



Florian Burgstaller, B.Sc.

Secure Cloud Password Manager (SPM)

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's Degree Programme: Computer Science

submitted to

Graz University of Technology

Supervisor

O.Univ.-Prof. Dipl.-Ing. Dr.techn. Reinhard Posch
Institute of Applied Information Processing and Communications (IAIK)


Advisor

Dipl.-Ing. Florian Reimair
Institute of Applied Information Processing and Communications (IAIK)

Graz, August 2016

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

Signaturwert	G3nzgLZhuD2SSzUjy1m2pIW77NfLqjhl+EdiV0ra+N6X23X0G4Rfh8Lheq/oY7IH7wPKTJDH bsdyCdg2aUz1Q==	
	Unterzeichner	Florian Burgstaller
	Aussteller-Zertifikat	CN=a-sign-Premium-Sig-02,OU=a-sign-Premium-Sig-02, O=A-Trust Ges. f. Sicherheitssysteme im elektr. Datenverkehr GmbH,C=AT
	Serien-Nr.	1069079
	Methode	urn:pdfsigfilter:bka.gv.at:binaer:v1.1.0
	Parameter	etsi-moc-1.2:ecdsa-sha256@60330bc6
Prüfinformation	Signaturprüfung unter: http://www.signaturpruefung.gv.at	
Hinweis	Dieses mit einer qualifizierten elektronischen Signatur versehene Dokument ist gemäß § 4 Abs. 1 Signaturgesetz einem handschriftlich unterschriebenen Dokument grundsätzlich rechtlich gleichgestellt.	
Datum/Zeit-UTC	2016-08-27T16:23:04Z	

Date

Signature

Abstract

Storing passwords in a secure environment has become common for many people nowadays. However, often there is a need to share passwords among a group of users. Group password managers are often implemented as trusted server-side encryption systems. Such systems encrypt the data only locally and are therefore more vulnerable to information leakage in case of an attack. Furthermore, many organisations that want to share passwords, do not have the means to run and maintain servers on their own. In this work, a system is presented, that solves the described problems by offering a fully equipped team password manager, that can mainly run on untrusted third party servers. This was accomplished by designing a distributed encryption system which implements different layers of trust. Consequently no keys have to be exchanged between the users of the password management system. Additionally the database server, that stores the password information, can be seen as untrusted and is not vulnerable to leak information in different kinds of attack scenarios. Those scenarios are discussed in the security analysis, which defines requirements and shows their fulfilment. The feasibility of the proposed system is demonstrated by the implementation of a prototype.

The outcome of this work can be seen as a step towards a more secure interaction with the internet in modern life.

Kurzfassung

Passwörter in einer sicheren Umgebung verschlüsselt zu speichern, ist heutzutage bereits Praxis. Jedoch ergibt sich immer öfter die Notwendigkeit, Passwörter mit anderen zu teilen. Das daraus resultierende Problem, verschiedene kryptographische Schlüssel zu verwalten wird meist mittels einem zentralen Key-Server gelöst. Im Falle eines Angriffs, könnten in solch einer Konfiguration sensible Daten gestohlen werden. Aus diesem Grund müssen diese zentralen Server in einer sicheren und geschützten Umgebung betrieben werden. Dies ist aus verschiedenen Gründen oft nicht möglich, da nicht jeder eigene Server betreibt oder die Sicherheit nicht garantieren kann.

In dieser Arbeit wird ein Password-Management System vorgestellt, welches diese Probleme löst, da es auch in einer unsichereren Umgebung betrieben werden kann. Dies wurde erreicht, indem ein verteiltes Verschlüsselungskonzept konzipiert wurde. Innerhalb dieses Systems müssen zwischen den Nutzern keine Schlüssel ausgetauscht werden. Da der Datenbank Server meist das erste Ziel von Angriffen ist, wurde dieser speziell konzipiert. In der Sicherheitsbewertung wird gezeigt, dass selbst ein Diebstahl von allen gespeicherten Informationen keine Gefahr für die Vertraulichkeit der Daten ist.

Die Umsetzbarkeit des vorgestellten Systems wird mithilfe einer Implementation gezeigt. Das Resultat dieser Arbeit kann als weiterer Schritt eines sicheren Umgangs mit dem Internet gesehen werden.

Contents

Contents	ii
List of Figures	iii
List of Tables	v
List of Listings	vii
1 Introduction	1
2 Related Work	3
2.1 Cloud Password Management Security Research	3
2.2 Proxy Re-Encryption	4
2.3 Offline Password Managers on Desktop Systems	4
2.4 Password Managers on Mobile Devices	5
2.5 Conclusions	6
3 Sharing Encrypted Cloud Storage	7
3.1 Requirements	7
3.2 Architecture	8
3.2.1 Cloud Storage Module	8
3.2.2 Cloud Encryption Module	9
3.2.3 Client Encryption Module	10
3.3 Encryption Scheme	10
3.4 Communications	12
3.4.1 Communication Workflow	12
3.4.2 Trust Classification	14
3.5 Conclusions	15
4 Background	17
4.1 CrySIL	17
4.1.1 Structure	17
4.2 Web Technologies	18

5	A Secure Cloud Password Manager	23
5.1	Specification	23
5.2	Building Blocks	23
5.2.1	Server	23
5.2.2	Client	25
5.3	Architecture	26
5.3.1	Feature Overview	26
5.4	Component Interaction	28
5.4.1	Create and Update User	28
5.4.2	Create Group	29
5.4.3	Update Group Permissions	30
5.4.4	Create and Update Password-Entry	31
5.4.5	Read Password-Entry	32
5.5	CrySIL Implementation	33
5.5.1	HTTP Interface	34
5.5.2	Session Management	35
5.5.3	Operations	35
5.6	SPM Server Implementation	39
5.6.1	Structure	39
5.6.2	Communication	41
5.6.3	Core and Management	46
5.6.4	Database	48
5.6.5	Permission and Session Management	50
5.7	SPM Client Implementation	52
5.7.1	Structure	53
5.7.2	Session Management	56
5.7.3	Web Security	58
5.7.4	Client Cryptography	58
5.7.5	CrySIL Protocol	62
5.7.6	Interaction	65
6	Evaluation	71
6.1	Component Classification	71
6.2	Assets	71
6.3	Threat Agents	73
6.4	Assumptions	73
6.5	Considered Attacks	73
6.6	Residual Risks	77
6.7	Performance	77
6.8	Conclusions	78
7	Conclusion	79
	Bibliography	81

List of Figures

3.1	Component Overview for Sharing Encrypted Cloud Storage	9
3.2	Structure of the Protected Wrapped Key	11
3.3	Generation of a New Group	12
3.4	Encryption of Data Items	13
3.5	Decryption of Data Items	13
3.6	Policy Update of a Group	14
4.1	Overview of the Internal Structure of CrySIL	18
5.1	Overview of Trust Zones in the SPM Environment	27
5.2	Component-Interaction During User Creation	29
5.3	Component-Interaction during Group Creation	30
5.4	Component-Interaction During Group Permission-Update	31
5.5	Component-Interaction During Password Generation	32
5.6	Component-Interaction During Password Decryption	33
5.7	CrySIL Pipeline	36
5.8	Protected Wrapped Key	37
5.9	SPM Server Overview	40
5.10	Simplified Class Diagram of the SPM Server	41
5.11	SPM Client Overview	53
5.12	Galois-Counter-Mode	59
5.13	Typed Arrays in JavaScript	62
5.14	Generation of a New Password-Group	66
5.15	Update Process of a Password-Group	67
5.16	Encryption Process of a Password-Entry	68
5.17	Decryption Process of a Password-Entry	68
5.18	Clientside CrySIL-Authentication Process	69
6.1	Time consumption of the Encryption Process	78

List of Tables

3.1	Security Requirements For The Proposed Encryption System	8
6.1	Summary of Attack Vectors and their Impact	78

List of Listings

4.1	SOP AJAX example	19
4.2	Cross Origin JavaScript Request	19
4.3	Simple CORS request	20
4.4	Preflight CORS Request	21
5.1	Angular Controller Example from official Documentation	25
5.2	Angular View Example from official Documentation	25
5.3	CrySIL HTTP-ProtocolHandler	34
5.4	CrySIL Schema Validation File	34
5.5	CrySIL Command Execution	35
5.6	Schema Template for a CrySIL Decryption-Request	38
5.7	Exposing Rest-Interfaces via JAX-RS	42
5.8	Example Rest-Service	42
5.9	Container-Class for the addGroup() interface	43
5.10	Java Servlet Filter to insert CORS Headers	46
5.11	Service Front-End for database manipulation	48
5.12	Base-class of SPM Server entities	49
5.13	Interceptor Used to Check Permissions	51
5.14	Shiro Configuration	52
5.15	Application-Route of SPMClient	53
5.16	Definition of Authentication Methods	56
5.17	Usage of Authentication Methods in the Authentication Handler	57
5.18	SPM Inner-Key Generation Result	59
5.19	SPM Inner Encryption Result	60
5.20	SPM Inner Decryption Result	60
5.21	SPM Key Export	61
5.22	SPM Key Import	61
5.23	Serialization Process for ArrayBuffers	62
5.24	Deserialization Process for ArrayBuffers	62
5.25	Interface functions of the crySILService	63
5.26	Interface functions of the crySILService	64
5.27	Request Header for CrySIL messages	64
5.28	Generation of a New Policy File	65
5.29	Generation of Password-Groups on the Client	66

Chapter 1

Introduction

The most common way of targeted attacks are made possible due to the fact, that many people reuse the same password across most of their online accounts. To stop the success of those attack types, the awareness of users needs to be raised. One way to add additional security is a multi factor authentication mechanism based on one-time passwords or certificates. However, many online services do not provide such kinds of mechanisms. Therefore it would be advisable to use different passwords with high entropy for each individual account. A problem with this approach is that it is very hard for a person to remember a high number of long and secure passwords.

Another process of ensuring the security of authentication is to periodically update the passwords. Different studies [Cranor, 2016] [Schneier, 2016] have shown that enforcing such password policies often even weaken the password strength. This implication is also based on the fact that users are not able to remember different strong passwords and tend to choose weak and strongly related passphrases.

For that reason, password managers currently notice an increase in popularity. This kind of software can support users in this task by automatically generating and persisting the authentication information. Since those applications contain aggregated and highly sensitive information, protection from unauthorized access needs to be ensured. The most common way to protect the information stored in this applications from unauthorized access is to encrypt the content based on a master password the user needs to enter. As a result, people will have to remember only one strong password in order to manage a wide range of different accounts. Still, password management applications can lead to security issues in case of implementation mistakes or if the user behaviour is wrongly estimated. A popular case for such an incident was the hack of the online password management provider *LastPass*. Attackers have gained access to the database of the online password management system and stole an unknown amount of information.

Another important requirement is the necessity to share passwords with other people. If password managers do not include a feature to share information, users are very likely to do so by other means. Often this is done by simply communicating sensitive information over email or chat applications. This is very dangerous and has been the attack vector for many social engineering attacks in the past.

The purpose of this work is to design and implement a system that solves the described problems with the means of cryptography. The application created within this master thesis is called *Secure Password Manager* (SPM), a distributed system that enables users to collectively manage and distribute passwords. The declared goal of this work is to reduce the trust relation between user and server. One of the main challenges was to introduce an untrusted server, that stores passwords without the need to distribute user specific keys among all users in the system. The Secure Password Manager solves many problems by splitting up the logic into a distributed system with three main components: the SPM Server, CrySIL¹ and the SPM Client. The advantage of this design is, that each component can be assigned a different

¹CrySIL is a system that is used for several cryptographic operations in this work. It will be closely described in Chapter 4

level of trust. It will be shown, that the presented system is able to provide a secure way of persisting and sharing password information and is still robust against various kinds of attacks

The first part of the thesis covers the theoretical and technical background of this work. In the beginning of this chapter, the requirements of the implemented system are defined and an abstract concept of sharing encrypted data in the cloud is introduced. Based on the requirements, the actual encryption workflow is presented. At the end, the actual encryption workflow is presented from an abstract perspective.

The second part of the thesis describes the technical implementation of the previously described concepts. In these chapters, a close look is taken at the implementation details and the challenges that had to be solved. At first, the underlying systems on which the SPM is based on, will be introduced. CrySIL, which is a cryptography provider and a fundamental part of the SPM system, will be covered at first. After that, a list of important web technologies and standards, which affected the implementation of SPM, are discussed. Before the actual implementation of SPM is described, the motivation and the building blocks are presented. The extensive implementation specification covers the different components, the communication flow and the encryption mechanism.

At the end of the thesis, the results of the security evaluation are presented. It will be shown, that the defined requirements could be met and the idea of adding security by introducing different levels of trust was successful.

Chapter 2

Related Work

This chapter provides an overview of ongoing research of shared encryption schemes. The first presented paper analyses the security of currently popular online password managers. Afterwards a work with a different approach of how encrypted information can be shared, is described.

2.1 Cloud Password Management Security Research

Since cloud password managers gained much popularity in the last years, a number of researchers have investigated the security of those applications.

Zhao, Yue, and Sun [2013] reviewed two popular web password managers: *LastPass and RoboForm*. The analysed applications are so called *Browser and Cloud based Password Managers*. They are cloud based in the sense they store the passwords on a server. They are browser based because extension for most popular browsers are available, offering auto form filling functionality. Both applications upload stored passwords to server, where they are encrypted and stored. Whenever a user needs access to one of the passwords, an authentication step against the provider-server needs to be performed. Afterwards the users's database can be queried. However to guarantee that the passwords are also available if there is no connection to the internet, the data is encrypted using a master password and stored locally on the client as well.

The main focus in the security analysis was put on the client side implementation of the password managers. Therefore, the way how data is stored in the SQLite database and communicated to the server was researched. Both reviewed password managers had major security flaws in their offline mode functionality. In contrast to the threat assumptions of this work, the server was mainly trusted in the presented analysis. However one of the security criteria stated, that the server should at least not be able to observe or decrypt the uploaded passwords. The paper shows, that under certain circumstances, passwords are stored entirely unencrypted at the user agent. In several client-side attack scenarios, the attacker was able to observe the plain-text passwords or brute force them withing seconds. In some scenarios, the attacker was even able to recover the master password. Since the encryption key was generated using a key derivation function with the master password as input, that is a very critical flaw. One of the two providers also sends the password information unencrypted to the server. Using transport layer security, the observation of the passwords should not be possible for an outside attacker, however the server itself is able to monitor all incoming traffic. In this case, even an honest but curios server assumption fails the security analysis. As a result of the security analysis, the following systematic flaws could be identified:

Passwords stored in the cloud are only encrypted at the server in many cases. This may protect the information from being revealed by database attacks, however the transport is not secure. Furthermore the password management server provider and potentially all insiders can gain access to the password

information.

The offline functionality that offers the possibility to access the password without a connection to the server, stores the passwords locally in an unsecure manner. The authors have shown, that the master password that was used as input for the encryption key, could be revealed.

2.2 Proxy Re-Encryption

Ateniese et al. [2005] describe a system to share encrypted content using proxy re-encryption. Based on proxy re-encryption, the authors explain different scenarios in which this system is used to share encrypted information.

Proxy re-encryption, is a function that transforms the ciphertext encrypted with someones public key, to a ciphertext that can only be decrypted using someone else's private key. This workflow is very well suitable for forwarding encrypted emails or granting access to encrypted files on a shared medium. As the authors point out, the designed proxy re-encryption system is partly based on the principle of delegating decryption rights presented by Mambo and Okamoto, 1997. The stated main advantage of the proxy re-encryption scheme is, that delegations are unidirectional, meaning that a delegation from one party to another does not require a delegation in the reverse direction. Another main benefit is, that the proxy server needs little to no trust, since it is not able to decrypt the data by its own.

The presented *Unidirectional Proxy Re-encryption Scheme* is based on a two-staged decryption process with two different secret keys. The encryption scheme itself is a variation of Elgamal. The secret key s of Alice is divided into two shares s_1 and s_2 . One key share is given to the proxy and one is given to Bob. If Alice shares a ciphertext with Bob, the proxy can partially decrypt the message and Bob can complete the decryption and gain access to the original information.

As an example of the presented encryption scheme, the setup of an encrypted file storage is described. The files are encrypted with symmetric encryption keys and stored at a shared block store. The file encryption keys are further encrypted using the public key of a user. To delegate the decryption, the owner of the file sends an encrypted block to the proxy server which re-encrypts it for another user.

Unidirectional Proxy Re-encryption Scheme could also be used to share passwords or any other protected information among a group of users. However some problems are encountered, that were solved in SPM as described in later sections. To share a block of data with a group of n people, it is necessary to encrypt the same data n times. This leads to significant run-time and space costs.

Further the proxy re-encryption scheme is based on a single writer - many reader principle. If therefore all of the members of a group should be able to change the information, the setup has to be duplicated for each user. This would lead to n^2 keys.

The problem of further delegations is left open in the presented paper. The cryptographic process therefore allows each user to further delegate the decryption permission to any other user. SPM offers the possibility to grant only special users the permission to invite new users to shared information.

2.3 Offline Password Managers on Desktop Systems

Traditional offline password management applications are in use since the beginning of the internet in the 90s. Usually they offer little or no integration with other applications, which leaves little space for side channel attacks. One of the currently most popular offline password managers is *KeePass*. It uses the Advanced Encryption Standard (AES, Rijndael) and the Twofish algorithm to encrypt its password databases. For the key derivation function SHA-256 is used, in order to transform the master password into the data encryption key. Subsequently the key encryption key, as the name suggests is used to encrypt the password database. The database is encrypted as a whole, therefore not only the passwords are protected but also the connected meta data like user names or URLs. Additionally advanced protection

mechanism like *In-Memory Passwords Protection* are in place as well. In-memory protection means, that the passwords are also encrypted while the application is running, therefore a dump of the currently running process does not lead to information leakage.

After all, much effort was put into security consideration of desktop password managers. However since those password managers are offline, there is no simple possibility to share passwords with other people, other than sharing the whole database together with the master password or directly sharing the password in plain, which often happens through an insecure channel. Both options do not seem acceptable with regard to the needs of many users nowadays.

2.4 Password Managers on Mobile Devices

Since already a wide range of people access their online services with smart phones, password managers for mobile applications need to be targeted as well. Neither one of the most popular mobile operating systems Android nor iOS do provide a way to store passwords out of the box. The Android built in browser is able to store passwords for web forms, however those are stored unencrypted in a file called *webview.db*. To satisfy the needs of many mobile platform users, a wide range of different password apps are offered in the app stores. There exist plenty of *Top 10 Mobile Password Manager* lists on the web, however those rankings focus on features and usability and barely on security aspects. Ziegler et al. [2014] analysed 45 password safe apps from the Android app store with partly stunning results. In 8 of the analysed apps, serious security problems were found. Most of the security vulnerabilities found in this paper were related to the used key derivation function. This type of function is used by the application, to transform the master password into a key, which is used to encrypt the local database. As the authors point out in the conclusion of the research, the following problem categories were discovered:

- **No Encryption**
In some of the analysed apps, the content was not encrypted at all, therefore all passwords could be extracted without any additional effort.
- **No Key Derivation Function**
In some cases no or a faulty key derivation function was used, therefore brute force attacks could be mounted in a very effective way.
- **Low number of iterations**
If a low number of iterations is used, effective brute force attacks can be applied even if the used key derivation function is adequate.
- **Static salt**
In cases of static salt usage, the precalculation of password hashes is possible and therefore passwords can be extracted using rainbow table attacks.

Another critical security implication regarding the clipboard of mobile operating systems was analysed by Fahl et al. [2013]. On mobile devices the workflow of entering passwords is usually to open the password app, copy the credentials into the clipboard and pasting it into the target application's password field. On Android the access to the clipboard is not restricted in any way, therefore any application is able to read or write the clipboard. Android also provides a callback function, that notifies apps in case the clipboard was updated. Malicious apps can register themselves at this callback interface and copy the values from clipboard. Another service of the Android operating systems communicates which app is currently in the foreground. This information can be used by the malicious app to determine for which application the password will be used by the user. The combined information of the password in the clipboard and the app which was used after the copy event, can be sent to a server afterwards. At the server, the password information can be automatically validated and stored in case of success.

2.5 Conclusions

Based on the findings stated in this section, the currently used mechanisms for password storage do not meet the needs of most users. Currently used Web based password management applications work with proprietary protocols and server environments, it is hard to verify the overall security. However it could be shown, that information can be leaked due to weak local protection or transport security. Especially applications for mobile devices seem to be error prone. Desktop and mobile password safes do not offer the feature to share passwords with other people. Therefore even if the password management was implemented acceptable, the transport of passwords between users happens over insecure channels. The application which is presented in this work, is able to provide the same level of security as an offline password manager with the additional opportunity to access it over the internet and share information with groups of different people.

Chapter 3

Sharing Encrypted Cloud Storage

This chapter will describe a system, that is able to securely store information among groups of users. In the beginning, different requirements which have to be fulfilled by the system, are defined. Based on these requirements, the general system composition and the encryption patterns are described afterwards. Therefore the resulting proposal for an encryption system should meet the defined goals.

3.1 Requirements

This section covers the requirements for a system that is able to share encrypted information over the cloud. The fundamental task of this encryption system is to enable the user to share information with other people. In the following description, users that share information are simply called a *group*. Any entity outside of this group should not be able to gain access to this data. Also any unauthorised manipulation of the information should be detected and communicated to the group.

