Salt Synchronization Service

Bachelor-Thesis von Roman Fojtik

Tag der Einreichung:

1. Gutachten: Prof. Dr. Johannes Buchmann
2. Gutachten: Moritz Horsch

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Theoretische Informatik
Kryptographie und Computeralgebra

Salt Synchronization Service

Vorgelegte Bachelor-Thesis von Roman Fojtik

1. Gutachten: Prof. Dr. Johannes Buchmann
2. Gutachten: Moritz Horsch

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den September 30, 2016

(R. Fojtik)

Abstract

The problem of managing individual and strong passwords for different user accounts services on different devices can be solved by the usage of password managers. The common approach of passwords managers is to store the passwords in an encrypted database on servers on the Internet. After a security breach at a server such a database is vulnerable to offline brute-force attacks which may result in full disclose of the passwords.

An alternative approach for synchronizing passwords is introduced by the PasswordLess Password Synchronization (PALPAS) scheme. Its first key feature is that the used synchronization server does not store any information about the passwords. This is done by generating the passwords from two sources, a seed and a salt. The seed is a randomly generated secret, stored on the devices while the salt is a random value, retrieved from the central server called Salt Synchronization Service (SSS). The salt is statistically independent of the password itself, which makes it non-critical to be stored at the server. And since the password generation process is deterministic, the same password is computed on each device. The second key feature is that no passwords are used for authentication, instead the client authentication at the server is based on public key cryptography.

This thesis presents the first full concept of the Salt Synchronization Service. The components for authentication, business logic, and data storage with an enhanced model for stored data are described. Furthermore the thesis deals with its detailed functionality and implementation. The thesis presents a reference implementation with its components. It also describes the internal communication between components and external communication between participants of PALPAS and the SSS. It is concluded that the presented solution meets all security, privacy, and functionality requirements. Thus it can be used to safely distribute salts and enable users to synchronize passwords with the PALPAS tool.

Zusammenfassung

Passwort-Manager bewältigen die Herausforderung, individuelle und starke Passwörter für verschiedene Nutzerkonten auf mehreren Geräten zu verwalten. Dabei ist ihre übliche Herangehensweise, die Passwörter in einer verschlüsselten Datenbank auf einem Server im Internet zu speichern. Allerdings kann durch das Ausnutzen einer geeigneten Sicherheitslücke und anschließender Entwendung der Datenbank, offline ein Brute-Force-Angriff auf die besagte verschlüsselte Datenbank ausgeführt werden. Dies kann dazu führen, dass die gespeicherten Passwörter offengelegt werden.

Das Schema von PasswordLess Password Synchronization (PALPAS) bietet einen alternativen Ansatz zum Synchronisieren von Passwörtern an. Sein erstes Hauptmerkmal ist, dass der verwendete Synchronisierungsserver keine Informationen über die Passwörter speichert. Dies wird durch das Generieren der Passwörter aus zwei Quellen, einem Seed und einem Salt, ermöglicht. Der Seed ist ein zufällig gewähltes Geheimnis, gespeichert auf dem Gerät, während der Salt ein zufälliger Wert ist, welcher von dem Synchronisierungsserver, im Folgenden als Salt Synchronization Service (SSS) bezeichnet, bezogen wird. Der Salt ist stochastisch unabhängig vom Passwort selbst, sodass kein Risiko dabei besteht, ihn auf dem Server zu lagern. Und dadurch, dass die Passwortgenerierung deterministisch verläuft, kann auf jedem Gerät dasselbe Passwort berechnet werden. Das zweite Hauptmerkmal besteht darin, dass keine Passwörter für die Authentifizierung verwendet werden. Stattdessen basiert das Authentifizierungsverfahren beim Server auf einem asymmetrischen Verschlüsselungsverfahren und Public-Key-Kryptographie.

Diese Bachelorarbeit stellt das erste, vollständige Konzept des Salt Synchronization Services vor. Es werden die Komponenten für Authentisierung, Geschäftslogik und Datenspeicher mit einem erweiterten Modell für gespeicherte Daten beschrieben. Darüber hinaus beschäftigt sich diese Arbeit mit der detaillierten Funktionalität des SSS und seiner Implementierung. Dafür wird im Laufe dieser Arbeit eine Referenzimplementierung präsentiert. Des Weiteren wird sowohl auf die interne Kommunikation zwischen den Komponenten eingegangen, als auch auf die äußere, zwischen den Akteuren von PALPAS und dem SSS selbst. Es wird der Schluss gezogen, dass die vorgestellte Lösung allen gestellten Anforderungen an Sicherheit, Privatsphäre und Funktionalität genügt und somit eingesetzt werden kann um mit PALPAS auf einem sicheren Weg Salts für Anwender zu synchronisieren.

Contents

List of Figures

1 Introduction

Internet services like online shopping, email, social networking sites, and video sharing sites accompany people all around the world on a daily basis. These services require users to create accounts and mainly use passwords to authenticate them. The concept of password-based authentication is easy to implement, device independent, and comprehensible for users. However, the security of the accounts highly depends on the users' security conscious behavior. In particular, to choose a strong and individual passwords for each account.

Moreover, due to the huge amount of accounts it is impossible for users to memorize all these different passwords. One solution to cope with the problem is to store the passwords locally. Nonetheless, this leads to further problems. First, this requires an additional protection mechanism to ensure that only legitimate users can access the data. In addition the availability of the passwords must be ensured. Locally stored passwords are not accessible from other user devices, e.g., tablets and smartphones. Since passwords can change, the local data does not only have to be moved once but kept synchronized between the devices to enable the user to access their passwords at any time and device. The most convenient way to synchronize data is through a synchronization server over the Internet. Most existing approaches for secure password storage, password synchronization and generation of individual strong passwords have severe drawbacks.

The usual way to implement password synchronization is by encrypting passwords locally with a key, derived from a user-chosen (master) password, and storing them on servers on the Internet. A security risk arises when adversaries gain access to the synchronization servers, copy the encrypted data and perform offline attacks (cf. [15], [7]). More precisely, a brute force attack on the encryption password. Therefore a synchronization scheme should be chosen that is not vulnerable to such attacks and user friendly at the same time. Furthermore the authentication to the synchronization servers is often based on passwords (same master password) to enable users to access their password vaults online. This leads to various possible attacks based on web vulnerabilities and phishing attacks (cf. [4], [30]).

As an alternative and to solve the mentioned issues, there is PasswordLess Password Synchronization (PALPAS) [18]. PALPAS is a password managing scheme for creating strong, service-specific passwords. It creates secure passwords by generating them out of a high entropy secret, called the seed, and a random salt value for each service. The seed is only stored at the local devices while the salts are stored at the server. The salts are statistically independent of the passwords, which means they do not reveal any information about them. Since no passwords are stored at the server, a security breach by an adversary, followed by an offline brute-force attack, can be considered non-critical. Furthermore, to eliminate the vulnerability to phishing attacks, PALPAS uses public key cryptography instead of passwords to authenticate user devices at the synchronization server.

This thesis presents the full concept of the Salt Synchronization Service (SSS). As indicated, its main responsibility lies in distributing salts between the user devices, while granting only their legitimate owners access to the data.

The thesis starts with summarizing the concept of PALPAS in Section 1.1, while focusing on the password creation and in Section 1.2, an overview of related work regarding password managers is given. Chapter 2 deals with the theory of the SSS by defining requirements, an attacker model and describing the components: API, data storage, access control and business logic. Chapter 3 presents an reference implementation and describes the details of the mentioned components. The API is specified and the communication between business logic and data storage, as well as the communication between the SSS and other PALPAS participants is illustrated. The thesis closes with a conclusion in Chapter 4 and outlines possible future extensions.

1.1 PALPAS - PasswordLess Password Synchronization

PALPAS [18] is a password managing scheme for creating service-specific, strong passwords and synchronizing them between devices with the help of a central server (the SSS). The key feature of PALPAS is that it does not store any users' passwords neither on the local device nor at the SSS. Further it does not use passwords to authenticate users in a privacy-preserving way at the SSS.

Since PALPAS does not store the users' passwords, every time a user wants to login at a service, the password is computed using three components: salt, seed, password policy. The resulting strong password is always completely random, unique and complies with the requirements of the service. As shown in Figure 1.1 a password is computed in two steps. First, a (cryptographically secure) pseudorandom generator (PRG) generates a pseudorandom value based on a seed and salt value. Second, a Password Generator (PG) derives the password from the pseudorandom value and ensures that it complies with a given password policy. The password policy specifies the password requirements of the service. Since the PRG and the PG are deterministic, the same password is generated using the same seed, salt and password policy. The password policies can be realized using the Password Requirement Markup Language [19].

The seed is a randomly generated, common secret shared by all user devices. PALPAS generates it at startup and it has to be transferred by the user to his or her additional devices once. The seed is encrypted on the devices using a secret derived from the user's chosen master pass-

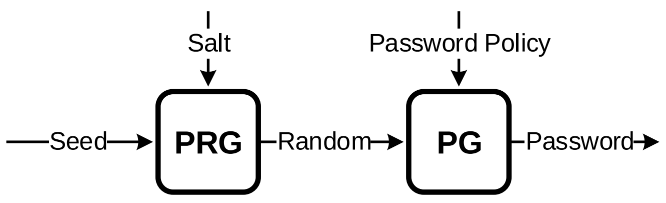

Figure 1.1: PALPAL password generation [18]

word. Notice that this secret and data encrypted using this secret do not leave the device, unless a new device has to be registered. Most importantly, they are never stored at the SSS.

The salt is a randomly generated, unique value in order to create different passwords for each service. It is generated by PALPAS, completely by random, which makes it statistically independent of the password or any privacy related information.

The password policy defines the password requirements of a service specific. It is provided by a mechanism of PALPAS and specifies for example the minimum and maximum password length and the allowed character sets.

PALPAS distributes the salts, to compute passwords on each user device, with the help of the SSS. In the following chapters it is described how the SSS operates, how it is set up and how the components function, i.e. data storage and access control. The communication works between the inner components is described as well as an API specification given for communication between user devices and the SSS.

1.2 Related Work

To implement synchronization between devices, most existing password managers encrypt passwords locally and then store them on central servers on the Internet. They derive the encryption key from a user-chosen master password. Popular examples are LasPass [6], KeePass [28], 1Password [20] and the password managers in browsers like Chrome and Firefox [25]. This approach bears the risk that in case of an security breach, an adversary can steal the encrypted data and perform an offline brute-force attack. PALPAS on the other hand does not rely only on encryption to ensure the confidentiality of passwords, because only the salts are stored at the SSS. To obtain the passwords, the adversary would still need to steal and encrypt the seed, stored in the user device.

Other methods to store passwords at central servers are hash-based. For example the PwdHash [29] scheme creates different passwords by hashing the user-chosen master password and the domain name of the service. The obvious downside is that an adversary already knows the domain names and after stealing one password he can perform a brute-force attack to retrieve the master password. Subsequently all remaining passwords can be generated. To improve the idea, the authors of Password Multiplier [16] impede brute-force attacks by strengthening the master password, but it does not solve the issue. Furthermore they propose to use additional, user chosen data to be included in the hashing process. Remembering additional information for each service means a downgrade in usability which is less attractive for the users.

