



# Security White Paper

Passbolt Pro Edition v2.11

August 2019

# Table of content

<b>Introduction</b>	<b>4</b>
<b>Application architecture</b>	<b>6</b>
Server component	6
Server side command line tools	7
Data stored on the server	8
Web client	10
AppJS	10
Web extension	11
Other clients	12
<b>Crypto Overview</b>	<b>13</b>
OpenPGP	14
Encrypted data scope	14
Data in use	14
Data in motion	15
Data at rest	15
Pseudo random number generator (PRNG)	15
Key management	16
Server key	16
Client secret key	17
Private Key storage	18
Passphrase storage	18
Public key exchange	18
Password Generator	19
<b>Authentication</b>	<b>20</b>
Server key verify steps	22
Login steps	22
Token format	23
Multiple Factor Authentication (MFA)	23
TOTP	24

Yubikey OTP	25
Duo	25
<b>Authorization</b>	<b>26</b>
Roles	26
System-wide roles	26
Group level roles	26
Resource level roles	26
Logical roles	27
<b>Risks mitigation strategies</b>	<b>28</b>
Phishing	28
Cross Site Scripting (XSS)	28
Persistent XSS	29
Reflected XSS	29
Unsafe methods	30
Javascript eval	30
PHP exec	30
Unsecure deserialization	30
SQL Injection	30
File upload	31
CSRF	31
Transport security	32
Security headers	32
Cookie security	32
Recovery risks	32
Web extension best practices	33
Trusted domain	33
Autofill	33
Other best practices	33
Continuous integration and testing	33
Authentication	33
Signed releases	34
Code review and publication	34
Security alerts	34
Static code analysis	34
Bug bounty	34
3rd Party Audits	34
<b>Residual risks</b>	<b>35</b>

Compromised Network	35
Man in the middle	35
Exposed metadata	36
Compromised client	36
Memory access	36
Filesystem / keylogged access	36
Clipboard access	36
Misconfigured client	37
Weak secret key passphrase / algorithm	37
Weak keys	37
Security token	37
Malicious visitor	37
User enumeration	37
Malicious logged in user	38
Data modifications / invalid encrypted content	38
Malicious public keys	38
Unsafe resource export	38
Malicious extension	38
Rogue vendor employee	38
Malicious admin	39
Server key modification	39
Malicious deserialization	39
Malicious third party website	39
Exposed plugin info	39
Included iframe	39
<b>Document Revision History</b>	<b>40</b>

# Introduction

When Passbolt team embarked on the ambitious journey of creating the best possible password manager designed for collaboration, we started by drawing a few lines in the sand. Here are some of the guiding principles that help us shape Passbolt and make it truly unique.

**End to end encryption.** The client is responsible for generating user keys, as well as encrypting and decrypting content. Moreover the code responsible for such sensitive operations is distributed through another trusted channel and thus can never be altered by 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 access means removing the ability to decrypt future versions.

**No passphrase derivatives server side.** An encrypted version of the passphrase (or even a hash) should never be seen server side. Passbolt relies on a challenge mechanism, instead of a password based one, in order to help a user login.

**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 software stack required to run the server should also be open source. The software should not include any mandatory proprietary component of any kind.

**No mandatory internet access.** It should not include any 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 bulk of the software must remain usable in a closed network with no internet access.

**No tracking.** The software should not report back about user behavior to any 3rd party website unless the administrator or the user explicitly opt-in.

If these principles are also important to you, there are good chances 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.

Feel free to contact us at [security@passbolt.com](mailto:security@passbolt.com) if you want to discuss any of this further.

# Application architecture

Some knowledge about the different layers of the application is needed in order to better understand the attack surface. This section briefly explains the overall software architecture of passbolt.

## Server component

Passbolt clients connect to a server component written in PHP 7 and more specifically CakePHP 3, following the Model-View-Controller (MVC) design pattern. This server application relies on a Mariadb database, and a caching system that can be configured to use either the local file system or Redis. The server uses a combination of Opengp-php library and PHP Gnupg extension, to validate keys and perform cryptographic operations related to the authentication.

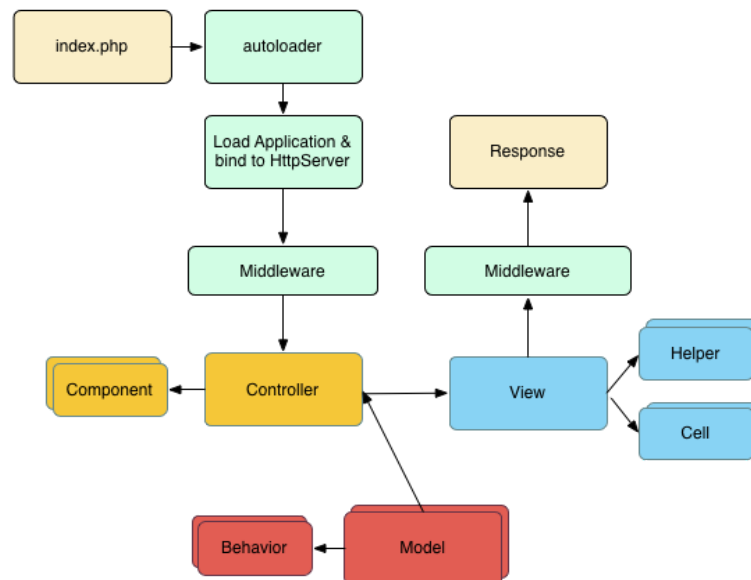


Fig. Typical server request lifecycle ([Source](#))

