passbolt

Security White Paper

Passbolt Pro Edition v3.1

April 2021

Table of content

Introduction

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

True end to end encryption. The client is responsible for generating user keys, as well as encrypting and decrypting content. 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 access means removing the ability to decrypt future versions.

No secret key derivatives server side. Even an encrypted or derived version of the user secret key should never be sent or stored server side.

Strong authentication. Passbolt must provide multi-factor authentication by default. By default it 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.

Privacy by design. No tracking. The software should not store unnecessary personal information. The software should not report back user behavior or analytics to any 3rd party website 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.

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 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 4, 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 Openpgp-php library and PHP GnuPG extension, to validate keys and perform cryptographic operations related to the authentication.

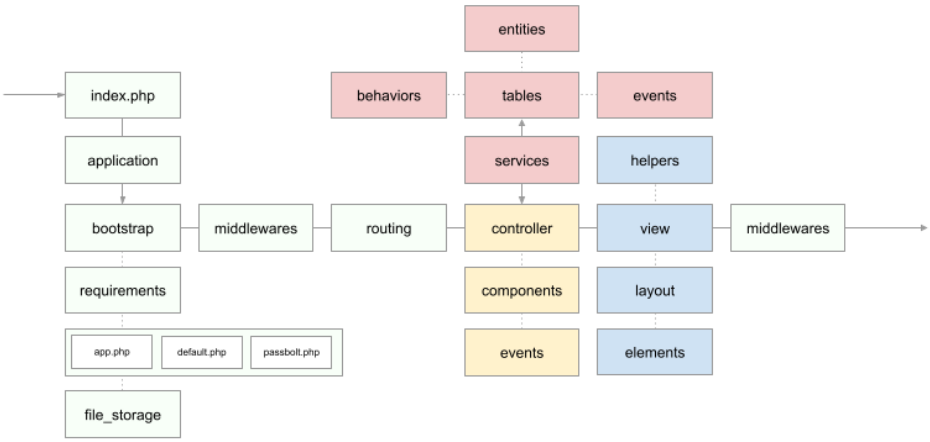

Fig. Typical server request lifecycle

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, 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 mostly composed of Restful API endpoints. It also includes some HTML documents such as the login page. The full documentation of the available API endpoints can be found online at 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 healthcheck 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), and 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.