In contrast to password managers that administer users' passwords, there is a different way to simplify authentication for a user at multiple services. Single-sign-on (SSO) like Facebook Connect [13] allows for a single login that enables users to seamlessly login at multiple systems or services. The obvious benefit is that a user has to authenticate himself manually only at the central authentication service and saves time by not having to login at the other services. The downside of SSO is that in case of identity theft, an adversary is able to impersonate the user at all participating services. An attacker could do so by utilizing phishing attacks [33]. In addition, SSO leads also to severe privacy issues [12], since the SSO identity provider can log and trace users' history of when and where someone performed a login. All in all, SSO bypasses the issue of managing multiple passwords and introduces new problems like the privacy issue.

2 Salt Synchronization Service

The Salt Synchronization Service (SSS) is the central synchronization service in the PALPAS system. It stores the salt values and synchronizes them between the different user devices. Furthermore it stores the usernames of the users' online accounts. Its key feature is that it neither store any users' passwords nor use passwords to authenticate users in a privacy-preserving way. In this chapter, we describe the architecture, components, and functionality of the SSS.

In Section 2.1 we define the attacker model. We describe the participants of the PALPAS system, such as the SSS, the attacker, and users clients, and the capabilities of the attacker.

In Section 2.2 we describe functional and non-functional requirements for the SSS. The functional requirements describe the general functionality of the SSS and the non-functional describe further requirements such as scalability, robustness, and extensibility. The implementation of these requirements is presented in Chapter 3.

In Section 2.3 we describe the components and functionalities of the SSS. First, we describe each component, as well as its function and purpose in the SSS. Second, we explain how these components communicate with each other.

2.1 Attacker model

The main actors in our environment are the SSS and the devices of a user. In general, the devices communicate directly with the SSS via a network connection, but not with each other. The SSS communicates with the devices by responding to incoming requests, this means that communication is always initiated by a device.

In this environment model, we assume an adversary that has the main goal of obtaining other users' usernames and passwords. To compute a password, first the attacker needs to steal the seed from a device. The security of the device lies primarily in the hands of the user. He or she have to ensure that no unauthorized person has access to the device. Furthermore, PALPAS supports the security of the device by encrypting the seed, so that an adversary has to decrypt it after stealing it from the device. Then he needs the salt, which is stored at the SSS and communicated with the client. To attack the network or the connection between an user device and the SSS itself, the attacker has the following capabilities:

- The attacker includes the rights of an user device in the system, i.e. registering and sending messages to the SSS.

- The attacker can overhear messages between any device and the SSS.
- The attacker can intercept messages between any device and the SSS.
- The attacker can modify messages between any device and the SSS.

By intercepting messages between the user device and the SSS or by flooding the SSS with messages, the attacker can deny service at the SSS. The first case can be avoided by using a different Internet entry point, rather than a potentially hijacked wifi hotspot at the local coffee shop. Second, a denial-of-service attack can be anticipated by redundancy of hardware or by also running multiple instances of the SSS on different locations. Moreover, there is more research on the topic of denial-of-service attacks, on their detection and their defense mechanisms [24], [34].

Furthermore we assume that an attacker might perform a security breach at the SSS but only temporarily. In that time he can e.g. read or destroy the data storage where the users' salts are stored. Hence, we do not expect an adversary that hijacks the SSS and operates it for a longer period of time.

2.2 Requirements

The listed requirements in this section are divided into functional and non-functional requirements. The functional requirements make statements about the functionality and non-functional requirements define further characteristics of the SSS.

2.2.1 Functional requirements

The functional requirements for the SSS can be mostly derived from the PALPAS paper [18].

FR 1: Register user

The user must be able to create an account at the SSS.

FR 2: Access restriction

The access to the data stored on the SSS must be restricted to the legitimated users. Only the owner of the salt values stored on the SSS is allowed to access and modify the salts.

FR 3: Add user device

The user must be able to register additional devices with his account at the SSS. The added device has the same privileges to operate on the user account.

FR 4: Store salt

The user must be able to store salts at the SSS, by sending them individually. The stored salts must be bound to the user account and to a respective service.

FR 5: Retrieve salt

The user must be able to retrieve salts with a device bound to his or her corresponding user account. The SSS must provide functionality to retrieve all salts bound to the user account or all salts bound to a specific service, or to retrieve them individually.

FR 6: Update salt

The user must be able to update a specific salt at the SSS.

FR 7: Delete data

The user must be able to delete his or her data from the SSS. The SSS must provide functionality to delete this data individually and also to delete the user account with all corresponding data at once.

2.2.2 Non-functional requirements

Unlike functional requirements which describe what a system should be capable of, non-functional requirements make a statement about the quality of the system.

NFR 1: Confidentiality

The SSS must ensure that users can only access data that are bound to their user account. Furthermore, the SSS should not have any knowledge about the content of the stored data, i.e. salt value, username and service domain name.

NFR 2: Availability

The SSS must be available at all times to provide salts for the users. Since a salt must be requested first to compute a password, availability of the SSS is critical for the users, otherwise no passwords can be generated.

NFR 3: Extensibility

The SSS must be easily extensible to allow significant extension of its capabilities without major rewriting of existing code or changes in its basic architecture. There is an outlook for potential changes and extended functionality for the SSS so it must be extensible, because otherwise addition of functionality would lead to an increased risk of errors and security flaws.

NFR 4: Robustness

The SSS must not crash due to invalid or erroneous inputs or other internal errors. The SSS must be able to handle errors during execution and recognize and handle erroneous inputs.

NFR 5: Scalability

The SSS must be scalable to efficiently manage a large amount of user accounts. This means that the amount of stored data at the SSS must not affect performance of the communication to the clients.

2.3 Components

The SSS consists of four main components (cf. Figure 2.1). The Application Programming Interface (API) consumes the incoming client request and the access control component checks the request for legitimacy. Then the business logic carries out the computation. The data storage is used to store and retrieve the users' data.

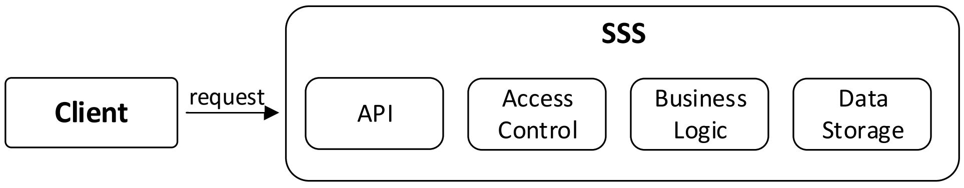

Figure 2.1: Component overview

2.3.1 API

The task of the server API is to provide the clients, i.e. user devices with a set of functions to interact with the SSS. These functions enable the clients to perform actions, which are described previously among the functional requirements (**FR 1** and **FR 3-7**) in Section 2.2. Furthermore, in Section 3.2 we explain how the API supports the scalability of the service.

2.3.2 Access Control

Some requests to the API, like requesting salts or registering a new device, have to be protected to ensure the legitimacy of the user. Therefore, to control access to these server requests, the user devices have to be authenticated.

Usually, password-based authentication mechanisms are popular approaches for user authentication at synchronization servers, but they also bear the risk of phishing attacks. The approach chosen by PALPAS is to use public key based authentication. To realize mutual authentication, server and client authentication are needed. Server authentication is reached easily, since the client only needs knowledge of the server's public key. Reaching client authentication is described in the following. In case the user has no account at the SSS, an authentication key pair, consisting of an independent secret K_{Auth} and a corresponding public key, is uniquely created on his or her device. This key pair is used to create the new account. At the creation of the user account, the SSS stores the corresponding public key of the client. In case the user wants to register an additional device to his account, he or she uses a registered device to request an authentication token T_{Auth} from the SSS. The new device generates a key pair as well and uses the token for authentication and assimilation of its own (and new) public key to the user account.

Hence the public key can be used by the SSS to encrypt further communication towards the client. Thus the premise of mutual authentication is fulfilled. As described in the PALPAS paper, the benefit of having authentication key pairs for each user device is fine-grained revocation. The user can simply delete the public key of its registered device from the SSS and therefore revoke all access rights to the user account from that device.

By controlling access at the SSS the functional requirement **FR 2** is fulfilled (implementation in Section 3.3).

2.3.3 Business Logic

The business logic of the SSS is responsible for executing API requests, i.e. to validate input data, to create, retrieve and store data to the storage, and to prepare data for the response.

The logic is separated into multiple units, one for each possible API call. After a client request reaches the API and passes the authentication, the business logic checks the input data. For instance this can be the authentication token when registering an additional device at the SSS. Then the credentials (certificate) for the new device are created and corresponding data (device name, certificate, device identifier) are written to the data storage. Finally, the response data are accumulated and send back to the client.

In support of the API, which provides corresponding API calls, the business logic implements the functional requirements **FR 1** and **FR 3-7**, application flows are illustrated in Section 3.5 and more detailed diagrams are shown in Section 3.4.2.

2.3.4 Data storage

The SSS stores the following data: salt blobs, identifiers, domain names, tokens, timestamps, certificates .

Most importantly the SSS stores salts. But unlike described in the PALPAS paper, the SSS stores a salt blob instead of just the salt value. A salt blob is an encrypted collection that contains the salt value with a timestamp, password policy version (PPV) of the service, domain and username for the specific service (cf. Figure 2.2).

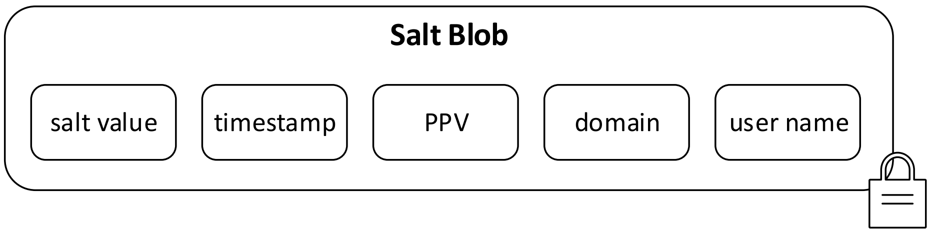

Figure 2.2: Salt blob composition

Thus the blob replaces the triple of salt, identifier and user data as described in the paper. The client encrypts the salt blob which protects the users' privacy, because the SSS cannot read the values of domain and user names. Moreover it prevents a possible seized SSS by an adversary from misleading clients by sending them illegitimate salt values.

Every salt blob has an unique identifier assigned to it, thus making it possible to retrieve a specific salt blob by providing an identifier in an API request. Other resources, like users and devices have an identifier, too. The identifiers are randomly generated at the SSS and therefore do not need further protection.

Each salt blob is bound to a service, which has the domain name as its identifier. Users can have multiple accounts at one service, and by providing the domain name in a request, the SSS returns the salt blobs corresponding to this identifier. The domain name is encrypted by the client before sent to the SSS in order to protect the users privacy. Hence, an attacker who stole the information of the data storage cannot draw any conclusions about the services, the user has accounts at.

Furthermore the SSS stores for each registered device a specific certificate for access control and to ensure a mutually authenticated communication. Since a certificate contains only the public key of the user device and it includes a fingerprint, which is a hash of itself, further protection is not required before storing it at the SSS.

To register additional devices to an user account, an authentication token is needed. The SSS generates this random, temporal token and stores it together with its respective timestamp in plaintext in the data storage. It then has to used for registration by the new user device in a given time frame. Subsequently the token is deleted after one use.

In conclusion, due to access control (Section 2.3.2) and due to the client side encryption of privacy sensitive data, the confidentiality requirement (**NFR 1**) can be fulfilled.