The shared information has to be available to any group-user at any time. The location of the cloud storage is not viewed as a trusted entity. This means, that unauthorised users may observe or manipulate the stored data. Also the transmission of information is seen as potentially insecure.

Only some special users should be allowed to invite new users to a group. Consequentially users should not have access to the key which is used for encryption. If this would be the case, the access to a group could not be controlled anymore, since any user could simply pass the encryption keys to group outsiders. Additionally, those special users should also be able to remove other users from the group. In this case, the removed user should not be able to decrypt group information any longer.

The entity which is in charge for the encryption process, should own the data encryption key. This cryptographic key must not leave the component. Export functionality is therefore not available to any user at any time. Furthermore, the authentication of users should be done at this entity. If therefore a users is willing to encrypt or decrypt shared information, authorisation information needs to be provided. The user provided data, that should be encrypted and shared later on, needs to be protected before it is transmitted to the encryption component. Therefore, this entity must not be able to observe the input data in plain.

Req. ID	Description	Involved Entity
1.	Shared information is available to any user in the group	all
1a.	Cloud Storage Module and Cloud Encryption Module are located in the cloud (untrusted locations)	Cloud Storage Module, Cloud Encryption
2.	Shared information is not available to group-outsiders	all
2a.	Shared information is securely stored in the cloud	Cloud Storage Module
2b.	Cloud Encryption Module can not learn from data during encryption/decryption	Cloud Encryption Module
2c.	Information can only be observed in plain at the Client Encryption Module	all
2d.	A person can only gain access to the group data, if she has the corresponding permissions	all
3.	Only special users are allowed to invite new users to existing groups	Client Encryption Module
3a.	Permission information is permanently bound to the data encryption key	Cloud Encryption Module
3b.	The client has no access to the data encryption key	all
4	The permissions of group-users can be revoked	Cloud Encryption Module
5	Encryption keys can not be extracted from the Cloud Encryption Module	Cloud Encryption Module

Table 3.1: Security Requirements For The Proposed Encryption System

Based on the above discussed properties, formal requirements listed in Table 3.1 can be extracted.

3.2 Architecture

We propose an architecture that is able to fulfil the requirement listed. Based on the requirements from Table 3.1 a distributed architecture is proposed that is illustrated in Figure 3.1. The to be described architectural concept includes three independent system components (Modules):

- Cloud Storage Module
- Client Encryption Module
- Cloud Encryption Module

Each of those components have different tasks to carry out and are assigned a level of trust depending on the environment and the data that is processed.

Figure 3.1 shows the components and their environments. It can be observed, that there are multiple client applications communicating with the central server applications.

3.2.1 Cloud Storage Module

The Cloud Storage Module is an application that is used to store data that needs to be persisted in the encryption system. This module is able to run in untrusted environments without exposing security

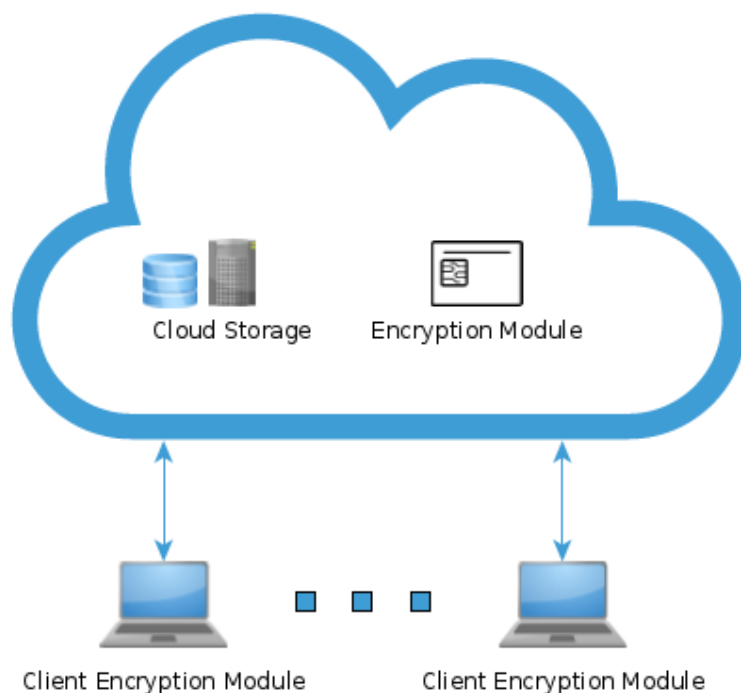


Figure 3.1: Component Overview for Sharing Encrypted Cloud Storage

vulnerabilities. With this requirement, the module is required to be resistant to unauthorised data manipulation. For that reason, all sensitive data is encrypted before it is transmitted to this component, in order to detect manipulation. Furthermore, all stored and processed data can potentially be observed by attackers. An authenticated encryption mode is used, in order to detect manipulation.

Since the permission information of this module could be manipulated by an attacker, the system security should not depend on it. The authentication and permission system of this module is only implemented for a better user experience and to stave web based attacks. If an attacker manages to manipulate permission information, only the user interface of this component is affected. For example, an unauthorized user may see a list of groups. However the protected information is not revealed by such kinds of attacks. This can be guaranteed, since the data has to be decrypted first by the Cloud Encryption Module, which works completely independent from the storage module.

3.2.2 Cloud Encryption Module

The Cloud Encryption Module which should act similar to a smart card, is used for data encryption. Keys that are stored inside this module, can not be extracted by any user in the system. To ensure, that sensitive information is not leaked by this module or during transmission, the client always sends already encrypted information. Therefore multiple layers of encryption carried out by the client and encryption module need to be implemented. The exact definitions of the layers are stated in Section 3.3.

The permission information that is used to decide whether a user is allowed to decrypt shared data, should not be stored at this location. Further on, the permission information should be directly bound to the data encryption key. To accomplish this, the data encryption key itself is encrypted together with the permission information using a key encryption key. The resulting cipher-text can be stored at the Cloud Storage Module without exposing further information. The combination of data encryption key and permission information will be called *wrapped key* from now on. The wrapped key is unwrapped

(decrypted) every time the encryption module needs to check permissions or use the data encryption key for cryptographic operations.

Two distinct types of permissions are managed in the wrapped key. The *group permission* tells if a user is allowed to decrypt. Each user in a group has this permission. Furthermore there is a *group administrator permission*, which includes users that are allowed to update the wrapped key. This has to be done if a user is invited or removed from a group.

3.2.3 Client Encryption Module

This module is the active component in the system. It communicates with both cloud services and initiates all actions that are taken. One layer of encryption, which is needed to hide the information from the encryption module, is done at the client.

The client is the only part of the system where the shared data can be observed in plain. However there is no data persisted at this location. Every time the user wants to view or share information, it is first loaded from the storage module and afterwards sent to the encryption module.

The authentication of clients is a multi-step process, since it is handled separately for the two cloud modules. If needed, the client authenticates itself against one of the cloud modules. The used credentials or certificates should not be shared or derived between those modules, since they are placed in different trust zones. The authentication information of the Cloud Storage Module is stored¹ in the Module's included database. In contrast, the Cloud Encryption Module does not store any permission or authentication information, this information is stored in a file that is not stored at this server. This file is called the *Wrapped key* and will be extensively described in the next section. The authentication process itself is explained in Section 3.3.

3.3 Encryption Scheme

This section is going to explain the cryptographic encryption system of the Encrypted Cloud Storage. Furthermore, the structure of exchanged data is discussed.

The encryption scheme is mainly based on the following four information blocks:

- **Wrapped key**
Protected container including an encryption key and the permission information for the Cloud Encryption Module. The only entity that is able to unwrap this file and observe the contents is the Cloud Encryption Module itself. This section will describe how the encryption key inside this file is used for the second layer data encryption. The mentioned permission information is located in the so called *policy file* and tells the Cloud Encryption Module which user is permitted to carry out a certain operation. Furthermore, the authentication information in form of username-password tuples is contained in the policy file as well. Figure 3.2 illustrates the internal structure of the wrapped key.
- **First layer encryption key**
Key for the first layer encryption of shared information. The encryption of shared data is a stream lined two step encryption process. Each entry of shared information is first encrypted by the Client Encryption Module and afterwards encrypted a second time by the Cloud Encryption Module. In Equation 3.2, this process is defined.

¹If a password is stored in the database, an iterative hashing approach like PBKDF2 [Kaliski and Okamoto, 2000] should be used.

- Second layer encryption key
Key for the second layer encryption of shared information. This key will be referenced as the data encryption key (DEK).
- Double encrypted shared data
Shared information in protected form. At this state, the information is encrypted twice and can be stored in the database of the Cloud Storage Module.

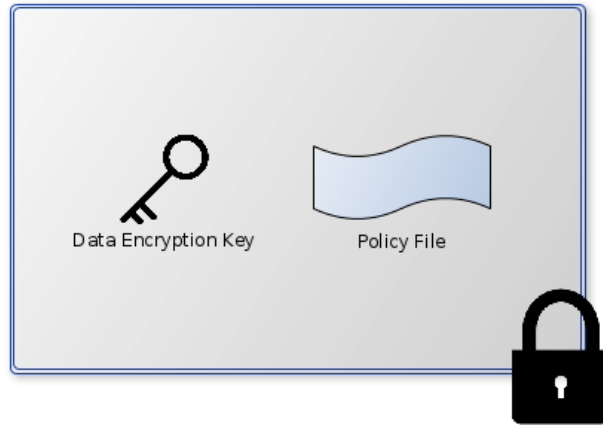


Figure 3.2: Structure of the Protected Wrapped Key

In the following paragraphs, the information that is exchanged between the system components, is described in detail. The wrapped key contains the data encryption key (DEK), which as the name suggests, is used to encrypt the shared data. Furthermore, the policy file containing user permissions for the corresponding key, is included in the wrapped key as well. A policy file must include the information whether a user is allowed to use the DEK for the decryption of protected shared data items. Additionally it contains permission information for users that are allowed to update the policy file itself. Each policy file has its own nonce that is updated every time the policy is altered. This nonce is also stored at the Cloud Encryption Module. Therefore if the permission of a user is checked, also the nonce inside the policy file is matched with the stored version of the nonce, in order to guarantee that the viewed version of the policy file is up to date. Equation 3.1 illustrates the structure of the wrapped key. Although the wrapped key is stored at the Cloud Storage Module, only the Cloud Encryption Module has access to the contents inside the wrapped key, since the key encryption key (KEK) is not known to any other entity.

$$wk := E_{KEK}(DEK, Policy) \quad (3.1)$$

In the following, the properties of the protected shared data are discussed. As mentioned in the requirement discussion, the shared data is stored in untrusted environments. Therefore it is cryptographically protected in form of two independent encryption rounds. Equation 3.2 defines the encryption process of protected shared data entries. The first layer encryption is performed at the Client Encryption Module and the second round is done at the Cloud Encryption Module.

$$protectedSDE := E_{DEK}(E_{K_{firstLayer}}(shared_data_entry)) \quad (3.2)$$

To access the contents of the protected shared data, the decryption process using the DEK needs to be initiated. The DEK itself is located inside the wrapped key and can therefore only be revealed by the owner of the KEK. This in other words means, that the decryption process of the protected shared data can only be started by the Cloud Encryption Module, because it is the only entity with access to the KEK. No other system component is able to observe the contents of the wrapped key. However the Cloud Encryption Module has no access to the first layer encryption key $K_{firstLayer}$ and can therefore not access

the shared data in plain. Therefore the decryption of protected shared data entries is a pipeline process. At first the client requests the Cloud Encryption Module to decrypt the second layer of the protection. When this task is finished, the Client Encryption Module continues to decrypt the first layer. As a result of this setup, the decryption process can only take place if both, the Cloud Encryption Module and the Client Encryption Module perform a decryption step.

The first layer encryption key, that is used in Equation 3.2 for the inner layer encryption of shared data entries, is located at the Cloud Storage Module. Again the Cloud Storage Module can not use this key to learn from the stored data entries, since they are encrypted with the DEK after the first encryption round.

One special property of this process is, that only the Client Encryption Module is able to observe the plain information. The resulting encryption scheme enforces the fusion of all three system components in order to decrypt the shared data. Furthermore it is guaranteed, that both cloud modules can not observe the shared information during the encryption and decryption process.

3.4 Communications

This section is going to describe the communication protocol between the three defined system components. The order of exchanged messages and the information that is sent is described for every task that is carried out by the encryption system.

3.4.1 Communication Workflow

This subsection covers the communication process during typical tasks of the encryption system. In the Sequence Diagrams 3.3, 3.4, 3.5 each arrow illustrates a call to the function, which is labelled next to the active block. The information that may be sent or returned from the activated function is placed in square brackets next to the interface function names.

Figure 3.3 illustrates the generation of a new group, in which data is shared. When a user with the corresponding permissions is willing to create a new group, a data encryption key has to be generated. To do so, the client sends a request to the Cloud Encryption Module in order to generate a new wrapped key, that will contain the data encryption key. The newly generated policy file inside the wrapped key will contain a nonce and all users that are attending the group. The nonce inside the wrapped key is changed each time the policy file is updated. After the wrapped key is returned to the client, it is stored at the Cloud Storage Module, in order to enable other group members to use it.

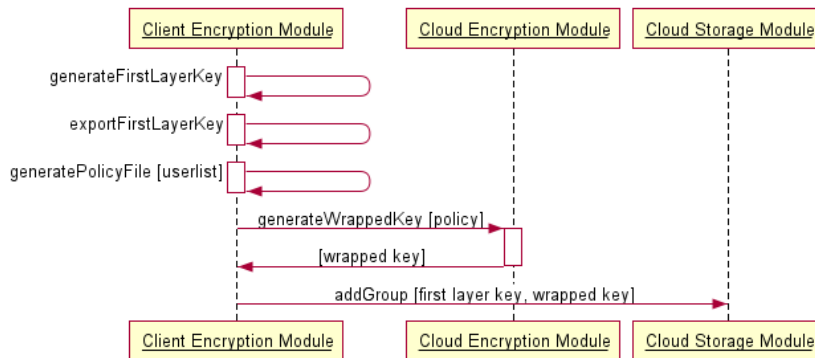


Figure 3.3: Generation of a New Group

Figure 3.4 shows how shared data items are encrypted. To accomplish this task, the cryptographic input data is loaded from the Cloud Storage Module. If one of the group users adds a new data entry to the

shared container, the first layer encryption is performed at the Client Encryption Module. The resulting cipher text (first layer encryption) is uploaded together with the wrapped key to the Cloud Encryption Module. At this component, the wrapped key can be unwrapped and the contained data encryption key is used for the second layer encryption. As a result of this operation, the protected shared data entry is sent back to the client and can be uploaded to the Cloud Storage Module. Any user inside the group, can now access the shared data entry. In order to do so, the following information is requested by the client:

- protected shared data
- wrapped key
- first layer encryption key

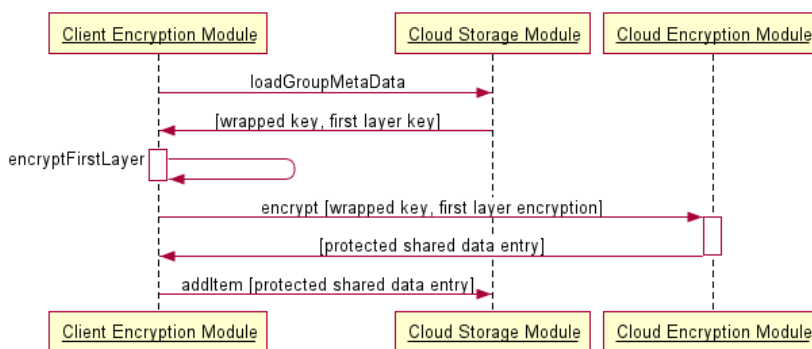


Figure 3.4: Encryption of Data Items

To observe the protected shared data, it has to be decrypted by the Cloud Encryption Module. This module will only start the decryption if the authenticated user is permitted to do so in the policy file. In Figure 3.5, the steps of the decryption process are illustrated. After the decryption was performed by the Cloud Encryption Module, the client can use the first layer encryption key to observe the original plain text.

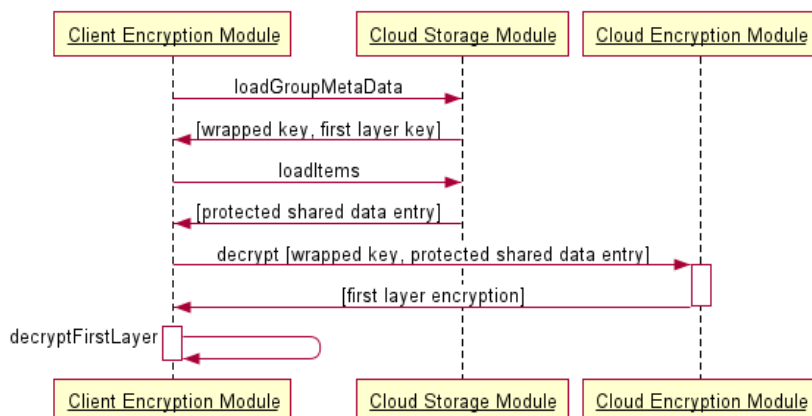


Figure 3.5: Decryption of Data Items

Users with special permissions are able to administrate groups by adding new users to the group or removing existing ones. This can be accomplished by updating the wrapped key, which can only be done by the owner of the key encryption key (Cloud Encryption Module). Figure 3.6 illustrates the sequence

of steps during the update process of a group. Before the policy file is updated, the Cloud Encryption Module checks if the authenticated user is permitted to do so in the policy file. Every time a policy file is updated, the contained nonce is changed as well. This nonce is also securely stored at the Cloud Encryption Module. Each time the Cloud Encryption Module is checking the permissions of a user, the stored nonce is compared with the nonce contained in the policy. If they do not match, the permission check must fail. Therefore it is guaranteed, that users can not use old versions of the wrapped key to decrypt protected shared data entries.

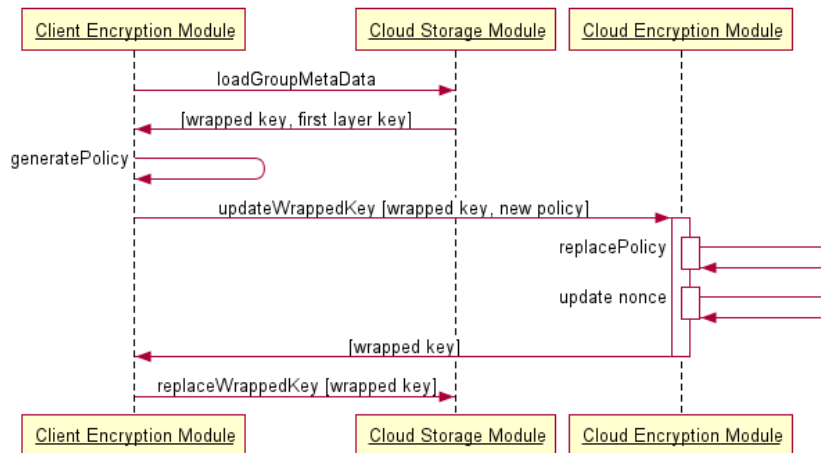


Figure 3.6: Policy Update of a Group

3.4.2 Trust Classification

Based on the requirements stated in the last section, each of the defined modules is assigned a level of trust. The trust level describes what a system component is allowed to observe. The following definitions are combined with descriptions that clarify why a certain trust classification holds for the presented encryption system.

- **Client Encryption Module - trusted**

The protected shared data is finally displayed to the user at the Client Encryption Module, therefore this component needs to be able to observe the original information. The only limitation from Table 3.1 regarding this module is *Requirement 3*. This requirement defines, that only special users are allowed to forward permissions. As defined earlier, the Client Encryption Module has no access to the content of the protected wrapped key. Permissions can only be updated if the policy file inside the protected wrapped key is updated. Therefore the client is not able to do so.

- **Cloud Encryption Module - honest but curious**

This module is classified as honest but curious, which means that all observed information can potentially be used by an attacker, however the module does not violate the defined protocol by performing unintended actions. In the presented design, the Cloud Encryption Module is not able to observe the shared data in plain. Furthermore, a potential attacker is not able to reveal the shared data, using all observed information at this module. This is because the first layer key which is needed for the decryption, is not known at this component.

- **Cloud Storage Module - untrusted**

This module is the most critical part of the system in terms of trust classification, since one of the major goals is to mount the storage module in untrusted environments. Therefore this component is

not allowed to store or observe data that may leak information about the shared data. Furthermore, this component should not be able to decrypt shared data or update permissions. Since neither the wrapped key content nor the second layer encryption key is known at this module, the classification holds for this module.

3.5 Conclusions

The encryption system presented in this section, defines three independent modules that communicate with each other, in order to encrypt or decrypt shared information. As described, neither one of the server modules is able to observe the plain information, which was one of the most important requirements for the new system. Furthermore, users that are removed from existing groups, can not longer decrypt the shared information of the group, because the permission information is protected and can only be updated or observed by the Cloud Encryption Module. Additionally, the Cloud Encryption Module is able to recognize the newest version of the policy file, that is used to persist permission information. Therefore users that are removed from groups, can not use old versions of the policy file in order to decrypt information.

Chapter 4

Background

The implementation of the Secure Cloud Password Manager is based on several technologies. This chapter is going to give an overview of the important building blocks.