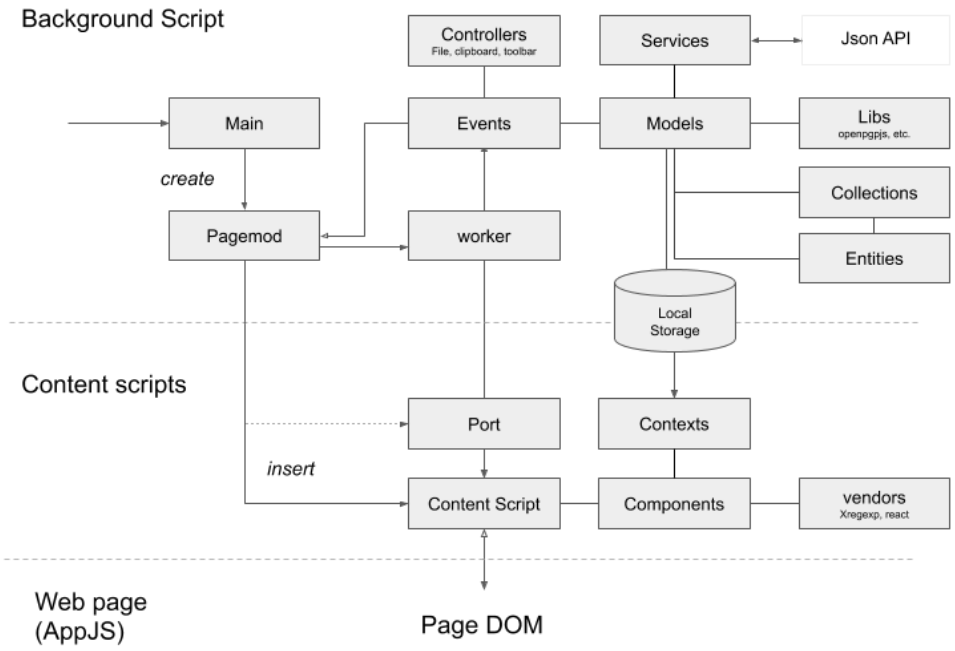

Fig. Passbolt web extension high level architecture

AppJS

A javascript application served by the server, also called AppJS, handles certain aspects of the application that are tied to the server version, such as the administration workspace that is used to configure the instance. It is used for example by an administrator to define which 2FA providers or email notifications are enabled.

This AppJS also handles the public forms such as the account registration or recovery, when the extension is not installed.

This AppJS is not allowed to access or interact with the webextension. Therefore it is not possible from a script hosted by the server to access sensitive content such as the user secret key, passphrase or the decrypted content.

Historically, prior to version 2, the AppJS and an webextension were more integrated, as this 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 and webextension have been fully decoupled.

Web extension

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

The web extension inserts content scripts in pages marked as trusted by the user during the setup. 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 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 content script. 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.

Most of the interactions, such as the ones on the user or password workspace, happen within such an iframe.

The background page and the content scripts (or scripts running inside the iframes) communicate using dedicated webextension ports. These environments also share a common set of data accessible via the extension local storage.

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 OpenPGP to achieve its interoperability goals.

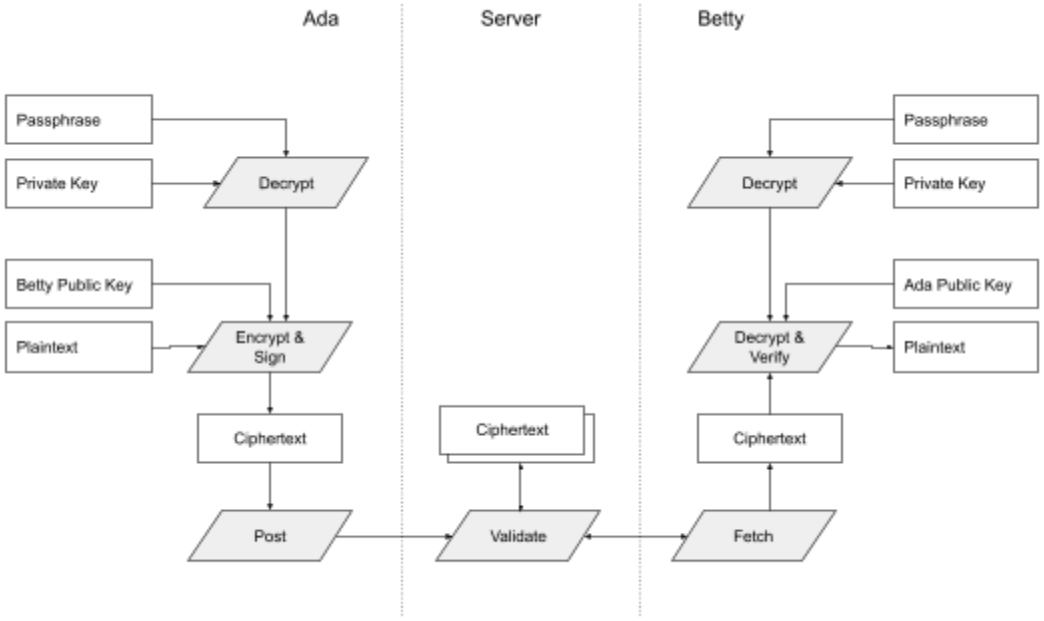

Fig. Encryption and decryption cycle in passbolt

OpenPGP

The different layers of passbolt rely on different OpenPGP implementations 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.

