable.

This MFA process takes place after the regular challenge based authentication. When the MFA process is completed a token is set in a 'passbolt_mfa' cookie linked to the trusted domain for the duration of the session. If the remember me option is checked, the cookie is set to expire after 30 days or if the user agent changes.

TOTP

The TOTP mechanism used is compatible with [RFC 6238](#). Given a secret generated for the user during the setup (256 random bytes), a provisioning URL will be created which in turn will be exposed as a QR code that can be scanned by the user. During the verification operation the server receives the TOTP code generated by the user device and uses the secret stored in the user settings database table entry to verify its validity.

As the whole setup is time sensitive it can be subject to DDOS if the clock of the server can be affected remotely.

Yubikey OTP

This provider relies on [Yubicloud](#) (Yubico's web service for verifying OTP) which requires an [API key](#) to work.

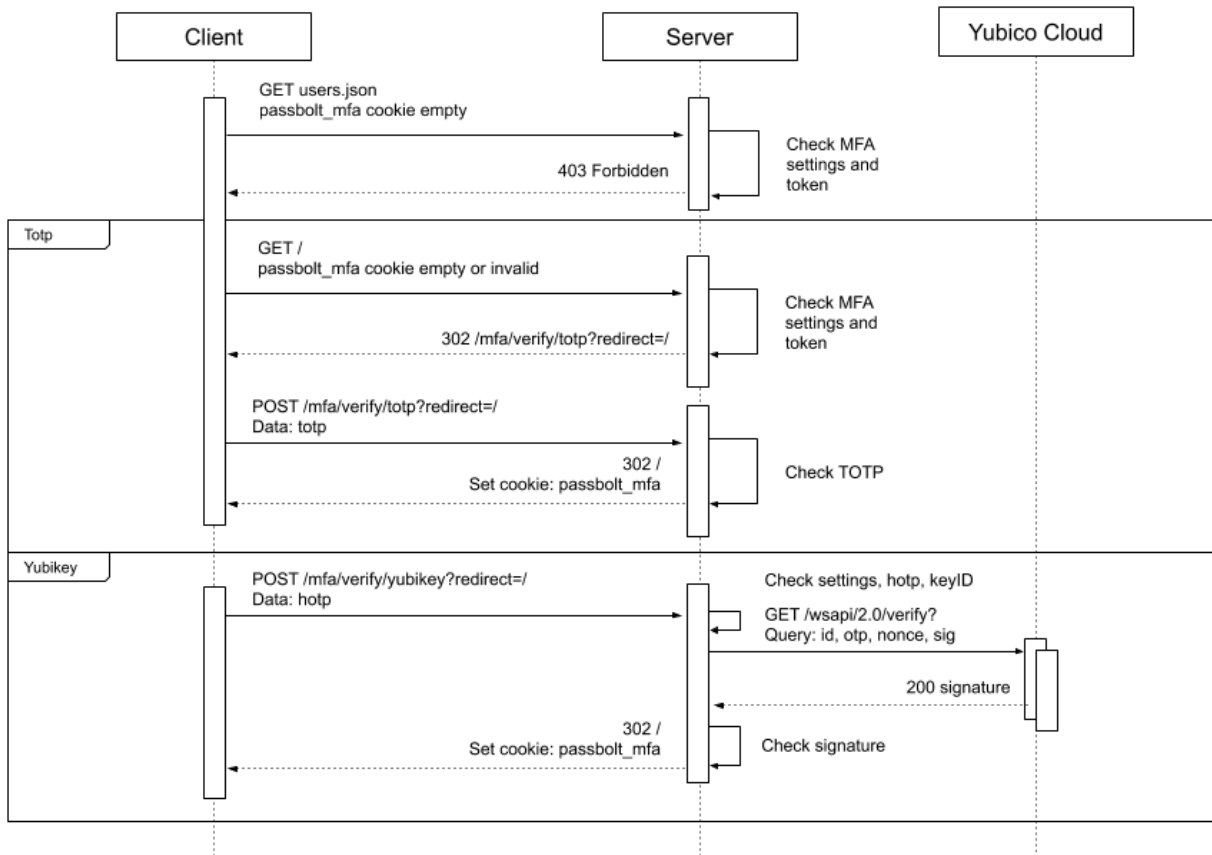


Fig. MFA sequence diagram

During a login attempt the passbolt will check if the key ID used by the user is the same that was used during setup. To change keys (if the key was lost for example) a user will need to first disable the Yubikey provider in their settings. In order for this service to work the server must allow outgoing connection to `api*.yubico.com`.

While some redundancy is built in the yubico service (multiple requests are triggered at once on multiple servers), if all servers are offline a user might be prevented from logging in.

Duo

This provider integrates with Duo using Web SDK v4 for multi-factor authentication. Authentication is initiated through a secure server-side flow: when a user starts the Duo setup or verification process, the Passbolt backend generates a signed authentication URL using Duo's OIDC-compliant PHP SDK (`duo_universal_php`). The user is then redirected to Duo's Universal Prompt hosted on Duo's servers, where they complete the second-factor authentication (e.g., push notification, SMS, or Yubikey).

Once authentication is completed, Duo redirects the user back to Passbolt along with a JWT (`duo_code`) and the original state parameter. The server verifies the JWT by using the configured Duo credentials (`clientId`, `clientSecret`, and `apiHostName`). This verification confirms the authenticity of the response and extracts the identity of the authenticated user. The redirect flow and JWT-based validation provide a secure and standards-based integration that ensures origin validation, tamper-proof authentication results, and minimal client-side exposure.

More information about this system can be found on the [Duo website](#).

Single Sign On (SSO)

This feature enables users to log in using their organization credentials, without manually entering their Passbolt passphrase each time. The system also accommodates hybrid login workflows: users who prefer or need to use their passphrase can still do so.

This allows administrators to leverage the authentication mechanism of their existing identity provider, such as Microsoft Azure Active Directory, Google, Okta, Keycloak, etc.

SSO Login Process

The login process adheres to OAuth 2.0 and OpenID Connect standards, using the Authorization Code Flow to verify user identity and securely deliver a decryption key to the browser, enabling access to the user's private OpenPGP key.