4.1 CrySIL

CrySIL was developed at the Graz University of Technology at the institute of applied information processing and communications (IAIK).

As Reimair, Teufl, and Zefferer [2015, page 2] say:

CrySIL provides a flexible architecture to use cryptographic protocols and algorithms in a heterogeneous environment and provides secure key storage and key handling capabilities. It features build-in authentication as well as transparent off-device key storage.

In this work, CrySIL represents the Cloud Encryption Module, which is in charge to maintain the *wrapped key* and perform the second layer encryption and decryption as explained in Section 3.3. Various cryptographic operations for encryption and signature generation are provided via REST interfaces for third party components. SPM is strongly based on the services provided by CrySIL. As described in later sections, CrySIL was also extended to meet the needs of SPM.

CrySIL communicates with the client component via HTTP and subsequently routes the requests to so called *Gatekeepers*. Gatekeepers can decide if an incoming request needs authentication or deny the access if the client does not have the appropriate permissions. At the end of this pipeline, there is the *Actor*, which is the component in this system, that carries out the command addressed by the client. CrySIL provides a flexible protocol, that allows the execution of operations on multiple Actors. The cryptographic keys, that are used in such commands can either come from a *keyStore* stored at an hardware security module (HSM) or are provided by the client itself. In the case, the client provides keys for the commands, those keys are protected in such a way, that only CrySIL is able to observe and use them.

4.1.1 Structure

The internal architecture of CrySIL is very flexible in order to support a wide range of different protocols. In this work only a subset of the interfaces and functions are used. The most fundamental entities of CrySIL can be summarised to:

- Receiver

The Receiver is the endpoint, that takes requests from third party entities. Each of those Requests

has to comply to a defined format. In order to support a wide range of applications, many different interfaces are available in CrySIL. SPM uses the *HTTPReceiver* to communicate with CrySIL. This Interface transforms HTTP Requests into a CrySIL specific request form and forwards them to the Router.

- Actor
Actors are the place where requests are handled and the actual cryptographic operations are carried out. Each Actor has a list of commands that can be identified using a unique name. This identifier has to be present in each request to CrySIL. Whenever a permitted request reaches the Actor, the corresponding command is executed and the resulting response is returned to the calling Receiver.
- Gatekeeper
Before any request is processed by an Actor, its Gatekeeper checks if the request needs authentication. If so, a response including a unique sessionId is returned. That tells the client to start the authentication process. When the client successfully finished the authentication step, the original request, which is stored along with the sessionId, is forwarded to the Actor. In this work the *XACMLGatekeeper* is used. It is able to process XACML files in order to decide if a client is allowed to use specific commands or keys.
- Router
Each Router maintains a list of different Actors. The task of a Router is to forward requests to the correct Actor and propagate the resulting response back to the Receiver. Responses can be intercepted and manipulated at this point, in order to allow special behaviour regarding authentication.

Figure 4.1 shows the communication structure of the different described CrySIL entities.

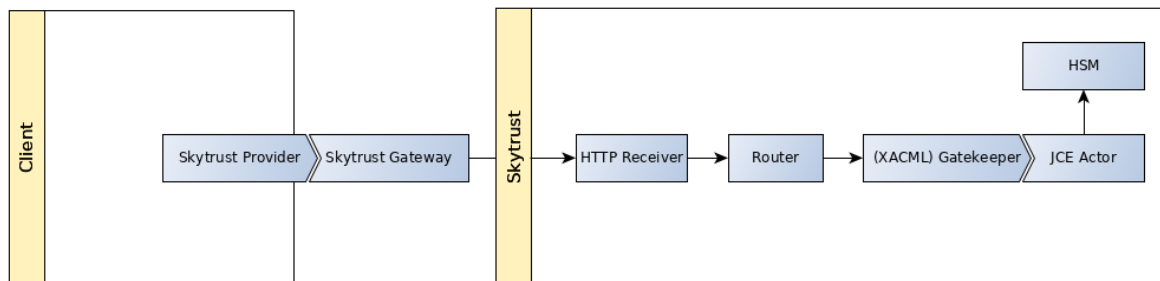


Figure 4.1: Overview of the Internal Structure of CrySIL

4.2 Web Technologies

Since SPM is a web based application, several limitations have to be considered. The following paragraphs will describe some of those limitations. At the end of this section the Webcrypto API, which is one of the most important technologies in this work, is introduced.

SOP

The Same Origin Policy (SOP) [Barth, 2011d] is a measure within web browsers to protect users from unauthorised access. One important part of SOP is that resources such as JavaScript files can only access other resources within the same origin. As a consequence, an Asynchronous JavaScript and XML (AJAX) call, may only request resources of servers with the same origin. The origin [Barth, 2011e] in this context is defined as the combination of *schme*, *host* and *port*.

User agents interact with content created by a large number of authors. Although many of those authors are well-meaning, some authors might be malicious. To the extent that user agents undertake actions based on content they process, user agent implementors might wish to restrict the ability of malicious authors to disrupt the confidentiality or integrity of other content or servers.

As an example, consider an HTTP user agent that renders HTML content retrieved from various servers. If the user agent executes scripts contained in those documents, the user agent implementor might wish to prevent scripts retrieved from a malicious server from reading documents stored on an honest server, which might, for example, be behind a firewall. [Barth, 2011c]

Another outcome of this strategy is, that JavaScript functions are not out of the box able to make Ajax calls to any other server. Therefore a script loaded from `http://www.example.org` will not be allowed to access resources at `http://sub.example.org`. Listing 4.1 illustrates this issue:

```
// document origin: https://example.org
var xhr = new XMLHttpRequest();
xhr.open('GET', 'http://sub.example.org');
xhr.send();
var xmlResp=xmlhttp.responseXML;
//this request fails with a user agent error
```

Listing 4.1: SOP AJAX example

As a consequence of this user agent behaviour, not only attackers but also developers who are interested in accessing resources in a cross-origin manner are affected.

Cross-Origin Resource Sharing

To overcome the limitations stated in the last paragraphs, different strategies were found:

- **Disabling Web Security**
As developers are facing this problem the first time, many react by simply disabling the web security features of their user agents in the first place. This is not only a major security risk but also not an option for productive systems, as it would require users to reconfigure their browsers.
- **JSON with Padding**
Some resource types such as JavaScript, CSS or Images can be loaded in a cross origin manner. As a consequence it is possible to load JavaScript files from other origins. This can be used to smuggle data inside JavaScript files from third origin servers.

Listing 4.2 shows, how JSON with Padding (JSONP) can be used. The server at `http://example.org` is able to evaluate the URL query parameter `queryvar=x` and responds with a script, that calls the function `processJSONResponse` with the requested data.

```
<script>
function processJSONResponse (jsonData) {
  // process the response
};
</script>

<script src='http://example.org?
jsoncallback=processJSONResponse&queryvar=x'></script>
```

Listing 4.2: Cross Origin JavaScript Request

However this kind of cross origin resource access is not viewed secure, as it is not possible to control which scripts are able to access the information.

- Reverse Proxy

From the official Apache documentation the definition of a reverse proxy is defined as:

A reverse proxy (or gateway), by contrast, appears to the client just like an ordinary web server. No special configuration on the client is necessary. The client makes ordinary requests for content in the namespace of the reverse proxy. The reverse proxy then decides where to send those requests and returns the content as if it were itself the origin.

A typical usage of a forward proxy is to provide Internet access to internal clients that are otherwise restricted by a firewall. The forward proxy can also use caching (as provided by `mod_cache`) to reduce network usage.

The forward proxy is activated using the `ProxyRequests` directive. Because forward proxies allow clients to access arbitrary sites through your server and to hide their true origin, it is essential that you secure your server so that only authorized clients can access the proxy before activating a forward proxy.

(Apache Docs, Apache Module `mod_proxy` ¹)

Because the SOP is implemented only by web browsers, other applications such as proxy servers can freely request resources on the web. Therefore reverse proxies can be used to overcome SOP limitation, because the client always requests resources from the proxy which is the same origin. Subsequently the proxy itself forwards those requests and their responses between client and origin server.

- Cross-Origin Resource Sharing

Since many of the presented methods to circumvent the limitations of SOP are either not practical or imply high security risks, the concept of Cross-Origin Resource Sharing (CORS) [van Kesteren, 2014a] was introduced. CORS allows web servers to decide whether the delivered content is allowed to be used at other origins. Browser and server communicate via *HTTP-Headers* [van Kesteren, 2014b]:

- **Origin:** Used by the user agent to communicate the origin of the context
- **Access-Control-Allow-Origin:** Used by the server to communicate the allowed origins
- **Access-Control-Allow-Methods:** Used by the server to communicate the allowed methods
- **Access-Control-Allow-Headers:** Used by the server to communicate the allowed HTTP headers

To illustrate the CORS workflow, the situation in Listing 4.1 is used. When the user agent sends the given AJAX request to the server, the HTTP header *origin* will be set with the value `https://example.org`. The server could subsequently produce a HTTP response and set the HTTP header *Access-Control-Allow-Origin* to the same value `https://example.org`. In the case the browser supports CORS, the response will be accepted and the resource can be used in the JavaScript context. Listing 4.3 shows the corresponding HTTP request and response pair.

```
GET / HTTP/1.1
Host: sub.example.org
Accept: text/html
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Connection: keep-alive
Origin: https://example.org

HTTP/1.1 200 OK
```

¹Documentation for Apache Module `mod_proxy`: http://httpd.apache.org/docs/current/mod/mod_proxy.html

```
Date:Thu, 07 Jan 2016 12:24:13 GMT
Server: Apache/2.0.61
Access-Control-Allow-Origin: https://example.org
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: application/xml
```

Listing 4.3: Simple CORS request

This already solves a substantial part of the issues with SOP. However some problems are not considered yet. Imagine the user agent sends a HTTP request, that should change the server state (for example updating a user profile) to a server which is not aware of CORS. In this case, the data on the server will be changed and the browser will only notice that the response is not valid, since no CORS response header is present. For this reason preflight requests are introduced:

Cross-site requests are preflied like this since they may have implications to user data. In particular, a request is preflied if:

It uses methods other than GET, HEAD or POST. Also, if POST is used to send request data with a Content-Type other than application/x-www-form-urlencoded, multipart/form-data, or text/plain, e.g. if the POST request sends an XML payload to the server using application/xml or text/xml, then the request is preflied. It sets custom headers in the request (e.g. the request uses a header such as X-PINGOTHER) Mozilla Developer Documentation ²

In those cases preflight requests are sent to the server. With that strategy the browser can first check if the server supports CORS and allows only requests with the given origin. This is realised by sending an *OPTIONS* request. Listing 4.4 shows how this process looks like.

```
OPTIONS / HTTP/1.1
Host: sub.example.org
Accept: text/html
Accept-Encoding: gzip,deflate
Connection: keep-alive
Origin: https://example.org
Access-Control-Request-Method: POST

HTTP/1.1 200 OK
Date:Thu, 07 Jan 2016 12:24:13 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: https://example.org
Access-Control-Allow-Methods: POST, GET, OPTIONS
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 0
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain
```

Listing 4.4: Preflight CORS Request

When the HTTP response is received by the user agent, the CORS headers can be checked and afterwards it can be decided if the original request should be sent or an error should be thrown.

²https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

Webcrypto API

The W3C Webcrypto API [Sleevi and Watson, 2015] offers web apps access to high-grade cryptography, that is implemented inside the user agent. This does not only bring a huge security gain, but also way better performance, since the crypto operations run in native code on the client machine.

A wide range of different cryptographic algorithms are supported. Depending on which algorithm is used, a subset of the following operations can be used: [Sleevi and Watson, 2015, 18.2. Concepts]

- encrypt
- decrypt
- sign
- verify
- deriveBits
- wrapKey
- unwrapKey
- generateKey
- importKey
- exportKey
- getLength

Generated keys can also be protected from export or other unintended use by setting the *KeyUsage* [Sleevi and Watson, 2015, 13. CryptoKey interface] parameter to permitted operations.

Chapter 5

A Secure Cloud Password Manager

This chapter is going to describe the technical details of the Secure Cloud Password Manager (SPM). This application implements the specification defined in Chapter 3 and therefore also fulfils the requirements specified earlier. Section 5.2 describes the underlying technologies used by SPM and is followed by an overview of the interacting components of the implementation. Afterwards a close look into the implementation details of the three main parts of the system is taken in Sections 5.5, 5.6 and 5.7.

The abstract system described in Chapter 3 can be used to share arbitrary data among groups of users. Password managers are one of the most common use cases of sharing information in organisations. Further, sharing passwords raises the need of strong security, since this type of information is highly sensitive. Therefore it was decided to implement a system which performs this task.

5.1 Specification

As defined in Chapter 3, the *Sharing Encrypted Cloud Storage* scheme consists of three main modules:

- Cloud Storage Module
- Client Encryption Module
- Cloud Encryption Module

In this chapter, the implementation of each of those specified modules is described. The implementation of the *Cloud Storage Module* is described in Section 5.6. The *Client Encryption Module* was implemented using Javascript and is called SPM Client. The definition of this component is stated in Section 5.7. The CrySIL framework is used for the implementation of the *Cloud Encryption Module*. The details of this component are described in Section 5.5.

5.2 Building Blocks

In this chapter, the relevant technologies and tools on which the SPM is based on are briefly described. The description is divided into server-side and client-side technologies.

5.2.1 Server

- JAVA Enterprise Edition
The Java EE Platform is the specification of the different services, which together carry out a transaction based middle-ware architecture. Especially web-applications have good support and

can be implemented quickly. The SPM server runs entirely in the Java EE context because this architecture supports all needed features for operation.

- JAX-RS

Java API for RESTful Web Services (JAX-RS) is part of Java Enterprise Edition 6 and is a API-specification¹ for the use of Representational State Transfer (RESTful) web-services. The reference implementation which was also used in the SPM Server is the open source project jersey². The whole communication between the SPM Client and the SPM Server was implemented with JAX-RS. The communication details of this components will be defined in Section 5.4

- Application Server:

Since the CrySIL and the SPM Server are deployed as so called Web application ARchive (WAR) files, they run inside application servers. These application servers are able to hand over HTTP requests to the contained WAR application, which runs inside the Java context of the server.

Of course these application servers do much more then just handing over HTTP requests to simple Java Servlets. The two server applications are based on different technologies and consequently run in different application servers.

The SPM server is a Java EE application and therefore has to run on an application server that implements the Java EE specification. The *Wildfly* (formly known as *JBoss*) server was chosen as container environment. The CrySIL implementation does not need all the features offered by *Wildfly* and can therefore run inside a lightweight *Tomcat* server.

- Shiro

Apache Shiro is an open-source security framework which is able to handle session management including authentication and authorization. It is also capable of carrying out cryptographic operations such as securely storing password-hashes. The Shiro-Project states that their *four cornerstones of application security* are:

- **Authentication:** Sometimes referred to as 'login', this is the act of proving a user is who they say they are.
- **Authorization:** The process of access control, i.e. determining 'who' has access to 'what'.
- **Session Management:** Managing user-specific sessions, even in non-web or EJB applications.
- **Cryptography:** Keeping data secure using cryptographic algorithms while still being easy to use.

Especially Java based web-applications benefit of the Shiro features. In the SPM Server project, authentication is handled for each REST interface separately via Java annotations. This works very clean by simply defining roles, which can be assigned to users. These roles can be enforced, when the client tries to call a REST method. Furthermore, the passwords are hashed by Shiro with a configurable algorithm (in the case of SPM this is a SHA-256 Hash with 1024 iterations and a salt).

- IAIK-JCE

The Java Cryptography Extension is a Java API Interface for providing basic cryptographic operations such as encryption, hash generation or key management. There are several implementations (also called providers) of the JCE interface, one of the most known is the IAIK-JCE:

¹<https://jax-rs-spec.java.net>

²Further information about the jersey project can be found under <https://jersey.java.net/>

The IAIK Provider for the Java™ Cryptography Extension (IAIK-JCE) is a set of APIs and implementations of cryptographic functionality, including hash functions, message authentication codes, symmetric, asymmetric, stream, and block encryption, key and certificate management. It supplements the security functionality of the default JDK. IAIK³

5.2.2 Client

- AngularJS

AngularJS is a dynamic HTML and JavaScript framework. Because it strongly supports the Model-View pattern, the code can be encapsulated into logical units and is therefore better maintainable.

One of the key concepts is the so called *Controller*, which is a structural entity that maintains its own scope and variables.

Those variables are automatically updated between model and view. Therefore updates to a variable, are automatically displayed in the HTML and vice versa.

In addition, Angular supports dynamic HTML templates, which automatically generate and update the Document Object Model (DOM).

```
var phonecatApp = angular.module('phonecatApp', []);

phonecatApp.controller('PhoneListCtrl', function ($scope) {
  $scope.phones = [
    {'name': 'Nexus S',
     'snippet': 'Fast just got faster with Nexus S.',
     'age': 1},
    {'name': 'Motorola XOOM with Wi-Fi',
     'snippet': 'The Next, Next Generation tablet.',
     'age': 2},
    {'name': 'MOTOROLA XOOM',
     'snippet': 'The Next, Next Generation tablet.',
     'age': 3}
  ];

  $scope.orderProp = 'age';
});
```

Listing 5.1: Angular Controller Example from official Documentation⁴

```
Search: <input ng-model="query">
Sort by:
<select ng-model="orderProp">
  <option value="name">Alphabetical</option>
  <option value="age">Newest</option>
</select>

<ul class="phones">
  <li ng-repeat="phone in phones | filter:query | orderBy:orderProp">
    <span>{{phone.name}}</span>
    <p>{{phone.snippet}}</p>
  </li>
</ul>
```

Listing 5.2: Angular View Example from official Documentation⁵

³Institute for Applied Information Processing and Communication: <http://jcewww.iaik.tu-graz.ac.at/>

⁴https://docs.angularjs.org/tutorial/step_02

The Listings 5.1 and 5.2 are taken from the official AngularJS documentation. The controller has a list of phones, that can be displayed in HTML without any JavaScript code. When the referenced variable *orderProp* is changed within the HTML, the change will be propagated in the other direction and the variable inside the JavaScript context is updated. Aside the described model binding, AngularJS offers many more features like unit-test support, dynamic imports and URL-routing.

5.3 Architecture

This section will give a brief overview of the features offered by SPM and will show how the different components, which are involved, interact with each other. Figure 5.1 illustrates these components and their corresponding trust zones. The working setup of SPM consists of the following components:

- **SPM Server**
The SPM server is responsible for storing and delivering all data. As discussed in later sections, this server is not trusted and therefore all sensitive information has to be protected against unauthorized access or manipulation.
- **CrySIL Server**
This entity is in charge of encrypting and decrypting all sensitive information. Keys, stored at this server, can not be exported. The *CrySIL Server* is seen as *Honest-but-curious*.

An honest but curious adversary is one which runs the programs and algorithms correctly, but might look at the information passed between entities. CRYPTUTOR⁶

Therefore all the information, which is observed by this entity, has to be protected in the first place.

- **SPM Client**
The client is implemented in JavaScript and is consequently executed in the browser of the user. It communicates with the two servers and is the active part in the SPM protocol. This is the only place in the SPM system where sensitive data appears in plain.

5.3.1 Feature Overview

The major goal of this project is to create a light-weight and easy-to-use password manager, that can be hosted on untrusted origins. Therefore the application has to offer all the basic features, which will be needed by users to organise and share passwords among groups of users:

- Group-Management
- User-Management
- Organising Passwords

All of those main categories will be described from the user's view in the following sections.

⁵https://docs.angularjs.org/tutorial/step_02

⁶Online encyclopedia covering theoretical cryptography: <http://crypto.cs.uiuc.edu/>

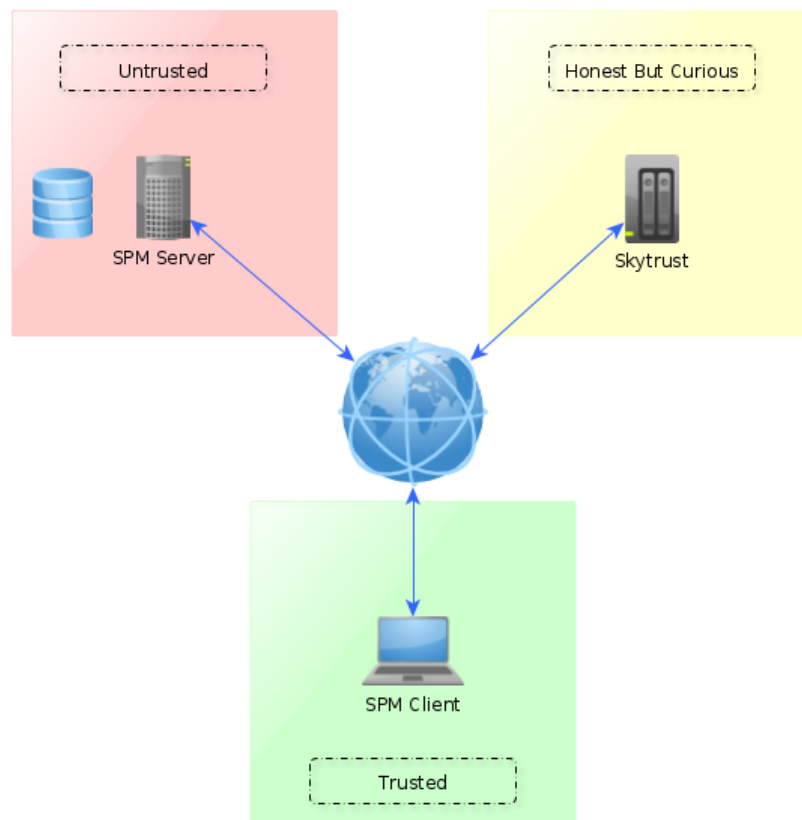


Figure 5.1: Overview of Trust Zones in the SPM Environment

Group-Management

There are different ways how passwords can be organized among users. The permission granularity reaches from letting each user access each password to manage passwords separately for each user. It was decided to use a strategy in between those extremes and therefore organize passwords in groups, because this approach satisfies the needs in most real world environments. A password group contains one to many so called password entries, which will be explained in Section 5.3.1. Each password group is connected to a list of users which are permitted to access the entries. Furthermore, there is also a list of users which are allowed to perform actions inside the groups. These users are so called *group admins*. To increase the usability of SPM, groups have names and a short description. On the main page of SPM, the users can search for groups by name and description. The resulting list only includes groups, in which the logged in user has access rights.