Type of data

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

Keys

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

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

Data

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

Data in passbolt is divided into two parts: the searchable non encrypted metadata called “resource”, and the encrypted part containing for example the passwords called “secret”.

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

```
{
  "resource":{
    "type":"object",
    "required":["name"],
    "properties":{
      "name":{
        "type":"string","maxLength":64
      },
      "username":{
        "anyOf":[
          {"type":"string","maxLength":64},{"type":"null"}
        ]
      },
      "uri":{
        "anyOf":[
          {"type":"string","maxLength":1024},{"type":"null"}
        ]
      },
      "description":{
        "anyOf":[
          {"type":"string","maxLength":10000},{"type":"null"}
        ]
      }
    }
```

```
  },
  "secret":{
    "type":"string","maxLength":4064
  }
}
```

Fig. example of resource type - the default “legacy” resource

Encrypted data scope

Data in use

In practice this means the metadata such as the name, the url, the comments, the folder it is in, the list of people who have access to the password are not encrypted, and are stored in plaintext both on the client and server side.

Passwords, and other optionally encrypted fields such as description, 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.

Secrets are encrypted once per user. When sharing with a group (or folder) the extension first fetches the group memberships (or folder permissions) and compiles the final list of recipients. The server in turn checks that all the recipients 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 efforts to enforce encryption for the data in motion. By default passbolt installation scripts will help the administrator setup 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 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 components requiring a secure PRNG are most notably the ones 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 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. 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. In 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[1].

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.

Client secret key

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

As of January 2021 support for keys based on elliptic curve cryptography (for example Ed25519) is not supported.

[1] See. GnuPG Agent Options

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 enforces 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).