The applications have a plugin oriented architecture. Each plugin also follow the same MVC paradigm. Plugins are organized following functional areas such as account settings, audit log, directory synchronization, email notification settings, export and import, multi factor authentication, tags, web installation wizard, etc.

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 server application is majoritarily composed of Restful API endpoints. It also includes some HTML documents such as the login page. The full documentation of the available API endpoint can be found online at [help.passbolt.com/api](http://help.passbolt.com/api).

```
GET /healthcheck/status.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.*

## 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 healthchecks, data integrity checks, add user command (useful to add the first user), etc.



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.

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

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

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

*Fig. example of server-side health check tool*

## Data stored on the server

The core application revolves around a set of core tables such as users, gpgkeys, roles, groups, permissions, resources (the password metadata), secrets (the encrypted content), authentication\_tokens. Additional plugins such as directory sync, audit logs extend this data model and mostly reference the core schema.

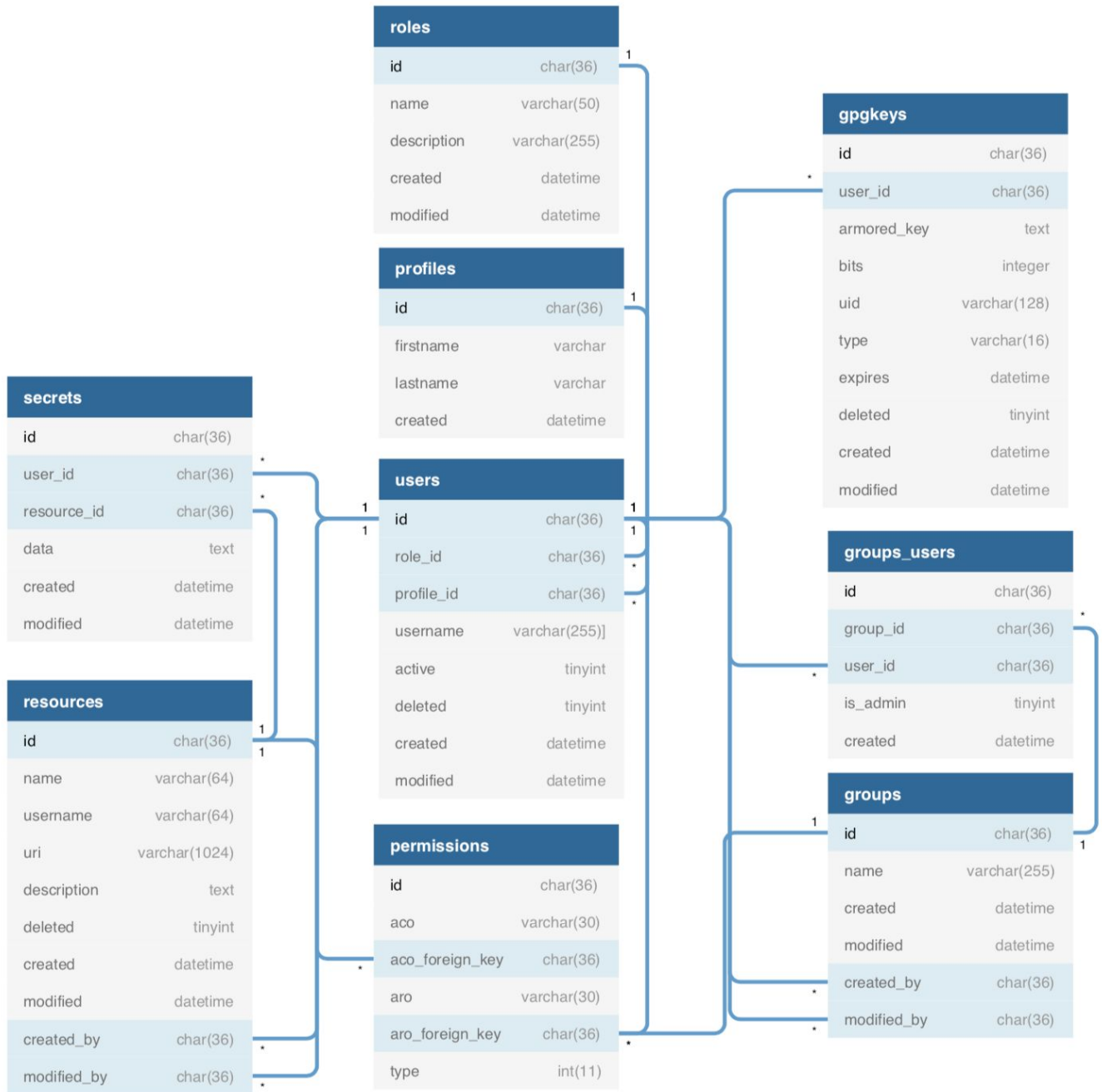
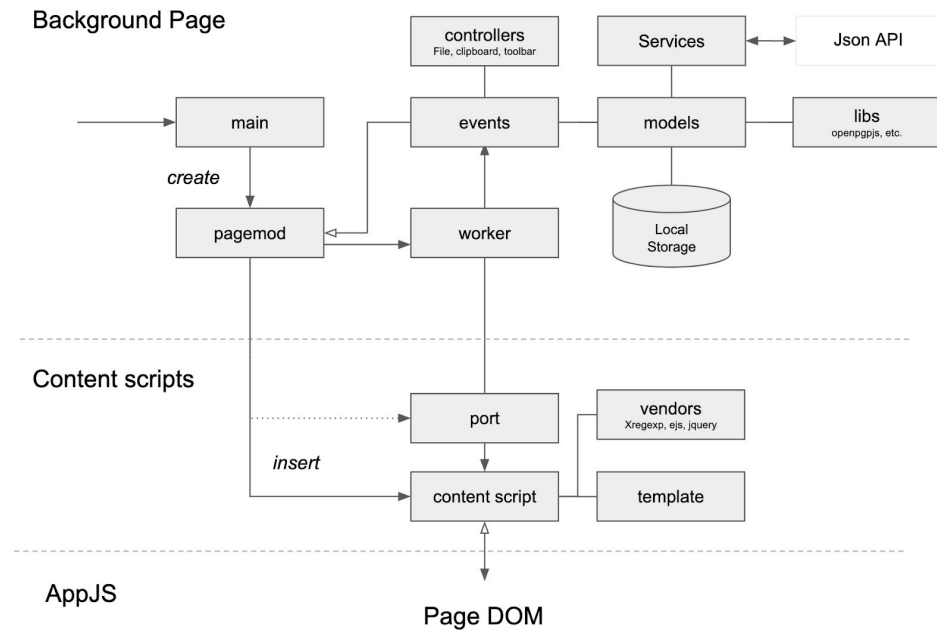


Fig. passbolt core data model

## Web client

Currently the web client is composed of two parts: an application served by the server (AppJS) and a web extension (composed of a background page and content scripts). Both are written in JavaScript using CanJS and React.



*Fig. Passbolt web extension high level architecture*

## AppJS

The javascript application served by the server, also called AppJS, handles all the non critical parts of the application such as the password and users workspace, including the left side navigation, the main grid, search bar and sidebar.

This AppJS is allowed to access or interact with the webextension through a limited set of events and therefore can trigger some operations such as opening a dialog, copy a username to clipboard, etc. However it is not able to interact with the content of these dialogs, or access

sensitive content such as the user secret key, passphrase or the decrypted content.

Historically this design was more relevant when Firefox and Chrome didn't have the same browser plugin architecture. Since they now share a unified webextension format (also supported by other browsers), the AppJS will progressively be moved to the content scripts in the webextension to reduce complexity.

## Web extension

The web extension takes care of the sensitive part of the application such as managing setup, login, password input, decryption, share, etc.