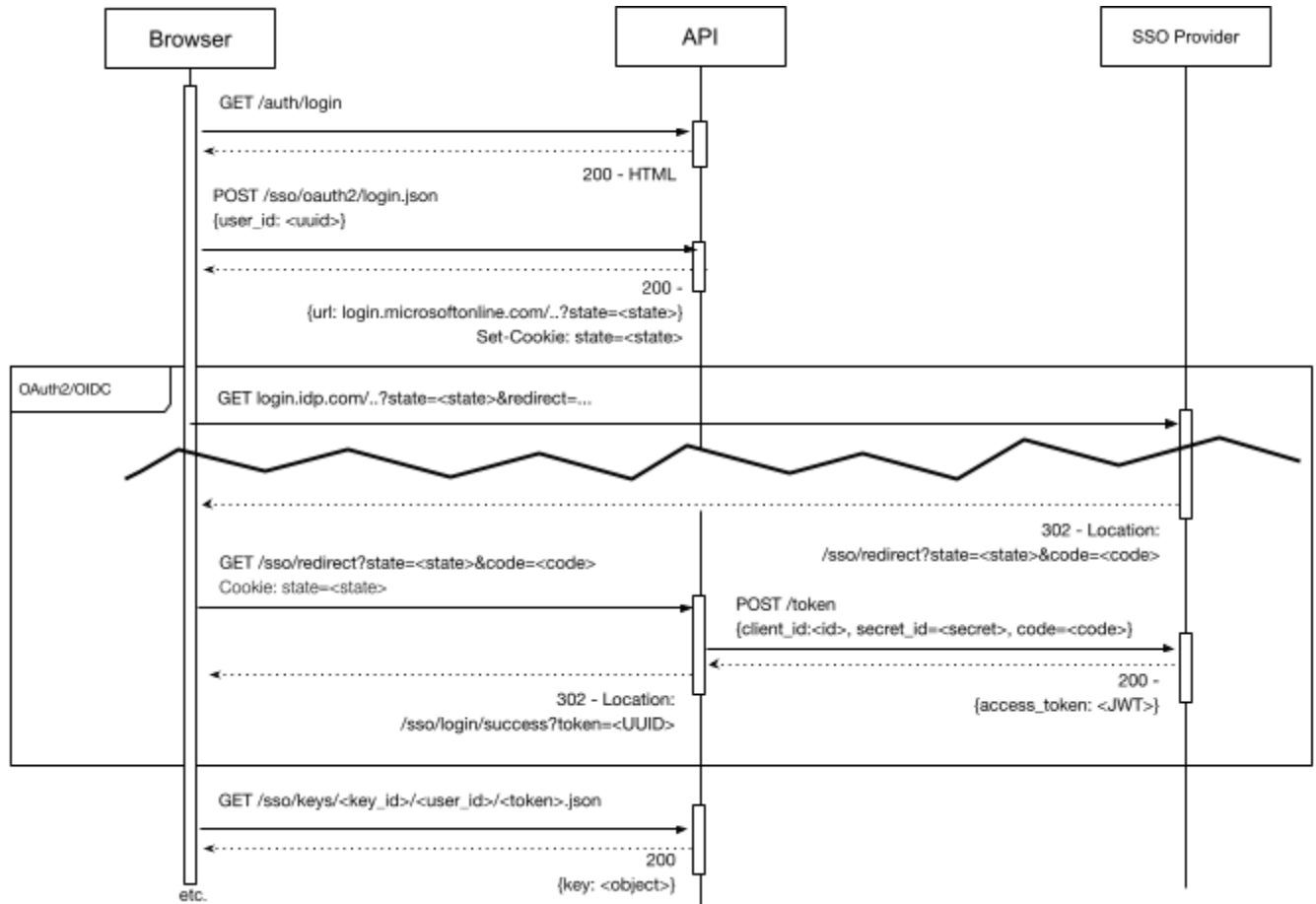


Fig. SSO sequence diagram

SSO keys

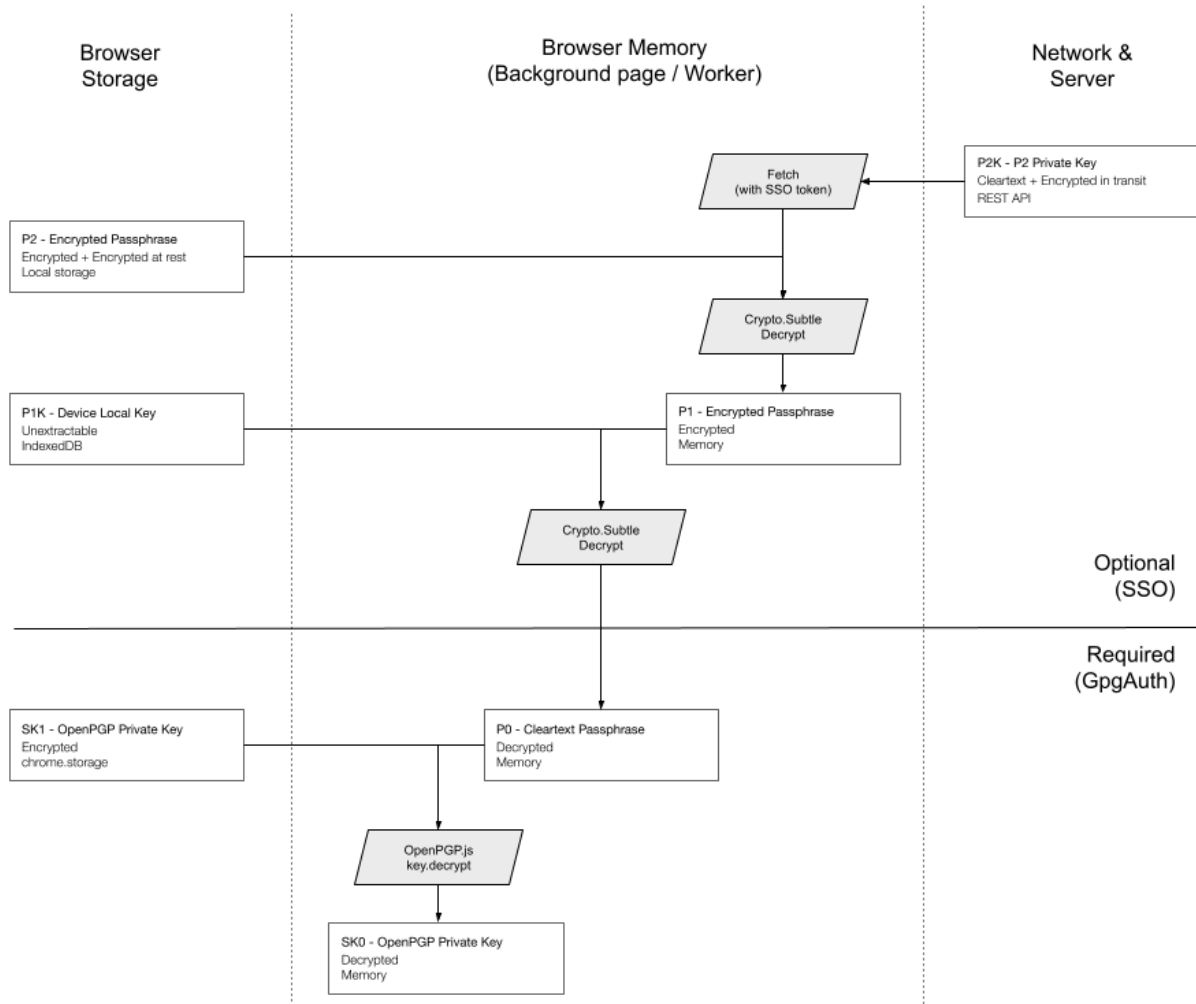


Fig. SSO Passbolt Crypto Scheme - Secret Key Decryption (BROWSER)

Technically, the browser extension securely stores a locally generated symmetric encryption key (P1K) that encrypts the user's passphrase. This encrypted passphrase (P1) is then wrapped with a second key (P2K), stored on the Passbolt server, and released only upon successful SSO authentication.

These keys are AES 256 keys used in GCM mode.

They are not generated or used using OpenPGP but using [SubtleCrypto API](#) in browsers, since it allows generating non extractable keys. When using the feature the new key landscape will look like:

Type of keys	Presence Type		
	Client Memory	Client storage	Server Side
Passphrase Key Key (P2K)	Cleartext	N/A	Cleartext
Passphrase Key (P1K)*	Cleartext	Cleartext	N/A
OpenPGP Secret Key Passphrase	Cleartext	Encrypted	N/A
OpenPGP Secret Key	Cleartext	Encrypted	N/A** or Encrypted
OpenPGP Public Keys	Cleartext	Cleartext	Cleartext

* Non extractable key.

** The OpenPGP private key is not stored server-side by default. It is only present server-side in encrypted form if the account recovery feature is explicitly enabled by the administrator.

Application architecture

Understanding the different layers of the application is essential for assessing the attack surface and evaluating Passbolt's security posture. This section provides a clear, concise explanation of the software architecture, so you know exactly how the components interact.

Application components

Passbolt follows a client-server architecture in which clients communicate with the server over the HTTP protocol. Clients can include the browser extension, mobile applications, command-line interfaces (CLIs), or other integrations built on top of the Passbolt API.

Server components

Passbolt clients connect to a server component written in PHP 8 and more specifically CakePHP 5, following the Model-View-Controller (MVC) design pattern. This server application relies on a MariaDB, MySQLi or PostgreSQL database, and a caching system that can be configured to use either the local file system or Redis. The server uses a combination of [Openpgp-php](#) library and [PHP GnuPG](#) extension, to validate keys and perform cryptographic operations, primarily the ones related to the authentication.

