Security Protocols homework

ProtoPass — Online, browser-based, end-to-end encrypted password manager application

Ádám BERECZKI (XKTB7Q)
Zoltán BOGÁROMI (A3H4H8)
Tamás Soma LUCZ (HQC99F)

Last revision: 3 May, 2018

Functional requirements

Our application is a browser-based password manager app, capable of storing passwords in key-value (K, V) format. This (K, V) pair is called a **password entry** — where K is a string entered by the user containing any associated data for the password, and V is another string, the password itself. Users can generate cryptographically secure passwords and store them in the app with additional data. Users can also enter their own passwords into the app with associated data, and use the app as a secure storage for their already existing passwords.

The password manager will be implemented as a web browser application communicating with a remote server via a secure, authenticated and encrypted (HTTPS) channel. The server stores all the user's password entries in an encrypted container, called **user profile**. After the user logs in to the application with their email address and **login password**, they are able to download their user profile, containing all their password entries, encrypted. The user can decrypt their profile and access their password entries with their **container key** derived from the **container password**, and access their passwords saved into the application. For better security, the user should not choose to have the same password as login and container password. Also, the user should not ever forget their container password: while the login password can be reset by an email-based password reset feature, the container password cannot be reset without losing access to all entries stored in the user profile.

When a user enters a new password to store in the app, it is encrypted on the client-side, before leaving the machine and will only be uploaded to the server as part of the encrypted user profile. As the server only stores the password entries as part of the user profile in encrypted form, there is no need to trust the server — even if someone downloads the user's profile from the server, they will not be able to decrypt it without possessing the key

(container password) of that profile. As the user profile containing the passwords are still encrypted in-transit as well, there is no need to trust the neither the internet service provider nor the communication channel.

As the application is a client-server application, the user can retrieve their passwords from anywhere in the world — in possession of internet connection, a modern web browser, their login password and their container password.

Functional requirements of the server

The concept of the server API is described at the end of the document, this is only a functional summary.

- Registering a user with an email address and a login password
- Validating the user's email address with an email sent to the address
- Logging in a user (acquiring a session) with an email address and a login password
- Logging out a user (destroying a session)
- Downloading the encrypted user profile, the container key salt and the initialization vector in possession of a valid session
- Uploading the encrypted user profile, the freshly generated container key salt and the freshly generated initialization vector in possession of a valid session
- Downloading the container password storage key in possession of a valid session
- Changing the login password of a user in possession of a valid session
- Resetting the login password of a user via email

Functional requirements of the client

- Decrypting the user profile with the container key (derived from the user's container password and the container key salt received from the server) and the initialization vector (received from the server)
- Encrypting the user profile with the container key (derived from the user's container password and a freshly generated container key salt) and the freshly generated initialization vector
- Generating cryptographically secure random string values with configurable character classes (lower- and uppercase alphabetic, numeric and special characters) and length — as password candidates
- Changing the container password of the user profile and re-encrypting the user profile with a new container key (derived from the new password and a freshly generated container key salt) and a freshly generated initialization vector

Attacker model

Goals and capabilities of the attacker

Our assumption is, the attacker can have one goal: to acquire one or all of a specified user's passwords in unencrypted, readable form. Given an email address, the attacker wants to find out if there is a matching user in the application's user base, and if so, the attacker wants to acquire the user's password entries.

Let's assume the attacker is qualified enough to find some vulnerabilities on the server enabling them to decide whether a user with a specified email address uses the password manager application or not. We also assume that the attacker is so skillful that they are able to download that specified user's profile from the server. But even in this case, if the attacker successfully downloads the user's encrypted profile from the server, they have no additional information about the user, as the user's profile, and thus all password entries are encrypted. Besides the user *using* the password manager application, no additional information is leaked from the application in possession of an encrypted user profile. (Of course, the attacker can make assumptions about how many passwords does the user store in the application).

All above assumptions require the user's profile to be encrypted with a practically secure encryption, meaning it must be impossible to decrypt the profile in foreseeable time without possession of the container key (the user's container password, the container key salt) and the initialization vector.

Another scenario would be the attacker hacking into the database server storing the user profiles associated to the email address. Even if downloading all user profiles, the attacker would not be able to find correlations among the separate encrypted user profiles, as the application uses non-convergent cryptography, described below.