The web extension insert content scripts in pages marked as trusted by the user during the setup. These content scripts can access and modify the web page DOM. They can also see any changes that were made to the DOM by the scripts that originated from the server. However the scripts served by the server can not see JavaScript properties added by content scripts (and vice-versa). This mechanism ensures that the two javascript environments are not maliciously affecting each other.

Moreover all the crypto functionalities are running in a third separate environment, called the background page, which implement the long-running logic of the webextension. Functionalities from the background page are exposed through another 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.

Whenever content scripts need to create html elements, and in order to create a "secure DOM" visible only by the web extensions scripts, passbolt relies on iframes inserted in the page by the webextension. The content of this iframe is served under a different domain (e.g. `chrome-extension://`), and therefore not accessible to the page served by the server, thanks to cross domain policy restrictions.

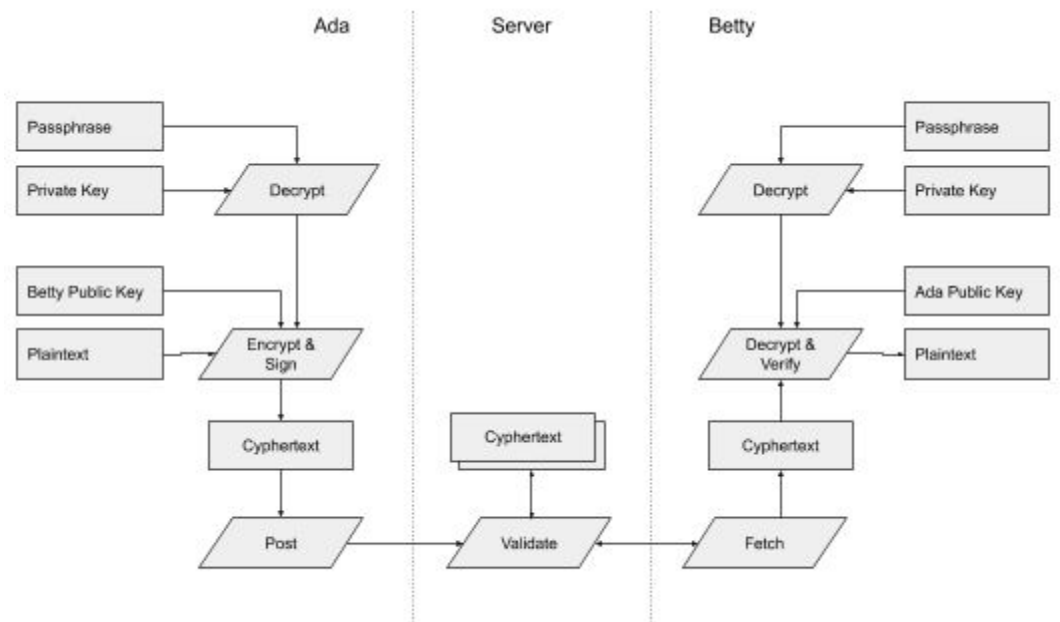
## Other clients

Additional clients are available or can be developed on top of Passbolt API. For example Passbolt team maintains a [Command Line Interface](#) client developed in NodeJS. Other clients are made available by the community such as [Wrench](#) a CLI client written in Python.

# Crypto Overview

The main difference with passbolt and other password managers, as you could guess from our guiding principles, resides in the fact that there is no symmetrically encrypted vault shared with multiple users.

Passbolt instead relies extensively on public key cryptography, and more specifically on the OpenPGP standard to achieve the interoperability goals.



*Fig. Encryption and decryption cycle in passbolt*

## OpenPGP

The different layers of passbolt rely on different OpenPGP implementation and/or bridges.

On the server-side passbolt relies on [openpgp-php](#) (a pure php implementation of OpenPGP protocol) to run a pre-validation of keys and messages prior to calling GnuPG using the native [gnupg-php](#) extension (which in turn uses [libgpgme](#)). Work has been done in 2019 to create an abstraction layer in order to support alternative backends in the future.

On the client side passbolt webextension relies on [OpenPGP.js](#) a library also used by projects such as ProtonMail or Mailvelope. This library has been reviewed multiple times by Cure53 in [2015](#) and [2018](#).

## Encrypted data scope

### Data in use

For the data in use, i.e data in the memory or file system on the server or client side, only the passwords are encrypted. Upcoming features that acts as password properties such as file attachment and custom fields will also be encrypted in the future. However the metadata such as the username, the url, the comments, the list of people who have access to the password with are not encrypted using OpenPGP, and are stored in plaintext both on the client and server side.