User-Management

A user account can have the following properties:

- **Group Admin**
The admin of a given group is permitted to invite or remove users from the group and declare other group admins.
- **Active**
As the name already suggests, the *active* property indicates whether the account is active or not. A user with an inactive account is able to log in to SPM but will not be able to read passwords or carry out any password related actions in the system.

- SPM Admin

An account with this property has SPM wide admin permissions and is able to perform user management actions, such as setting an account active/inactive or adding admin permissions to other users. Furthermore, SPM admins are permitted to delete users and groups from the database. However SPM admins are not automatically allowed to read or organize password groups. The reason for this is, that the SPM server itself is not trusted and therefore the SPM admin permission only enables users to manage the structure of the system but not to gain knowledge about secret information.

Organising Passwords

As already discussed, password entries are organized in so called *password groups*. Each of those entries include meta data and the password itself:

- Name to identify the entry
- Description of the entry
- Account name
- URL (for web accounts such as web-mail)
- Hostname (for local accounts such as the root password of servers)
- Password

On the group-page, all password entries are listed. The actual password of each entry is hidden and can be shown by clicking on an icon.

5.4 Component Interaction

This section describes the abstract interaction between the different components of the SPM environment. Any inter-component interaction is carried out over the internet. Of course this is not a requirement for SPM to function correct, but the communication channel between the components is viewed as not controllable by the user. The description of the ongoing communication is divided into various operations that are triggered by the client.

5.4.1 Create and Update User

In the process of creating and activating SPM users, only the SPM client and the SPM server are active. Managing the group permissions of users would require interaction with CrySIL, this use case is handled in Section 5.4.3. Anyone is able to register an account at the SPM server, however the account has to be activated by an administrator in order to be able to gain permissions for a group. Since the SPM Server is not trusted, storing passwords in a secure way is very important. Also the user management of SPM and CrySIL are strictly separated and it is not recommended to use the same passwords for both services. The hashing strategy for storing passwords is covered in Section 5.6.5.

1. A new user registers an account at the SPM Server
2. After validation, the account meta data and the login credentials will be stored in database.
3. One of the administrators of the SPM Server instance can then activate that user.

4. The server checks if the administrator has the required permissions and updates the user in the database. After activation the user is able to login and get invited into password groups.

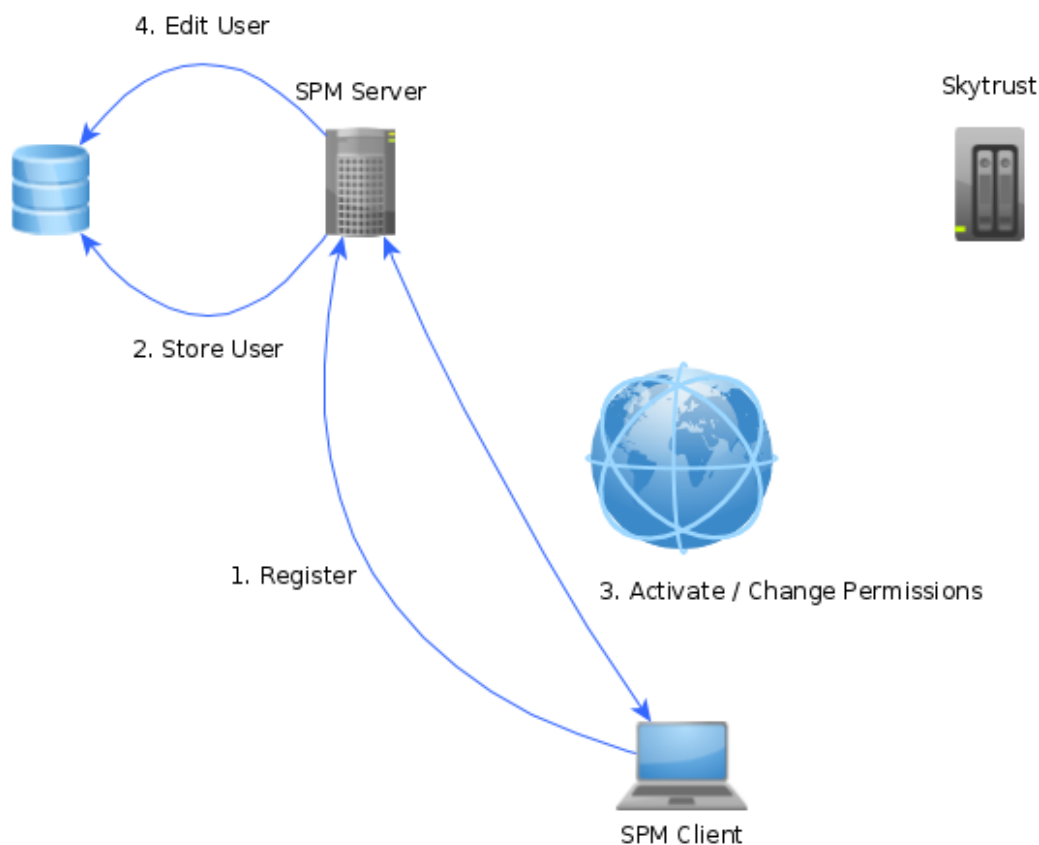


Figure 5.2: Component-Interaction During User Creation

5.4.2 Create Group

As already mentioned, a group in the context of the SPM system is a container for passwords. Therefore any group in this system needs a key that can be shared among the people permitted to read the passwords. In this case, every group has two keys, since password entries are encrypted twice. The key for the inner encryption will here be called *local key* because it is locally generated on the client inside the JavaScript context. The second key, which is used for the outer encryption, will be called *wrapped key*, because it is wrapped and therefore protected with a hybrid encryption scheme, that is described in Section 5.5. The protection of the wrapped key is strongly needed, because it is stored inside SPM Server's database. In order to create a new wrapped key, CrySIL needs a policy file from the client. Therefore such a policy is automatically generated by the SPM Client only allowing the active user to read or manipulate the new group.

1. A new key and a default policy file is generated on the client-side
2. The client requests CrySIL to generate a wrapped key. (This request includes the policy file generated on the client side)

- (a) If this type of request is sent the first time in this session⁷, CrySIL sends a response back to the client requesting for authentication
 - (b) After user interaction, the client sends a new request including the authentication data
 - (c) CrySIL checks if the user is permitted to carry out this operation
 - (d) CrySIL generates a new wrapped key, including the key itself and the sent policy file
3. The group meta data and the two keys are sent to CrySIL
 4. CrySIL validates the data and stores it in the database

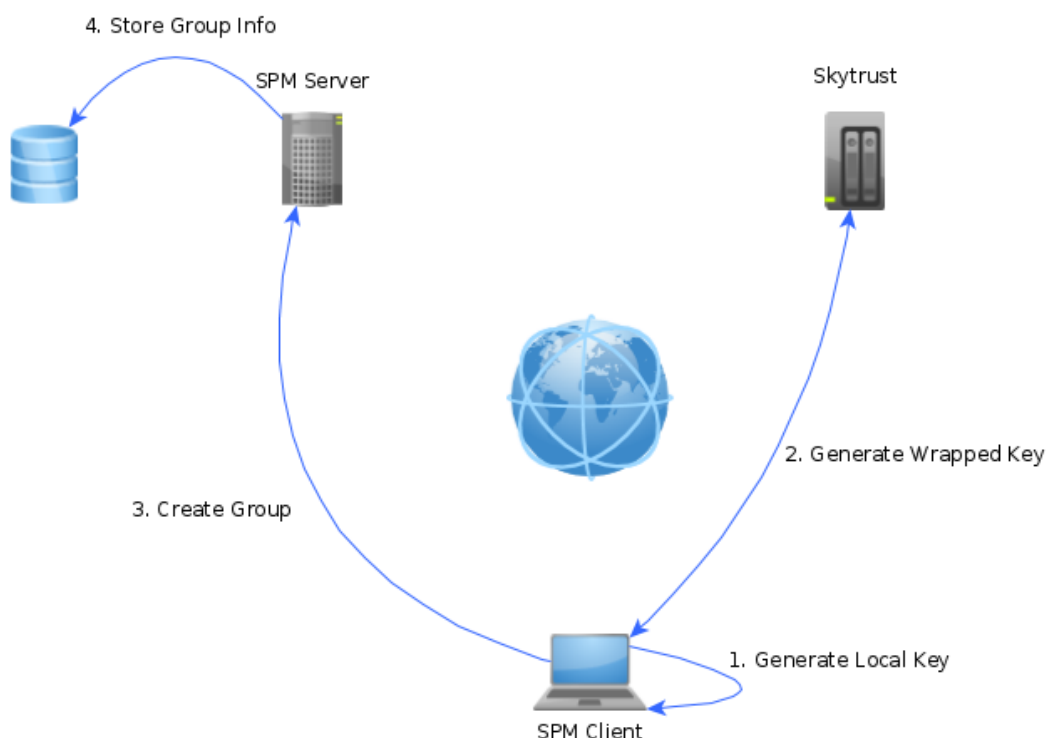


Figure 5.3: Component-Interaction during Group Creation

5.4.3 Update Group Permissions

The information about which user is permitted to read the passwords of a group (group member) and which user is permitted to update the group permissions itself (group admin) is stored in the policy file. In this context, updating group permissions means inviting/removing users to a group or granting other users to update the group permissions (declaring group admins). As already mentioned, the wrapped key is protected by a hybrid encryption scheme, carried out by CrySIL. If therefore the permissions of a given group shall be updated, the wrapped key has to be sent to CrySIL, where it is unwrapped and the policy file is manipulated. Another issue, that is concerned by permission updates, is the information freshness. Since the wrapped key is not stored by CrySIL itself, there may exist several versions of it, when updated over time. However CrySIL needs to know which version is the latest and only valid one. Therefore a nonce is generated and added to the file every time the policy is updated. This nonce is then also stored on the server for verification.

⁷CrySIL has its own session management, a user only needs to authenticate once per session and command. The session management is described in Section 5.5.2

1. Depending on the permission changes a new policy file is generated
2. The policy file is sent together with the wrapped key of the corresponding group to CrySIL
 - (a) If this type of request is sent for the first time in this session, CrySIL sends a response back to the client requesting for authentication
 - (b) After user interaction, the client sends a new request including the authentication data
 - (c) CrySIL checks, if the user is permitted to carry out this operation
 - (d) The wrapped key is unwrapped and the existing policy is checked, if the authenticated user is a group admin
 - (e) If both checks succeeded, a new nonce is generated and added to the new policy file
 - (f) At the end, the old policy file is replaced with the updated one and the key is wrapped again
3. The new wrapped key is sent to the SPM Server
4. The server updates the wrapped key in its database

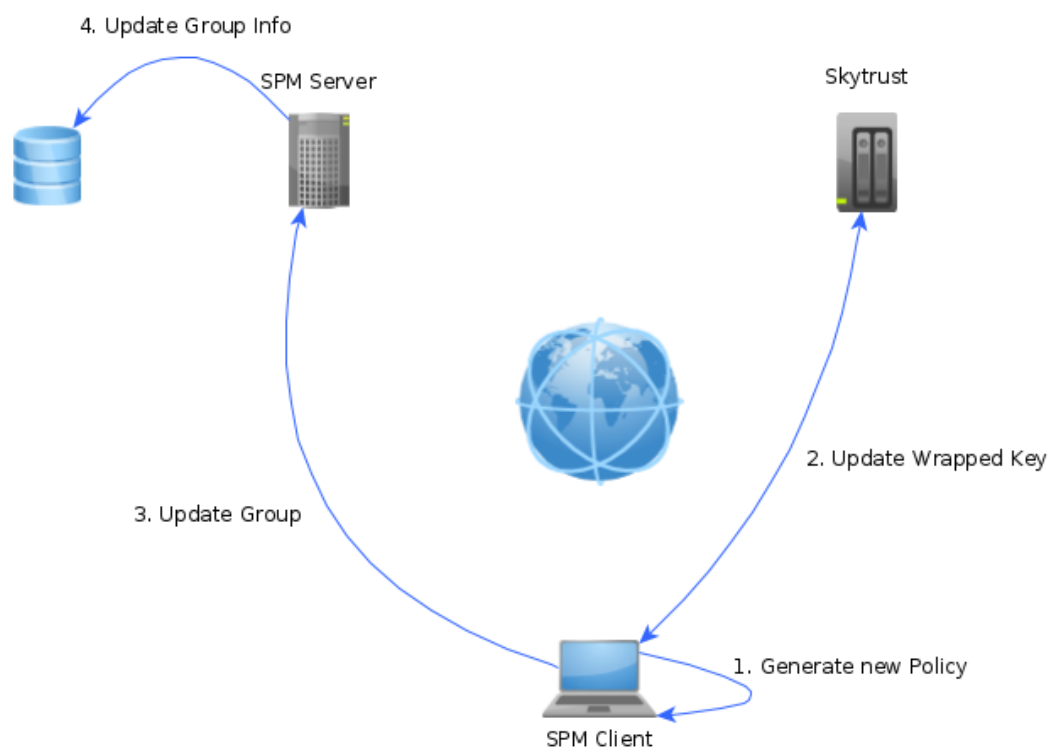


Figure 5.4: Component-Interaction During Group Permission-Update

5.4.4 Create and Update Password-Entry

Adding new password entries to a group can be done by any group member. The first step in this process requires the user to enter the password information. Subsequent this information is serialized by the SPM Client into the JavaScript Object Notation (JSON) and afterwards encrypted. Cipher information, such as algorithm identifier, IV and the serialized key⁸, is loaded from the SPM Server and imported into the

⁸The key serialization strategy is explained in Section 5.7.4

SPM Client environment automatically. In order to store the password entry, the SPM Server checks if the user has permissions for the corresponding group.

1. The client requests the group meta data including the encryption keys
2. The new password entry is serialized and encrypted using the local key
3. A request including the encrypted password entry is sent to CrySIL for the outer encryption
 - (a) If this type of request is sent for the first time in this session, CrySIL sends a response back to the client requesting for authentication
 - (b) After user interaction, the client sends a new request including the authentication data
 - (c) CrySIL encrypts the request data and sends it back to the client
4. The double encrypted entry is sent together with the encryption IVs and a group identifier to the server
5. The SPM Server validates the data and stores it in the database

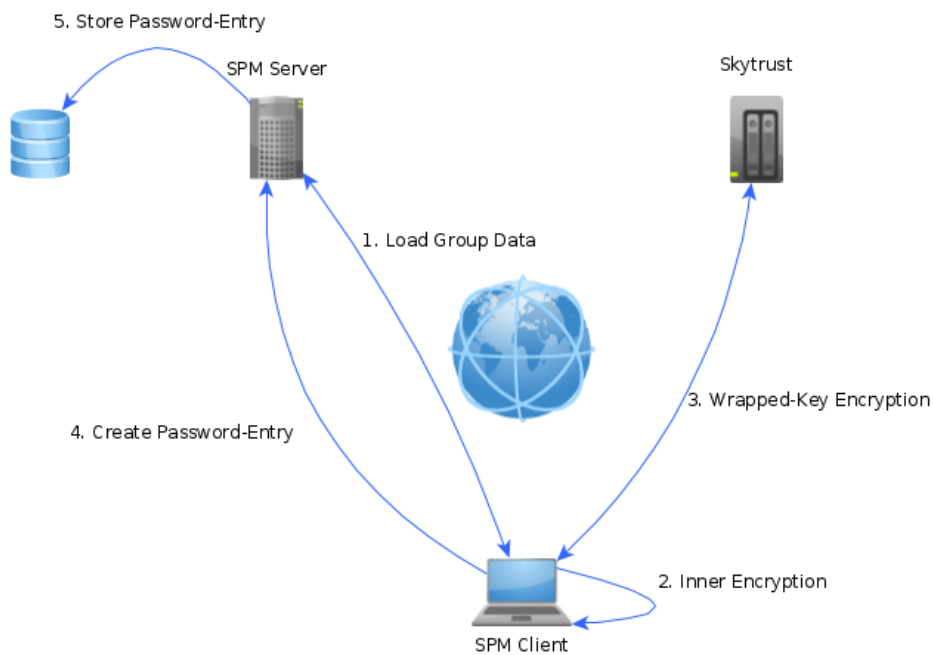


Figure 5.5: Component-Interaction During Password Generation

5.4.5 Read Password-Entry

The decryption of password entries is the most critical command in this list. It requires the user to have accounts and permissions for the SPM Server and CrySIL as well as the decryption permission in the group policy. In order to speed up the decryption process, the CrySIL interface is able to directly receive a list of passwords instead of just one at a time.

1. The client requests the group meta data
2. The list of encrypted entries is sent to CrySIL for decryption

- (a) If this type of request is sent the for first time in this session, CrySIL sends a response back to the client requesting for authentication
 - (b) After user interaction, the client sends a new request including the authentication data
 - (c) The wrapped key is unwrapped and the existing policy is checked, if the authenticated user is a group admin
 - (d) CrySIL decrypts the entries and sends the resulting list back to the client
3. The list is then iteratively decrypted on the SPM Client using the local key
 4. At the end, the resulting plain-text is presented to the user

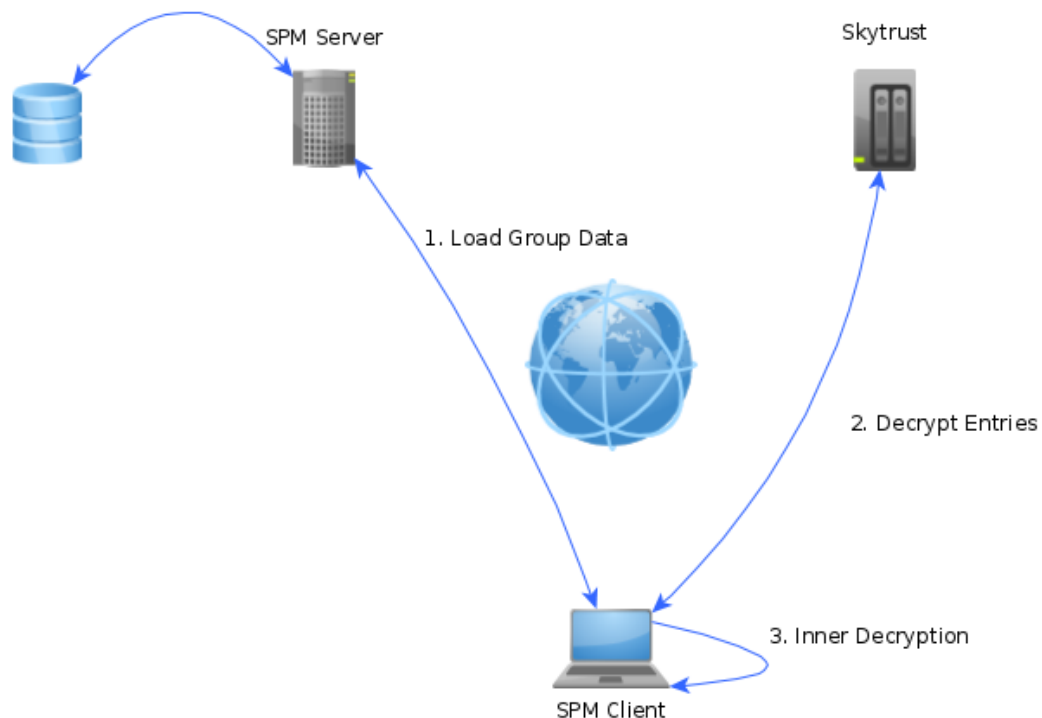


Figure 5.6: Component-Interaction During Password Decryption

5.5 CrySIL Implementation

As shortly mentioned before, CrySIL represents the Cloud Encryption Module from Chapter 3 in this implementation. The underlying platform, on which this system component is executed, is left open. The only important requirement is a connection to the cloud. As Reimair, Teufl, Kollmann, et al. [2015] have pointed out, CrySIL can even be successfully deployed on mobile devices. CrySIL was chosen as platform for the Cloud Encryption Module, because it offers a very flexible API, that can be easily called from the client component. Furthermore, there are already implementations for many of the needed cryptographic operations available. This section will describe the internal structure of the encryption component in SPM, which is CrySIL. After discussing the overall structure and the session and authorization management, each relevant cryptographic operation is described.