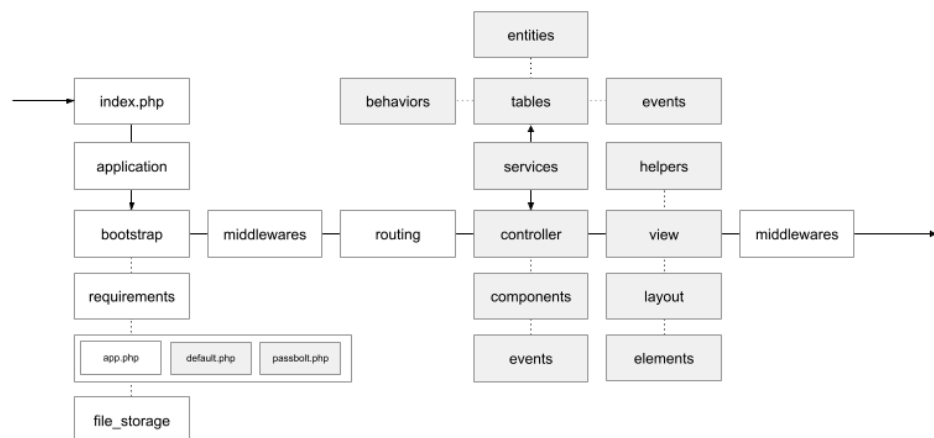


Fig. Typical server request lifecycle

The core of the application handles the baseline functionalities such as managing users, resources (the password metadata), secrets (the encrypted content), groups, group memberships and the associated permissions.

The applications have a plugin oriented architecture. Each plugin also follows the same MVC paradigm. Plugins are organized following functional areas such as account settings, audit log, user directory synchronization, email notification settings, multi factor authentication, tags and folders management, single-sign-on, account recovery, etc.

JSON API

The server application is primarily composed of Restful JSON API endpoints. The full documentation of the available API endpoints can be found online at passbolt.com/docs/api.

```
GET /healthcheck/status.json
Authorization: Bearer eyJhbGciOiJIUzIsInR5cCI6IkpXVCJ9...
Accept: application/json

{
  "header": {
    "id": "3304d332-31e0-4e45-b4f2-da4e52f5f5d5",
    "servertime": 1563219221,
    "action": "f52ecf6c-8e82-5f3d-ab73-f6df46eb71b5"
    "code": 200
  },
  "body": "OK"
}
```

Fig. example of server side response.

Authenticated endpoints support either session cookies or bearer tokens (JWT) for user authentication persistence.

Server-side JavaScript

Several JavaScript applications are served by the server and they play a minimal role. One application mostly handles the public forms such as the account registration or recovery, when the extension is not installed. Another application handles the initial setup of the server application, e.g. the last part of the installation process.

These small JavaScript applications served server side are not allowed to interact with the sensitive logic or data of the webextension. Therefore it is not possible from a script hosted by the server to access secret content such as the user's private key, passphrase or the decrypted content.

Server side command line tools

Additional command line interface tasks are made available to the administrator to ease the maintenance of the passbolt instance such as system configuration health checks, data integrity checks, action log purge, etc.

```
$ ./bin/cake passbolt cleanup --dry-run
```

```
-----  
Cleanup shell (Dry-run mode)
```

```
-----  
No issue found, data looks squeaky clean!
```

Fig. example of server-side command line tool

Cron Jobs & email queue

Emails are managed through a queue that requires a cron job to run. This allows decoupling the job of sending email notifications from the user action triggering them. Optionally emails of the same types that are part of the queue can be grouped via an email digest system.

Database

The core application revolves around a set of core tables such as users, roles, resources (the credential metadata), secrets (the credential secret content), resource types, permissions, groups, folders, and authentication_tokens.

Additional plugins such as directory sync, audit logs extend this data model and mostly reference the core schema.

folders	folders_relations	gpgkeys	groups
id \emptyset char(36) NN	id \emptyset char(36) NN	id \emptyset char(36) NN	id \emptyset char(36) NN
metadata \emptyset mediumtext	foreign_model varchar(30) NN	user_id char(36) NN	name \emptyset varchar(255)
metadata_key_id \emptyset char(36)	foreign_id char(36) NN	armored_key text NN	deleted \emptyset tinyint(1) NN
metadata_key_type \emptyset varchar(100)	user_id char(36) NN	fingerprint varchar(51) NN	created datetime NN
created datetime NN	folder_parent_id \emptyset char(36)	deleted \emptyset tinyint(1) NN	modified datetime NN
modified datetime NN	created datetime NN	created datetime NN	created_by char(36) NN
created_by char(36) NN	modified datetime NN	modified datetime NN	modified_by char(36) NN
modified_by char(36) NN			

groups_users	metadata_keys	metadata_private_keys	permissions
id \emptyset char(36) NN	id \emptyset char(36) NN	id \emptyset char(36) NN	id \emptyset char(36) NN
group_id \emptyset char(36)	fingerprint varchar(51) NN	metadata_key_id char(36) NN	aco varchar(30) NN
user_id \emptyset char(36)	armored_key text NN	user_id \emptyset char(36)	aco_foreign_key char(36) NN
is_admin \emptyset tinyint(1) NN	created datetime NN	data mediumtext NN	aro varchar(30) NN
created datetime NN	modified datetime NN	created datetime NN	aro_foreign_key \emptyset char(36)
	expired \emptyset datetime	modified datetime NN	type int(11) NN
	deleted \emptyset datetime	created_by \emptyset char(36)	created datetime NN
	created_by \emptyset char(36)	modified_by \emptyset char(36)	modified datetime NN
	modified_by \emptyset char(36)		

resource_types	resources	roles	secrets
id \emptyset char(36) NN	id \emptyset char(36) NN	id \emptyset char(36) NN	id \emptyset char(36) NN
slug varchar(64)	metadata \emptyset mediumtext	name varchar(50) NN	user_id char(36) NN
name \emptyset varchar(64)	metadata_key_id \emptyset char(36)	description \emptyset varchar(255)	resource_id char(36) NN
description \emptyset char(255)	metadata_key_type \emptyset varchar(100)	created datetime NN	data mediumtext NN
definition \emptyset text	deleted \emptyset tinyint(1) NN	modified datetime NN	created datetime NN
deleted \emptyset datetime	expired \emptyset datetime		modified datetime NN
created datetime NN	created datetime NN		
modified datetime NN	modified datetime NN		

users
id \emptyset char(36) NN
role_id char(36) NN
username varchar(255) NN
active \emptyset tinyint(1) NN
deleted \emptyset tinyint(1) NN
disabled \emptyset datetime
created datetime NN
modified datetime NN

Fig. passbolt core data model tables

Browser Extension

The flagship client of Passbolt is the browser extension, which builds are distributed most notably on Chrome, Edge and Firefox webstores.

The web extension handles the primary interface, e.g. the main application that presents the resource workspace, the user workspace, the account setup and recovery wizards, the login box, the settings, etc.

The browser extension also serves the secondary applications, such as the ones used when integrating with web pages to generate, autofill, and manage passwords directly within login forms (aka “In-form Menu”) or within the browser extension contextual windows that appears when a user clicks on the passbolt icon in the browser (aka “QuickAccess”).