Passwords can be made available in a decrypted form at some point, for example when using the quick access functionality or by copying a secret to the clipboard, but they will never be stored in plain text on the filesystem on either the client or server side.

Passwords are encrypted once per user. When sharing with a group the extension first fetch the group memberships and compile the final list of recipient. The server in turn checks that all the recipient are included when a new version of the secret is published.

## Data in motion

For the data in motion, i.e. on 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 effort to enforce encryption for the data in motion. By default passbolt installation scripts will help the administrator setup certificate on the server. Similarly the default configuration of passbolt server makes sure non encrypted request is redirected to its https counterpart. While it is possible to disable that behavior, it will trigger the display of an “unsafe mode” banner in the footer.

## Data at rest

For the data at rest, for most of the clients and servers, it is also possible to encrypt the database at the [file system](#) level as well. This additional encryption layer can be useful, for example, in the case where the machine running passbolt is seized or stolen. However this extra configuration is not handled or enforced by passbolt.

## Pseudo random number generator (PRNG)

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

On the server side, passbolt uses GnuPG which implements its own [random number architecture](#) which in turn relies on configurable entropy gathering modules such as `/dev/random`. If passbolt runs on a virtualized system, the size of the entropy pool will be affected and passbolt administrators can therefore consider using additional hardware sources, or if not possible, gather additional entropy with the help of software packages such as [Haveged](#).



Similarly the PHP application uses [random bytes](#) which in turn relies on Linux [getrandom](#) syscall. The component requiring a secure PRNG are most notably the one generating the random UUIDs used for the authentication challenge or the 2FA authentication proofs.

On the webextension side the application uses native browser crypto api [getRandomValues](#). The webextension requires a secure PRNG for generating the random UUIDs used for the authentication challenge and for OpenPGP related operations such as key generation or secret encryption and for the password generator functionality.

On the Nodejs client side the application uses the node [crypto api](#), which in turn relies on the OpenSSL library.

## Key management

### Server key

The OpenPGP server key is generated during the setup. It can be generated via `openpgpjs` 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. This server key is mostly 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.