3 Implementation

In this chapter we present an exemplary implementation of the SSS based on the concept presented in Chapter 2. The implementation was done in Java and provides a RESTful Web Service [1] which is based on the Jersey 2 framework [9]. Data types are specified in XML Schemas and used through the JAXB framework [8]. The data storage was realized with a traditional MySQL database [31] and cryptographic components are based on Bouncy Castle [27].

In Section 3.1 the implementation overview is given and the project setup is described. In Section 3.2 the Application Programming Interface (API) of the SSS is presented. We describe the commands and messages for the communication with the SSS, the content of the messages, and status as well as error codes. In Section 3.5 we present exemplary application flows for all interactions with the SSS.

3.1 Overview

The Java implementation of the SSS is divided into two sub-projects, namely common and service. As illustrated in figure 3.1 the common project contains only the XSD schemes which consist of the specification for the resource response data classes. In detail, the XSD specifies the data structures for users, salts, devices, tokens, and services. The XSD schemes are compiled into Java classes using JAXB. The advantage is that in the client development the XSD schemes can be reused. The incoming responses from the SSS are then transformed into Java objects.

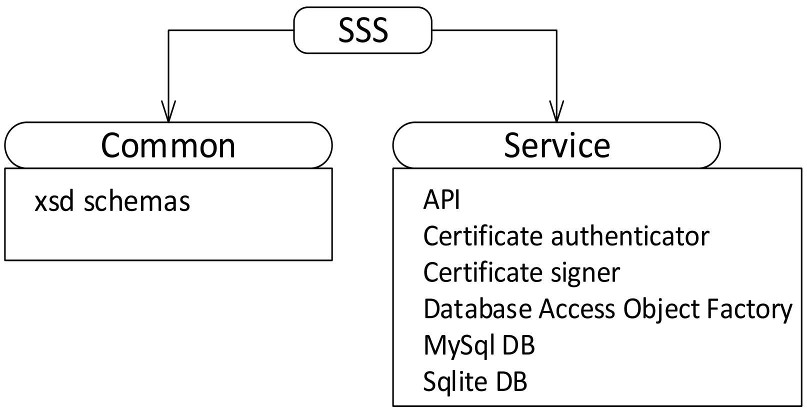

Figure 3.1: Project setup

The project service contains the actual implementation of the SSS and is divided into the following packages containing the user authentication, certificate creation and verification, the business logic for requests, exception classification, and database access logic.

Every exception that can potentially be thrown in the code is caught and categorized. This means there are packages with defined exceptions that can be thrown in the respective components. After throwing the adapted exception, it is caught a second time in the resources component, where the business logic for the API of the SSS is implemented. In case the exception is not resolved at this point, the resource sends a response to the client with a corresponding error code. Due to the thorough and uncluttered exception handling, the requirement of robustness can be achieved (**NFR 4**). Robustness is accomplished through tests, too. JUnit tests [23], which are written in support of the REST Assured framework [17], cover the different kinds of requests to the SSS. That way all other components are implicitly tested as well.

The implementation itself cannot fulfill the requirement of availability (**NFR 2**). To prevent failures due to overload or even denial-of-service attacks, hardware redundancy should be considered. Making use of a load balancer is also an option, to handle requests through multiple instances of the SSS which are wired to a common data storage. These solutions are mostly independent of the implementation but are critical to make use of, because without the SSS, the users cannot generate any passwords.

3.2 API

The API is the component which allows clients to communicate with the SSS. Clients can manage their data at the SSS by sending requests via the stateless protocol HTTP to specific URLs. It was decided to design a RESTful API that promises benefits like reusability, extensibility and scalability.

The general idea of a RESTful API is that there are specific requests for creating, deleting, and retrieving information which are tied to resources. The principles of REST are to identify resources, allow manipulation of these resources through representations, self-descriptive messages, and hypermedia as the engine of application state. Hypermedia means multimedia content in connection with hypertext [14]. Since XML and JSON are one of the most used data formats in the area of Web Services [22], we use both of them to respond with data to the client. The restriction to the use of HTTP GET, POST, PUT and DELETE requests enables simplicity and clarity of the API. By identifying resources in Table 3.1 we can manage the associated data by a single request. This makes the SSS highly scalable (**NFR 5**), since updating a salt, requires only the update of one salt blob instead of updating a whole password database as existing solutions do.

Mimetype	Description
`users`	The resource represents a user at the SSS.
`users/tokens`	The resource represents an access token retrieved by the user to register an additional device.
`users/devices`	The resource represents the list of user devices.
`users/services`	The resource represents the services stored for each user.
`users/services/salts`	The resource represents the salt blobs, which is encrypted data containing the salt, username, and PPV, stored for each user. Each salt blob is bound to a specific service.

Table 3.1: Resource Representations

In our case, we defined the services resource to be bound to the salts resource. This way we can have a request which gathers all the salt values for a specific user account at once by retrieving the service list (cf. Section 3.2.4). This was done for the sake of simplicity and to take away some logic and computation from the client.

Beside data, HTTP status codes (cf. Section 3.2.6) are sent by the SSS to identify the response type. They are most helpful when a error occurs. Due to the code it can be identified, for example whether the request was malformed or an error occurred on the server.

Extensibility

The API of the SSS (see 3.2) is extensible (**NFR 3**) due to the RESTful design. Resources can be added independently without impacting the current implementation. This gives the possibility for clients to request and manage additional data at the SSS. Since RESTful APIs handle requests for single resources at a time, it is up to the client to carry out the requests it needs. Adding information/resources would mean adding routes to the API.

3.2.1 Users

This section specifies the user-related functionality of the SSS. Mainly adding a new user and deleting an existing one.

Get User

Receive confirmation for the existence of a user by sending an HTTP GET request to `/users/[uid]`. This request can be used to check the client authentication at the SSS.

URL	`https://example.com/api/users/[uid]`
Method	GET
Requires Authentication	Yes
Returns	200 OK: The user exists. 401 Unauthorized: User authentication is required. 500 Internal Server Error: The server encountered an unexpected condition which prevented it from fulfilling the request.

Create New User

Creating a new user and registering the initial device is done by sending an HTTP POST request to `/users`. The request requires a name for the device and a Certificate Signing Request (described in 3.3) which is then signed and returned from the SSS.

URL	`https://example.com/api/users`
Method	POST
Requires Authentication	No
Request Body	Content-Type: application/x-www-form-urlencoded csr: Contains the Certificate Signing Request (CSR) for the user device. name: Contains the name of the user device.
Returns	201 Created: The user account and a device were successfully created. The user was returned. 400 Bad Request: The request was malformed, for instance the CSR was not encoded correctly. 500 Internal Server Error: The server encountered an unexpected condition which prevented it from fulfilling the request.

Request body (encoded string):

```
name=My_Laptop&csr=ABCD_CSR
```

XML Response message:

```
<?xml version="1.0"?>
<user>
   <uid>123456</uid>
   <certificate>ABCD_CRT</certificate>
</user>
```

JSON Response message:

```
{"uid":"123456", "certificate":"ABCD_CRT"}
```

Delete User

Delete a user by sending an HTTP DELETE request to `/users/[uid]`.

URL	`https://example.com/api/users/[uid]`
Method	DELETE
Requires Authentication	Yes
Returns	200 OK: The user account was removed. 401 Unauthorized: User authentication is required. 500 Internal Server Error: The server encountered an unexpected condition which prevented it from fulfilling the request.

3.2.2 Tokens

Tokens are needed to register further devices at the SSS.
Each user can request one token at a time to register his new device. A token expires as soon as a new token is requested or after a 5 minutes timeout.

Get Token

Receive authentication token by sending an HTTP GET request to `/users/[uid]/tokens`.

URL	`https://example.com/api/users/[uid]/tokens`
Method	GET
Requires Authentication	Yes
Returns	201 Created: Token was successfully created and returned. 401 Unauthorized: User authentication is required. 500 Internal Server Error: The server encountered an unexpected condition which prevented it from fulfilling the request.

XML Response message(tokens):

```xml
<?xml version="1.0"?>
<token>
   <value>1453</value>
</token>
```

JSON Response message:

```
{"value": "1453"}
```

3.2.3 Devices

Each user can register multiple devices at the SSS.
New Devices can be registered by an existing device that requests a token first. Since all devices have the same permissions for the user account, a device can be deleted remotely or it can delete itself.

Get Devices List

Get the users device list by sending an HTTP GET request to `/users/[uid]/devices`.

URL	`https://example.com/api/users/[uid]/devices`
Method	GET
Requires Authentication	Yes
Returns	200 OK: List of devices. 401 Unauthorized: User authentication is required. 500 Internal Server Error: The server encountered an unexpected condition which prevented it from fulfilling the request.

XML Response message(devices):

```
<?xml version="1.0"?>
<devices>
   <device>
      <did>1234</did>
      <name>Tablet</name>
      <certificate>certA</certificate>
   </device>
   <device>
      <did>1563</did>
      <name>My_Smartphone</name>
      <certificate>certB</certificate>
   </device>
</devices>
```

JSON Response message:

```
{"device": [{"did":"1234", "name":"Tablet", "certificate":"certA"},
{"did":"1563", "name":"My_Smartphone", "certificate":"certB"}]}
```

Get Device Information

Get device information for a specific user device by sending an HTTP GET request to `/users/[uid]/devices/[did]`.

URL	`https://example.com/api/users/[uid]/devices/[did]`
Method	GET
Requires Authentication	Yes
Returns	200 OK: Device information successfully returned. 401 Unauthorized: User authentication is required. 404 Not Found: The server has not found anything matching the Request-URI. 500 Internal Server Error: The server encountered an unexpected condition which prevented it from fulfilling the request.

XML Response message(devices):

```
<?xml version="1.0"?>
<device>
   <did>876</did>
   <name>My_Laptop</name>
   <certificate>ABCD</certificate>
</device>
```

JSON Response message:

```
{"device": {"did":"876", "name":"My_Laptop", "certificate":"ABCD"}}
```

Add User Device

An additional user device is added to SSS by HTTP POST request to `/users/[uid]/devices`. After retrieving an authentication token on a registered device, the new device sends it with a Certificate Signing Request and a device to the SSS.

URL	`https://example.com/api/users/[uid]/devices`
Method	POST
Requires Authentication	No
Request Body	Content-Type: application/x-www-form-urlencoded csr: Contains the Certificate Signing Request (CSR) of the user device. token: Contains the token for the new user device to register at the SSS. name: Contains the name of the user device.
Returns	201 Created: The additional user device was successfully added and a certificate returned. 400 Bad Request: The request was malformed, for instance the CSR was not encoded correctly or token has expired. 401 Unauthorized: User authentication is required. 500 Internal Server Error: The server encountered an unexpected condition which prevented it from fulfilling the request.

Request body (encoded string):

```
name=My_Laptop&csr=ABCD_CSR&token=1234
```

XML Response message:

```
<?xml version="1.0"?>
<device>
   <did>876</did>
   <name>My_Laptop</name>
   <certificate>ABCD</certificate>
</device>
```

JSON Response message:

```
{"device": {"did":"876", "name":"My_Laptop", "certificate":"ABCD"}}
```

Update Device

Update the device, especially the name by sending an HTTP PUT request to `/users/[uid]/devices/[did]`.

URL	`https://example.com/api/users/[uid]/devices/[did]`
Method	PUT
Requires Authentication	Yes
Request Body	Content-Type: application/x-www-form-urlencoded name: Contains the new device name.
Returns	200 OK: The device name was successfully updated. 400 Bad Request: The request was malformed and it was not understood by SSS. 401 Unauthorized: User authentication is required. 404 Not Found: The server has not found anything matching the Request-URI. 500 Internal Server Error: The server encountered an unexpected condition which prevented it from fulfilling the request.