Security requirements

Our application's security is based on three principles:

- Only an authenticated user can download a user profile, the container key salt and the initialization vector, and these can only be their own data.
- The user profile can only be decrypted in possession of the container key derived from the user's container password, known only by the user.

- The user profile in unencrypted form is only present on the logged in user's machine while the user is using the app. If the profile leaves the app (it is uploaded to the server), it is encrypted. If the user logs out or closes the app, the last version of the profile is encrypted and uploaded to the server, and the local unencrypted profile is deleted from the machine.

The application's security requirements are the following:

- The application should never persist the user's profile in unencrypted, unserialized form.
- The application should never transfer the user's profile in unencrypted, unserialized form.
- The application should work on zero-knowledge principles: it should never store any sensitive information in a way it can be obtained without the container password only known by the user, and it should never store the container key or password in a way they can be obtained.
- The communication between the client and the server should only happen via an encrypted and authenticated channel. (As the profile itself is encrypted with authenticated encryption, and other communication cannot contain any sensitive information about the user, it is not an absolute must, but an additional layer of security.)
- The login process should be implemented in a way that the user's login password does not leave the client machine.

Architecture

The application has a classic client-server architecture:

- The server stores and provides encrypted data to the user.
- The client verifies and decrypts the data, modifies the data and sends it back to the server.

The server exposes a REST API for the client, the client connects to the server via HTTPS requests. Except for registration, login and password reset requests, all requests must be authenticated with a SessionId header, containing a valid session key obtained during the login process - the server authenticates and authorizes user requests by this session key.

Cryptographic protocols

The used cryptographic protocols are the following:

- Secure Remote Password (SRP) protocol, version 6a for the login process. This is to provide a secure password-authenticated key agreement between the client and the server, and to prevent the login password leaving the machine in any form.
- Advanced Encryption Standard (AES) “protocol” with 256-bit key size used in Galois-Counter Mode (GCM). This is to provide a strong *authenticated* encryption for the user’s profile.
- Transport Layer Security (TLS) protocol, version 1.2 — implicitly, for the HTTPS.
- Scrypt “protocol” for the derivation of the container key from the container password.

Since the feature set of the application does not contain any share-like features involving multiple parties accessing the same encrypted data, we do not use any asymmetric cryptography in the application (besides the SRP’s implicit Diffie-Hellman-like PKI-based key exchange and the key exchange of TLS).

Key exchange and key management

Key exchange between the client and the server is implicitly managed by the TLS protocol. Besides correct configuration (modern, secure cipher suites and suitable keys), it is to be supported by the server and the client, we do not have to implement it.

At the login, password key exchange between the client and the server is managed by the SRP protocol. We will use a cryptographic library implementation of SRP, we will not implement it ourselves.

The user’s profile is encrypted with a container key derived from the user’s container password with scrypt. We will use a library implementation of scrypt. The salt for the password derivation is stored on the server, and can be downloaded with the user profile in possession of a valid session. The container itself is encrypted with AES256-GCM. The initialization vector is stored on the server and is sent to the client with the encrypted user profile. The container password and the container key is never stored in a persistent storage neither on the server, nor on the client’s machine.

The user profile is not stored by the client for longer amount of time than it is explicitly needed for a transaction. This means that every time the user wants to operate with something in the profile (view a password, add a new password, delete a password from the profile), the following happens:

1. The profile is downloaded from the server with the salt and IV used at the encryption.
2. The profile is decrypted by the client using the container password, salt and IV.
3. The necessary operations are performed on the profile (e.g. displaying a password on the GUI — copying the one readable password to be displayed to a GUI variable or inserting a new password into the profile).
4. If the profile was changed, it is re-encrypted with a fresh salt and IV, and uploaded to the server together with the salt and IV. If the profile was not changed, its encrypted and decrypted versions are simply deleted from the client.

The above solution comes with generating some network traffic. But, considering that the uploaded user profile is an (encrypted, serialized) JSON object containing only password entries, it will not grow to unbearable sizes. A power user having — very much — 100 password entries (K, V) in their profile with an average K size of 20 characters and an average V size (password length) of 50 characters, will only generate ~16 KB of network traffic per the above cycle of 4 operations (if the profile was changed).