Layers & logic separation

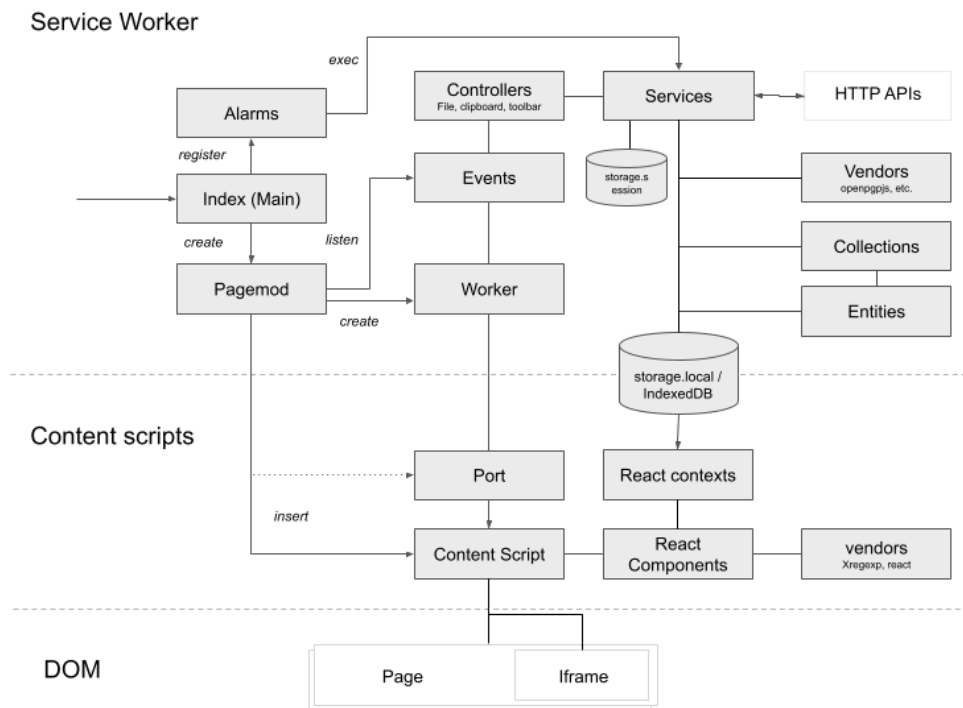


Fig. Passbolt web extension high level architecture

Currently the browser extension supports both the MV2 and MV3 formats. The browser extension is written in JavaScript using a minimal set of dependencies such as [OpenPGP.js](#) and [React](#).

In practice the web extension inserts the application content scripts in pages. These content scripts can access and modify the web page DOM. However the scripts served by the server can not see JavaScript properties added by the browser extension content scripts (and vice-versa). This mechanism ensures that the two javascript environments are not maliciously affecting each other.

Whenever a content script needs to create html elements, and in order to create the aforementioned “secure DOM” visible only by the web extensions scripts, passbolt relies on iframes inserted in the page. The content of this iframe is served under a different domain (e.g. `chrome-extension://`), and therefore is not accessible to the page served by the server, thanks to cross domain policy restrictions. Most of the interactions, such as the ones on the user or password workspace, happen within such an iframe.

Moreover all the cryptographic functionalities are running in a third separate environment, called the Service Worker (also historically known as Background Page in Extension Manifest v2 format), which implement the long-running logic of the webextension. Such sensitive functionalities from the Service Worker are exposed through a set of event APIs to the content scripts. This layered architecture is useful to guarantee the integrity of the high level cryptographic functionalities and restrict access to sensitive data.

The service worker and the content scripts (or scripts running inside the iframes) communicate using dedicated webextension [ports](#).

Type of storage

These environments also share a common set of data accessible via the extension local storage (e.g. [storage.local](#), not to be confused with `window.localStorage`).

The extension local storage also belongs to the extension context, not the website. Only the extension's own scripts (background, popup, content scripts injected by the extension) can access it.

The secret key along with the user configuration is stored in the web extension local storage. This local storage is in turn stored on the user file system with the [browser profile data](#).

Additionally the extension uses session storage (e.g. `storage.session`, not to be confused with `window.sessionStorage`), to persist items, such as the user passphrase. Such items may be stored in memory for the duration of the browser session and are not persisted to disk.

Finally the extension uses `indexedDB` in service workers context (e.g. not to be confused with the regular webpage `Window` context) to store the non-extractable device specific symmetric key generated for Single-Sign On (SSO) purposes, as described further in this document.

Browser Extension Permissions

- The **host** permission is needed because since passbolt can be self-hosted, the url of the passbolt server cannot be predicted, and therefore the web client can run on any domain and that's why the `**/*` match pattern is required.
- The **active tab** permission is needed to temporarily access the currently active tab when the user clicks on browser action, in order to enable filling up login forms with passwords or prefill the password creation form.
- The **Tabs** permission is needed to insert the scripts needed to run the passbolt main interface on the domain selected by the user during the setup. The Tabs permission is also required so that the web application can continue to work even when the user switches tabs. By Instance, when a user starts a bulk share operation and navigates to another tab, then the operation will not be interrupted and the operation will continue.
- The **Download** permission is needed in order to trigger downloads, for example of the user's private key.

- The **clipboard** write permission is needed to allow users to copy a secret or username to the clipboard.
- The **background** permission is needed in order to implement the long-running logic of the webextension.
- **Chrome local storage** is used to store the user configuration, and application local cache, as seen before.
- **Unlimited storage** permission is required in order to store a large number of items.
- The **scripting permission** is required to insert passbolt applications inside pages served by a self-hosted API, as well as in all pages the users visit.
- The **Alarm** permission is needed in order to schedule routine tasks. A task is there to control the user session validity and adapt the user experience accordingly. While others are there to flush caches and mitigate risks that could be linked to keeping sensitive data in memory.
- **Cookie** permission is required to obtain the current CSRF token.
- The **offscreen** permission is necessary to allow the Passbolt browser extension to effectively manage interactions with Passbolt APIs that utilize SSL certificates accepted at the browser level but not at the operating system level². It is also needed for clipboard interactions.

² Ref. Chromium Issue #40882068, Comment 13

Risks mitigation strategies

This section describes the risk mitigation strategies implemented across the solution. It outlines the main measures taken to reduce potential threats and limit impact where risks cannot be fully avoided. While not exhaustive, it gives a clear view of what has been considered within scope and how common risks have been addressed in practice.

OWASP Category	Risk mitigation section(s)
A01: Broken Access Control	Access Control and Authorization,
A02: Cryptographic Failures	Authentication, Key Validation, Transport Security, JWT, Signed releases
A03: Injection	Input Validation and Sanitization, SQL Injection, LDAP Security, PHP exec
A04: Insecure Design	Secure Development Best Practices, Authentication, Browser Extension, CSRF
A05: Security Misconfiguration	Server Side, Transport Security, Security Headers, Cookie Security, File Upload
A06: Vulnerable and Outdated Components	Dependency Monitoring, Static Code Analysis, Annual Security Audits
A07: Identification and Authentication Failures	Authentication, MFA Rate Limiting, JWT, Browser Extension
A08: Software and Data Integrity Failures	Signed Releases, Code Review and Publication, Developer Authentication
A09: Security Logging and Monitoring Failures	Security Logging and Monitoring, SIEM Integration
A10: Server-Side Request Forgery (SSRF)	Server-Side Request Forgery (SSRF) Protection

Fig. Mapping of OWASP Top 10 with risk mitigation documentation

Security Logging and Monitoring

Passbolt includes logging and monitoring features to ensure traceability, detect misuse, and support compliance.

User action logs

All user actions that modify data or access sensitive information, such as login attempts, credential updates, or resource sharing, are recorded in the audit trail.

Administrative changes, including role updates or system settings modifications, are also logged and trigger email alerts.

Each entry captures the user identity, timestamp, source IP, and a description of the action to provide a reliable audit trail.

Application error logs

Runtime errors and exceptions are logged via syslog or local log files. Security-relevant issues like invalid tokens, cryptographic failures, or unexpected behavior are captured to allow external review and long-term storage.

SIEM Integration

Passbolt supports integration with Security Information and Event Management (SIEM) systems. Logs follow structured formats for easy parsing and correlation, enabling centralized analysis, anomaly detection, and retention according to organizational policies.