By default passbolt does not encourage using passphrases on the server secret key to facilitate the deployment and reduce support. To our opinion the benefits of using a passphrase are quite limited, since the passphrase will either be stored unencrypted on file or using environment variables. However, it should be possible to set one up by editing the GnuPG configuration to allow pinentry via loopback<sup>1</sup>.

The client downloads the server public key during the setup (or account recovery) and ties it in the configuration of the webextension to the domain. If the key changes an error will be displayed. At the moment the user must perform an account recovery to accept any server key changes.

---

<sup>1</sup> See. [Gnupg Agent Options](#)

## Client secret key

The client secret key is generated during the initial setup. By default it uses RSA 2048 keys, but is possible to use larger RSA key by importing one.

As of July 2019 support for keys based on elliptic curve cryptography (Ed25519) is not official and still experimental but should be made official in the course of the year.

The client secret key is encrypted using a passphrase selected during the setup (or in another system if the user is importing an existing key). When generating a key via passbolt by default the webextension only enforce the secret key passphrase to be at least 8 char in length. It provides additional prompt for complexity and information such as if the passphrase has been compromised in a breach before (using [haveibeenpwned](#) service).

### Set your passphrase

---

This is the only passphrase you will need to remember from now on, so make sure you choose it wisely!



click here to view in clear text

Some tips for choosing a strong passphrase:

- It is at least 8 char in length
- It contains lower and uppercase character
- It contains letters and numbers
- It contains special characters (like / or \* or %)
- It is not part of a dictionary

*Fig. set passphrase screen in passbolt webextension setup*

## Private Key storage

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<sup>2</sup>.

## Passphrase storage

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. The passphrase is stored in memory, along with the timeout function to reset it, in the background page, as a property of the user singleton object.

It is possible for an administrator to configure server side the timeout values that are allowed or disable this feature entirely. By default the following options are proposed: 5 minutes, 15 minutes, 30 minutes, 1 hour, until logout. If the browser is closed the background page will be destroyed and the passphrase along with it.