Request body (encoded string):

```
name=My_Favorite_Laptop
```

Remove User Device

A User Device can be removed from the SSS by sending an HTTP DELETE request to `/users/[uid]/devices/[did]`.

URL	`https://example.com/api/users/[uid]/devices/[did]`
Method	DELETE
Requires Authentication	Yes
Returns	200 Removed: The device was successfully removed. 401 Unauthorized: User authentication is required. 404 Not Found: The server has not found anything matching the Request-URI. 500 Internal Server Error: The server encountered an unexpected condition which prevented it from fulfilling the request.

3.2.4 Services

The SSS stores salt blobs for each service of a user. By retrieving or deleting a service, all corresponding salt blobs are included as well.

Add Service

A service is added to SSS by HTTP POST request to `/users/[uid]/services`.

URL	`https://example.com/api/users/[uid]/services`
Method	POST
Requires Authentication	Yes
Request Body	Content-Type: application/x-www-form-urlencoded domain: Contains the encrypted domain value.
Returns	201 Created: The service was successfully added and a certificate returned. 400 Bad Request: The request was malformed and it was not understood by SSS. 401 Unauthorized: User authentication is required. 500 Internal Server Error: The server encountered an unexpected condition which prevented it from fulfilling the request.

Request body (encoded string):

```
domain=e01
```

XML Response message:

```
<?xml version="1.0"?>
<service>
   <domain>e01</domain>
   <salts/>
</service>
```

JSON Response message:

```
{"domain": "e01", "salt": []}
```

Get Services List

Gets the list of services with salt blobs for a user by sending a HTTP GET request to `/users/[uid]/services`.

URL	`https://example.com/api/users/[uid]/services`
Method	GET
Requires Authentication	Yes
Returns	200 OK: List of services successfully returned. 401 Unauthorized: User authentication is required. 500 Internal Server Error: The server encountered an unexpected condition which prevented it from fulfilling the request.

XML Response message(salts):

```
<?xml version="1.0"?>
<services>
   <service>
      <domain>e10</domain>
      <salts>
         <salt>
            <sid>s11</sid>
            <value>saltblob11</value>
         </salt>
      </salts>
   </service>
   <service>
      <domain>e20</domain>
      <salts>
         <salt>
            <sid>s21</sid>
            <value>saltblob21</value>
         </salt>
      </salts>
   </service>
</services>
```

JSON Response message:

```
{"service":[{"domain":"e10","salts":{"salt":[{"sid":"s11","value":"saltblob11"}]}},
{"domain":"e20","salts":{"salt":[{"sid":"s21","value":"saltblob21"}]}}]}
```

Remove a service

Delete a service with corresponding salt blobs from the SSS by sending an HTTP DELETE request to `/users/[uid]/services`.

URL	`https://example.com/api/users/[uid]/services`
Method	DELETE
Requires Authentication	Yes
Request Body	Content-Type: application/x-www-form-urlencoded domain: Contains the encrypted domain value.
Returns	200 OK: The service was successfully removed. 400 Bad request: The request was malformed and it was not understood by SSS. 401 Unauthorized: User authentication is required. 404 Not Found: The server has not found anything matching the Request-URI. 500 Internal Server Error: The server encountered an unexpected condition which prevented it from fulfilling the request.

Request body (encoded string):

```
domain=e10
```

3.2.5 Salts

The SSS stores a separate salt blob and an associated identifier for each service of a user. A salt blob can be updated or retrieved. The service identifier allows to request the salt blob for a particular service.

Get Salt List

Get a salt blob list for a user for a service by sending an HTTP POST request to `/users/[uid]/services/salts`.

URL	`https://example.com/api/users/[uid]/services/salts`
Method	POST
Requires Authentication	Yes
Request Body	Content-Type: application/x-www-form-urlencoded domain: Contains the encrypted domain value.
Returns	200 OK: List of salt blobs successfully returned. 401 Unauthorized: User authentication is required. 500 Internal Server Error: The server encountered an unexpected condition which prevented it from fulfilling the request.

Request body (encoded string):

```
domain=e10
```

XML Response message(salts):

```
<?xml version="1.0"?>
<salts>
   <salt>
      <sid>s12</sid>
      <value>a10</value>
   </salt>
   <salt>
      <sid>s23</sid>
      <value>a20</value>
   </salt>
   <salt>
      <sid>s34</sid>
      <value>a30</value>
   </salt>
</salts>
```

JSON Response message:

```
{"salt":[{"sid":"s12","value":"a10"},{"sid":"s23","value":"a20"},
{"sid":"s34","value":"a30"}]}
```

Get Current Salt Value

Get current salt blob by sending an HTTP GET request to `/users/[uid]/services/salts/[sid]`.

URL	`https://example.com/api/users/[uid]/services/salts/[sid]`
Method	GET
Requires Authentication	Yes
Returns	200 OK: Value of salt blob successfully returned. 401 Unauthorized: User authentication is required. 404 Not Found: The server has not found anything matching the Request-URI. 500 Internal Server Error: The server encountered an unexpected condition which prevented it from fulfilling the request.

XML Response message(salts):

```
<?xml version="1.0"?>
<salt>
   <sid>s12</sid>
   <value>a10</value>
</salt>
```

JSON Response message:

```
{"salt":{"sid":"s12","value":"a10"}}
```

Add New Salt

Add a new salt blob to SSS by HTTP POST request to `/users/[uid]/services/salts`.

URL	`https://example.com/api/users/[uid]/services/salts`
Method	PUT
Requires Authentication	Yes
Request Body	Content-Type: application/x-www-form-urlencoded domain: Contains the encrypted domain value. value: The salt blob value.
Returns	201 Created: Salt blob was successfully saved and returned. 400 Bad Request: The request was malformed and it was not understood by SSS. 401 Unauthorized: User authentication is required. 404 Not Found: The server has not found anything matching the Request-URI. 500 Internal Server Error: The server encountered an unexpected condition which prevented it from fulfilling the request.

Request body (encoded string):

```
value=a60&domain=e10
```

XML Response message(salts):

```
<?xml version="1.0"?>
<salt>
   <sid>s16</sid>
   <value>a60</value>
</salt>
```

JSON Response message:

```
{salt":{"sid":"s16","value":"a60"}}
```

Update Salt

Update the salt blob by sending an HTTP PUT request to `/users/[uid]/services/salts/[sid]`. To update a salt blob the current salt blob and the new salt blob must be sent to the SSS. The SSS checks whether the current salt blob corresponds with the salt blob stored in the database and replaces it with the new value.
In case the current salt blob value does not match the salt blob at the SSS, which implies that the salt blob is currently being updated, error code 409 is returned.

URL	`https://example.com/api/users/[uid]/services/salts/[sid]`
Method	PUT
Requires Authentication	Yes
Request Body	Content-Type: application/x-www-form-urlencoded currentValue: The current salt blob value. newValue: The new salt blob value.
Returns	200 OK: The salt blob for this user device was successfully updated. 400 Bad Request: The request was malformed and it was not understood by SSS. 401 Unauthorized: User authentication is required. 404 Not Found: The server has not found anything matching the Request-URI. 409 Conflict: The request could not be completed due to a conflict with the current state of the resource. 500 Internal Server Error: The server encountered an unexpected condition which prevented it from fulfilling the request.

Request body (encoded string):

```
currentValue=e60&newValue=e6000
```

Remove a salt

Delete a salt blob from SSS by sending an HTTP DELETE request to `/users/[uid]/services/salts/[sid]`.

URL	`https://example.com/api/users/[uid]/services/salts/[sid]`
Method	DELETE
Requires Authentication	Yes
Returns	200 OK: The salt blob was successfully removed. 400 Bad request: The request was malformed and it was not understood by SSS. 401 Unauthorized: User authentication is required. 404 Not Found: The server has not found anything matching the Request-URI. 500 Internal Server Error: The server encountered an unexpected condition which prevented it from fulfilling the request.

3.2.6 HTTP status codes

Successful 2xx

200	OK	The request has succeeded.
201	Created	The request has been fulfilled and resulted in a new resource being created.

Error 4xx and 5xx

400	Bad Request	The request could not be understood by the server due to malformed syntax.
401	Unauthorized	The request requires user authentication.
404	Not Found	The server has not found anything matching the Request-URI.
405	Method Not Allowed	The method specified in the Request-Line is not allowed for the resource identified by the Request-URI.
409	Conflict	The request could not be completed due to a conflict with the current state of the resource.
415	Unsupported Media Type	The server is refusing to service the request because the entity of the request is in a format not supported by the requested resource for the requested method.
500	Internal Server Error	The server encountered an unexpected condition which prevented it from fulfilling the request.

3.3 Access Control

In Section 2.3.2 we described that we use public-key cryptography for user authentication at the SSS with a random authentication secret K_{Auth} and respective public key. The implementation uses the X.509 [5] standard for certificate creation and uses these certificates for authentication.

To establish a secure and authenticated connection between the SSS and a user device, mutual authentication is needed. Since establishing a secure connection to the SSS with server authentication is commonly done by HTTPS (HTTP over TLS), it makes sense to use TLS for client authentication as well [11]. So to create a two-way-authenticated connection between the user and the SSS, client and server authentication is needed. In the following it is explained how those two goals are reached.

Server authentication

For the client to know that the SSS is authentic, the SSS needs a verified certificate by a certification authority (CA) which the client trusts. This is the case when the client trusts the mentioned CAs root certification authority. Alternatively the client needs the public key, also known as the certificate of the SSS, in its list of trusted CAs.

The verification takes place at the beginning of the communication between server and client. It is called the TLS handshake. Whether the SSS has a self-signed certificate or it has its certificate signed by a CA is not important for the implementation, since it is a matter of configuration before starting the server. While testing we used a X.509v3 self-signed 2048-bit RSA certificate for the SSS.

Client authentication

Setup

For the SSS to know that the client is authentic, the client needs to present a certificate signed by the SSS. The client authentication to the SSS is setup as follows. A secure connection via HTTP over TLS is established with server authentication for the client's first request at the SSS. This request contains a Certificate Signing Request (CSR) [26] with the public key. Any other information contained in the CSR is not relevant to the SSS, therefore it can be omitted. Based on the CSR, the SSS creates a X.509 certificate, signed with the private key of the SSS. The X.509v3 specification provides a number of properties to be set in the certificate. To build the certificate, the SSS sets a 64 bit serial number, generated with the `SecureRandom` class from `java.security`. It sets its own certificate as the CA certificate and sets the expiration date 30 years into the future. Furthermore extensions are added, which are defined in section 4.2.1 of

RFC 5280 [5]. First the Basic Constraint (ASN.1 Object Identifier 2.5.29.19) is set which attests that the subject of the certificate is not a CA. Second, key usage (OID 2.5.29.15) is set which indicates the use for digital signature and data encipherment. Third, extended key usage (OID 2.5.29.37) is set to assure client authentication.

The SSS stores the certificate in the database together with the device name and user identification and sends it back to the client. From now on the client has to present this certificate every time when accessing the SSS, because apart from registration, the other requests for managing data at the SSS require client authentication.

Adding additional devices

Since every device needs its own signed certificate, the way to register an additional user device at the SSS works as follows. The user with his already registered device requests an authentication token T_{Auth} from the SSS. The SSS saves the token bound to the user account in the database and sets a timeout of 5 minutes on it. Within this time the user has to transfer the token to the new device which creates its own key pair and a respective CSR. Then it sends the token and the CSR to the SSS. This time, analogous to registering a user at the SSS, only server side authentication via HTTPS is required. Finally, the SSS signs the certificate, sends it back and deletes the token. Because of the authentication token only the user can add new devices using registered ones.