Every time the user profile is encrypted, serialized and uploaded to the server, it is encrypted with a fresh container key and initialization vector. The fresh container key is derived from the same container key password, but a freshly generated container key salt. At the upload, besides the user profile, the freshly generated container key salt and initialization vector is uploaded to the server as well. This is to ensure that next time the client wants to open the user profile, it is able to do that.

To avoid that the user is needed to type in the container password every time an operation happens on the profile (e.g. a user wants to view a password, or wants to add a new one, or the profile is encrypted to be uploaded to the server), the container password needs to be stored by the client. The client stores the container password locally, in-memory (non-persistently), encrypted with AES256-GCM by a key derived with scrypt from the **container password storage key** acquired from the server with a valid session.

The methodology of securely storing the container password locally is the following:

1. After login, the client requests a container password storage key from the server, which is a freshly generated random string.

2. When the user is first requested to type in their container password after the login to open their user profile, the container password is encrypted with the container password storage key, a fresh initialization vector and salt, and the encrypted container password is stored locally (together with the IV and the salt). The unencrypted container password is never stored.
3. After the encryption of the container password finished and the encrypted password is saved locally with the IV and the salt, the container password storage key is deleted from the client.
4. After this, when an operation needs the container password — e.g. the user wants to view a password (profile decryption) or adds a new password to the app (modified profile encryption and upload to the server) —, the following happens:
    a. the client requests the container password storage key from the server,
    b. derives the necessary key from the container password storage key and the locally stored IV and salt,
    c. after the derivation is completed, the client deletes its locally stored container password storage key (so when it needs the container password again, it needs to request the container password storage key from the server again to be able to decrypt it),
    d. opens the locally stored, encrypted container password, thus acquires the unencrypted container password in readable form,
    e. decrypts the user profile with the container password, performs the operation requested by the user to the user profile,
    f. encrypts the user profile with the unencrypted container password, a fresh salt and a fresh IV,
    g. uploads the encrypted user profile, the salt and the IV to the server,
    h. deletes the locally stored unencrypted container password (so when it needs the container password again, it needs to start with point a. again).

This whole process is necessary to avoid that the container password sits in the memory in unencrypted form, and can be easily read out by a malware on the client's machine.

The container password storage key is invalidated by the server every 10 minutes (or if forceFresh = true, see below): in this case, the server returns a freshly generated container password storage key. This means that the user will need to type in their container password every 10 minutes, as the client will not be able to decrypt the encrypted container password. Considering the use-cases of a password manager app, this is a reasonable time to perform user operations, and will not cause inconvenience.

If the client cannot decrypt the encrypted container password with the server-provided container password storage key and the locally stored salt and IV, that is not a fatal error: in this case, the container password is needed to be typed in again by the user, and it needs to be stored locally with the same logic as described above.

Brief security analysis

The aforementioned security requirements are fulfilled by various design choices and cryptographic solutions, which will be discussed in this chapter by detail on a per requirement basis.

User's profile

"The application should never persist or transfer the user's profile in unencrypted, unserialized form."

As the information contained in the user's profile is encrypted on the client, it does not leave the client machine in unencrypted form. Moreover, the data between the client and the server is sent through and encrypted HTTPS channel. The client does not store any user profile data in a persistent way, neither transfers it unencrypted. The server, which stores user profile data, receives all data encrypted, without any information about the password or the derived key, so there is no way to decode the data using the information stored on the server only.

Zero-knowledge principle

"The application should work on zero-knowledge principles: it should never store any sensitive information in a way that can be obtained without the container password only known by the user, and it should never store the container key or password in a way they can be obtained."

All user profile data on the server is encrypted using the AES-256 standard, as discussed in the Cryptographic protocols section. This is a widely used cryptographic standard, which cannot be cracked in a foreseeable amount of time with today's computing capacity. Without the container password only known by the user, the profile cannot be decrypted. The container key is never stored neither on the client nor on the server, and a fresh one is generated every time before the profile is encrypted, serialized and uploaded to the server. The container password is only stored on the client, and only in encrypted form.