## 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 system relying on OpenPGP.

Passbolt removes the need for manual keyring management and for a web of trust by providing a central authority to access the keys of the users of a given domain. Therefore the web extension 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 the public key of the user is sent to server. It is validated then added to the database in a record associated with the user.

---

<sup>2</sup> See. <https://chromium.googlesource.com>

This public key is then distributed to the other users using an API endpoint that returns all the OpenPGP keys that have been saved on the server with an optional parameter to retrieve only get the keys modified since a given timestamp. By default the web extension stores such timestamp each time it queries the endpoint to make sure not to re-import keys of existing users marked as unchanged by the server.

This key synchronization takes place prior to all encryption involving multiple users.

## Password Generator

Passbolt offers a cryptographically secure password generator. This generator creates high-entropy passwords of 17 char in length with upper/lower case letters, numbers and special characters. As of July 2019 passbolt does not offer options to further customize/configure these default routes.

# Authentication

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 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 common password. Moreover a different secret is used for every authentication attempts.

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

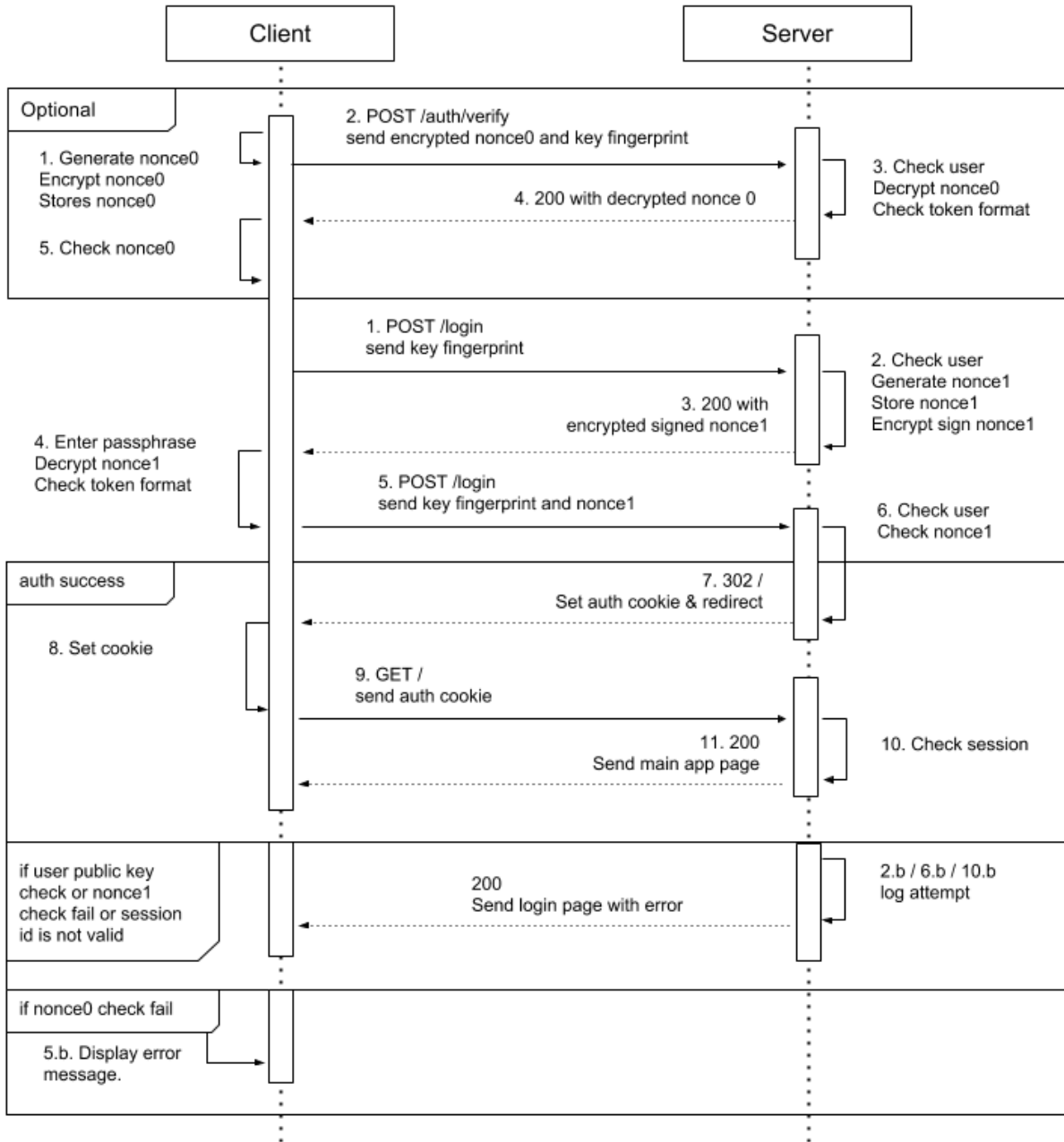


Fig. Sequence diagram of a GPGAuth based authentication

## Server key verify steps

This part of the authentication is optional but always enforced by passbolt at the moment. This server identity verification should not be understood as an end to end server authentication, e.g. it does not protect against an attacker performing a man in the middle attack.

However it can help in certain unlikely scenarios such as when a domain name is seized.

This section of the authentication works as follows:

1. The client generates an encrypted token of random data (encrypted with the server public key), and stores the unencrypted version locally.
2. That encrypted token is sent to the server along with the user key fingerprint.
3. Based on the user key fingerprint the server checks if the user exists and is active. If it is the case the server decrypt the nonce and checks if it is in the valid format.
4. The server sends back the decrypted nonce.
5. The client checks if the nonce match the previously recorded one. If it does not match the client warns the user that the server identity cannot be verified.

## Login steps

1. The user sends their key fingerprint.
2. The server checks to see if the fingerprint and user associated with 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 enter their private key passphrase, the client decrypt the nonce and check 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.

## 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 mean 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
```