Verification

Once a user device has the signed client certificate, a connection via HTTPS with mutual authentication can take place. This is realized through the TLS handshake protocol [11]. The protocol verifies that the server trusts the client's certificate and vice versa. Trust in this case means, that the certificate of the proving party lies in the trusted certificate chain of the verifying party. This process is handled by Tomcat.

Before processing the client request further to the API, the implemented filter method `AuthenticationFilter` is triggered, which overrides the standard filter of `javax.ws.rs.container.ContainerRequestFilter`. This is the only way to access the client certificate by code. The triggering of the filter has been implemented with Name Binding, which supported by JAX-RS 2.0.

At this point, the SSS proceeds with dealing with the resource request. The business logic for creating, deleting and retrieving resources, e.g. saving and retrieving salt blobs, lies here as well. The SSS performs CRUD (Create, Retrieve, Update, Delete) operations in a database with the help of data access objects.

Then the SSS can check further properties, to determine whether an incoming request will be granted. First it checks the expiration date of the client certificate. Then it checks the signature of the client certificate. It must be signed by the SSS, otherwise the request is rejected. Last it checks in the database whether the user identifier in the request matches the identifier which was bound to the certificate. It it needs to be noticed that it is not checked which device is issuing the request as long as it belongs to the correct user account. This means that every device has full rights to operate on the user account.

Outlook

A possibility to further extend the functionality of the SSS with modifying the current implementation is to add device dependent permissions. A real life use case would be that user A lends his secondary mobile phone to user B who is authorized to have access to only a subset of services and act on user A's behalf. The secondary device would then be authorized to only request salt blobs for these services at the SSS. The SSS would write the device identifier into the certificate when creating it out of a Certificate Signing Request and store it into the database. An additional table in the database would store permissions for different requests of different devices. Any request at the SSS that requires client authentication triggers a filter method in the `AuthenticationFilter`. At this point we can access the client certificate, look up the device identifier, redefine the `javax.ws.rs.core.SecurityContext` of the incoming request by adding the identifier into the context. After the request reaches the business logic, it can be determined, whether the requested resource is allowed to be accessed by the current device and possibly decline the request.

3.4 Data storage

The purpose of the SSS is to distribute and store data. This implementation uses data access objects (cf. Section 3.4.1) to store data into a MySQL database. Section 3.4.2 describes the functionality of the SSS in regard to the data storage.

As described in Chapter 2 the SSS stores most importantly salt blobs, which contain of a collection of data regarding a specific service account. Since the salt blob is encrypted the SSS does not have to take further cryptographic action before writing it into the database. The only precaution that is taken is that the execution of SQL commands is implemented with `java.sql.PreparedStatement`. This serves to defend against SQL injections [3]. Otherwise it would be possible for an attacker to read, modify and even delete large parts of the database. It is also notable that to store data, the SSS does not have to be on the same server as the SSS. On startup, the SSS connects to a MySQL database via a JDBC driver. In case the database is reachable under a different host, the connection can be secured e.g. with the JDBC Thin Driver [10], or through ssh tunneling using the JCraft framework [32].

Beside the salt blob, the SSS stores the encrypted domain name in order to categorize the salt blobs and to handle requests which ask for salt blobs of a specific domain. Furthermore, upon creating any resource except for a token, the SSS creates a 64 bit serial number to identify the resource. It is generated with the `java.security.SecureRandom` class and stored in the database as well. For more details, the table in figure 3.2 shows the structure of the database tables. The MySQL type for all the columns is *text* and every column has a primary key of type *integer* which is automatically incremented and not exposed to the business logic of the SSS. The reason why we use randomly generated identifiers instead of the auto incremented primary keys of the database is security. An attacker should not be able to draw any conclusions from the identifier about the state of the database of the SSS.

table	**column**			
users	uid			
tokens	uid	token	timestamp	
devices	uid	did	E_k(name)	certificate
salts	uid	E_k(domain)	sid	salt blob
services	uid	E_k(domain)		

Table 3.2: Database structure

3.4.1 Data Access Objects

A data access object is an object that serves as an interface for the communication between the business logic in the resource classes and the database. There are data access objects for all the resource types, namely device, salt, service, token and user. The implementation uses the DAO pattern without a transfer object in combination with the abstract factory pattern [2].

The abstract factory pattern allows to implement multiple factories to create DAO objects for different database types and also to select the factory at runtime. The implementation includes a `MySqlDAOFactory` and `SQLiteDAOFactory` (for SQLite databases [21]). Both factories create specific DAOs for their corresponding database type. Any other database can be supported by implementing the DAO interfaces, which helps the extensibility of the service (**NFR 3**). The table in figure 3.3 shows what functionality the interfaces define for each data access object.

	create	**update**	**delete**	**get**	**list**	**verify (existence)**
Device	x	x	x	x	x	x
Salt	x	x	x	x	x	x
Service	x		x		x	x
Token	x		x	x		x
User	x		x			x

Table 3.3: DAO functionality

3.4.2 Sequence diagrams

In this Section we have a closer look at different kind of processes which are possible with the SSS, with respect of the data storage.

Note that for reasons of clarity, the API and the business logic are represented as a single entity.

Registration at SSS

A user is registered at the SSS usually when PALPAS is set up on a client device. The client device sends a request to create a new user (cf. Section 3.2.1). Since the request does not require client authentication, the SSS creates a certificate out of the CSR (cf. Figure 3.2). Then a 64 bit serial number is randomly generated to serve as the user identifier. The user DAO stores the user identifier in the users table of the database (cf. database tables 3.2) and with the help of the device DAO the device information, i.e. corresponding user identifier, signed certificate and device name are stored in the database. An object of the user class is created to be returned to the client. It includes the identifier and the certificate for the initial device.

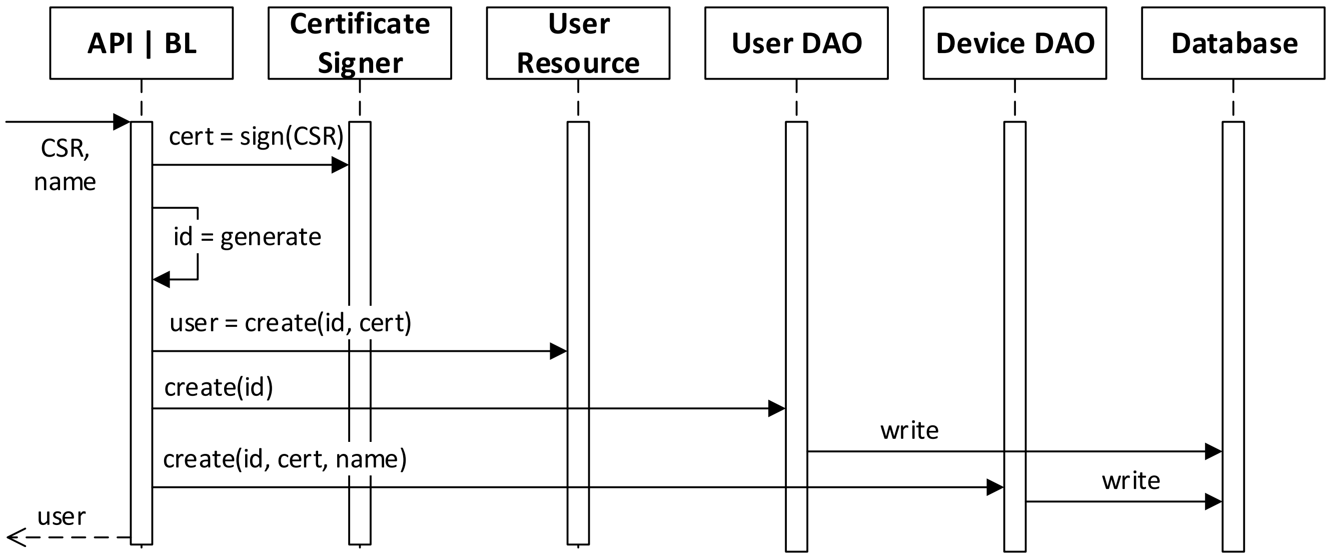

Figure 3.2: Create user

Registration of an additional device

To register an additional device a client has to request an authentication token first (cf. Figure 3.3). Since this request requires client authentication, the SSS uses the access contro lcomponent to validate the request, i.e. whether provided certificate matches the user identifier of the request. A random 64 bit token value is generated and stored in the database through the token DAO together with the user id and a timestamp. The token is sent back to the client.

After initializing the new device, it sends a CSR, device name and the authentication token to the SSS (cf. Figure 3.4). It is checked, whether the token is not expired yet. Otherwise the request is rejected. The certificate signer creates a certificate out of the CSR and a 64 bit device identifier is generated. The token can be deleted through the token DAO and the device DAO stores the device identifier, certificate and name in the database. Subsequently the device object with the signed certificate is returned to the client.