5.5.1 HTTP Interface

This component processes and validates HTTP Post calls and subsequently calls the according requested functions at the CrySIL back-end. The HTTP Interface is implemented using the Java Spring framework, therefore a so called *Controller* is used to handle incoming requests. Before any HTTP request is processed by the CrySILProtocolHandler, it needs to go through a CORS Header manipulation step. At this step, the HTTP *Access-Control-Allow* headers are added in order to enable clients with different origins to communicate with CrySIL.

Listing 5.3 shows parts of the *CrySILProtocolHandler*. The JSON validation of the HTTP Post body is done using *Schema files*. Listing 5.4 includes the part of the schema file which validates the request to generate a new group key. Each incoming request can be identified using its unique *type* field. In the presented schema part, the only parameter which needs to be added by the client, is the policy file. When the validation succeeded, the JSON data is transformed into a Java object which can be passed to the *Receiver* in line 14.

```

1 @Controller
2 public class CrySILProtocolHandler {
3     @RequestMapping(value = "/json", method = RequestMethod.POST, produces =
4         "application/json; charset=utf-8")
5     @ResponseBody
6     public ResponseEntity<SResponse> handleCrySILCommand(@RequestBody String
7         crySILRawRequest) {
8         HTTPReceiver receiver = (HTTPReceiver)
9             WebAppInitializer.element.getReceiver("HTTPReceiver");
10
11         if (receiver.isValidSchema() && !JsonUtils.isValidJSON(crySILRawRequest,
12             WebAppInitializer.requestSchema)) {
13             return new ResponseEntity<>(HttpStatus.NOT_ACCEPTABLE);
14         }
15         ...
16         crySILResponse = receiver.forwardRequest(crySILRequest);
17         ...
18         return new ResponseEntity<>(crySILResponse, HttpStatus.OK);
19     }
20 }
21 }

```

Listing 5.3: CrySIL HTTP-ProtocolHandler

```

1 "generateWrappedSymKeyRequest": {
2     "properties": {
3         "type": {
4             "enum": ["generateWrappedSymKeyRequest"]
5         },
6         "policy": {
7             "$ref": "#/definitions/base64string"
8         }
9     },
10    "required": ["policy"],
11    "additionalProperties": false
12 }

```

Listing 5.4: CrySIL Schema Validation File

5.5.2 Session Management

Each Actor in the CrySIL environment has its own Gatekeeper instance. Those Gatekeepers decide if the user sending a request, is allowed to execute a corresponding command. Furthermore, a user can be requested for authentication if needed.

In this work, the *XacmlGatekeeper* is used. This special Gatekeeper uses XACML files to decide if requests are permitted.

These files are located inside the wrapped key, which needs to be provided by the client. Therefore the Gatekeeper needs to decrypt the wrapped key before the policy can be evaluated. For this reason the *JaikJceActor* provides a method for key unwrapping that can be used by its Gatekeeper. During the unwrapping step, the policy nonce is checked as well. The policy extraction process including the nonce logic, is described in Subsection 5.5.3.

Permission files are very flexible and can be used in different ways. The policies used by SPM include a list of operations that can be called at CrySIL. Each of those operations is connected to a group of user entries. Such an entry defines how the user needs to authenticate against CrySIL. If the Gatekeeper decides, that a user needs to perform an authentication step, before the command can be executed, a special response is returned to the client.

In this case, the SPM Client will automatically detect such a response and display an authentication window. Once the client has successfully sent an authentication request, the Gatekeeper hands over the original request to the actor. The original request in this context, is the request that lead to the need for an authentication step. In order to do so, CrySIL needs to connect individual requests using a client session. As soon as the Gatekeeper decides, that an authentication step needs to be performed, the current request is stored in a session map and a the corresponding session id is added to the response. Since the session id will be present in the following requests, the Gatekeeper is able fetch back the original request from the map. Figure 5.7 shows the different steps of the authentication process during the decryption of a password entry.

5.5.3 Operations

This Subsection covers the Operations that are carried out by the Actor. After a general discussion concerning the overall structure of the Actor architecture, the concrete Commands which are relevant for this work will be described.

As mentioned, request objects are directly passed from the HTTP Interface to the Receiver. Each of those Receivers has a list of Actors that can be called. Depending on whether a concrete Actor supports an incoming requested command, it will produce a result. The Actor has a list of Commands, which can be uniquely identified using the command name. If the Actor owns a Command with the corresponding name referenced in the request, it will be executed.

Listing 5.5 shows the relevant parts of the command execution process of the *jaikJceActor*. In line 6, the Gatekeeper is reviewing the authentication state of the incoming request. If this check fails because the request was not authenticated yet, an exception is thrown, which will result in a response that requests the client to authenticate. Line 12 and 14 shows how Commands are extracted and executed.

```

1  public SResponse take(SRequest crySILRequest) {
2  try {
3      String commandName = crySILRequest.getPayload().getType();
4
5      BasicCommand command =
6          providedCommands.get(CmdType.getCmdTypeFromName(commandName));
7      Result result = gatekeeper.process(crySILRequest);
8      if (! result.isPermitted()) {
9          return
10             Gatekeeper.createAuthenticationFailedResponse(crySILRequest.getHeader()),

```

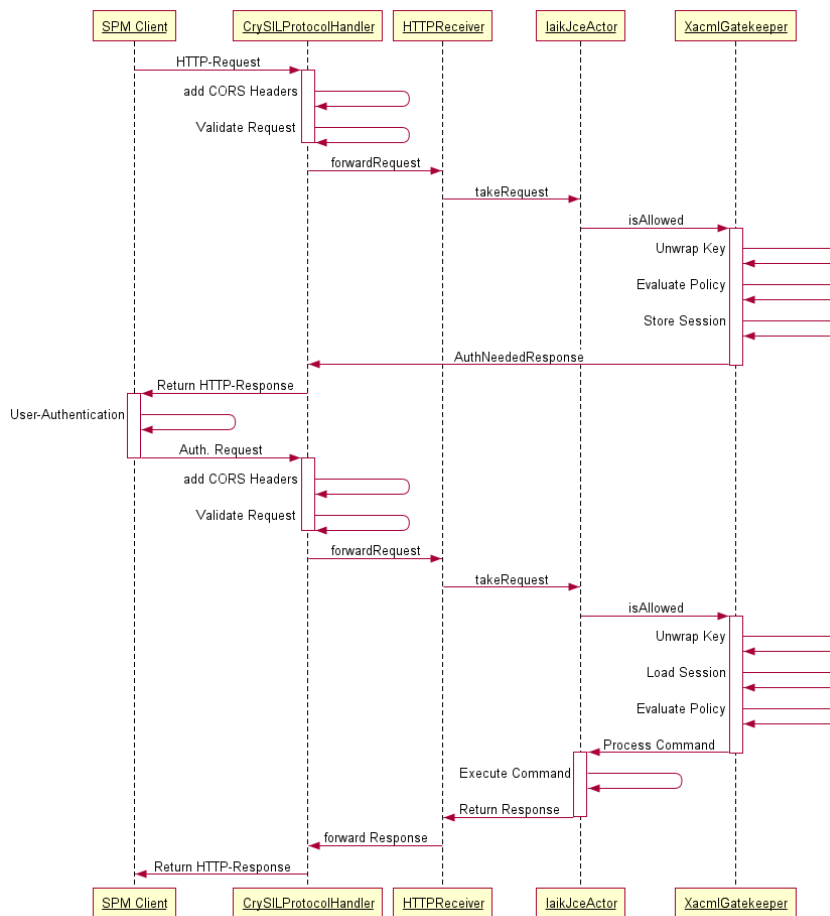


Figure 5.7: CrySIL Pipeline

```

    CrySILError.AUTH_FAILED);
9      }
10     SRequest sRequest = result.getOriginalRequest();
11     commandName = sRequest.getPayload().getType();
12     command = providedCommands.get(CmdType.getCmdTypeFromName(commandName));
13
14     return command.handle(sRequest);
15 } catch (AuthenticationRequiredException e) {
16     return e.getResponse();
17 }
18 }

```

Listing 5.5: CrySIL Command Execution

As mentioned, the relevant Commands in this work, are owned by the *IaikJceActor*. The cryptographic operations used in the following listed Commands are implemented via the Java Cryptography Extension (JCE). This framework enables users to carry out various different cryptographic services. JCE itself, does not implement the provided operations on itself, it only specifies the interfaces. The actual implementation is done by so called providers.

Java Cryptographic Extensions (JCE) is a set of Java API's which provides cryptographic services such as encryption, secret Key Generation, Message Authentication code and Key Agreement. OWASP⁹

⁹Open Web Application Security Project Documentation: <http://www.owasp.org>

The *IAIK-JCE*, which is described in Section 5.2 is one such provider and was used in this work. The following paragraphs are going to explain the purpose and the implementation details of each of those Commands.

- **JaikJceGenerateWrappedSymKeyRequest**

This command is executed in order to generate a new wrapped key, used for group content encryption. The assets contained in this structure are the symmetric key and a policy file. Therefore the user needs to send a policy file including user permissions along with the request.

In order to parse the policy, it is *base64* decoded and transformed into a String. To check the freshness of the policy file, a nonce has to be inserted. Therefore a 130 bit random number is generated and stored into a properties file, which resides on the CrySIL server. During the nonce generation process a unique identifier is chosen for the nonce. Both the id and the nonce itself are then inserted into the policy file.

In the next step, a new AES-256 key is generated using a key generator from the *IAIK* provider. Both entities are then put into the key object and parsed into a JSON String, which is called plain JSON key.

The resulting String is then signed with the Actor's signing key. The encryption process now encrypts the plain JSON key together with its signature using a hybrid encryption scheme. In this scheme, the input is first *AES256-GCM* encrypted. The symmetric encryption key for this container is then asymmetrical encrypted with a list of keys. In the case of the SPM setup, there is only one asymmetric key used for key encryption, because there is only one CrySIL instance involved in the process and only this entity should be able to decrypt the container. It is crucial, that this asymmetric key does never leave CrySIL, since the whole system security depends on this key. The resulting encrypted container is again signed with the Actor's private signature key, in order to enable the client to verify the wrapped key. The result of this is called the protected wrapped key.

Figure 5.8 illustrates the structure of the protected wrapped key.

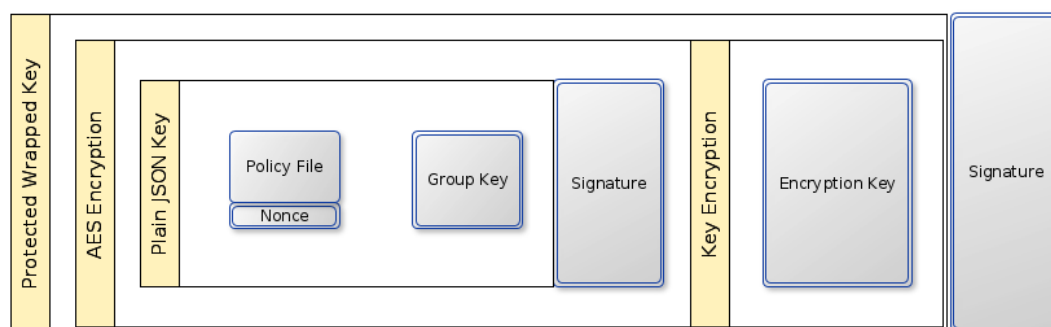


Figure 5.8: Protected Wrapped Key

In order to transport the protected wrapped key back to the requesting user, it is base64 encoded and added to the HTTP response body. The result of this operation has to be sent along each other request, since the policy file needs to be checked. The new group key can now be used to encrypt or decrypt password entries.

- **JaikJceUpdatePolicy**

This command is executed in order to update the policy file. This has to be done if the permission of users needs to be updated. In order to perform this operation, the calling user needs to be a SPM group admin. This means, that the user is allowed by the policy file to execute the *JaikJce-UpdatePolicy* command. Permission checks of the policy file are done inside the Gatekeeper.

To update the policy, the user needs to provide the current wrapped key and a new version of the policy file, that should replace the current one. Before a wrapped key is updated, the new policy file is parsed and sanity checked.

The internal logic of this command is very similar to the *IaikJceGenerateWrappedSymKeyRequest*. A new protected wrapped key is generated, however the contained group key stays the same. This is mandatory, since old password entries could no longer be decrypted, if the group key was updated in this process. One of the most important parts of this process is to update the policy nonce. The update needs to be propagated into the property file on the CrySIL server as well.

As a result of this operation, a new wrapped key is returned with an updated version of the policy file and the old version of the group key.

- **IaikJceSymEncryptRequest**

This command is used for the encryption of password entries. Therefore the client needs to provide a list of password entries, a protected wrapped key and an optional algorithm identifier.

If all permission checks succeed, this command encrypts the list of entries using the specified algorithm. In case no algorithm is specified by the client, *AES/CBC/PKCS5Padding* is used.

The Cipher Block Chaining (CBC) is a mode of operation where each block of plaintext is XORed with the previous ciphertext block before being encrypted. Therefore an IV needs to be used for the XOR operation of the first block. The 16 byte IV is generated using the *SecureRandom* interface. Since the size of the plaintext needs to match a multiple of the block length, padding is applied. In the default configuration, PKCS5Padding [Housley, 2009] is used to accomplish this by adding the same byte *N* times. The value of the added byte is *N*, which is the number of bytes that are added.

In order to start the encryption process, the protected wrapped key is first unwrapped. The *iaikJceActor* provides an interface to all commands, which are used for key unwrapping. Therefore CrySIL uses the stored asymmetric key encryption key, to start the decryption of the hybrid scheme. The result is used to decrypt the actual content encryption key. When the decryption succeed, the content encryption key is returned to the command.

As a result, a JSON list of encrypted password-entries in base64 encoding is returned to the client. Additionally an IV for each entry is added to the result, since it will be needed for decryption.

- **IaikJceSymDecryptRequest**

Same as the encryption Command, also the decryption can process a list of password-entries in one request. This is especially useful in the SPM setup, since all password entries inside a group are encrypted using the same wrapped key. Each decryption request needs to provide at least the protected wrapped key, the encrypted entries and the IVs. Listing 5.6 shows parts of the schema file, that is used to validate incoming decryption requests. Line 27. shows all mandatory fields of the request.

```

1  "symDecryptRequest": {
2    "properties": {
3      "type": {
4        "enum": ["symDecryptRequest"]
5      },
6      "encodedWrappedkey": {
7        "$ref": "#/definitions/base64string"
8      },
9      "encryptedData": {
10       "type": "array",
11       "minItems": 1,
12       "items": {
13         "$ref": "#/definitions/base64string"

```

```

14         }
15     },
16     "algorithm": {
17         "$ref": "#/definitions/finiteLengthString"
18     },
19     "ivs": {
20         "type": "array",
21         "minItems": 1,
22         "items": {
23             "$ref": "#/definitions/base64string"
24         }
25     }
26 },
27 "required": ["encodedWrappedkey", "encryptedData", "ivs"],
28 }

```

Listing 5.6: Schema Template for a CrySIL Decryption-Request

Before the command is executed, the protected wrapped key is unwrapped and the policy file is checked. Afterwards the content encryption key, contained in the wrapped key is used to start the decryption process. The algorithm used for this decryption needs to be the same, as for the encryption. If the signature check during the decryption succeeds, the resulting plaintext will be returned to the client.

5.6 SPM Server Implementation

The role of the SPM Server in this work, is to maintain and manage the data which is processed by the SPM Client and CrySIL. Therefore most of the logic in this component validates information and stores it in the database. The SPM Server is implemented as a Java EE container and therefore features like *EJB* annotations can be used. To this end, the resulting *WAR* file, which is generated when building this component, can be run inside a Java EE compatible application server. In this work, the *wildfly* application server was used.

5.6.1 Structure

As illustrated in Figure 5.9, the SPM Server can be divided into the following logical parts:

- **Interface:** The interface component handles the HTTP communication with the SPM client. To this end, serialization and deserialization is also done here. Incoming requests including JSON data are automatically transformed into *POJOs*. Before any request is processed by the corresponding Rest component, it is intercepted and the permission management performs a verification process. When a response is generated, another interceptor is activated, to add CORS attributes in order to conform to the Same Origin Policy.
- **Core:** This component is the entry point of the server execution. At application start, the connection to the database is established and mock data is inserted. Afterwards tests are executed and the Rest configuration is processed by the *StartupManager*. In the last step of the bootstrap process, configuration is read from a maven configuration file and processed by the *ConfigurationManager*.
- **Database:** This component is in charge to communicate with the Rest interface and perform database level manipulation. The object mapper *Hibernate* is used for database communication. To this end, entity classes that represent tables in the database are used for data representation. Those entities are automatically serialized and deserialized if needed. One of the big advantages of this design is that the database provider is transparent to the REST of the application. Currently *Postgresql* is used, but the database can be switched without any change to the SPM Server code.

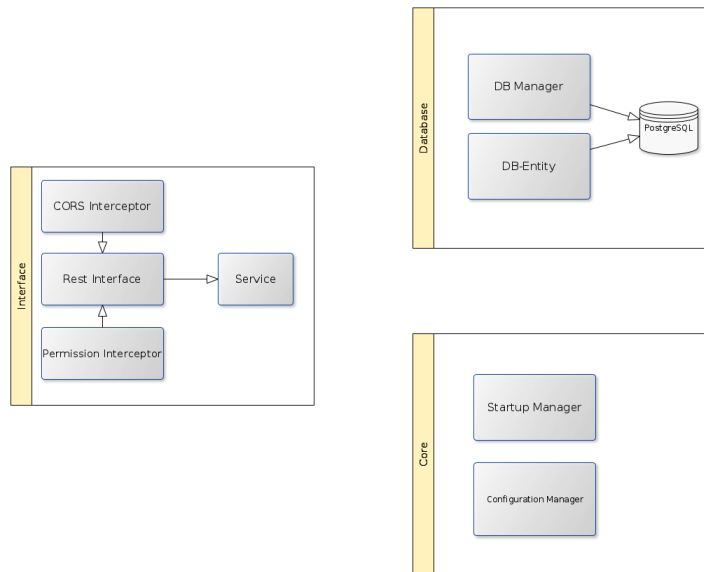


Figure 5.9: SPM Server Overview

The components in Figure 5.9 again abstract subsystems that will be described in the following Subsections. For a better understanding of the context of each subsystems that will be described, a simplified version of the class diagram was added in Figure 5.10.

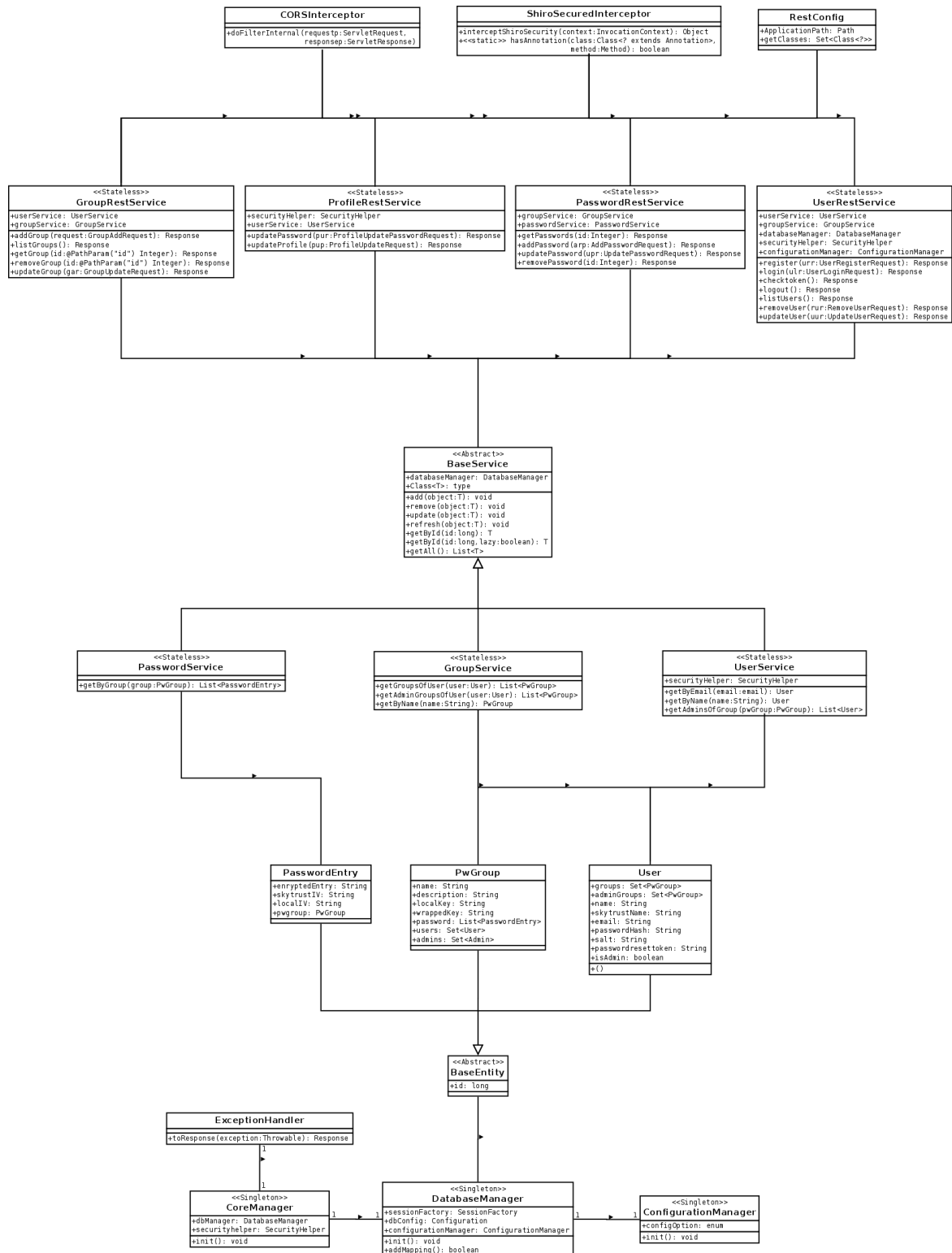


Figure 5.10: Simplified Class Diagram of the SPM Server

5.6.2 Communication

In this section, the party of the system which is in charge to communicate with the SPM client is viewed closer. This communication is transported over HTTP GET, POST and DELETE calls. To this end, the

interface classes implement the JAX-RS specification.