Authentication

Oracle Attacks

The authentication protocol includes mitigations that render oracle attacks impractical³. The challenge-response mechanism requires the

³ See. PBL-12 Audit-Report Passbolt Oracle Attack Scenario 12.2024 Cure53

client to prove possession of the private key, meaning an attacker gains nothing unless the decrypted challenge is valid and usable. Additionally it suppresses all decryption error messages, preventing error-based oracles. Moreover, even if RSA decryption proceeds with malformed padding, the symmetric encryption layer secured by OpenPGP's integrity checks ensures any tampering results in failure.

JWT

Access token capture via cross-site scripting (XSS) is mitigated by ensuring that tokens have a short lifetime of five minutes and are not stored in cleartext.

The risk of forged JWT tokens is addressed through strict validation of token headers and thorough verification of digital signatures.

To prevent refresh token attacks, refresh tokens are designed to be single-use and are transmitted through secure channels, either encrypted with the user's public key or via HTTP-only cookies.

In the event of a server private key compromise, administrators have the ability to rotate the server key, which immediately invalidates all previously issued tokens.

MFA Rate Limiting

Multi-factor authentication (MFA) endpoints implement rate limiting to prevent brute-force attacks against authentication codes. After a configurable number of failed attempts, the user needs to start the login from scratch again.

Server Side

Access Control and Authorization

The application implements strict access control mechanisms where authorization checks are centralized ensuring uniform validation across all API endpoints. Administrative endpoints are protected by multiple validation layers including role verification. Where applicable database

queries systematically include user ID restrictions to enforce data isolation between users.

Input Validation and Sanitization

User inputs undergo comprehensive validation at multiple layers. The application uses framework-provided ORM parameterized queries to prevent SQL injection. All URL parameters are validated against expected patterns (such as UUID format) before processing. Mass assignment vulnerabilities are prevented through careful entity mapping and validation rules.

Key Validation

Imported PGP keys undergo comprehensive validation including algorithm strength verification, key size requirements (minimum RSA-2048 or equivalent), and capability checks to ensure keys can perform required operations. Weak cryptographic algorithms like DSA and ElGamal are rejected.

PHP exec

The server application does not use the [exec](#) function except in some rare cases such as tasks that can be run by the administrator only using command line when logged in on the server, for example to create a backup of the database using `mysqldump`.

Unsecure deserialization

Malformed serialized data could be used to abuse application logic, deny service, or execute arbitrary code, when deserialized. The application does not rely on [unserialize](#) and avoids serialization of PHP objects and rely on JSON serialization instead.

SQL Injection

The server side application almost never uses direct SQL queries but instead accesses the data through the framework ORM. By default these database abstraction layers prevent most SQL injection issues. User

input used by the ORM is always carefully validated. [Remaining risks](#) are mitigated using code reviews.

File upload

File upload is another sensitive area for web applications. Passbolt at the moment only supports file upload in relation to the management of user avatars. The file size limit can be controlled by the administrator using the PHP / webserver environment variables. Passbolt validates the mime types (image/jpeg, image/png, image/gif), the file extension (png, jpg, gif), and checks for [file upload errors](#). Images are then resized to allowable dimensions. The application stores all relevant metadata in a database record and uses a random filename for the actual cache local filesystem storage.

CSRF

Cross Site Request Forgery is an attack scenario where a user's action on a malicious third party site would trigger a modification of data on a website, such as editing a resource or deleting it, by crafting a malicious image url or by having the user submit a form from another domain.

Several mechanisms are in place to limit the attack surface. First, in the spirit of Restful API, HTTP GET operations do not trigger change in the data. Secondly, every entity is identified by a UUID and this identifier is required for all DELETE and PUT operations, making URLs hard to guess for an attacker.

Finally a PSR-7 Middleware is used to protect against the remaining CSRF risks on POST, PUT and DELETE operations. It works by setting a csrfToken in a cookie and in a hidden input field in forms. The token will be submitted along with the cookie and as part of the request data. Alternatively for Ajax the tokens can be submitted through a special X-CSRF-Token header. The middleware component will compare the request data & cookie value and if the data is missing or the values mismatch an error will be returned.

Server-Side Request Forgery (SSRF) Protection

Administrative functionality that connects to external services implements host and port restrictions and/or can be locked using environment variables. For example the SSO or DirectorySync plugin includes configuration controls that limit which hosts and ports can be accessed, preventing unauthorized network scanning or access to internal services.

Transport security

Security headers

By default passbolt will fallback to HTTPS if an HTTP request is made. It also uses a PSR-7 middleware to apply the following security related headers:

- X-Content-Type-Options nosniff
- X-Download-Options noopen
- X-Frame-Options sameorigin
- X-Permitted-Cross-Domain-Policies all
- Referrer-Policy sameorigin

Cookie security

By default the cookie used for the session and the MFA token have the SetCookie header set with the “HttpOnly” and “secure” flags on. The HttpOnly flag mitigates the risk of client side scripts accessing the protected cookie. The secure flag mitigates the risk of the cookie being sent in the clear over http.

LDAP Security

Directory synchronization features implement strict input validation for LDAP queries. Custom filters undergo proper escaping to prevent LDAP injection attacks. Field mapping configurations are statically defined rather than dynamically processed to prevent arbitrary data exfiltration from LDAP directories.

Browser Extension

Passphrase Phishing

Several mechanisms are present in the Passbolt web extension to prevent phishing. In this attack scenario an attacker would create a page looking like a regular login page, or inject an additional javascript on a legitimate passbolt page.

Passbolt mitigates this type of attack by using a “security token” that is present whenever the user needs to enter their passphrase. To prevent an attacker from placing a transparent input dialog on top of the input next to the security token, for example to capture the passphrase, a field focus event displays an additional interaction in place with color changes.

Clickjacking Protection

Web-accessible resources are carefully managed to prevent malicious websites from embedding extension components. Frame ancestors CSP directives restrict which domains can frame extension content, preventing credential theft through clickjacking attacks.

Memory Security

Sensitive data, like a passphrase, is cleared from memory immediately after use and upon logout. Extension popups implement secure memory management to prevent credential recovery through memory inspection after logout.

Cross Site Scripting (XSS)

Persistent XSS

In this scenario an attacker would submit malicious data to the server that would then be executed on the page when accessed by another user. We assume it is the responsibility of the clients to treat the server information as hostile and take care of the sanitization.

In practice in most cases this means only rendering the information as text and using escaping facilities provided by the templating libraries used by passbolt (React) and [template literals](#). In the rare cases where using text is not an option, such as when making the URL of a password clickable, additional filtering is in place.

It is possible to verify this claim by for example looking at the use of unsafe method such as `innerHTML` or `jquery` equivalents⁴ in the code. These issues are captured by the static coding tools used by Mozilla uploading a new version of the extension. Moreover Passbolt's selenium testsuite includes a set of common XSS prone strings to check against regressions.

Additionally passbolt includes default content security policy to prevent running inline javascript or including javascript files from third party domains.

Reflected XSS

In this scenario an attacker would craft a link (for example in an email) to run arbitrary code when the user navigates on the passbolt domain.

In certain cases passbolt uses parameters provided in URLs. URL parameters are used, for example, to directly access a resource (which therefore allows sharing a link to the resource with a coworker) or prefill a form with data (to help an admin add a user).

The attack surface for reflected XSS is reduced by running validation on the requested route content (for example the input must match the uuid of a resource or a user) and by not displaying the route content back on the screen as html.

Content Security Policy

The extension implements strict Content Security Policy rules that prevent inline JavaScript execution and restrict resource loading to approved domains. Default CSP configurations are enhanced with additional directives including `base-uri`, `form-action`, and `frame-ancestors` for comprehensive protection.

⁴ For example: `.html()` `.append*()` `.insert*()` `.prepend*()`, etc.

URL Validation and Suggestion Security

The extension implements robust URL validation for password suggestions, using normalized URL comparison through the window.URL API to prevent parser differentials. The system correctly handles IPv6 addresses, different IP representations, and implements parent hostname checking to prevent cross-domain credential suggestions.