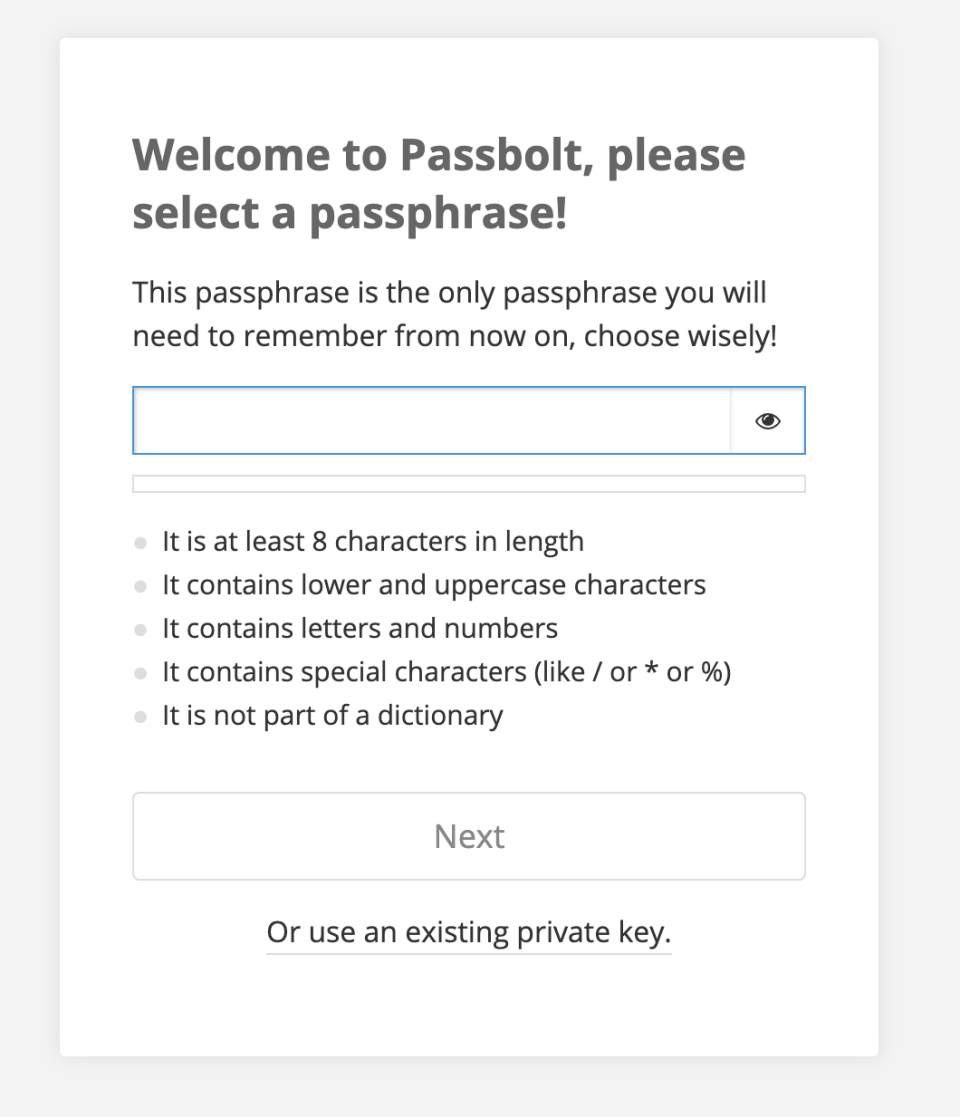

Fig. set passphrase screen in passbolt webextension setup

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.

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 systems 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, after a validation 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 all the OpenPGP keys that have been saved on the

server with an optional parameter to retrieve only the keys modified since a given timestamp. By default the web extension stores such a 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 (of new or modified keys) 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 January 2021 passbolt does not offer options to further customize/configure these default routes.

Authentication

GpgAuth

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 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 password.