## Multiple Factor Authentication (MFA)

While one could argue that GPGAuth is already a multi factor authentication mechanism, Passbolt Pro Edition provides additional providers 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.

When an MFA is completed a token is set in a 'passbolt\_mfa' cookie linked to the trusted domain. Such token is reset after a configurable period (72h by default) or if the user agent changes.



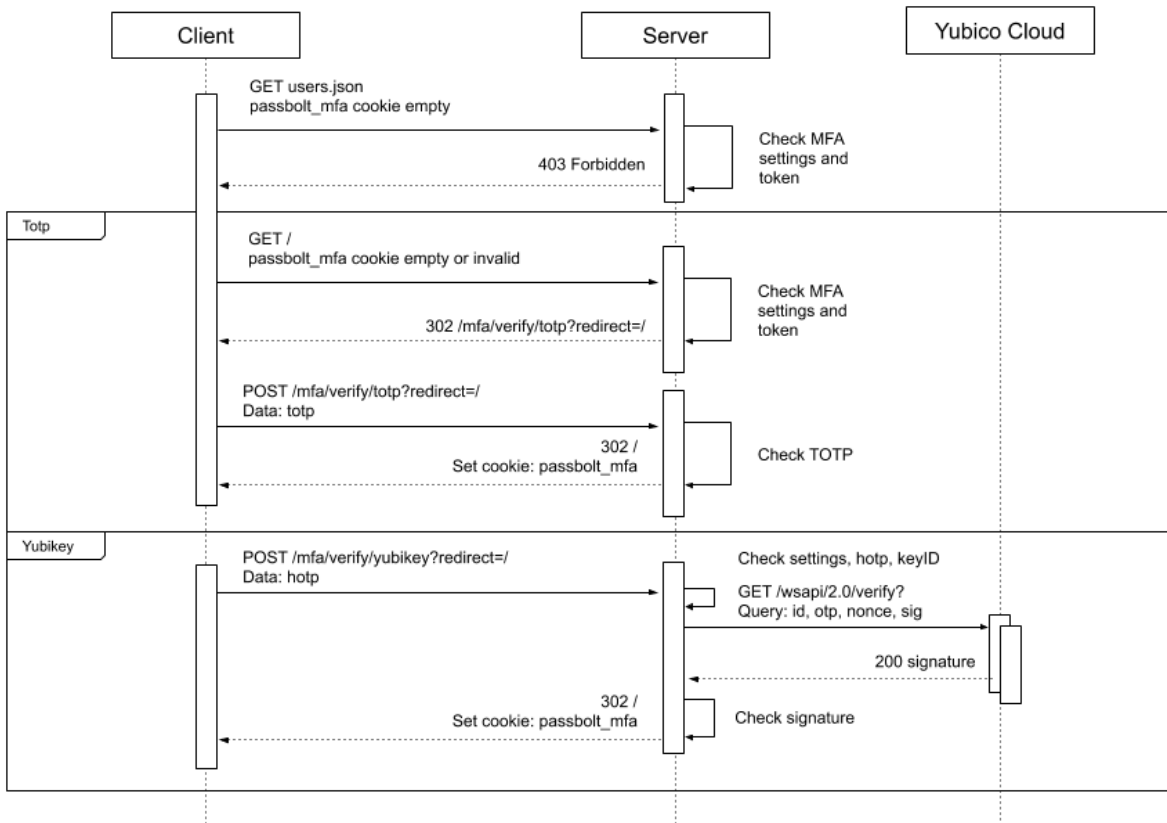


Fig. MFA sequence diagram

## 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 receive the TOTP code generated by the user device and using 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 webservice for verifying OTP) which requires an [API key](#) to work. It is possible to [host yourself the OTP validation server](#), but this approach is not supported by Passbolt at the moment.

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 key (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 to login.

## Duo

This provider relies on Duo. For Duo authentication, a signature (`sig_response`) is captured in the client via an iframe loading a 3rd party service. This iframe drives the authentication of the user with the Duo service, triggering for example a push on a configured mobile device, or a phone call, etc. The behavior is configurable by the admin directly in the Duo service. This signature is then verified server side: using the integration key, secret key, integration secret key, and the signed response as input, if the response is valid, it will return the username of the authenticated user.

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

# Authorization

## Roles

### System-wide roles

The system proposes by default two system roles “admin” and “user”. This system is the first line of the authorization mechanism performing authorization check directly at the application controller level and/or at the model level using User Access Control objects.

In a nutshell, an administrator manages the instance. In practice it means that they can manage organization wide settings such as the content of the email notifications or which multiple factor authentication provider is enabled. Another responsibility is to create or delete users, manage groups and group managers, 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 administrator or appoint a regular user.

Due to the nature of the encryption in passbolt, 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).

### Resource level roles

Passbolt offers three permissions on the resource level.

**Owner:** can manage share settings, delete, update, read.

**Update:** can update the record and delete.

**Read:** can only read and use the password metadata and secret.

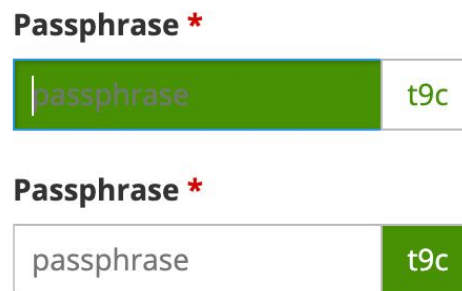
## Logical roles

Finally another layer of logical access control can be enforced depending on the context, most notably when the user is the creator of given item that does not have a complex permission system associated with it. For example, only the user that created a comment can edit it.

# Risks mitigation strategies

## Phishing

Several mechanisms are present in 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.



*Fig. anti-phishing security token*

Passbolt mitigates this type of attacks 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, an additional interaction with color changes on a field focus event is in place.

## Cross Site Scripting (XSS)

High on the list of web application vulnerabilities a XSS vulnerability would allow an attacker to run arbitrary content on the page.

## 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. While historically passbolt server used to provide server side sanitation in version 1, it is not the case for version 2. 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 EJS) 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 equivalents`<sup>3</sup> in the code. These issues are captured by the static coding tools used by Mozilla uploading a new version of the extension. Moreover Passbolt 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.