Use of unsafe methods

Javascript eval

The execution of eval statements is not allowed in the webextension thanks to the [default policy restrictions](#) of web extension and server side Content Security Policy.

Trusted domain

In order for a page to interact with the webextension it must be on a trusted domain, this trust is enforced by the user during the setup. The web extension will not insert the main application content script or iframe in a non trusted domain.

Autofill

Several best practices are implemented to reduce the risks associated with leaking secrets in relation with the autofill functionality. Most importantly the autofill functionality requires user input in a trusted part of the extension. The user will have to select a suggested entry in a browser popup and click on a "fill on this page" button in order to trigger the autofill functionality. Moreover the autofill functionality does not try to enter the credentials in iframes inserted on the page.

Background / Content Script message spoofing

The extension implements origin-based validation to prevent cross-domain credential injection and use secure communication channels between background scripts, content scripts, and web pages to prevent message interception or spoofing.

Manifest Security

The extension manifest implements security best practices including minimal required permissions, proper externally_connectable configurations, and content script injection controls. Web-accessible resources are limited to only necessary files with appropriate access restrictions.

Secure development best practices

Continuous integration and testing

Passbolt code client and server side have an adequate level of coverage. Automation is in place on the continuous integration server to enforce execution of both unit tests and functional tests (selenium) as part of the delivery pipelines, with broad tests matrix (e.g. all the supported versions of the underlying components). The tests also include XSS scenarios executed in a real browser through selenium to make sure that there are no regressions at this level.

Developer authentication

Every developer of the passbolt team must use a strong password and wherever possible a multiple factor authentication system in order to access the systems needed to publish code. This policy compliance is regularly reviewed.

Signed releases

Passbolt's release team uses digital signatures for tags to help the administrator ensure the integrity of each release. All official Passbolt packages, including RPM and DEB distributions, are cryptographically signed, allowing system administrators to verify their origin before installation. Similarly the web extension releases are signed to ensure that only a legitimate passbolt extension can be installed / updated. Additionally, passbolt contributors sign each commit with their OpenPGP key.

Code review and publication

Only a small number of people are responsible and allowed to publish code. Before being pushed on a sensitive branch, each pull request is reviewed and validated by another maintainer.

Dependency monitoring

The Passbolt team has access to automated reporting and security alerts related to possible vulnerabilities in libraries used by passbolt through tools provided by vendors such as [Github](#) and [Snyk](#).

Static code analysis

The Passbolt team uses multiple tools to perform code analysis such as eslint, phpcs, phpstan, psalm, webext-lint, etc. These checks are enforced as part of the continuous delivery pipeline.

Responsible Disclosure

Passbolt maintains a [responsible disclosure policy](#) and program to encourage security researchers and users to report vulnerabilities responsibly. Submissions are reviewed promptly and rewarded when eligible, in accordance with defined severity levels and scope.

In the interest of transparency and community trust, validated vulnerabilities and their remediation are publicly disclosed through the Passbolt [incidents page](#) once mitigations have been deployed.

Annual Security Audits

To ensure continuous assurance and alignment with industry best practices, Passbolt undergoes at least two independent security audits per year. This includes:

Source Code Audit

At least one yearly security-focused code review, focused on one or more aspects of the codebase, is conducted by external security experts

to identify potential vulnerabilities, verify the effectiveness of implemented controls, and ensure the application continues to adhere to secure coding standards.

SOC 2 Type II Audit

Passbolt is regularly evaluated against the SOC 2 Trust Services Criteria, with annual assessments conducted by certified auditors. This audit validates the operational effectiveness of our controls in areas such as security, availability, and confidentiality.

Packaging

Passbolt is provided in a variety of ways to install. Some of these methods come prepared for a quicker installation experience, such as a docker image or a virtual appliance in OVA format. In these cases some additional security options are present.

CIS Benchmarks

To help provide a more secure installation method when providing preconfigured images we apply the CIS benchmarks during the installation. This helps reduce the risk of a misconfiguration when adding this to an environment. Administrators should still review the security of these provided installation methods to ensure they fit the necessary security requirements of their environments.

Non-root Docker Image

In addition to the standard docker image provided, we also provide a non-root image for use. This image does not come with a root user. This is to reduce the risk involved in the event an attacker is able to gain access to the image.

Residual risks

We believe passbolt is a software solution that provides a level of risk that is acceptable for most organizations. For many organizations, especially those not using a password manager, the benefits far outweigh the risks.

However for organizations that are under serious threat by well funded attackers passbolt may not be the best option.

Indeed no software is perfect and no reasonable software vendor can make the promise of perfect security. We believe it is very important to remain transparent on the risks that must be further worked on (or accepted) by a passbolt administrator and the end users.

Compromised cryptographic primitives

Quantum resistance

Passbolt relies on public-key primitives (RSA, ECDSA, etc.) which can be targeted by a powerful-enough quantum computer in the future. While no quantum computer that threatens the security of public-key cryptographic primitives such as ECDSA (default), RSA (optional, previous default prior to v5), and more, exists as of the date of the formulation of this paper, the emergence of such technology has been imminent and anticipated by the Cryptographic community.

In order to mitigate this risk and protect data from future attacks Passbolt will transition to post-quantum primitives when such new standards emerge⁵.

Weak server side random number generation

Since Passbolt supports deployment on native as well as containerized environments, a weak entropy source could weaken the cryptography.

⁵ Ref. <https://datatracker.ietf.org/doc/draft-ietf-openpgp-pqc/>

The following recommendations are proposed for optimal results when it comes to Pseudo-Random Number Generation:

- Ensure sufficient entropy is available at system startup, especially in virtualized environments, to allow `getrandom()` to complete securely.
- Consider enabling `only-urandom` in `/etc/gcrypt/random.conf` if you experience blocking behavior or entropy starvation during key generation in containerized environments.
- Using an external entropy source or using one of the *league of entropy* providers when applicable.

Compromised Network

Man in the middle

If an attacker is able to break the TLS connection between the client and the server they will have access to the unencrypted data and as such be able to capture the session cookie to perform actions on behalf of the user (see malicious client section).

By default such an attacker won't be able to decrypt the encrypted data but it can use the application mechanism to delete data. They could also inject a public key in the keyring synchronization request and wait until the user shares a password with them.

To reduce these risks a passbolt administrator [must ensure](#) that the server SSL configuration is configured to modern standards for example by disabling support for weak algorithms.

As an indication, as of when this report is being written, the following TLS setup is recommended:

- Protocols:
 - TLS 1.3 (recommended) should be the default protocol.
 - TLS 1.2 (legacy support only, to be deprecated) does not support post-quantum algorithm extensions and will not

be enhanced in that direction. Its support should be treated as a transitional measure and fully deprecated in a future release.

- Cipher suites (TLS 1.3):
 - TLS_AES_128_GCM_SHA256:TLS_AES_256_GCM_SHA384:TLS_CHACHA20_POLY1305_SHA256
- Cipher suites (TLS 1.2, legacy support only):
 - ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305
- TLS curves: X25519, prime256v1, secp384r1
- Certificate type: ECDSA (P-256) (recommended), or RSA (2048 bits)
- DH parameter size: 2048
- HSTS: max-age=31536000 (one year); includeSubDomains; preload
- Certificate lifespan: 90 days (recommended) to 366 days
- Cipher preference: client chooses

It is also possible to further reduce the risk by restricting network access to the passbolt server (by using a VPN or adding another layer of authentication, etc.).

Exposed metadata (applicable to v4 resource types)

For version 4 resource types, an attacker with access to data in motion or at rest may be able to view unencrypted metadata, such as the names of the websites a user is accessing or the associated usernames. While the secret (i.e., the password itself) remains encrypted, this metadata leakage can still be valuable in a targeted phishing campaign, or used to manipulate account recovery mechanisms on third-party services by revealing where the user holds accounts and under what identities.