There are several ways to expose JAX-RS to the outside world. In this work, a way that does not require any XML configuration was preferred. There is one configuration Interface that subclasses from the JAX-RS *Application* and declares the URL path from which it should be called. Inside this class, there is only one method, returning the classes which contain the actual REST interfaces. The JAX-RS library is able to retrieve this list at run-time and extracts the needed information (via Java reflection). All the information which is needed to decide which method should be called, after a HTTP Request was retrieved, is declared via annotations. As a result no *web.xml* file is needed. Listing 5.7 shows the above described process of exposing Rest endpoints. After application start, those endpoints will be available at the deployed server via the path *server_url/rest/endpoint/method*.

```
@ApplicationPath("/rest/")
public class RestConfig extends Application {

    public Set<Class<?>> getClasses() {
        return new HashSet<Class<?>>(Arrays.asList(CORSFilter.class,
            ShiroSecuredInterceptor.class, ExceptionHandler.class,
            UserRestService.class, ProfileRestService.class,
            GroupRestService.class, PasswordRestService.class));
    }
}
```

Listing 5.7: Exposing Rest-Interfaces via JAX-RS

Furthermore, each of the Rest classes declare a path for the endpoint and a path for each method. For the given example in Listing 5.8, the URL to call the *addGroup* method would look like *server_url/rest/group/addGroup*. The following list describes the annotations for the *addGroup* method:

- *@POST*
Indicates that only HTTP POST requests are accepted.
- *@Path("addGroup")*
The URL path under which this method can be reached.
- *@Consumes(MediaType.APPLICATION_JSON)*
Identifies the encoding and structure of incoming requests.
- *@Produces(MediaType.APPLICATION_JSON)*
Identifies the encoding and structure of outgoing responses.
- *@RequiresAuthentication*
Indicates that the *Shiro* interceptor needs to validate whether the requesting client is authenticated.
- *@RequiresRoles("acceptedUser")*
Indicates that the *Shiro* interceptor needs to validate whether the requesting client has a specific role.

The last two annotation types concerning user-permissions are closely discussed in Section 5.6.5.

```
@Path("/group")
@Stateless
@Interceptors(ShiroSecuredInterceptor.class)
public class GroupRestService {

    @EJB
    private UserService userService;
```

```

@EJB
private GroupService groupService;

@POST
@Path("addGroup")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@RequiresAuthentication
@RequiresRoles("acceptedUser")
public Response addGroup(GroupAddRequest gar) throws Exception {
    if (!gar.isValid())
        return Response.status(Response.Status.BAD_REQUEST).entity("").build();

    // process the request
}

```

Listing 5.8: Example Rest-Service

The life-cycle of the Rest endpoints is defined via the `@Stateless` annotation. Stateless in this context means, that the class itself can not hold any internal state, since it can be created and destroyed at any time inside the application server. A closer look at the different life-cycles of JAVA beans is taken in Section 5.6.3.

As mentioned in the above listing of different annotations, input and output types are declared. In the given example, the client is demanded to send input data, encoded into a *JSON* structure inside the HTTP Post request-body. As it can be seen in Listing 5.8, in the JAVA context the input is processed in form of a POJO. The transformation of the incoming JSON into Java objects, as well as the reverse transformation from POJOs into JSON, is done automatically by the library. To this end, container classes need to be created for each request and each response. To indicate whether a value of the input is required or optional, again Java annotations are used. All request containers are inherited from the base class *BaseRequest* which offers an *isValid()* method. When this method is called, it will check all class members that are annotated with `@Required`. As in Listing 5.8, this validation method should be invoked at the beginning of any Rest method.

In case a special validation is needed, the *valid()* method is overridden by the container class. In the case of the referenced *GroupAddRequest*, this is done. Listing 5.9 is presenting the relevant subset of the logic in the *GroupAddRequest* container. The *isValid* method first calls its super class to perform the basic validation and afterwards performs the special validation code. In this case, it is checked if all admins of the new group are users as well, since it would lead to inconsistent permission behaviour if this requirement is not met.

```

@XmlRootElement
public class GroupAddRequest extends BaseRequest {

    @Required
    private String name;
    private String description;
    @Required
    private List<String> users;
    @Required
    private List<String> admins;
    @Required
    private String local_key;
    @Required
    private String wrapped_key;

    @Override
    public boolean isValid() {
        if (!super.isValid())

```

```

        return false;

    for(String admin : admins) {
        if (!users.contains(admin))
            return false;
    }

    return true;
}

```

Listing 5.9: Container-Class for the addGroup() interface

In the following paragraph, all Rest-Endpoints and their corresponding methods are listed and briefly described. Next to the name, the type of the HTTP method is listed. The type POST indicates, that the client is required to send the input for the serialization process, formatted in JSON .

- PasswordRestService

This Endpoint is in charge to maintain the password entries. Each method that can be called in this Rest-Service, requires group permissions.

- listPasswords - GET

This call requires a specific group id, that is transmitted as part of the URL. The URL for calling this service has the following structure: (server/password/listPasswords/[id]). All encrypted password entries for a group are loaded from the database and returned to the user.

- addPassword - POST

After this call is processed, a new password entry should be stored in the database. The client is required to send the encrypted data, the IVs and a group id, in order to perform the validation successfully.

- updatePassword - POST

A specific password entry that is referenced by its id is replaced in the database.

- removePassword - DELETE

The password entry with the given id is hard deleted from the database.

- UserRestService

This Rest-Service handles the user registration and sessions functions. Each user is also able to maintain a small profile, that can be updated here.

- register - POST

When this request is handled, a new user is created and stored in the database. Also a new user session is generated, so that the client does not have to login as an extra step. However the new user has to be activated by an SPM administrator, to perform any further actions.

- login - POST

To perform the login procedure, the client has to provide an email address and a password. Via the email address the specific user can be identified, since this attribute has to be unique among the users in the database. Afterwards the information is handed over to Shiro to perform the password validation and session generation. This process is described in Section 5.6.5.

- checktoken - GET

Since user sessions on the client side should not break when the page is reloaded or the browser history is used, the session has to be checked with a dynamic process. This process itself is explained in Section 5.7.2. The checktoken method is called in this process and

returns the user information that is needed by the client. Therefore it needs a unique session identifier that is created in the login procedure.

- logout - POST
Removes the user session out of the session map and afterwards instructs the client to remove the session cookie.
- listUsers - GET
As mentioned, SPM administrators are able to perform different actions on the user database. To do so, first all users have to be listed. This method returns all available users in the SPM database to the client.
- removeUser - DELETE
The removeUser method soft deletes the user with the given user id from the database.
- updateUser - POST
This method is used to update the SPM permissions of a given user. To be able to carry out this process, the executing user needs to be an SPM administrator.

- GroupRestService

In this code part, all the logic is located that performs actions on the password groups. Although the SPM Server itself is not security critical to the system security, much attention was paid to the permissions in this logic. Even though the cryptography can not be harmed in this place, data loss would be possible, if permissions are not checked correctly. In contrast to the last Endpoint, the important permission flag here that is checked, is the *group administrator* not the *SPM administrator*. This flag indicates, that the given user is allowed to manage a given group, not the user system itself. First of all, each method in this class has the annotations *@RequiresAuthentication* and *@RequiresRoles("acceptedUser")* which ensure, that the executing user is at least authenticated and active.

- addGroup - POST
With this method each active user is able to create a new password group and invite users to it. The executing user will become automatically the group administrator of the new group. In order to call this method, the user needs to provide a list of users and a wrapped key.
- listGroups - GET
This method is used for the group overview. A short summary of the information for each group is returned.
- getGroupPermissions - GET
The group administrator is able to switch to a management section, that lists all users and their permissions in the given group. This method returns the needed information for the management section.
- removeGroup - DELETE
This logic deletes the given group and its passwords from the SPM database.
- updateGroup - POST
The updateGroup command changes the group permissions. Therefore users can be removed or added to the group, as well as new administrators can be assigned. If a user was removed from the group, this user should not be able to decrypt the passwords of the given group in the future. In order to make sure, that decryption permission is revoked from this point in time, the updateGroup command also takes a new wrapped key.

- ProfileRestService

This very small REST service is in charge to manage the user specific information.

- updatePassword - POST

In order to update the SPM password, the user needs to provide the current and a new password. If the check succeeds, the new password will be iteratively hashed, salted and stored in the SPM database.

- updateProfile - POST

The user profile includes a profile image, a name and the email address of the user. Additionally, the user can enter the CrySIL user name. Only if this field is set, the user can get invited to other groups.

Since the general setup of the SPM environment assumes, that every part is hosted on a different server, the user agent which executes the SPM client is running into problems concerning the Same-Origin Policy. For this reason a Request-Interceptor was implemented to set the CORS headers. The most common way how this can be done, is by extending from the JAVA EE *OncePerRequestFilter*. For each request-response pair, the method *doFilterInternal* is called. Listing 5.10 shows a simplified excerpt of this method. It can be seen, that response headers allowing the origin and different HTTP Headers are added to the responses.

On normal requests, two CORS headers are set to the response and the REST of the application logic is called via *filterChain.doFilter(request, response)* to process the requests.

However if a so called preflight request is sent, a response is directly returned to the client, without the processing of the request. Those preflight requests just check if the next request that is going to be sent from the user agent, is allowed to be executed on the server.

```
@Override
protected void doFilterInternal(ServletRequest requestp, ServletResponse
    responsep,
    FilterChain filterChain) throws ServletException, IOException {
    HttpServletRequest request = (HttpServletRequest)requestp;
    HttpServletResponse response = (HttpServletResponse)responsep;

    response.addHeader("Access-Control-Allow-Origin", serverOrigin);
    response.addHeader("Access-Control-Allow-Credentials", "true");

    if (request.getHeader("Access-Control-Request-Method") != null &&
        "OPTIONS".equals(request.getMethod())) {
        //preflight
        response.addHeader("Access-Control-Allow-Methods", "GET, POST, PUT,
            DELETE");
        response.addHeader("Access-Control-Allow-Headers", "Content-Type");
    }else {
        filterChain.doFilter(request, response);
    }
}
```

Listing 5.10: Java Servlet Filter to insert CORS Headers

5.6.3 Core and Management

This subsection covers the different core and management parts of the SPM Server. In contrast to the SPM Client and the CrySIL interface, the SPM Server has relatively little management operations to carry out, since its job is to check permissions and act as a database front-end. However there are several important tasks, that need to be described.

Before jumping into great detail of the different management components, the EJB life-cycles need to be briefly discussed. All active components in the SPM Server setup are so called Java Session Beans:

A session bean encapsulates business logic that can be invoked programmatically by a client over local, remote, or web service client views. To access an application that is deployed on the server, the client invokes the session bean's methods. The session bean performs work for its client, shielding it from complexity by executing business tasks inside the server.

A session bean is not persistent.

(Oracle Documentation ¹⁰)

There are three different types of session beans in the context of life-cycles:

- **Stateful Session Beans**
As the name suggests a session bean is able to maintain its current state through its instance variables. A stateful session bean keeps a conversation with a client. Therefore those kind of beans can not be shared among several clients. A new bean is created during the session build up and is destroyed when the client ends the session. Therefore the state is retained for the duration of the session. This type of bean is used, when the state of the bean itself represents the interaction between client and server.
- **Stateless Session Beans**
A stateless bean only contains state during the invocation of a method. Therefore no state is shared between two calls, which implies that there is no session information available. The behaviour enables EJB to reuse such beans. If a client invokes a method of such a bean, one stateless bean can be taken out of a pool, to carry out the process. Because they can support multiple clients and are therefore reusable, stateless session beans can offer better scalability.
- **Singleton Session Beans**
A singleton session bean is created at the application start and exists until the application is shut down. As the name says, there is only one single instance of such a bean that is shared among all clients. As the singleton session bean maintains its state between client invocations, it is used when all clients in the system have to share the same state.

The REST services discussed in the last subsection, are stateless session beans, since there is no need for a state inside the bean. All the state information that is needed such as the authentication and permission flags are kept in the session map of the servlet context. In contrast to that, much of the management logic does in fact need to maintain a state. Generally the management components are mainly used for configuration and initialization purposes.

- **ConfigurationManager**
The Configuration Manager is started at the application start and reads in a file called *app.properties*. This property file includes key value pairs for different configuration options like file paths. All those options are server specific. Therefore a configuration file rather than hardcoded string constants is used for them. Since the Configuration Manager is marked as *@Singleton* all EJB beans access the same and only instance of it.
- **DatbaseManager**
This management instance is in charge to set up and configure a connection to the database. Since hibernate is used as an object mapper, a configuration file called *hibernate.cfg.xml* and a *data source* has to be provided. When a connection is established, for each table in the database, a

¹⁰<http://docs.oracle.com/javaee/6/tutorial/doc/gipjg.html>

so called *entity class* is mapped to them and a Session is created that can be used by services to manipulate the database. Each table corresponds to an Entity mapped here. For the different configuration options, the ConfigurationManager is used.

- CoreManager

This is the main entry point to the SPM Server. The CoreManager is annotated with *@Startup* and is therefore constructed at the application start. The *init()* method is directly executed when the instance is constructed. This is done via *@PostConstruct*. When this method is called, the DatabaseManager and the ConfigurationManager are called to perform the mentioned tasks. In case the the SPM Server is started the first time, the database is initialized with data. A new user is constructed with all permissions, so that the person installing the SPM Server, has a valid account to configure and set up the application.

5.6.4 Database

The access to the database is organised into two parts. First there is the *DatabaseManager* that sets up the connection and maintains the *SessionFactory*. This *SessionFactory* handles the database transactions and rollbacks in case an error occurs. Different service classes act as a front-end to the database. Those services expose different interfaces to the REST classes, in order to update the database.

For the communication with the database itself, there are entity classes. Those entities are normal POJOs with special annotations for database meta information. Each entity extends the base class *BaseEntity*.

All service classes are inherited from the *BaseService*. This special class holds an instance of the DatabaseManager in order to get the current session. All basic database operation that should be provided for all entities such as add, remove, update are implemented here in a generic way.

Listing 5.11 shows, how the diamond operator is used for the database entity manipulation. Each service provider that extends the *BaseService* has to provide a specific entity class.

```
public abstract class BaseService<T extends BaseEntity> {

    @EJB
    protected DatabaseManager dbManager;

    private final Class<T> type;

    public BaseService(Class<T> type) {
        this.type = type;
    }

    public void add(T object) {
        Session ses = dbManager.getSessionFactory().getCurrentSession();
        ses.saveOrUpdate(object);
        ses.flush();
    }

    public T getById(long id, boolean lazy) {
        Session ses = dbManager.getSessionFactory().getCurrentSession();
        Criteria c = ses.createCriteria(type);
        c.add(Restrictions.idEq(id));

        if(!lazy)
            c.setFetchMode("gallery", FetchMode.JOIN);

        @SuppressWarnings("unchecked")
        T ret = (T) c.uniqueResult();
        return ret;
    }
}
```



```

    }

    public List<T> getAll() {
        ..
    }

    public void remove(T object) {
        ..
    }

    public void update(T object) {
        ..
    }

    public void refresh(T object) {
        ..
    }
}

```

Listing 5.11: Service Front-End for database manipulation

```

@MappedSuperclass
public abstract class BaseEntity implements Serializable{

    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "increment")
    private long id;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }
}

```

Listing 5.12: Base-class of SPM Server entities

Each service is able to execute Hibernate SQL (HQL) for selecting and updating database records. For these queries, prepared statements are used in order to prevent SQL injection vulnerabilities. In the following list, the three most important services and their interfaces are briefly described. It can be recognized, that only a small amount of interface methods are needed for the database interaction, since most of the work is already done in the *BaseService*.

- GroupService
 - getGroupsOfUser
Returns all password groups the provided user belongs to.
 - getAdminGroupsOfUser
Returns all groups in which the provided user has administrator permissions.
 - getByName
Returns the password group using the provided unique SPM group name.
- PasswordService

- `getByGroup`
Returns all password entries for a given group.
- `UserService`
 - `getByEmail`
Returns the user based on the provided email address.
 - `getByName`
Returns the user based on the provided unique SPM user name.
 - `getAdminsOfGroup`
Returns all group administrators of the provided group.

5.6.5 Permission and Session Management

This subsection covers the permission and its corresponding users session system. At this point, please note, that the SPM server permission system is not seen as part of the trusted environment. This means, that the SPM server does not see or process sensitive information at any point in time. However the possibility of unauthorized manipulation leading to data loss, is considered during the implementation of this component.

As already mentioned, the *Apache Shiro* framework was used in this project to control the session management. The following main tasks are carried out by the permission system:

- **Registration**
The registration process is unauthorised and can therefore be carried out by any client. To create a new user in the SPM database, the password has to be stored conforming to the Shiro configuration. This is important because otherwise Shiro will not be able to match the password later on. For this reason, the *SPM SecurityHelper* offers the interface *generateShiroConformHash*. This method will generate a *Sha256Hash* with a given salt and a number of iterations that is read from the later explained shiro.ini file. Currently the number of iterations is configured to be 1024. The salt for the password is a 130 bit alphanumeric String that is generated using the Java *SecureRandom* interface.
- **Login**
The first step in the login process is to check the password. This is done by putting the username and the password into a container and handing it over to Shiro. If the call succeeds, the user object is added to the session map. This is possible, since after the call to Shiro, there is already an existing session. The Shiro session is identified via a browser cookie stored on the client and sent along with each request. From this point in time, every REST interface of the SPM Server has access to the user object and all its corresponding permissions.
- **Logout**
When a logout is triggered, Shiro is informed to remove the Session and all its corresponding information from its session map. Afterwards the user agent is instructed to remove the session cookie as well.
- **Session Check**
The SPM Client needs to check if the stored session cookie is still valid in many cases. For example if the user reopens or reloads the browser tab. For this reason, an interface method was implemented, which checks if the session identifier is still contained in the session map. In case there is no corresponding user object inside the map, the user agent is requested to remove the cookie and redirect to the welcome page.

- Action Permissions

The SPM Server permission system enables the developer to specify individual roles and authentication modes for each interface. Those permissions are enforced by the *ShiroSecuredInterceptor* before the actual interface call is handled.

The SPM permission system was implemented in such a way, that it is able to cooperate with Shiro in terms of session permissions. For this purpose the *ShiroSecuredInterceptor* and the *SecurityHelper* components were implemented. The *ShiroSecuredInterceptor* uses the mentioned Shiro session information in the defined Java annotations to determine if a concrete request should be permitted. This component is a so called EJB Interceptor. Interceptors have no state and just process information before or after a request is sent. The method *interceptShiroSecurity* is annotated with *@AroundInvoke*, therefore it is called before the actual method. Listing 5.13 shows, how the different types of annotations are checked via reflection against the Shiro session map.

```
@AroundInvoke
public Object interceptShiroSecurity(InvocationContext context) throws
    Exception {
    Subject subject = SecurityUtils.getSubject();
    Class<?> c = context.getTarget().getClass();
    Method m = context.getMethod();

    if (!subject.isAuthenticated() && hasAnnotation(c, m,
        RequiresAuthentication.class)) {
        throw new SpmUnauthenticatedException("Authentication required");
    }

    if (subject.getPrincipal() != null && hasAnnotation(c, m,
        RequiresGuest.class)) {
        throw new SpmUnauthenticatedException("Guest required");
    }

    if (subject.getPrincipal() == null && hasAnnotation(c, m,
        RequiresUser.class)) {
        throw new SpmUnauthenticatedException("User required");
    }

    RequiresRoles roles = getAnnotation(c, m, RequiresRoles.class);

    if (roles != null) {
        try {
            subject.checkRoles(Arrays.asList(roles.value()));
        } catch (Exception e) {
            throw new SpmUnauthenticatedException("wrong role");
        }
    }

    RequiresPermissions permissions = getAnnotation(c, m,
        RequiresPermissions.class);

    if (permissions != null) {
        subject.checkPermissions(permissions.value());
    }

    return context.proceed();
}
```

Listing 5.13: Interceptor Used to Check Permissions

The Shiro security framework, receives the needed information from the configuration file *shiro.ini*.