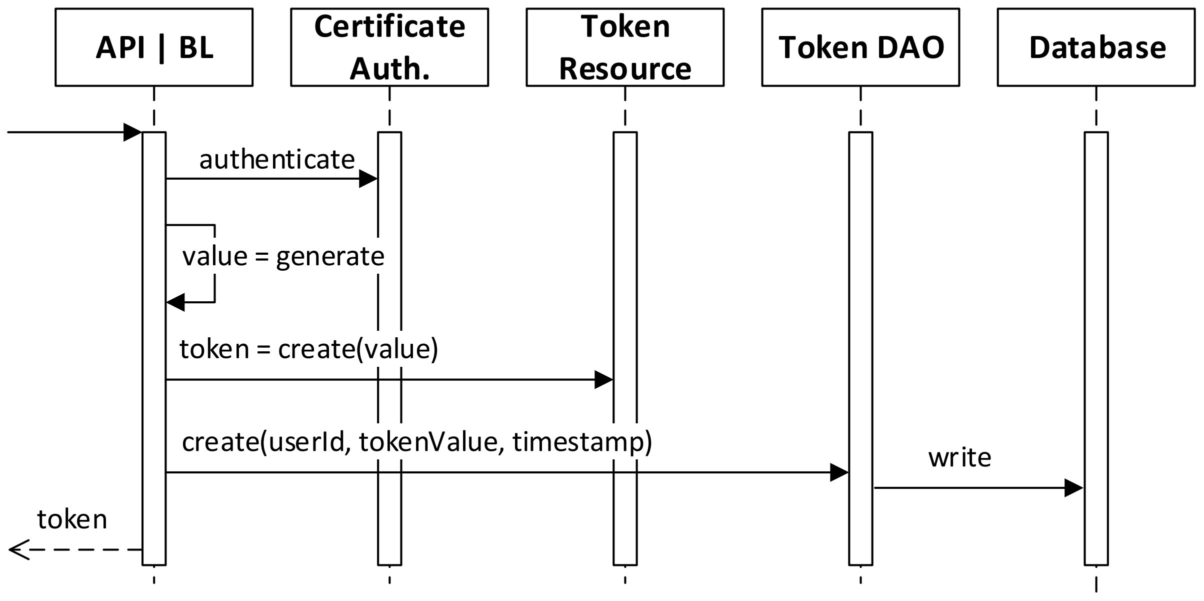

Figure 3.3: Request token

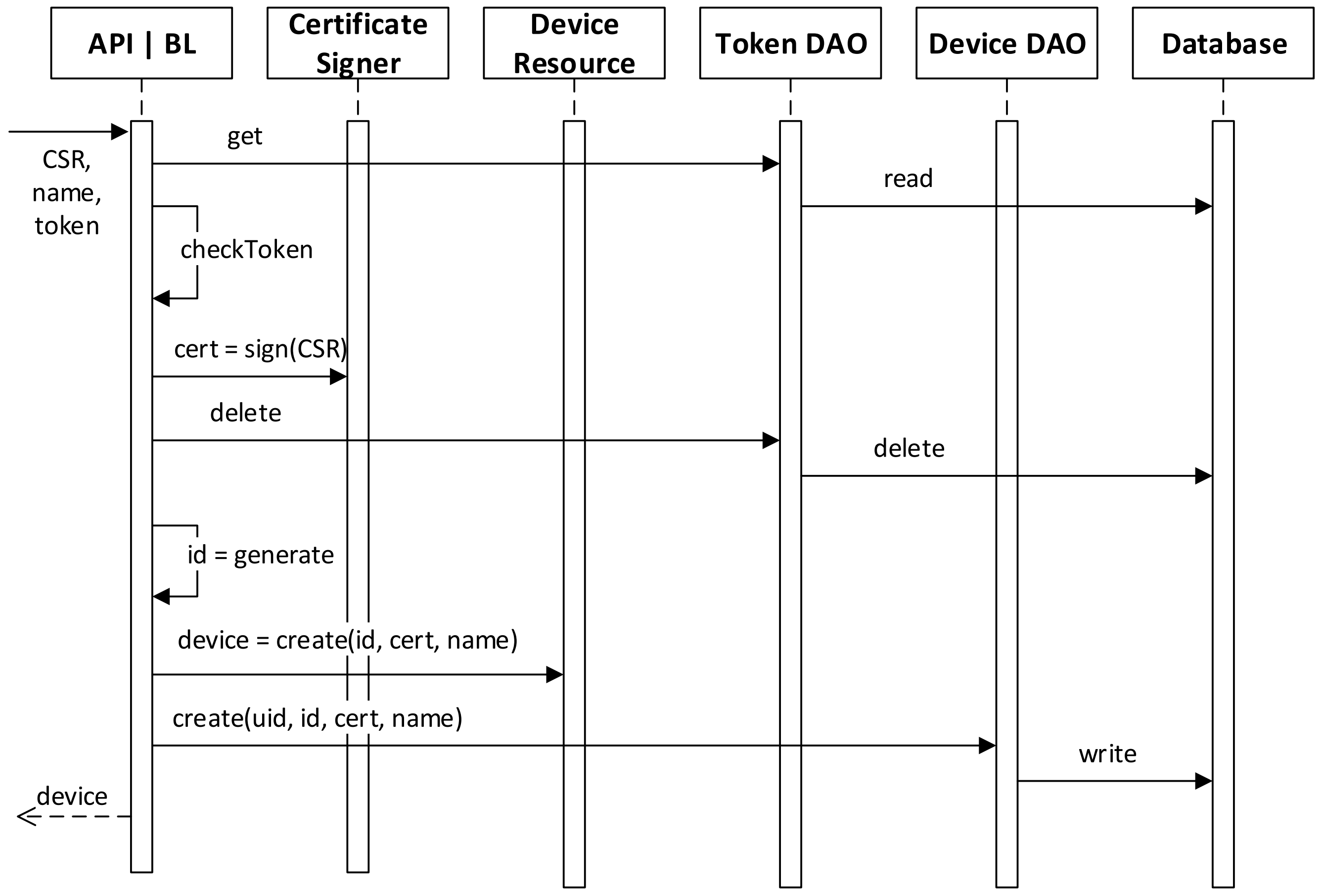

Figure 3.4: Register device

The main interactions with salts at the SSS are presented in this subsection. Note that retrieving and deleting other resources works analogous.

Adding salt blob

To add a salt blob at the SSS the registered client sends the salt blob, as well as the encrypted domain value to the SSS (cf. Figure 3.14). The certificate authenticator validates the request. And if the salt belongs to a new service, an entry in the database is created with the provided domain value. A random 64 bit salt identifier is generated and stored in the database with the help of the salt DAO together with the user id, domain and the salt blob. Then, a salt object including i.a. the salt identifier is returned to the client.

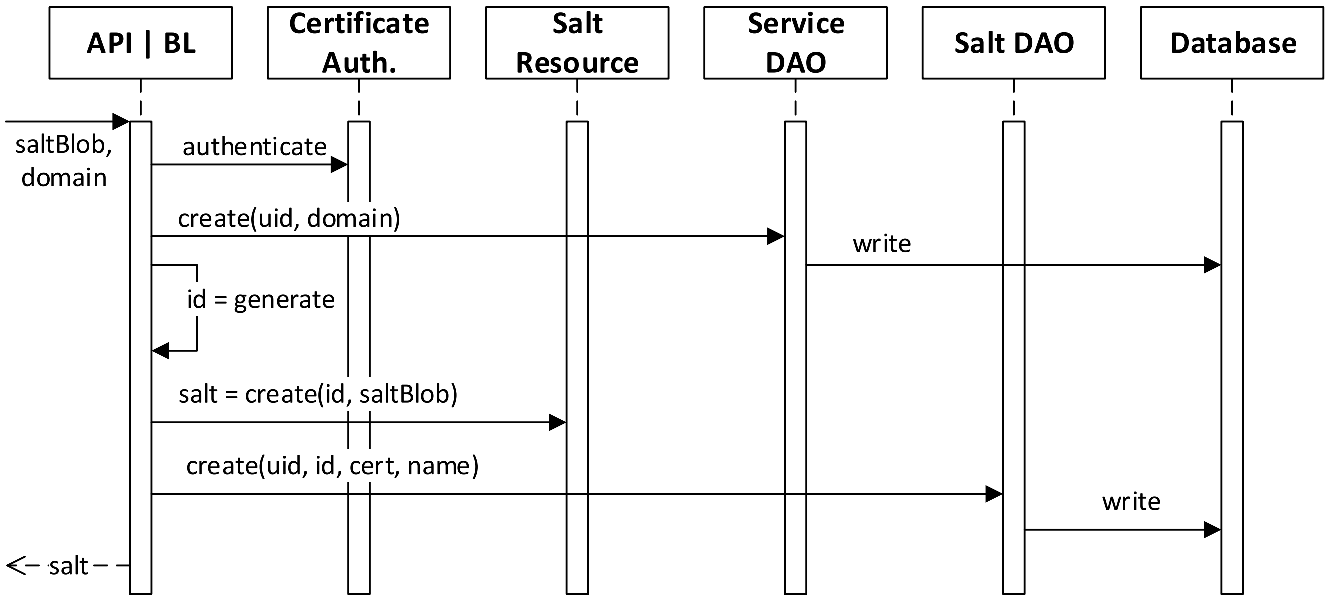

Figure 3.5: Adding salt blob

Updating salt blob

Figure 3.6 shows the update of a salt blob at the SSS. First the client has to generate a new salt and create the new salt blob. Then the domain, the new, as well as the current salt blob are sent to the SSS. The requested is authenticated and the SSS checks whether the currently stored salt blob matches the one in the request. The salt DAO is used to retrieve the data from the database. If the salt blob does not match, the request is aborted. It means that the client sent the wrong value or that the salt blob is currently being updated by another device. In that case the client has to determine how to dissolve the issue. Once the verification is successful, the salt DAO updates the salt blob in the database.

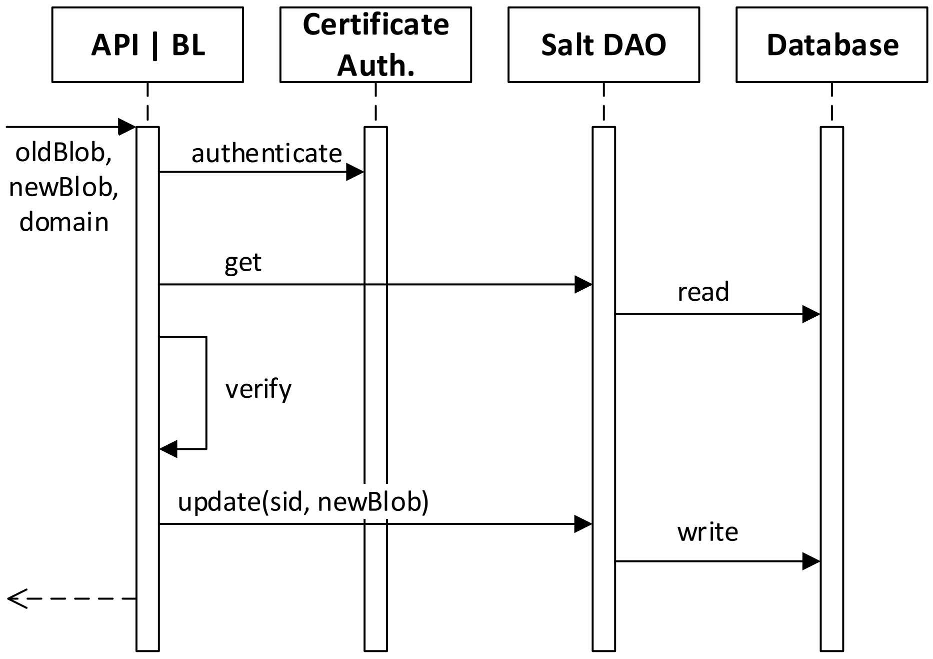

Figure 3.6: Updating salt blob

Retrieving salt blob

If a user wants to obtain a salt from the service, he or she sends an authenticated GET request with his user id and the salt id in the URL to the SSS (cf. API in Section 3.2.5). After the request is authenticated the salt blob is retrieved from the database through the salt DAO (cf. Figure 3.7 Then a salt object is created including the salt blob and sent back to the client.

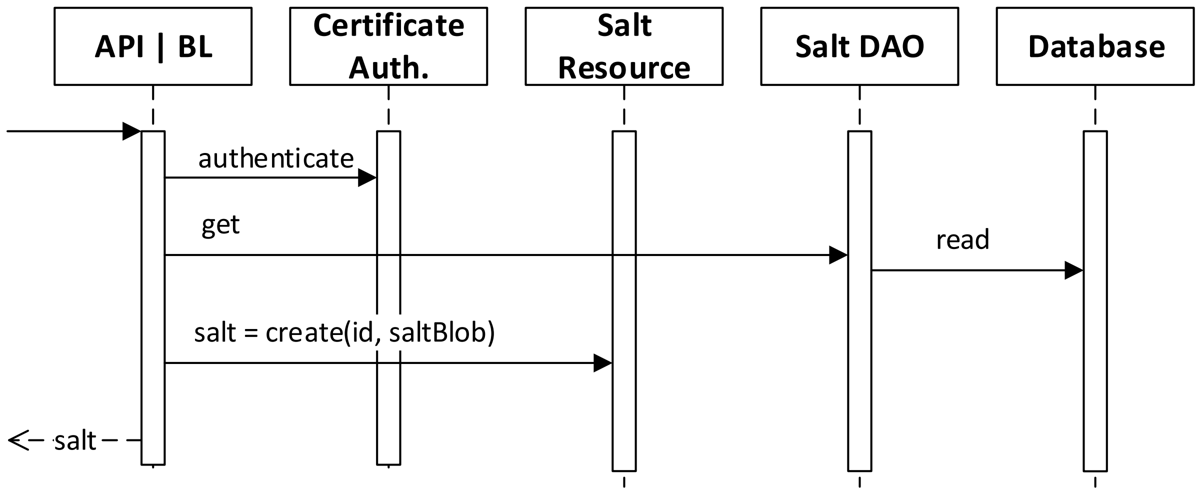

Figure 3.7: Retrieving salt blob

Deleting salt blob

Figure 3.8 shows the deletion of a resource, more specifically a salt blob at the SSS. The client sends a request, which is authenticated by the certificate authenticator. Then the existence of the salt blob is verified with the help of the salt DAO. Last, the entry is deleted and the request successfully completed.