Passbolt uses in certain cases parameters provided in URLs for example to access directly a resource (and therefore allow sharing a link to resource with a coworker for example) or prefill a form with data (to help an admin add a user for example).

---

<sup>3</sup> For example: `.html()` `.append*()` `.insert*()` `.prepend*()`, etc.

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.

## Unsafe methods

### Javascript eval

The execution of eval statement is not allowed in the webextension thanks to the [default policy restrictions](#) of web extension Content Security Policy. In the javascript application served by the server, the only context where eval could run, eval or equivalent is run only in trusted libraries (such as well tested template libraries). This policy is enforced using static code analysis<sup>4</sup>.

### 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 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 uses serialization in some rare scenarios such as the EmailQueue plugin. To mitigate the risks passbolt does not handle serialized / deserialized user data without a thorough validation first.

## SQL Injection

The server side application almost never uses direct SQL queries but instead access the data through the framework ORM. By default this database abstraction layers prevent most SQL injection issues. User input used by the ORM is always carefully validated. [Remaining risks](#) are

---

<sup>4</sup> <https://github.com/passbolt/passbolt-appjs/blob/master/.eslintrc.json#L35>

mitigated using code reviews. Independent reviews have not highlighted any security issues in this area in the past.

## **File upload**

File upload is another sensitive area for web applications. Passbolt at the moment only support file upload in relation to the management of user avatars. The filesize 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 right formats. The application store all relevant metadata in a database record and use a random filename for the actual filesystem storage.

By default passbolt a local file adapter and places the avatar in the application webroot under the public image directory. Passbolt offers the option to use a different file adapter in order to for example host the files in AWS S3 or Google Cloud Buckets.

## **CSRF**

Cross Site Request Forgery is an attack scenario where a user 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 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.

## 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
- X-XSS-Protection 1; mode=block
- Referrer-Policy sameorigin

### Cookie security

By default the cookie used for the session and the mfa token are set http only with secure flag on. The HttpOnly flag mitigate the risk of client side script accessing the protected cookie. The secure flag mitigate the risk of the cookie being sent in clear over http.

## Recovery risks

In order to recover an account a user must first validate that they have access to the associated email by clicking on a verification link. They must then import their private key and uses their passphrase to decrypt it, in order to complete an authentication challenge. This process makes an attack scenario quite complicated, but reduces the usability for the end user by introducing chances of losing access to the account permanently if the private key and/or passphrase is lost.

## **Web extension best practices**

### 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 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 in a browser popup a suggested entry and click on “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.

## **Other 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.

### Authentication

Every developer of 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 release team use digital signatures for tags to help the administrator ensure the integrity of each release. Similarly the web extension releases are signed to ensure that only a legitimate passbolt extension can be installed / updated.

## Code review and publication

Only a small amount of people are responsible and allowed to publish code. Before being pushed on a sensitive branch, each pull request is therefore reviewed and validated by different maintainers.

## Security alerts

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

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

## Bug bounty

Passbolt has a bug bounty program running on the <https://yeswehack.com> platform that includes monetary rewards.

## 3rd Party Audits

- [Openpgpjs security review \(2018\) by Cure53](#)
- [Openpgpjs penetration test report \(2015\) by Cure53](#)
- [CakePHP Security Assessment \(2017\) by NCC Group](#)
- [Passbolt server application code review \(2018\) by CakeDC](#)

## Residual risks

We believe passbolt is a software solution that provides a level of risk that is acceptable for most organization. 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 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 share a password with them.

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

An attacker with access to the data in motion or at rest would be able to access the passwords metadata such as the sites the user is using and the associated username. This can be proved useful as part of a larger targeted phishing campaign, or to try to game the recovery mechanisms of such 3rd party site.

## 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.

### Filesystem / keylogged 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 / algorithm

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 import a key that has a trivial passphrase.

### Weak keys

By default passbolt allows a user to generate a strong key (2048) while installing the extension, but it does not enforce minimum requirements on imported keys. It is therefore possible for a user to use a small key size and/or weak algorithms.

### Security 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.

## 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 malicious key. Additional work will be scheduled in the future for the webextension to automatically cleanup 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.

## Malicious extension

### Rogue vendor employee

An attacker with access to the Mozilla or Chrome webextension webstores would be able to distribute a malicious extension. To mitigate this risk, as explained in the [best practices](#) section, every developer of 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. Moreover notifications are sent to the maintainers after a publication.

## **Malicious admin**

### Server key modification

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.

### Malicious deserialization

Data such as emails that are placed in the Email Queue database table can be serialized as objects. Therefore some residual risks are present in a scenario where an attacker would have access to edit the data in database directly and therefore bypass data validation to be able to run arbitrary code on the web server. Limiting and securing access to the database is therefore a very important step during a passbolt installation.

## **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 hamper the user 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.



# Document Revision History

## Summary

### 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.



© 2019 Passbolt SA, All Rights Reserved.

Passbolt is 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.