This configuration file is very flexible and comprehensive. The three most important information parts in this file are:

1. How the session can be identified
2. How user information can be retrieved
3. How a password can be matched

Listing 5.14 gives a short extract how this information is specified.

```
#Session identification
cookie = org.apache.shiro.web.servlet.SimpleCookie
cookie.name = spmsession
cookie.path = /
sessionManager.sessionIdCookie = $cookie
securityManager.sessionManager = $sessionManager

# password hashing specification
sha256Matcher = org.apache.shiro.authc.credential.HashedCredentialsMatcher
sha256Matcher.hashAlgorithmName=SHA-256
sha256Matcher.hashIterations = 1024
sha256Matcher.hashSalted = true
sha256Matcher.storedCredentialsHexEncoded = false

# JDBC Realm configuration
jdbcRealm = com.spm.management.helper.JdbcRealmImpl
jdbcRealm.permissionsLookupEnabled = false
jdbcRealm.authenticationQuery = SELECT password, salt FROM usertable WHERE email
= ?
jdbcRealm.userRolesQuery = SELECT type FROM usertable WHERE email = ? OR
oauthidentifier = ?
jdbcRealm.jndiDataSourceName= java:jboss/datasources/PostgresDS
jdbcRealm.credentialsMatcher = $sha256Matcherreturn context.proceed();
```

Listing 5.14: Shiro Configuration

The session identification tells Shiro how the id can be extracted from HTTP requests, by providing the cookie name and path. This session cookie will be set via the HTTP *SET-COOKIE* header which is generated by Shiro itself.

The password hashing specification from the shiro.ini is relatively self explanatory, the password is hashed with SHA-256 with an iteration count of 1024. In the end, the realm configuration specifies how user information can be received. In this case, Shiro is instructed to connect to a database and perform SQL queries. It can be seen, how the password and salt combination or the user permissions are selected.

5.7 SPM Client Implementation

The SPM Client is the part of the system, where all the information comes together. It directly communicates with the SPM Server and CrySIL. Apart from information gathering the SPM Client is also in charge to carry out cryptographic operations.

This web component is a single page application. This means, that the application only reloads parts of the content of the page, which makes it feel like using a desktop application. Therefore the data which is loaded from the SPM Server is gathered in an asynchronous way (AJAX).

The following sections will describe the concept of controllers and services. The overview of the logical parts in Figure 5.11, shows the four main environments. The controllers and services are pro-

cessing data, whereas the interface components implement the corresponding server protocol in order to communicate with the SPM Server and CrySIL.

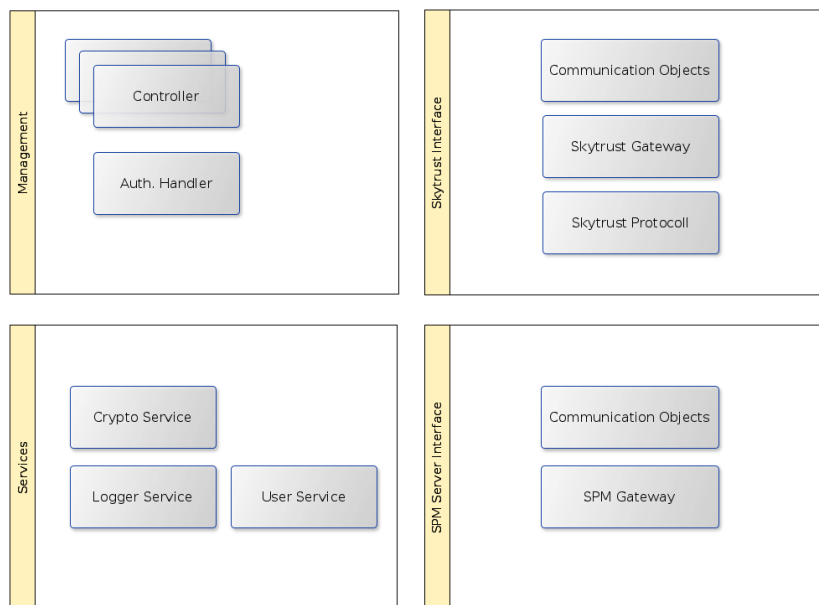


Figure 5.11: SPM Client Overview

5.7.1 Structure

For the JavaScript implementation, the framework *AngularJs* was used. This is reflected in the way how the project is structured, since *AngularJs* has a concept of how JavaScript code is structured in a pseudo object oriented way.

This concept is mainly based on *Controllers* and *Services*. A service in this context performs special processing, that is used by several parts of the system. Such services can be injected into controllers.

Angular services are substitutable objects that are wired together using dependency injection (DI). You can use services to organize and share code across your app. Angular services are: Lazily instantiated – Angular only instantiates a service when an application component depends on it. Singletons – Each component dependent on a service gets a reference to the single instance generated by the service factory. [AngularJs documentation](https://docs.angularjs.org/guide/services)¹¹

A controller itself is responsible for the logic corresponding to one page in the web application. So there is always one HTML file and one controller for each logical site.

The entry point to the application is a file called *SPMRoute*. At this place, the *AngularJs* module is created and the so called routes are defined. A route is always connected with a *templateUrl*, which is the part of the URL that comes after the URL hash key. The manipulation of this part of the URL will not trigger a browser reload. This behaviour is desired, since only some parts of the application should be re-rendered, when a link is clicked. Therefore such a route connects the *templateUrl* with a controller and the HTML file which is injected at a defined place in the DOM. Listing 5.15 shows how the route is configured in the way explained above. Another interesting feature, that can be seen in this listing, is how URL parameters are defined. So for example the *PasswordController* is able to access the variable *groupId* from the URL via an *AngularJs* service.

```
var app = angular.module('SPMClient', [ 'ngRoute', 'ui.bootstrap' ]);
```

¹¹<https://docs.angularjs.org/guide/services>

```

...
app.config([ '$routeProvider', function($routeProvider) {
    $routeProvider.when('/pentry/:groupId', {
        templateUrl : 'static/html/main/passwordList.html',
        controller : PasswordController
    }).when('/newgroup', {
        templateUrl : 'static/html/main/groupManagement.html',
        controller : GroupController
    }).when('/usermanagement', {
        templateUrl : 'static/html/main/userManagement.html',
        controller : UserManagementController
    }).when('/groupmanagement/:groupId', {
        templateUrl : 'static/html/main/editGroup.html',
        controller : GroupManagementController

        ...

    }).otherwise({
        redirectTo : '/welcome'
    });
} ]);

```

Listing 5.15: Application-Route of SPMClient

In the following paragraphs the different controllers are briefly described. A more detailed analysis of the important logic is given in the following sections.

- **groupController**

The `groupController` is responsible for displaying a searchable list of all groups, the logged in user is attending. Therefore it communicates with the SPM interface to gather the needed data. Furthermore, new groups can be created here. To fulfil this task, a local key has to be generated at the `cryptoService` and a wrapped key is requested from CrySIL. In order to do these steps, the JavaScript promise architecture is used. The detailed explanation of this workflow is described in Section 5.7.4.

- **groupManagementController**

This controller is used at the detail page of a group. The user is able to view and modify all the meta information including access permissions such as group access and group administration. To update this permission information, this controller has to modify the wrapped key itself, since the information is stored at this place.

- **headerMenuController**

The main menu is displayed at the top of the page and is the only component that is never replaced on reload events. The corresponding controller is relatively light weight, since no complex operations have to be carried out here. It has to check the permissions of the currently logged in user and present the corresponding allowed menu options. It may happen, that the JavaScript context is wiped, if for example a manual page reload is performed. In those cases, the information has to be loaded again from the SPM Server. For all this operations, the user service is used.

- **passwordController**

The `passwordController` is the central point where most of the information runs together. The actual decryption process of the passwords is started here. In addition, new passwords are generated and encrypted here as well. Therefore this controller strongly uses the communication interfaces and the `cryptoService`. Again the multi-step workflow of the operations contained in this controller are fully described in Section 5.7.4.

- **userProfileController**
Each SPM user has a user profile that includes an image, login credentials and a recovery e-mail address. Additionally, an active user can set the CrySIL username to be able to get invited into password groups. All this information is loaded from the SPM Server and can be managed at this place.
- **userRegistrationController**
The registration process of new users is started in this controller. The user has to perform a registration by himself. Afterwards a new inactive account is stored in the SPM database. Additionally a profile for the new user is generated and stored as well.
- **userManagementController**
As mentioned earlier, there are two distinct levels of permissions. The SPM usermanagement covers permissions concerning managing meta information, such as activating a new registered user. This controller is only accessible by SPM administrators and presents a list of all registered users and their access rights. The administrator is able to update the permissions of other users here. After a user was updated, all other SPM components are informed by the SPM Server. Therefore no re-login interval is needed to flush the current permission state.
- **CrySILAuthController**
CrySIL has its own session bound permission management. If the users should perform a authentication, CrySIL sends a special response back to the SPM Client. When response is received, the SPM Client displays a login form and waits until the user has entered the credentials. Afterwards the server resumes to process the original request.

The following services are available for each controller in the SPM system:

- **cryptoService**
A substantial part of the cryptographic operations that are needed to protect the passwords, is carried out on the client side. The cryptoService offers interface functions that are used by the different controllers. The offered functionality covers key import and export, key generation, encryption and decryption. The cryptoService itself uses the *Web Cryptography API* to execute the cryptographic primitives.
- **loggerService**
This service offers three log-levels to controllers in order to log information. The log-level can be dynamically switched during run-time and completely turned off in production. There is also the possibility to log to a remote server, if needed.
- **crySILGatewayService**
The crySILGatewayService communicates with the CrySIL Server and relays the transmitted information. The communication itself is carried over HTTP via REST calls. Furthermore, the gateway has to handle transmission errors and execute, depending on the returned response code one of the callback functions. Also the CrySIL session handling is managed here.
- **crySILProtocolService**
This service generates CrySIL requests. The CrySIL protocol exactly specifies which payload attributes and HTTP Headers have to be set at a given request. Also the XACML Policy file which protects the password access, is generated in this service.
- **spmGateway**
Similar to the crySILGateway, this service is used to handle the direct HTTP communication. Since the session handling with the SPM Server is based on a stored cookie, the service itself is not responsible for that.

- **userService**
The userService provides the controllers with information about the currently logged in user. This information includes permissions and profile data. In case the JavaScript context is deleted and the information is locally not available anymore, a special REST call is sent to the SPM Server. The server in this case checks the cookie session id and responds with the current user information. Another special case happens, when the information is updated while the corresponding user is logged in. Then this service needs to reload the information and inform all controllers about the update.
- **translationService**
This service was implemented to support multiple languages and in user interface. This component holds references to property files, which include the language strings that are displayed to the user. For each supported language, there is one corresponding property file. The HTML files only include the language keys, which are then replaced by the translation service.

5.7.2 Session Management

The session management of the SPM client consists of the SPM Server and the CrySIL session. These are treated differently since both have their own session information and authentication mechanism.

- **CrySIL Session**
The CrySIL session handshake is initiated when the first request is sent to the CrySIL server. In this request, the session identifier is empty. Therefore the server responds with a so called *authChallengeRequest*. In the payload of this response the type of authentication will be transmitted. Currently there is just one client side implementation for authentication available. This authentication type requests the user for a username and a password. However the SPM client has a generic authentication type management, that can dynamically call registered authentication handlers. If the crySILGateway is called, those authentication types can be handed over as parameters and will be called if an *authChallengeRequest* is received with the matching *authType*.

Listings 5.16 and 5.17 shows how the authHandler is generated and used.

```

$scope.authMethods = {
  "UserNamePasswordAuthType": function(requestObject) {
    var modalInstance = $modal.open({
      animation: true,
      templateUrl: 'static/html/main/crySILAuthModal.html',
      controller: 'crySILAuthModalController',
      size: 'sm'
    });

    modalInstance.result.then(function (authInfo) {
      crySILProtocolInterface.sendAuthRequest(requestObject,
        authInfo.username, authInfo.password);
    }, function () {
      requestObject.reject();
    });
  }
};

```

Listing 5.16: Definition of Authentication Methods

In Listing 5.16 the authentication types are defined in the way how the CrySIL server adds them to the responses, whereas Listing 5.17 directly takes the type out of the response and dynamically calls the corresponding authentication method. The authentication modal view requests the user to

enter the username and a password. The result callback then automatically sends a authentication request back to CrySIL in order to resume the processing of the original request.

```

if (requestObject.getPayload().type === "authChallengeRequest") {
    var authTypes = requestObject.getAuthTypes();
    authTypes.forEach(function(authType) {
        if (authMethods[authType] !== null) {
            authMethods[authType](requestObject);
            return;
        }
    });
}

```

Listing 5.17: Usage of Authentication Methods in the Authentication Handler

The next CrySIL response will afterwards include an HTTP header called *sessionId*. This id is parsed and stored in the JavaScript context by the SPM Client, since it needs to be set at each further request. The session that is identified by this id also stores the information for which type of request the client is authenticated and permitted. So the client has to authenticate for each type of request when sent the first time. After the authentication, the permission information is cached for further requests.

- SPM Session

The interface of the SPM Server is structurally different from the CrySIL Server interface. Since each command on the Server has its own URL path and different HTTP methods as well as different payloads, each call to the server is done by an independent REST function. The main difference in the way the session handling works, is how the session identifier is stored and transmitted. The *sessionId* is a browser cookie and can not be accessed by the JavaScript itself, since the *httpOnly* HTTP flag is set.

The *HttpOnly* attribute limits the scope of the cookie to HTTP requests. In particular, the attribute instructs the user agent to omit the cookie when providing access to cookies via "non-HTTP" APIs (such as a web browser API that exposes cookies to scripts). Barth, 2011b

This flag is set, to reduce the risk of session hijacking, since injected code will not be able to steal cookie information. Therefore the identifier will be sent along each HTTP request automatically as long as the session is valid. This time window is limited by one of these actions (whatever happens before):

- The user logs out from the SPM Client
If this is the case, the JavaScript controller will overwrite the current value of the session cookie.
- The user agent's session is terminated
In all tested user agents, the session is terminated by the time the browser is closed. This happens because neither the *Expires* nor the *Max-Age* attribute is set. Therefore it can be guaranteed, that no session information is persisted on the system, after the session is closed by the client.

If a cookie has neither the *Max-Age* nor the *Expires* attribute, the user agent will retain the cookie until "the current session is over" (as defined by the user agent). Barth, 2011a
- The session is terminated by the SPM Server
The SPM Server has a configurable session max age, that removes the session information on the server side.

5.7.3 Web Security

In contrast to discussing the general security design of SPM as in most parts of this work, this subsection takes a look at the implementation security of the SPM client. Common attack vectors on web applications are:

- SQL injection
- Cross-Site-Scripting
- Cross-Site Request Forgery
- Session Hijacking

AngularJs has several ways how content can be stored in the JavaScript context and subsequently rendered in the DOM. In this work the only used way is the function called *ng-model*, that is attached to an HTML tag. This function connects the content of a scope variable to the content inside the HTML component. This connection is called a two way binding, because an update to one of the two sides will result in automatic synchronisation. The *ng-model* function calls the *\$sanitize* service out of the box. This service sanitizes and escapes the input based on whitelisting.

Sanitizes an HTML string by stripping all potentially dangerous tokens. The input is sanitized by parsing the HTML into tokens. All safe tokens (from a whitelist) are then serialized back to properly escaped html string. This means that no unsafe input can make it into the returned string. AngularJs documentation¹²

Since all the content that is rendered and therefore interpreted by the user agent comes from static HTML templates or is included with *ng-model*, there is no way of executing script injected in a Cross-Site-Scripting attack.

AngularJs also offers a client side mechanism to prevent Cross-Site Request Forgery, based on commonly accepted XSRF mitigation techniques. If a request is sent with the *\$http* service, AngularJs automatically reads the value of a session cookie and adds an HTTP Header with that value. Since only JavaScript that is loaded from the same origin can read the value of this cookie, the SPM Server can be sure, that the call is originating from code that was loaded from the SPM Client domain.

One of the most common attack vectors, the SQL injection, is prevented on the server-side in the SPM setup. As explained in the SPM Server section, prepared statements, that prevent all kinds of injections are used. In this case, an error response code is sent back to the client and no actual update on the database is performed.

To prevent session hijacking session ids are protected by Transport Layer Security (TLS), since all the HTTP traffic is encrypted after the TLS handshake has happened.

The agent acting as the HTTP client should also act as the TLS client. It should initiate a connection to the server on the appropriate port and then send the TLS ClientHello to begin the TLS handshake. When the TLS handshake has finished. The client may then initiate the first HTTP request. All HTTP data MUST be sent as TLS "application data". Rescorla, 20000

5.7.4 Client Cryptography

In this section, the cryptographic functions that are used to carry out the CrySIL protocol are described. As mentioned, the client uses the functions provided by the *Web Cryptography API*.

It was decided to use the *AES-GCM* (Galois/Counter Mode) mode of operation with a key-length of 256 Bit, because it is widely used and is accepted in the scientific community. As Mozaffari-Kermani

¹²[https://docs.angularjs.org/api/ngSanitize/service/\\$protect \T1\textdollar sanitize](https://docs.angularjs.org/api/ngSanitize/service/$protect%20T1%20textdollar%20sanitize)

and Reyhani-Masoleh, 2012 point out, AES-GCM can be executed highly efficient on parallel hardware architectures. It was an important factor to use a mode of operation that provides authenticated encryption.

Rule - Use strong approved Authenticated Encryption

E.g. CCM or GCM are approved Authenticated Encryption modes based on AES algorithm.. OWASP Documentation¹³

Rule - Use Authenticated Encryption of data

Use (AE) modes under a uniform API. Recommended modes include CCM, and GCM as these, and only these as of November 2014, are specified in NIST approved modes, ISO IEC 19772 (2009) "Information technology — Security techniques — Authenticated encryption", and IEEE P1619 Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices OWASP Documentation¹⁴

GCM is based on the Counter Mode (CTR) with the additional property of authenticated encryption.

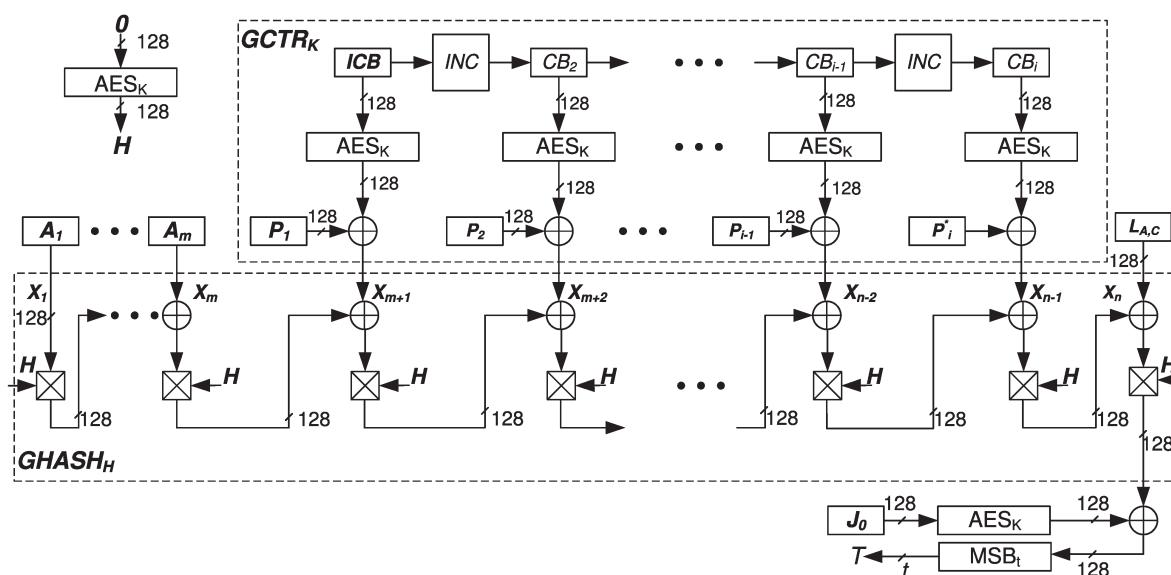


Figure 5.12: Galois-Counter-Mode [Mozaffari-Kermani and Reyhani-Masoleh, 2012]

The following list, describes all interface functions that are offered by the *cryptoService* to all SPM client controllers.

- generateAESKey

This interface generates a new Advanced Encryption Standard (AES) key that will be used for the symmetric inner encryption of a specific password group. The newly generated key needs to be exported later on, since it will be stored in the SPM database.

```
return window.crypto.subtle.generateKey(
  {name: "AES-GCM", length: 256}, //algorithm specification
  true, //flag wether the key can be exported
  ["encrypt", "decrypt"] //permitted operations
).then(function(key) {
  return key;
});
```

Listing 5.18: SPM Inner-Key Generation Result

¹³https://www.owasp.org/index.php/Cryptographic_Storage_Cheat_Sheet

¹⁴https://www.owasp.org/index.php/Cryptographic_Storage_Cheat_Sheet

- encryptAES

The encryption interface, is used for password encryption. Each group shares one symmetric key for this operation. As mentioned, the GCM mode of operation is used for encryption. AES-GCM takes four input parameters:

- Key
Secret-Key with the specified key-length.
- Plaintext
Data to be encrypted.
- Initialization Vector (IV)
AES-GCM-ESP IV field with 8 or 16 octets
- Additional Authenticated Data (AAD)
Combined by the *Galois Mult function* with the ciphertext to produce an authentication tag.

The encryption interface accepts these values, except the IV, as parameters. The IV is generated inside the function using the *RandomSource.getRandomValues()* function of the *Web Cryptography API*.