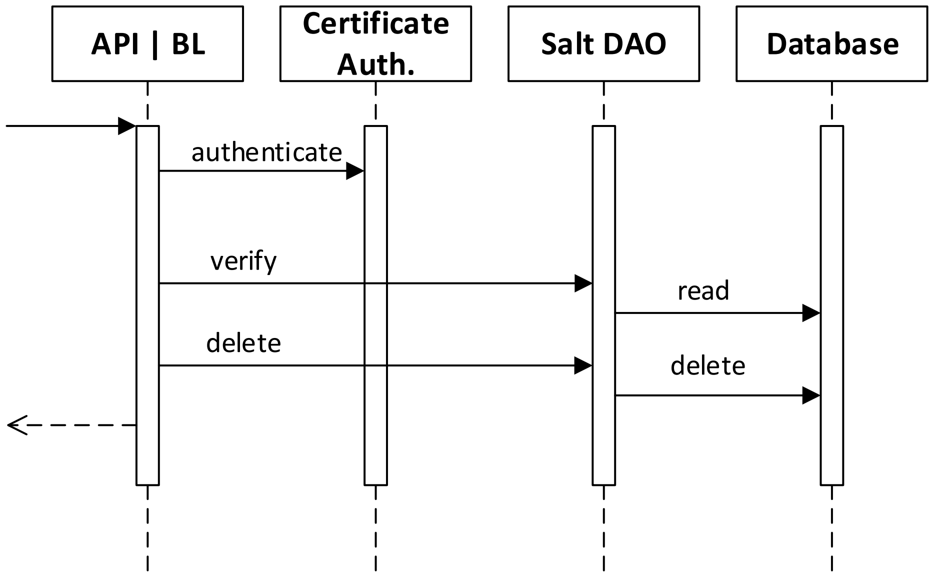

Figure 3.8: Deleting salt blob

Retrieving list of salt blobs

If a client wants to retrieve a list of salt blobs for a particular service, it can be done by sending a request with the domain data as parameter (cf. API in Section 3.2.5). After the request is authenticated, it is verified whether there are any salt blobs registered for the particular service. Then the salt DAO gets the list of entries for the service from the salt table in the database. Since the entries contain more information that are necessary for the response, a salts object is created out of the salt blob list and sent back to the client.

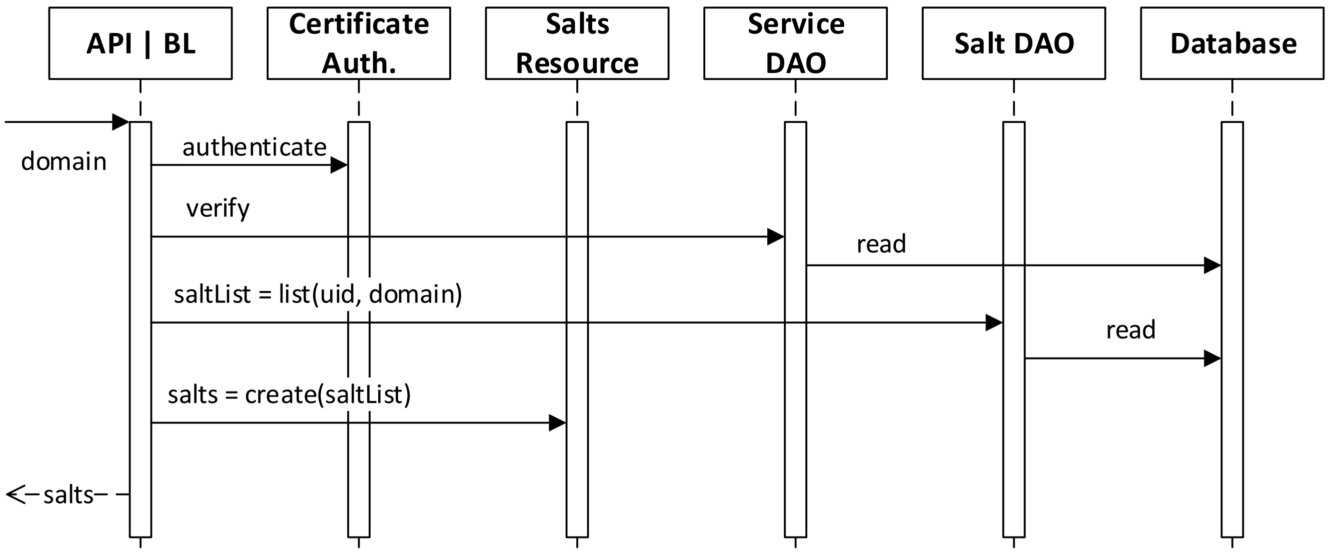

Figure 3.9: Retrieving list of salt blobs

3.5 Application flows

This Section provides figures and descriptions of the protocol flows for the functional requirements **FR 1** and **FR 3-6**, namely for the registration of a new user, an additional user device, for the storage, retrieval and update of salt blobs. The application flow for deletion of data is similar to their retrieval and the corresponding requests are defined in the API in Section 3.2. The following diagrams contain the Salt Synchronization Service (SSS) and a user device (D) running PALPAS. Since the responses consist of objects, the object name, as well as the content is displayed.

Note that the API (cf. Section 3.2) allows more interaction than represented in the diagrams, but they work analogously.

Registering an User

As illustrated in Figure 3.10, a client running PALPAS sends a Certification Signing Request (CSR) via a HTTP POST request to `/users` (cf. API in Section 3.2.1). The SSS signs the public key certificate in the CSR with its private key and sends the signed certificate back to the client together with the user identifier.

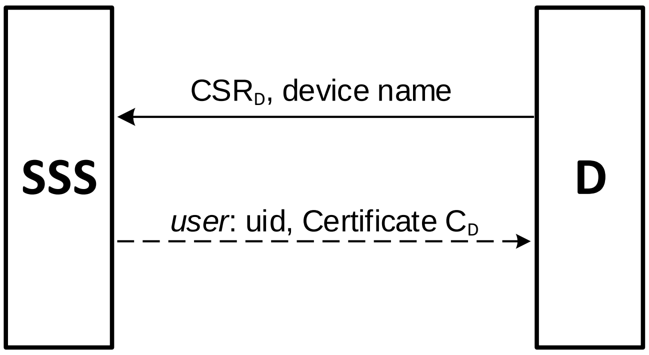

Figure 3.10: Registering an user

Adding a Device

To add an additional device, the client has to perform two steps. As illustrated in Figure 3.11, first an authentication token T_{Auth} is requested with the already registered device D. This is done by sending a HTTP GET request to `/users/tokens` (cf. API in Section 3.2.2). Then, after setting up PALPAS on the new device D', which includes transferring T_{Auth}, a HTTP POST request is sent to `/users/[uid]/devices`, providing the transferred authentication token, a CSR and a encrypted device name for the new device (cf. API in Section 3.2.3). This request does not require client authentication, since the new device cannot be registered yet. The SSS signs the certificate and sends it back to the new client. From now on, the new device uses the certificate to create a mutually authenticated connection to the SSS, for further requests.

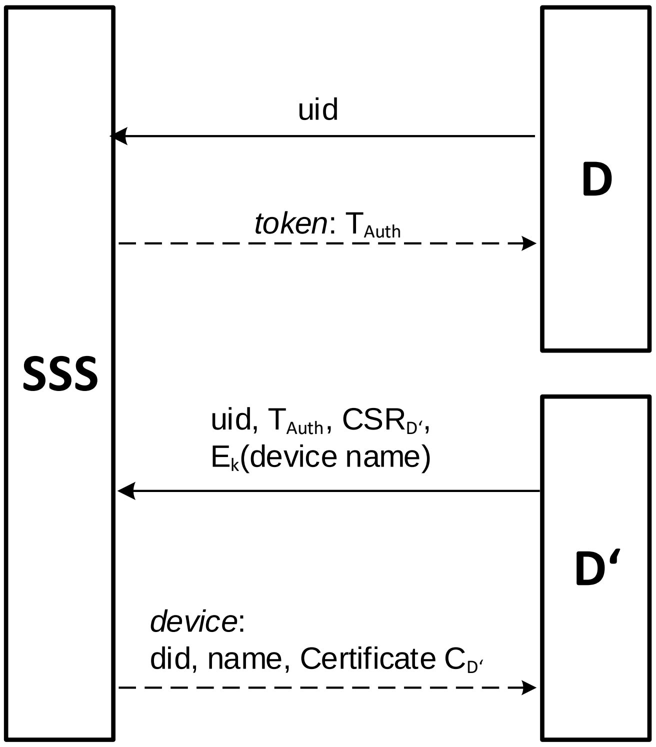

Figure 3.11: Adding a device

Retrieving Salt blobs

As depicted in Figure 3.12, to retrieve a single Salt blob, the client sends a HTTP GET request to `/users/[uid]/services/salts/[sid]`, providing his user identifier and the salt identifier and the SSS responds with the salt object, containing the salt identifier and the salt blob itself (cf. API in Section 3.2.5).

To retrieve a list of salt blobs which belong to the same service, the client sends a HTTP POST request to `/users/[uid]/services/salts`, providing the encrypted domain instead of the salt id. As shown in Figure 3.13, the SSS accumulates a list of corresponding salt objects and sends them back to the client.

Furthermore the client has the possibility to retrieve all stored salt blobs from all services. Apart from the user id, no information has to be provided to execute the HTTP GET request `/users/[uid]/services`.

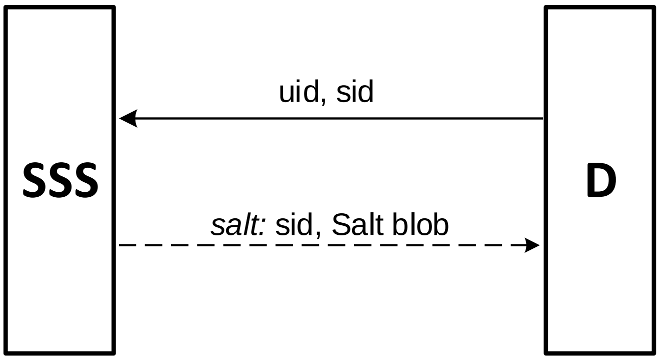

Figure 3.12: Retrieving single salt blob

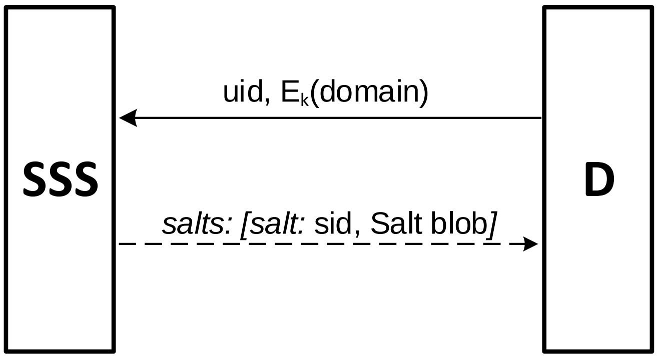

Figure 3.13: Retrieving list of salt blobs

Adding a Salt blob

As depicted in Figure 3.14, the clients sends a HTTP POST request to `/users/[uid]/services/salts`, providing the salt blob and the encrypted domain to add a salt blob to the SSS (cf. API in Section 3.2.5). The SSS checks, whether a service with the given domain exists and creates a service entry if needed, and stores the salt blob with an associated and randomly generated identifier. The client receives a response with the salt object containing the salt id which can be used to retrieve the salt blob at a later time again.