Other Exposed metadata

Additionally, certain metadata elements, such as folders, tags, and comments, are not yet encrypted in Passbolt, including in version 5, even though the API supports it.

These components are currently stored in plaintext on the server and will be progressively encrypted in upcoming releases within the 5.x series as client compatibility with the new API improves.

Misconfigured server

Unverified Email in SSO Mapping

If an identity provider (IdP) allows users to self-assign or modify their email address without verification, and this field is mapped to the username in Passbolt, an attacker could exploit this misconfiguration. While this does not compromise authentication or access to encrypted secrets, it can expose part of the SSO key materials.

Compromised server

Server Access in Non-Zero Knowledge Mode

In non-zero knowledge mode, which is currently the default in Passbolt version 5, the server has access to the symmetric metadata key used to encrypt and decrypt resource metadata (such as resource names, usernames, and descriptions). Although this key is generated by the client, it is encrypted for and shared with the server, which then redistributes it to users during key synchronization. This design simplifies user on-boarding but a compromised or malicious server could decrypt, inspect, or alter metadata before passing it on to clients.

Metadata & secret integrity issues

Passbolt encrypts resources and secrets using either the user's public key or a shared key, and signs this data to ensure authenticity. However, current versions of Passbolt clients (including the browser extension) do not verify these signatures during metadata or secret decryption, which introduces a potential vector for forgery.

An attacker with access to the Passbolt database could alter the encrypted metadata. They could modify the signed metadata, and the clients would presently not detect this tampering. As a result, end users

may unknowingly act on forged metadata (e.g., trusting an altered label or resource URL).

User key integrity / group membership integrity issues

Similarly an attacker with database access would be able to add themselves as a user (or replace an existing user public key), add themselves to a group, and wait for a user to share encrypted data with them. In the future additional security could be put in place to sign the user keys with for example the administrators keys.

Compromised client

Memory access

Passbolt does not protect the end user in a scenario where the attacker would be able to read the content of the memory on the client, e.g. a scenario where an attacker is capable of breaking the browser sandbox. Regular organizations should be fine by making sure the end user browsers are up to date and without malicious extensions installed.

Filesystem / keylogger access

Similarly, passbolt does not fully protect the end user in a scenario where the attacker has read access to the local filesystem or has a keylogger installed. It would be possible in this scenario for example to gather the secret key from the local storage and the passphrase using a keylogger. Enforcing general end user endpoint security best practices (such as having an anti-virus in place, patched operating system, etc.) should be enough to mitigate the remaining risks to an acceptable level for most organizations.

Clipboard access

Similarly, passbolt cannot prevent another application or web extension with clipboard access right to listen to clipboard changes and protect the password if the user chooses to use this functionality in a compromised environment. Reducing the number of installed applications and extensions to a minimum is a good risk mitigation measure.

Misconfigured client

Weak secret key passphrase

At the moment there are no rules to enforce that a passphrase must be strong even though it is encouraged. For example it is possible for a user to select a passphrase that looks like it has strong entropy but is trivial for the attacker to guess in context.

Weaker key size / algorithms

Passbolt allows a user to generate a strong key by default while installing the extension, but it allows for less strong requirements on imported keys (ex. RSA 2048).

Anti-phishing token

Our research shows that the majority of users do not understand the concept of phishing and therefore do not understand the concepts behind the security token. Additional training and prompt may be required for this mechanism to be useful.

Malicious visitor

User enumeration

Since the authentication is challenge based, an attacker could ask for a challenge for a given public key to find out if the associated user is registered on a given domain. One easy way to mitigate this risk is to use a public key that is specific to passbolt and not advertised on public key servers. By default keys generated by passbolt are not uploaded on public key servers.

Homoglyph-based User Impersonation

When registration is open, an attacker who can create email addresses within an allowed domain may register using an address that appears

visually similar to that of a legitimate user by leveraging characters that look alike but are different (e.g., a (Cyrillic) vs a (Latin)). This could lead users or administrators to mistakenly trust the malicious identity, believing it to be associated with the legitimate user.

Malicious logged in user

Data modifications / invalid encrypted content

In this scenario a disgruntled (or clumsy) user would delete or edit the data and render it unusable. While passbolt will keep an audit log of the user action it will not provide tools to recover the lost or modified data by default. It is therefore important that the administrator in charge of the passbolt instance make sure that the database is securely backed up and that such backups are working.

Malicious public keys

Similarly, passbolt does not prevent a user from uploading a malicious key. Additional work could be scheduled in the future for the webextension to perform an automatic cleanup of the keys (for example remove unused signatures, or general key bloat).

Unsafe resource export

It is possible for a rogue user to craft a malicious resource and share it, so that when exported and opened by the victim, they will trigger an issue in a third party software. For example it is possible to create a malicious resource name containing OS commands, that would then be exported as a CSV file, and trigger operations once opened in a spreadsheet software. While escaping data to avoid code execution in CSV is possible, it is not yet implemented as it requires modifying the exported data.

Malicious extension

Rogue vendor employee

An attacker with access to the Mozilla or Chrome web extension web stores would be able to distribute a malicious extension. To mitigate this risk, every developer of the passbolt team must use a strong form of authentication in order to access the systems needed to publish code. Moreover notifications are sent to the maintainers after a publication.

Malicious third party website

Exposed plugin info

In order to provide relevant information to the user during the setup process passbolt injects some information on all the pages that acts as a passbolt application. This behavior can be used to find out if the user has passbolt installed and hampers the user's privacy through fingerprinting.

Included iframe

If the attacker is able to guess the extension ID (that is random on Firefox but not Chrome) they can insert an iframe. While the attacker will not be able to access the data, this behavior can be misleading and additional work is required for passbolt iframe to display some warnings back to the user and prevent misuse.

Best Practice Checklist

This section consolidates the key security recommendations for deploying and administering a Passbolt instance. It is intended as a practical reference for administrators, complementing the more detailed guidance scattered throughout this document. After any significant configuration change, administrators are encouraged to run the built-in healthcheck tool to verify that the instance remains in a healthy and expected state.

```
$ ./bin/cake passbolt healthcheck
```

Metadata Encryption Configuration

Passbolt v5 introduces support for encrypted resource metadata, including resource names, URIs, and other descriptive fields. Administrators should review and configure the metadata encryption mode that best fits their organization's threat model and operational requirements.

For personal resources, metadata can be encrypted with the user's personal OpenPGP key, ensuring that only the individual user can access it. This is the recommended configuration for personal vaults or high-sensitivity personal accounts.

For shared resources, administrators should select an encryption mode appropriate to their compliance and privacy requirement. Organizations with strict auditability requirements, such as financial or regulated environments, may prefer server knowledge mode or restrict users to shared keys only. Organizations with stronger privacy requirements may enforce zero-knowledge mode.

Metadata key rotation should be performed whenever a user with access to shared resources leaves the organization, following the same discipline applied to secret access revocation. Rotation procedures should be tested before they are needed.

HTTPS and TLS

Passbolt must be served exclusively over HTTPS. TLS 1.3 should be the default protocol, with TLS 1.2 permitted only as a legacy transitional measure and treated as a candidate for deprecation. Recommended cipher suites, curves, and HSTS configuration are detailed in the Compromised Network section of this document.

Certificates should use ECDSA (P-256) or RSA (2048-bit minimum), with a lifespan not exceeding 366 days and ideally 90 days to reduce exposure in case of compromise. Certificate rotation should be automated where possible. The security headers described in the Transport Security section (X-Content-Type-Options, X-Frame-Options, Referrer-Policy, etc.) must be present and verified after each deployment or reverse proxy configuration change.

Database

The database user account used by Passbolt should follow the principle of least privilege, restricted to only the operations the application requires. Direct root or superuser access should be disabled for the application connection.

Encryption at rest for the database should be enabled at the storage layer, as Passbolt does not manage this directly. Audit logging should be enabled on the database engine to capture access and modification events. Credentials and connection strings should be rotated on a defined cadence, and rotation procedures should be tested before they are needed.