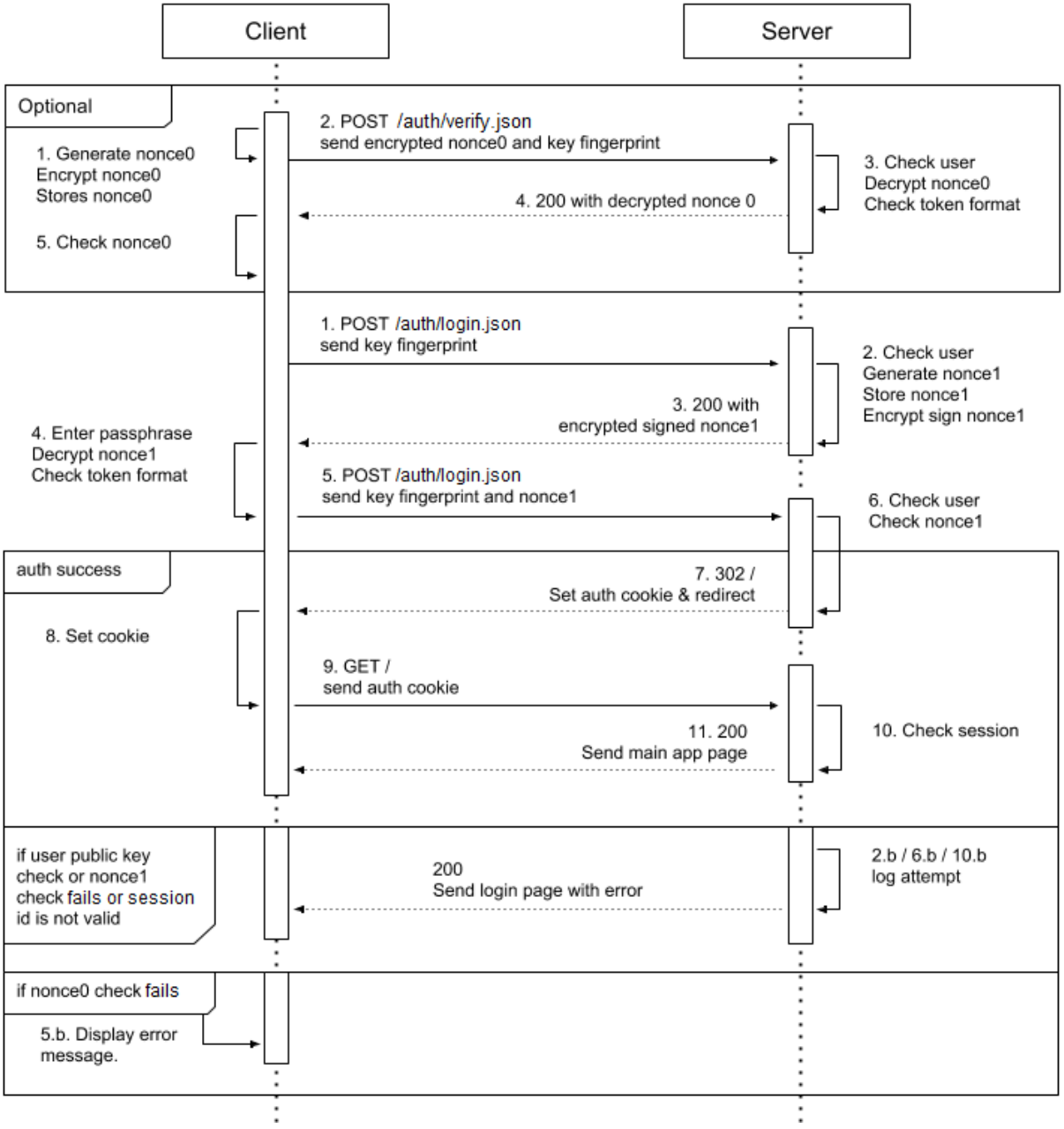

Fig. Sequence diagram of a GPGAuth based authentication

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 decrypts 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 matches 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 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.