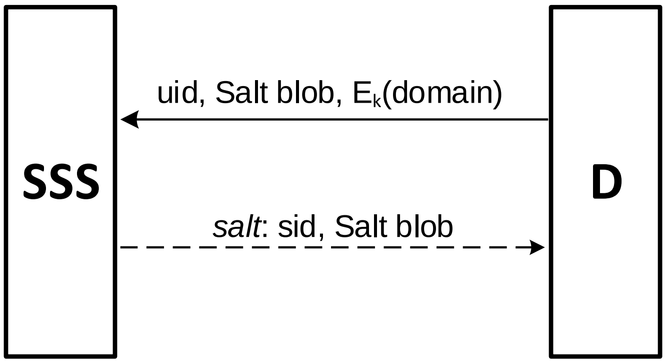

Figure 3.14: Adding a salt blob

Updating Salt blob

To update an existing salt blob at the SSS, the client has to retrieve the current salt blob first. After PALPAS updates the password at a service, the new, as well as the old salt blob are sent to the SSS (cf. Figure 3.15). A HTTP PUT request to `/users/[uid]/services/salts/[sid]` is used (cf. API in Section 3.2.5). The SSS verifies that the given old value matches the stored salt blob and overrides it with the new value in the database. This mechanism protects salt blobs from being incorrectly overwritten, when multiple devices try to update a salt blob at the same time. The response is either a success or an error message. The error message usually indicates that a different device is currently trying to update the same salt blob.

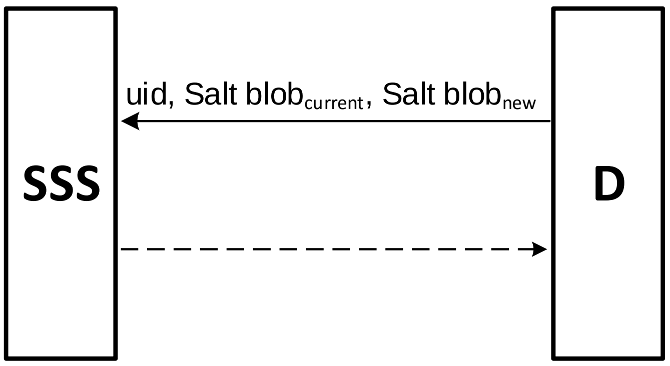

Figure 3.15: Updating salt blob

4 Conclusion

Typically, synchronizing passwords is done by encrypting the passwords and storing them on a central server. In case an adversary manages to steal the encrypted data, an offline brute-force attack can be performed which makes a disclosure of all stored passwords possible. PALPAS solves the problem and provides secure password synchronization. It does not store the passwords at a server, only data that is statistically independent of the passwords. This data, i.e. the salts, in combination with a local secret are used to compute the password on the user devices. Since privacy relevant data is encrypted first before storing it at the SSS, the protection of privacy of the user data can be ensured. Furthermore, the usage of public key authentication makes the PALPAS scheme invulnerable against phishing attacks.

The SSS is the key component in the PALPAS scheme. Its main role lies in distributing salts between the legitimate user devices. In this thesis we presented the full-fledged concept of the SSS. The attacker model was described to provide insight about the assumed operational environment. Then we defined requirements to meet the needs that are expected e.g. from the PALPAS scheme. Last, the architecture was explained by elaborating on the components the SSS consists of, such as the API, access control, business logic and data storage component.

Moreover the thesis shows how the SSS can be realized by presenting an implementation. It is explained how the components are implemented and how their communication functions internally. Our implementation provides client certificate creation, client and server authentication via TLS, a RESTful API and the usage of either of two types of databases for data storage.

The SSS can be operated with a self-signed or a CA signed certificate. The usage of device specific certificates for client authentication not only enables identifying the user account of the request but also opens up possibilities for device specific features.

The RESTful API provides a clear, extensible, and reusable interface for clients. It enables the clients to retrieve resources like salts individually as well as a coupled list in order to facilitate the interaction with the SSS. As proposed by Horsch et al. [18], the implementation supports multiple salt data per service. The extensibility of a RESTful API enables possible features like sharing salts with other users, and also, in combination with the implemented access control, restricting access to a subset of salts for specific devices. Hence offering the user the opportunity to have devices that are authorized to compute only critical or less important passwords.

Another advantage of the implementation is the modularity of the data storage and the possibility to choose the database type at runtime, which makes it adaptable to the needs of the operator. Interfaces for more databases can be implemented, currently MySQL and SQLite databases can be used.

In summary it can be stated that the presented concept of the SSS fulfills all security and functional expectations. Furthermore the implemented solution meets the requirements of the described concept so that it can be used within the PALPAS scheme.

Bibliography

[1] Subbu Allamaraju. *RESTful Web Services Cookbook*. O'Reilly, 2010. Cited on page 11.

[2] Deepak Alur, Dan Malks, and John Crupi. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001. Cited on page 35.

[3] Kasra Amirtahmasebi, Seyed Reza Jalalinia, and Saghar Khadem. A survey of SQL injection defense mechanisms. In *Proceedings of the 4th International Conference for Internet Technology and Secured Transactions, ICITST 2009, London, UK, November 9-12, 2009*, pages 1–8, 2009. Cited on page 34.

[4] Karthikeyan Bhargavan and Antoine Delignat-Lavaud. Web-based attacks on host-proof encrypted storage. In *6th USENIX Workshop on Offensive Technologies (WOOT'12)*, pages 97–104. Usenix, aug 2012. Cited on page 1.

[5] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile. RFC 5280, RFC Editor, May 2008. `http://www.rfc-editor.org/rfc/rfc5280.txt`. Cited on pages 31 and 32.

[6] LastPass Corporate. Lastpass - the last password you have to remember. `https://lastpass.com`, 2016. Cited on page 4.

[7] LastPass Corporate. Lastpass security notification. `https://blog.lastpass.com/2011/05/lastpass-security-notification.html`, 2016. Cited on page 1.

[8] Oracle Corporation. Java architecture for xml binding (jaxb). `https://jaxb.java.net`, 2016. Cited on page 11.

[9] Oracle Corporation. Project jersey. `https://jersey.java.net`, 2016. Cited on page 11.

[10] Jean de Lavarene. Ssl with oracle jdbc thin driver. `http://www.oracle.com/technetwork/topics/wp-oracle-jdbc-thin-ssl-130128.pdf`, April 2010. Cited on page 34.

[11] T. Dierks and E. Rescorla. The transport layer security (tls) protocol version 1.2. RFC 5246, RFC Editor, August 2008. `http://www.rfc-editor.org/rfc/rfc5246.txt`. Cited on pages 31 and 32.

[12] Serge Egelman. My profile is my password, verify me!: The privacy/convenience tradeoff of facebook connect. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 2369–2378, New York, NY, USA, 2013. ACM. Cited on page 4.

[13] Facebook. Add facebook login to your app or website. `https://developers.facebook.com/docs/facebook-login`, 2016. Cited on page 4.

[14] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887. Cited on page 12.

[15] Paolo Gasti and Kasper Bonne Rasmussen. On the security of password manager database formats. In *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, pages 770–787, 2012. Cited on page 1.

[16] J. Alex Halderman, Brent Waters, and Edward W. Felten. A convenient method for securely managing passwords. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, pages 471–479, New York, NY, USA, 2005. ACM. Cited on page 4.

[17] Johan Haleby. Rest assured - java dsl for easy testing of rest services. `http://rest-assured.io`, 2016. Cited on page 12.

[18] Moritz Horsch, Andreas Hülsing, and Johannes A. Buchmann. PALPAS - password less password synchronization. In *10th International Conference on Availability, Reliability and Security, ARES 2015, Toulouse, France, August 24-27, 2015*, pages 30–39. IEEE Computer Society, 2015. Cited on pages 1, 2, 3, 6, and 47.

[19] Moritz Horsch, Mario Schlipf, Johannes Braun, and Johannes A. Buchmann. Password requirements markup language. In Joseph K. Liu and Ron Steinfeld, editors, *Information Security and Privacy - 21st Australasian Conference, ACISP 2016, Melbourne, VIC, Australia, July 4-6, 2016, Proceedings, Part I*, volume 9722 of *Lecture Notes in Computer Science*, pages 426–439. Springer, 2016. Cited on page 2.

[20] AgileBits Inc. 1password - when it's too important to go anywhere else. `https://1password.com`, 2016. Cited on page 4.

[21] Jay A. Kreibich. *Using SQLite - Small. Fast. Reliable. Choose any Three*. O'Reilly, 2010. Cited on page 35.

[22] Maria Maleshkova, Carlos Pedrinaci, and John Domingue. Investigating web apis on the world wide web. In *8th IEEE European Conference on Web Services (ECOWS 2010), 1-3 December 2010, Ayia Napa, Cyprus*, pages 107–114, 2010. Cited on page 12.

[23] Vincent Massol and Ted Husted. *JUnit in Action*. Manning Publications Co., Greenwich, CT, USA, 2003. Cited on page 12.

[24] Anupama Mishra, B. B. Gupta, and R. C. Joshi. A comparative study of distributed denial of service attacks, intrusion tolerance and mitigation techniques. In *Proceedings of the 2011 European Intelligence and Security Informatics Conference*, EISIC '11, pages 286–289, Washington, DC, USA, 2011. IEEE Computer Society. Cited on page 6.

[25] Mozilla. Firefox accounts/sync protocol. `https://github.com/mozilla/fxa-auth-server/wiki/onepw-protocol`, 2016. Cited on page 4.

[26] M. Nystrom and B. Kaliski. Pkcs #10: Certification request syntax specification version 1.7. RFC 2986, RFC Editor, November 2000. Cited on page 31.

[27] The Legion of the Bouncy Castle Inc. Bouncy castle crypto apis. `https://www.bouncycastle.org`, 2016. Cited on page 11.

[28] D. Reichl. Keepass password safe. `http://keepass.info`, 2016. Cited on page 4.

[29] Blake Ross, Collin Jackson, Nick Miyake, Dan Boneh, and John C. Mitchell. Stronger password authentication using browser extensions. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association. Cited on page 4.

[30] Ben Stock and Martin Johns. Protecting users against xss-based password manager abuse. In *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, pages 183–194, 2014. Cited on page 1.

[31] Michael Widenius and David Axmark. *MySQL reference manual - documentation from the source*. O'Reilly, 2002. Cited on page 11.

[32] Dr. Atsuhiko Yamanaka. Jcraft. `http://www.jcraft.com`, 2016. Cited on page 34.

[33] Chuan Yue. The devil is phishing: Rethinking web single sign-on systems security. In *Presented as part of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats*, Washington, D.C., 2013. USENIX. Cited on page 4.

[34] S. T. Zargar, J. Joshi, and D. Tipper. A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks. *IEEE Communications Surveys Tutorials*, 15(4):2046–2069, April 2013. Cited on page 6.