Communication

“The communication between the client and the server should only happen via an encrypted and authenticated channel.”

The communication between the server and the client happens through a secure TLS 1.2 channel. TLS is a widely used protocol over the web, which ensures confidentiality of the communication. Version 1.2 is the latest, secure version of it, which is widely adopted (as TLS 1.3 was just released, it is not widely adopted).

Login

“The login process should be implemented in a way that the user’s login password does not leave the client machine.”

During the login process, SRP protocol is used that ensures, that the user password does not leave the client.

Conceptual endpoint description details

The communication between the client and the server will be achived using a REST API.

/register

- role: Registering a user with an email address and a login password.
- URI: /register
- method: POST
- query parameters: -
- request body parameters:
    - email: string - the user’s email address
    - salt: string<hex> - the binary salt used for generating the verifier
    - verifier: string<hex> - the SRP verifier (number)
- headers: -
- response: -
- exceptions:
    - 400/BadInput - invalid request body format
    - 400/EmailNotValid - not a valid email address
    - 400/SaltNotValid - the salt is not a base64 string
    - 400/VerifierNotValid - the verifier is not a hex string
    - 403/UserAlreadyExists - there is a user with this email
    - 500/UnexpectedException - internal server error

/validate

- role: Validate the user's email address with an email sent to the address
- URI: /validate?email=<email>&id=<id>
- method: GET
- query parameters:
    - email: string - the user's email address
    - id: string - the email validation id
- headers: -
- response: -
- exceptions:
    - 400/BadInput - invalid or missing query parameters
    - 400/EmailNotValid - not a valid email address
    - 403/UserNotExists - no user exists with this email
    - 403/UserInWrongState - the user already registered
    - 403/IdNotValid - the provided id is not valid
    - 500/UnexpectedException - internal server error

/challenge

- role: Starting the SRP login process for a user with an email address and a login password.
- URI: /challenge
- method: POST
- query parameters: -
- request body parameters:
    - email: string - the user's email address
    - clientChallenge: string<hex> - the challenge (number) created by the client
- headers: -
- response:
    - salt: string<hex> - the binary salt used for generating the verifier
    - serverChallenge: string<hex> - the challenge (number) created by the server
- exceptions:
    - 400/BadInput - invalid request body format
    - 400/EmailNotValid - not a valid email address
    - 400/ChallengeNotValid - challenge is not a hex string
    - 403/UserNotExists - no user exists with this email
    - 500/UnexpectedException - internal server error

/authenticate

- role: Finishing the SRP login process for a user with an email address and a login password.
- URI: /authenticate
- method: POST
- query parameters: -
- request body parameters:
    - email: string - the user's email address
    - clientProof: string<hex> - the client's calculated password proof (number)
- headers: -
- response:
    - salt: string<hex> - the salt used for generating the verifier
    - serverProof: string<hex> - the server's calculated password proof (number)
    - sessionId: string - the server-generated sessionId, which the user will use to authorize themselves after the login process
- exceptions:
    - 400/BadInput - invalid request body format
    - 400/EmailNotValid - not a valid email address
    - 400/ClientProofNotValid - the client proof is not a hex string
    - 403/ClientProofIncorrect - the client proof is incorrect (wrong password)
    - 403/UserNotExists - no user exists with this email
    - 500/UnexpectedException - internal server error

/logout

- role: Logging out a user, destroying its session.
- URI: /logout
- method: GET
- query parameters: -
- headers: Authorization: LoginSession <sessionId>
- response: -
- exceptions:
    - 403/InvalidAuthorization - invalid authorization header type
    - 403/InvalidAuthorization - invalid authorization header value
    - 403/InvalidSession - invalid session id
    - 500/UnexpectedException - internal server error

/downloadUserProfile

- role: Downloading the encrypted user profile, the container key salt and the initialization vector.
- URI: /downloadUserProfile
- method: GET
- query parameters: -
- headers: Authorization: LoginSession <sessionId>
- response:
    - encryptedUserProfile: string<base64> - the encrypted user profile
    - containerKeySalt: string<base64> - the binary container key salt
    - initializationVector: string<base64> - the IV used for encrypting the profile
- exceptions:
    - 403/InvalidAuthorization - invalid authorization header type
    - 403/InvalidAuthorization - invalid authorization header value
    - 403/InvalidSession - invalid session id
    - 404/UserProfileNotFound - user profile is not found
    - 500/UnexpectedException - internal server error