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
```

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 a token is reset after a configurable period (72h by default) or if the user agent changes.

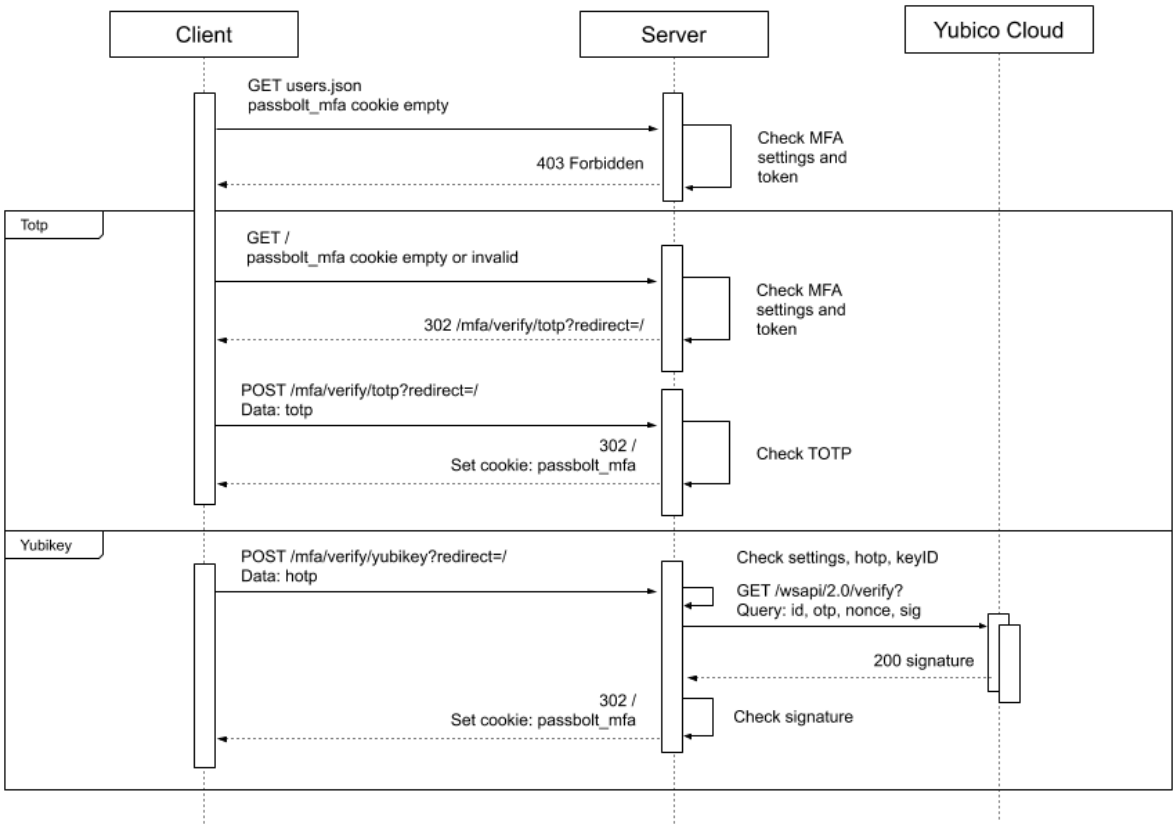

Fig. MFA sequence diagram

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 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 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 from logging in.

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 checks 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 a 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.

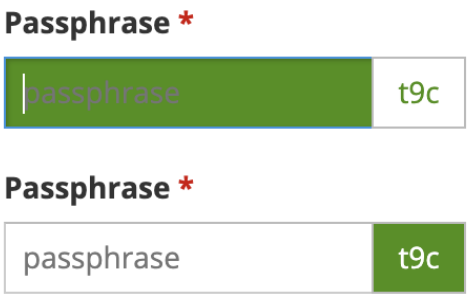

Fig. anti-phishing security token

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, a field focus event displays an additional interaction in place with color changes.

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 API used to provide server side sanitation in version 1, it is not the case for version 2 or 3. 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 equivalents[2] 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).

[2] 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 statements is not allowed in the webextension thanks to the default policy restrictions of web extension and server side Content Security Policy.

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 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 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. 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 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 filesystem storage.

By default passbolt uses 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 host the files remotely (for example in AWS S3 or Google Cloud Buckets).

CSRF

Cross Site Request Forgery is an attack scenario where a users 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
- 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.

Recovery risks

To recover an account a user must import their private key and use 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 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.

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 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. 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 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

Passsbolt team uses multiple tools to perform static code analysis such as eslint, phpcs, codacity, 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

- Passbolt Whitepaper Security Audit (2021) by Cure53
- Passbolt Web extension Security Audit (2021) by Cure53
- 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 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 RSA, ECDSA, 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 emerge.

Weak server side random number generation

Since Passbolt's supports deployment on native as well as containerized environments, a weak entropy source could yield identical server keys and predictable authentication tokens for all deployments.

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

- Using the only-urandom configuration flag under /dev/gcrypt/random.conf which disables the use of the blocking /dev/random call, replacing it with a call to /dev/urandom when applicable.
- Using an external entropy source or using one of the *league of entropy* providers when possible.
- Ensuring that */dev/urandom* was properly initialized prior to using its output for pseudo-random value generation.

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 shared 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, TLS 1.2
- Cipher suites (TLS 1.3):
    - TLS_AES_128_GCM_SHA256:TLS_AES_256_GCM_SHA384:TLS_CHACH A20_POLY1305_SHA256
- Cipher suites (TLS 1.2):
    - 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:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-GCM-SHA384
- TLS curves: X25519, prime256v1, secp384r1
- Certificate type: ECDSA (P-256) (recommended), or RSA (2048 bits)
- DH parameter size: 2048
- HSTS: max-age=63072000 (two years)
- 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

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 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 / 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 a malicious key. Additional work will 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.

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, as explained in the best practices section, 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. 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 directly edit the data in the database 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 hampered 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.

Acknowledgements

The security researcher community especially:
Cure53: Dr.-Ing. Mario Heiderich, Dr. Nadim Kobeissi.
Mailvelope: Thomas Oberndörfer.

Thank you for your contributions.

Document Revision History

Summary

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.