Backup

Database backups should be performed on a regular schedule appropriate to the organization's recovery point objective. Backups must be stored in a location isolated from the primary server, including at least one copy held offline or in cold storage. Restoration procedures should be tested periodically to confirm that backups are valid and recovery is achievable within the expected time window. Retention policies should balance recovery needs with data minimization principles.

OS Hardening

The operating system hosting the Passbolt server should be hardened to at least CIS Level 1 baseline. Unnecessary services and network-facing daemons should be disabled. A strict patch policy should be in place, with security updates applied promptly. System time must be synchronized using NTP to avoid authentication and logging issues, particularly for TOTP-based MFA. The filesystem permission model should follow the principle of least privilege, and logging should be configured to capture authentication events, privilege escalations, and system errors.

An intrusion detection system (IDS) is recommended to monitor for suspicious activity at the host level. The preconfigured images provided by Passbolt already apply CIS benchmarks, but administrators should review and adapt these to their own environment requirements.

Network Segmentation

A Web Application Firewall (WAF) is recommended for public-facing deployments. The Passbolt server should be placed behind a reverse proxy or load balancer. Ingress should be restricted to HTTPS (port 443) and, where applicable, a management port accessible only from trusted administrative networks. Egress should be limited to the specific external services the instance requires, such as the Yubico API for OTP verification, external LDAP or SMTP servers, and any configured identity provider. Unused ports should be blocked at the firewall level.

Reverse Proxy

When Passbolt is deployed behind a reverse proxy, TLS termination should occur at the proxy level using the cipher suites and protocols described in this document. The proxy must forward the correct headers (X-Forwarded-For, X-Forwarded-Proto) so that the application can correctly determine the client origin and enforce HTTPS redirects. Rate limiting should be configured at the proxy level to reduce the impact of brute-force attempts. Request body size limits and timeout values should be tuned to match the expected usage patterns of the API, preventing resource exhaustion. The proxy should also be configured to enforce the

security headers described in the Transport Security section if they are not already set by the application.

Identity Provider and MFA

When SSO is configured, the identity provider should use OpenID Connect or OAuth 2.0 with the Authorization Code Flow, as described in the SSO section. The IdP must be configured to verify email addresses before mapping them to Passbolt accounts, to prevent account takeover via unverified email claims. Session lifetime at the IdP level should be aligned with the organization's access policy, and inactive session expiry should be enforced.

Multi-factor authentication should be enabled and enforced by policy for all users, with at least one recovery method available. Supported MFA types in Passbolt include TOTP (compatible with standard authenticator applications), Yubikey OTP, and Duo. MFA enrollment and recovery procedures should be documented and communicated to users before rollout. MFA events should be included in audit log monitoring.

Endpoint and Extension Control

Browser extensions installed on user endpoints should be limited to a known and approved set. The Passbolt extension is signed and distributed through official browser web stores, and automatic update mechanisms should remain enabled to ensure security patches are applied promptly. Administrators should establish and communicate a clear policy on approved browsers and extension installations.

Endpoint protection including EDR and antivirus should be in place and kept up to date, as Passbolt cannot protect secrets if the host environment is compromised.

Acknowledgements

The security researcher community especially:

Cure53: Dr.-Ing. Mario Heiderich, Dr. Nadim Kobeissi.

Mailvelope: Thomas Oberndörfer.

Quarkslab: Philippe Teuwen, Angèle Bossuat.

Thank you for your contributions.

Document Revision History

Summary

March 2026

- Clarify in the random generation section of the SW that, on the browser side, the `crypto.getRandomValues()` function is used as a seed, never directly as a key.
- Clarify default key configuration: ECDSA (default), RSA (optional, previous default prior to v5).
- TLS 1.2 marked as legacy support only.
DHE-RSA-AES128-GCM-SHA256 and
DHE-RSA-AES256-GCM-SHA384 have been removed from the recommended cipher suites.
- Clarify OpenPGP private key is not stored server side by default when using SSO, only when using account recovery.
- Added best practice checklist.

August 2025

- Update with v5.4 changes.
- Add packaging information

April 2025

- Update with v5 changes.

April 2021

- Add more explanations on residual risks based on PBL-01 security audit by Cure53.

January 2021

- Update with v3 changes.

August 2019

- Add default CSP info
- Add CSV command injection to residual risks.

July 2019

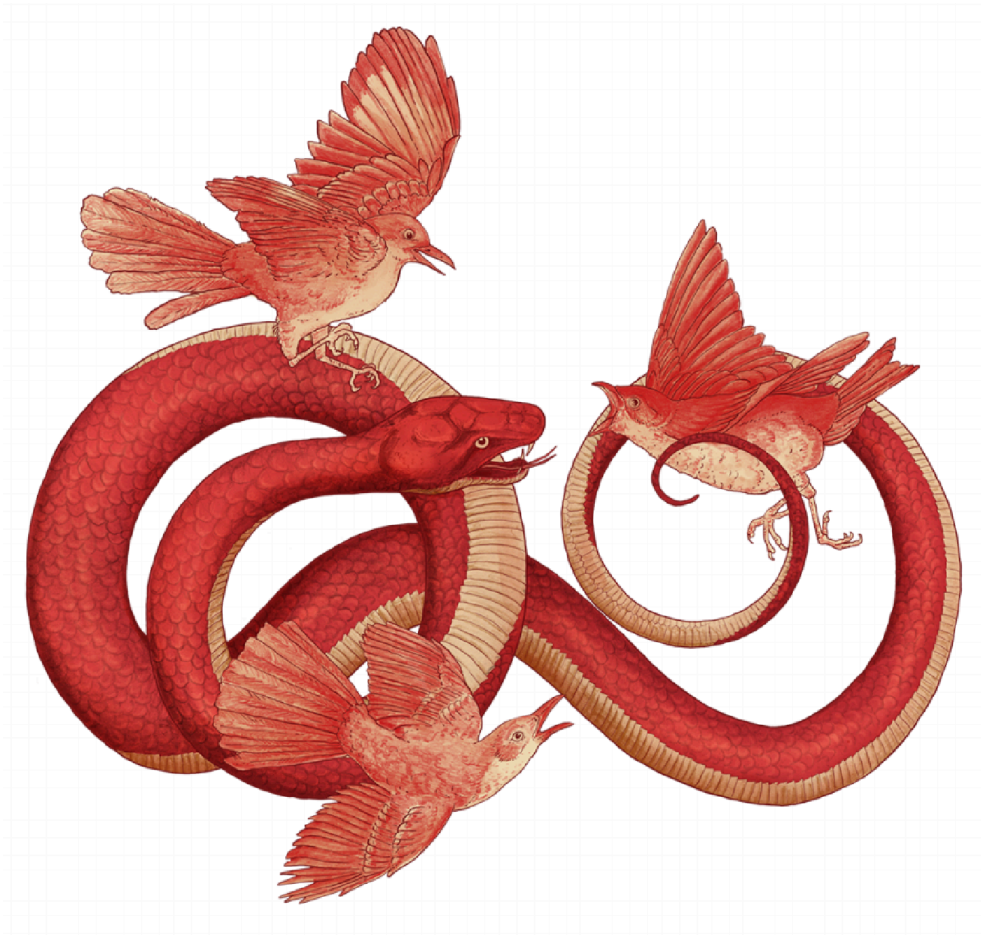
- Reconsolidation of several documents into one.
- Risk mitigation strategies.
- Residual risks.

May 2019

- Application Architecture.
- Crypto overview.

January 2018

- Authentication and authorization.



© 2026 Passbolt SA, All Rights Reserved.

Passbolt is a registered trademark of Passbolt SA. Other product and company names mentioned herein may be trademarks of their respective companies. Product specifications are subject to change without notice. Made with love in Luxembourg.

The content of this document is made available under Creative Common BY-SA 4.0 license.