/uploadUserProfile

- role: Uploading the encrypted user profile, the container key salt and the initialization vector.
- URI: /uploadUserProfile
- method: PUT
- query parameters: -
- request body parameters:
    - encryptedUserProfile: string<base64> - the encrypted userProfile
    - containerKeySalt: string<base64> - the binary container key salt
    - initializationVector: string<base64> - the IV used for encrypting the profile
- headers: Authorization: LoginSession <sessionId>
- response: -
- exceptions:
    - 400/BadInput - invalid or missing query parameters
    - 400/BadInput - invalid request body format
    - 403/InvalidAuthorization - invalid authorization header type
    - 403/InvalidAuthorization - invalid authorization header value
    - 403/InvalidSession - invalid session id
    - 412/ContainerKeySaltNotFresh - the container key salt is the same as the previous one
    - 412/InitializationVectorNotFresh - the IV is the same as the previous one
    - 500/UnexpectedException - internal server error

/getStorageKey

- role: Download the container password storage key.
- URI: /getStorageKey?forceFresh=<'true' | 'false'>
- method: GET
- query parameters:
    - forceFresh: boolean - if true, the server generates a new key (even if the previous was still valid), and deletes the previous one
- headers: Authorization: LoginSession <sessionId>
- response:
    - containerPasswordStorageKey: string<base64> - the binary container password storage key
- exceptions:
    - 403/InvalidAuthorization - invalid authorization header type
    - 403/InvalidAuthorization - invalid authorization header value
    - 403/InvalidSession - invalid session id
    - 500/UnexpectedException - internal server error

/changePassword

- role: Change the login password of a user.
- URI: /changePassword
- method: POST
- query parameters: -
- request body parameters:
    - salt: string<hex> - the binary salt used for generating the new verifier
    - verifier: string<hex> - the new SRP verifier (number)
- headers: Authorization: LoginSession <sessionId>
- response: -
- exceptions:
    - 400/BadInput - invalid request body format
    - 400/SaltNotValid - the salt is not a base64 string
    - 400/VerifierNotValid - the verifier is not a hex string
    - 403/InvalidAuthorization - invalid authorization header type
    - 403/InvalidAuthorization - invalid authorization header value
    - 403/InvalidSession - invalid session id
    - 500/UnexpectedException - internal server error

/resetPasswordRequest

- role: Send a password reset request to the user's email address with a link like: https://protopass-frontend.azurewebsites.net/reset-password?id=<id>&email=<email>
- URI: /resetPasswordRequest
- method: GET
- query parameters:
    - email: string - the email address of the user
- headers: -
- response: -
- exceptions:
    - 400/BadInput - invalid or missing query parameters
    - 400/EmailNotValid - not a valid email address
    - 403/UserNotExists - user is not found
    - 500/UnexpectedException - internal server error

/resetPassword

- role: Reset the password by an password reset id received in email.
- URI: /resetPassword
- method: POST
- query parameters:
    - id: string - the password reset id
- request body parameters:
    - salt: string<hex> - the binary salt used for generating the new verifier
    - verifier: string<hex> - the new SRP verifier (number)
- headers: -
- response: -
- exceptions:
    - 400/BadInput - invalid or missing query parameters
    - 400/BadInput - invalid request body format
    - 400/SaltNotValid - the salt is not a base64 string
    - 400/VerifierNotValid - the verifier is not a hex string
    - 403/InvalidId - the provided password reset id does not exist
    - 500/UnexpectedException - internal server error

The proposed schema for the user profile

The user profile in unencrypted form is a JSON object. The following is an example of a valid user profile. The profile does not contain any user-specific info. Even is somebody unauthorized acquires an open, unencrypted user profile object, they will not be able to use the passwords, since they do not know, which username/email address owns that password. (Except if the user types in the account name as a key.)

```
{
  "PasswordEntries":
  {
    "facebook": "TitkosFacebookJelszavam1994",
    "twitter (az accom emailje: luczsoma@gmail.com)": "Twitter9"
  }
}
```