The *RandomSource.getRandomValues()* method lets you get cryptographically random values. The array given as the parameter is filled with random numbers (random in its cryptographic meaning). MDN Documentation ¹⁵

The interface function itself retruns a JavaScript promise, that will resolve when the algorithm operation has finished. As shown in Listing 5.19, when the encryption is done, the algorithm's output is the resulting ciphertext and the authentication tag, that has to be used to start the decryption process.

```
return crypto.subtle.encrypt(alg, key, buffer).then(function(result) {
  return {cipher: cryptoService.ab2str(result), iv:
    cryptoService.ab2str(iv)};
});
```

Listing 5.19: SPM Inner Encryption Result

- decryptAES

The decryption function works according to the specification of the encryption interface. Of course the same algorithm type is used here. Again the Additional Authenticated Data (AAD) can be provided for verification. Since the ciphertext is converted and stored as a String, it has to be converted to a JavaScript Array-Buffer in the first place. Afterwards, the decryption process can be started using the provided parameter values, which are the ciphertext, the key, the IV and the AAD. In the callback function that is executed when the decryption has finished, the resulting plain text will be again converted to a String. If the verification of the AAD fails or the decryption fails for some other reason, the error callback will be called. In this case, an error object will be returned to the controller. Most of the controllers will show a error message and redirect the user to the error page.

```
return crypto.subtle.decrypt(alg, key, buffer).then(function(plainText) {
  return cryptoService.ab2str(plainText);
}, function(error) {
  return error;
});
```

Listing 5.20: SPM Inner Decryption Result

¹⁵<https://developer.mozilla.org/de/docs/Web/API/RandomSource/getRandomValues>

- **exportKey**

A key object in the *Web Cryptography API* is called *cryptokey*. It does not actually hold the key information itself, it is an identifier, so that the Web Cryptography API can discover the corresponding key. This is done by design, since the key material should never be exposed to the JavaScript itself, unless it is explicitly exported.

Since the local key is uploaded and stored at the SPM server, the export function that serializes the key material, is essential. To mount this operation on a given key, the export flag which is set during the generation process has to be true. During the serialization phase, the key material is transformed into a string of a given format. The SPM client uses the JSON key format (JWK) to represent keys. Listings 5.21 and 5.22 show how the JWK format is used for import and export.

When this specification says to parse a JWK, the user agent must run the following steps:

1. Let data be the sequence of bytes to be parsed
2. Let Json be the Unicode string that results from interpreting data according to UTF-8.
3. Convert Json to UTF-16.
4. Let result be the object literal that results from executing the JSON.parse internal function, with text argument set to a JavaScript String containing json.
5. Let key be the result of converting result to the IDL dictionary type of JsonWebKey.
6. If the "kty" field of key is not defined, then throw a *DataError*.
7. Return key.

(W3C Editor's Draft [Sleeve and Watson, 2015])

```
return window.crypto.subtle.exportKey("jwk", key)
    .then(function(keyMaterial) {
        return keyMaterial;
    });
```

Listing 5.21: SPM Key Export

- **importKey**

In the beginning of the decryption phase, the local key is loaded from the SPM server and is imported into the *Web Cryptography API* context. As a result, the callback function returns the key interface (*cryptokey*). The import function needs the key specification including algorithm and the serialization format, that was used in the export process.

```
return window.crypto.subtle.importKey("jwk", jwk_key,
    {name:aesAlgorithm.name}, false, ["encrypt",
    "decrypt"]).then(function(keyInterface) {
        return keyInterface;
    });
```

Listing 5.22: SPM Key Import

- **str2ab**

JavaScript has introduced so called *typed arrays*, which are array like objects that contain binary data. Those typed arrays are able to grow and shrink automatically. For more dynamic use and better performance, the concept of typed arrays are split up into *ArrayBuffers* and *ArrayViews*. A buffer is just a container that includes binary data. It is not possible to read or manipulate a buffer directly, since it has no context of the data it holds. The content could for example be an integer with the length of 8 bit or a Float with 64 bit length. Therefore the concept of *ArrayViews* was added. This view specifies what the content of such a buffer represents. So a buffer can be manipulated using one (or many) views.

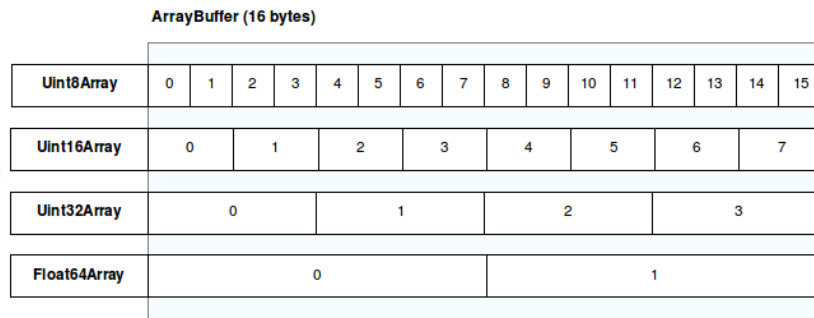


Figure 5.13: Typed Arrays in JavaScript ¹⁶

This JavaScript functionality is very important for the SPM client, since the data that should be encrypted or decrypted is always represented as `ArrayBuffers`. In order to serialize and upload these buffers, the `cryptoService` offers helper functions to convert the binary data into strings. The `str2ab` function, takes a buffer and mounts a `Uint8Array` View on top of it. This way, each element in this view can be uniquely transformed into a char. Listings 5.23 and 5.24 show the serialisation of array buffers and done in SPM.

```
var view = new Uint8Array(buffer);
for (var i = 0; i < view.byteLength; i++) {
    serialisedBuffer += String.fromCharCode(view[i]);
}
return serialisedBuffer;
```

Listing 5.23: Serialization Process for `ArrayBuffers`

- `ab2str`

In order to let the Web Cryptography API functions process the data, that was loaded from the SPM Server, the strings have to be converted back into buffers. This function does exactly that, by again using the same type of `ArrayView`.

```
var bufferView = new Uint8Array(str.length);
for (var i=0; i < str.length; i++) {
    bufferView[i] = str.charCodeAt(i);
}
return bufferView;
```

Listing 5.24: Deserialization Process for `ArrayBuffers`

5.7.5 CrySIL Protocol

The client side CrySIL protocol service (`crySILService`) generates the messages that are exchanged with the CrySIL Server. Those messages have to conform with the format enforced by the server. By the time of development, the current protocol version¹⁷ is `2.0`. This protocol specifies the JSON format for requests and responses of the CrySIL web interface. However the protocol of the current version was extended in order to implement the new CrySIL functionality.

¹⁶Taken from the official Mozilla documentation: https://developer.mozilla.org/de/docs/Web/JavaScript/Typed_arrays

¹⁷<https://demo.a-sit.at/skytrust/protocol-2-0/>

The *crySILService* has interface functions that can be called by the SPM controllers for each command that is called at the CrySIL Server. There are the following interfaces available in the *crySILService*:

- **encrypt**
Takes a list of Strings and the wrapped key. CrySIL will encrypt the list using the symmetric key contained in the wrapped key.
- **decrypt**
Takes a list of Strings and the wrapped key. CrySIL will decrypt the list using the symmetric key contained in the wrapped key.
- **generateWrappedSymKey**
Takes a list of users and a list of admins. CrySIL will generate a new wrapped key including a policy file. This policy file is generated in the internal part of the *crySILService* using the lists of users and admins.
- **updateWrappedSymKey**
Takes a list of users, a list of admins and a wrapped key. CrySIL replaces the policy file with the correctly updated user permission.
- **sendAuthRequest**
Takes a username and a password to send an authentication call to CrySIL.

Each of the interface functions will create a new JavaScript promise that will resolve when the newly generated request was answered by the server. Listing 5.25 shows a small part of interface functions. It can be observed, that in the promise body, which is executed asynchronously, always a *cryptoObject* is created. This object represents the actual request that is going to be sent.

```

var crySILProtocolInterface = {

  encrypt : function (encodedWrappedkey, plainDataList, algorithm) {
    return new Promise(function (resolve, reject) {
      var cryptoObject = new CryptoObject();
      setEncryptRequest(cryptoObject, encodedWrappedkey, plainDataList,
        algorithm);
      cryptoObject.resolve = resolve;
      cryptoObject.reject = reject;

      var header = generateHeader();
      cryptoObject.setHeader(header);
      crySILGateway.makeServerRequest(cryptoObject, authMethods);
    });
  },

  decrypt : function (encodedWrappedkey, encryptedData, algorithm, iv) {
    return new Promise(function (resolve, reject) {
      var cryptoObject = new CryptoObject();
      setDecryptRequest(cryptoObject, encodedWrappedkey, encryptedData,
        algorithm, iv);
      cryptoObject.resolve = resolve;
      cryptoObject.reject = reject;

      var header = generateHeader();
      cryptoObject.setHeader(header);
      crySILGateway.makeServerRequest(cryptoObject, authMethods);
    });
  },
};

```

...

Listing 5.25: Interface functions of the crySILService

These cryptoObjects are created in the internal part of the cryptoService, which can not be accessed from outside the service. The internal message generation functions create and set the message payload according to the protocol specification.

In Listing 5.25, the internal function *setEncryptRequest* is called with the needed parameter values from the interface function. This internal function, which is shown in 5.26 generates the payload for an encrypt request. When this request is sent to CrySIL, the data which is contained in the field *plainDataList* will be encrypted with the symmetric key included in the wrapped key. CrySIL is able to identify the correct command which should be executed using the *type* field in the request. The server side request validation is based on the *type* field in the request, since it is unique.

```
var plainDataListBase64 = [];
for ( var i = 0; i < plainDataList.length; i++) {
    plainDataListBase64.push(btoa(plainDataList[i]));
}
var payload;
if(algorithm === null) {
    payload = {
        "type": "symEncryptRequest",
        "encodedWrappedkey": encodedWrappedkey,
        "plainDataList": plainDataListBase64;
    }else{
        payload = {
            "type": "symEncryptRequest",
            "encodedWrappedkey": encodedWrappedkey,
            "plainDataList": plainDataListBase64,
            "algorithm":algorithm;
        }
    }
cryptoObject.setPayload(payload);
```

Listing 5.26: Interface functions of the crySILService

As mentioned, the internal part of the cryptoService has a function that will generate the payload for each interface that is exposed. However there are some more tasks that are done in this part of the service. Furthermore, the request header for each request is generated here. The header has the same format for each request, as can be seen in Listing 5.27, the only variable fields are the *commandId* and the *sessionId*.

```
var header = {
    "type": "standardCrySILHeader",
    "commandId": commandId,
    "sessionId": sessionId,
    "path": ["java-api-instance"],
    "protocolVersion": "1.3"
};
```

Listing 5.27: Request Header for CrySIL messages

CrySIL is able to identify the correct command which should be executed using the *commandId* field in the request. The *sessionId* field is managed in the crySILGateway and identifies the session permission of the corresponding authenticated user.

As mentioned, the interface functions used to generate or update a wrapped key, take users and admins as input. Before the message payload can be set, a policy file with the given permission information has to be generated. This task is done as well in the internal functions. As described in Section 5.5, the

policy is used to check if a given user is permitted to access the information contained in a password group or to update the policy file itself. The policy is a XACML file containing rules for each of the commands that need a permission check.

To generate such a file, a policy template is parsed in the service and is dynamically extended with the users from the interface parameter lists. Listing 5.28 contains parts of this dynamic permission file generation process. The listing shows, how users and admins are attached to the condition checks of the corresponding rules.

```

for(var i=0; i<users.length; i++){
    //create new user entry
    var user = jQuery.parseXML( userTemplate );
    user = jQuery(user);
    user.find( "AttributeValue" ).text( users[i] );
    //append user to rule
    policyTemplate.find('Rule').each(function() {
        var rule = jQuery(this);
        if(rule.attr('RuleId') === 'readPasswords' ){
            rule.find('Condition').children(0).append(user.children(0));
        }
    });
}

for(var i=0; i<admins.length; i++){
    //create new admin entry
    var user = jQuery.parseXML( adminTemplate );
    user = jQuery(user);
    user.find( "AttributeValue" ).text( admins[i] );
    //append user to rule
    policyTemplate.find('Rule').each(function() {
        var rule = jQuery(this);
        if(rule.attr('RuleId') === 'changePolicy' ){
            rule.find('Condition').children(0).append(user.children(0));
        }
    });
}
return new XMLSerializer().serializeToString(policyTemplate[0]);

```

Listing 5.28: Generation of a New Policy File

5.7.6 Interaction

In this section, the functionality of the SPM Client is summarized by describing how the parts of the client work together, in order to carry out the major workflow functions. It therefore also describes how information is processed and exchanged with the two server endpoints. Only the most important SPM functions that communicate with both servers are described in this section. In most of the commands, a specific order of the messages has to be complied, because the information needed for one request has to be requested first from another part of the system. The description of the workflow is divided into the main actions that are carried out by the SPM Client.

- **Generate Password-Group:**

As mentioned, all passwords inside a group are encrypted with the same inner and outer keys. For that reason, each group has its dedicated keys that are not shared with other groups. When a new group is created, those keys have to be generated and the key usage has to be defined.

The sequence diagram in Figure 5.14 shows an asynchronous communication flow between the controller and the different services. Each step in the workflow of creating a new group requires the

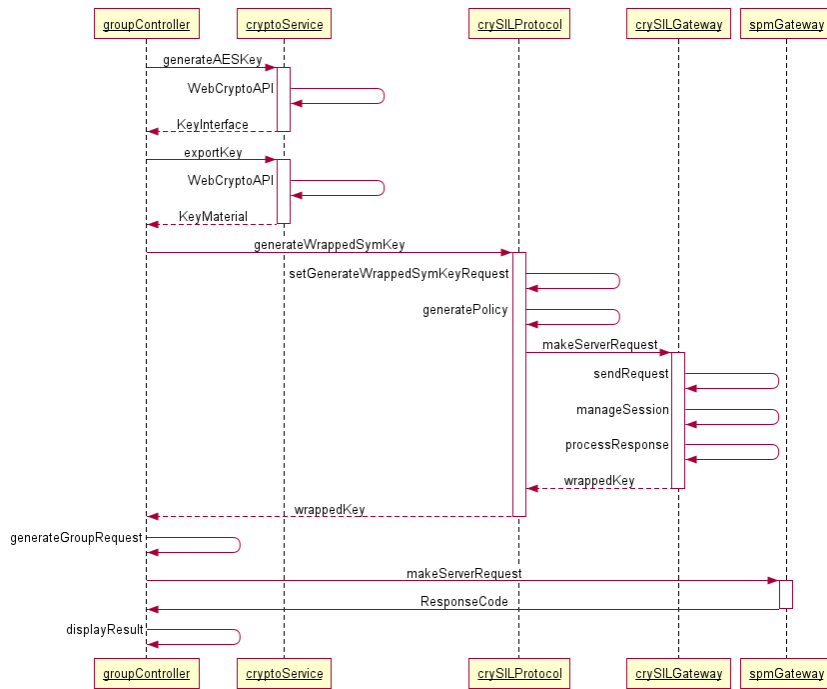


Figure 5.14: Generation of a New Password-Group

previous step to be completed before continuing. Since all calls to the services are asynchronous, this is implemented using JavaScript promises. As shown in Listing 5.29, each step in the workflow is started inside the success callback of the previous one. Once the last promise is fulfilled, the user interface will be updated and the group is successfully stored on the server.

```

//generate new local key
cryptoService.generateAESKey().then(function(key) {
  console.log(key);
  //export local key to jwk
  cryptoService.exportKey(key).then(function (localKey) {
    //generate new crySIL key
    crySILProtocol.generateWrappedSymKey($scope.users,$scope.admins).
    then(function (crySILResult) {
      var encodedkey = crySILResult.encodedWrappedSymKey;
      var newGroup = new NewGroup($scope.newGroupName,
        $scope.newGroupDescription, localKey, encodedkey);
      //store group in spm database
      spmGateway.sendNewGroupRequest(newGroup).then(function (status)) {
        $scope.refreshPage(status);
      });
    });
  });
});

```

Listing 5.29: Generation of Password-Groups on the Client

- Update Password-Group: In order to change the permissions of a group, the wrapped key has to be unwrapped by CrySIL and the policy file has to be replaced. The SPM client first generates an updated version of the policy XML file and sends it to the server together with the wrapped key. At this point, CrySIL will check if the current user has the permission to update the policy file (group admin permission).

Finally the new generated policy file has to be uploaded to the SPM Server.

Once the new wrapped key is generated on the CrySIL Server, users will not be able to use the old wrapped key. For this reason, there is no need to change any key.

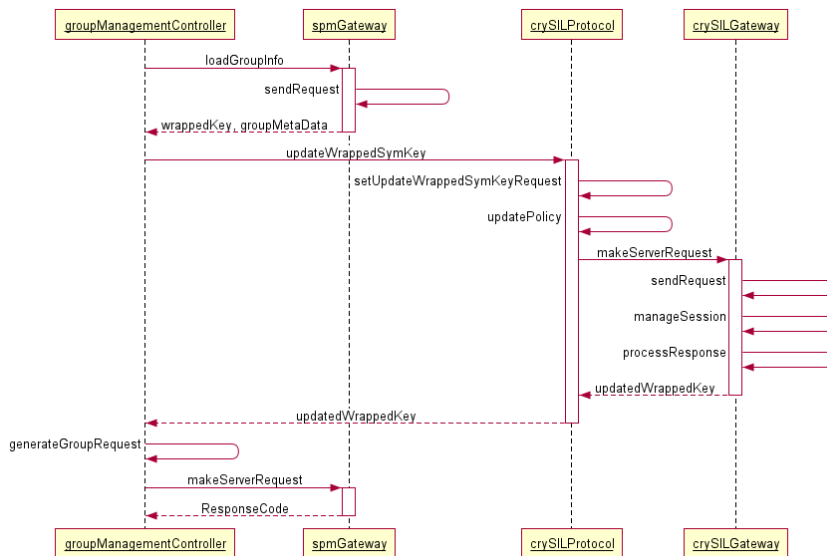


Figure 5.15: Update Process of a Password-Group

- **Encrypt Password:**

To add a new password to an existing group, it has to be encrypted locally and on CrySIL. Afterwards it can be stored in the SPM database. To being able to decrypt later on, the meta information including the IVs and the corresponding group identifier has to be uploaded together with the ciphertext.

- **Decrypt Password:**

When the user opens the group detail page, all passwords that are contained in the group should be displayed in a list. Therefore all passwords should be decrypted first. For performance reasons, the decryption process should not iterate over each individual entry. All passwords are first loaded from the SPM Server and are then decrypted at once at the CrySIL side. The information loaded from the SPM Server additionally contains the IVs which are needed for decryption. Figure 5.17 illustrates the decryption and shows, that the steps are processed in reverse order.

- **CrySIL Authentication:**

For simplicity, the dynamic authentication process against the CrySIL Server was skipped from the above workflow descriptions. Every time the crySILGateway calls the *sendRequest* functions and the user is not yet authenticated for the requested command, the authentication workflow is triggered. Depending of the kind of authentication the server is requesting, an action is triggered on the client. The yet implemented authentication type requests the user to enter a username and a password. After the authentication is successfully finished, the request which led to the need for authentication, will be processed.

Figure 5.18 shows, that after the authentication request is sent to CrySIL, the following message will be the response to the original request.

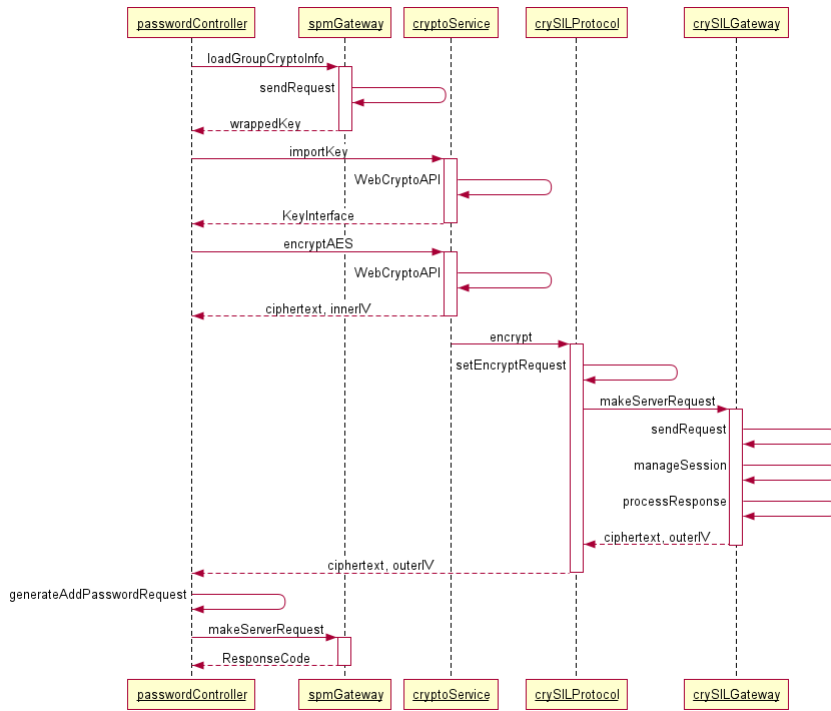


Figure 5.16: Encryption Process of a Password-Entry

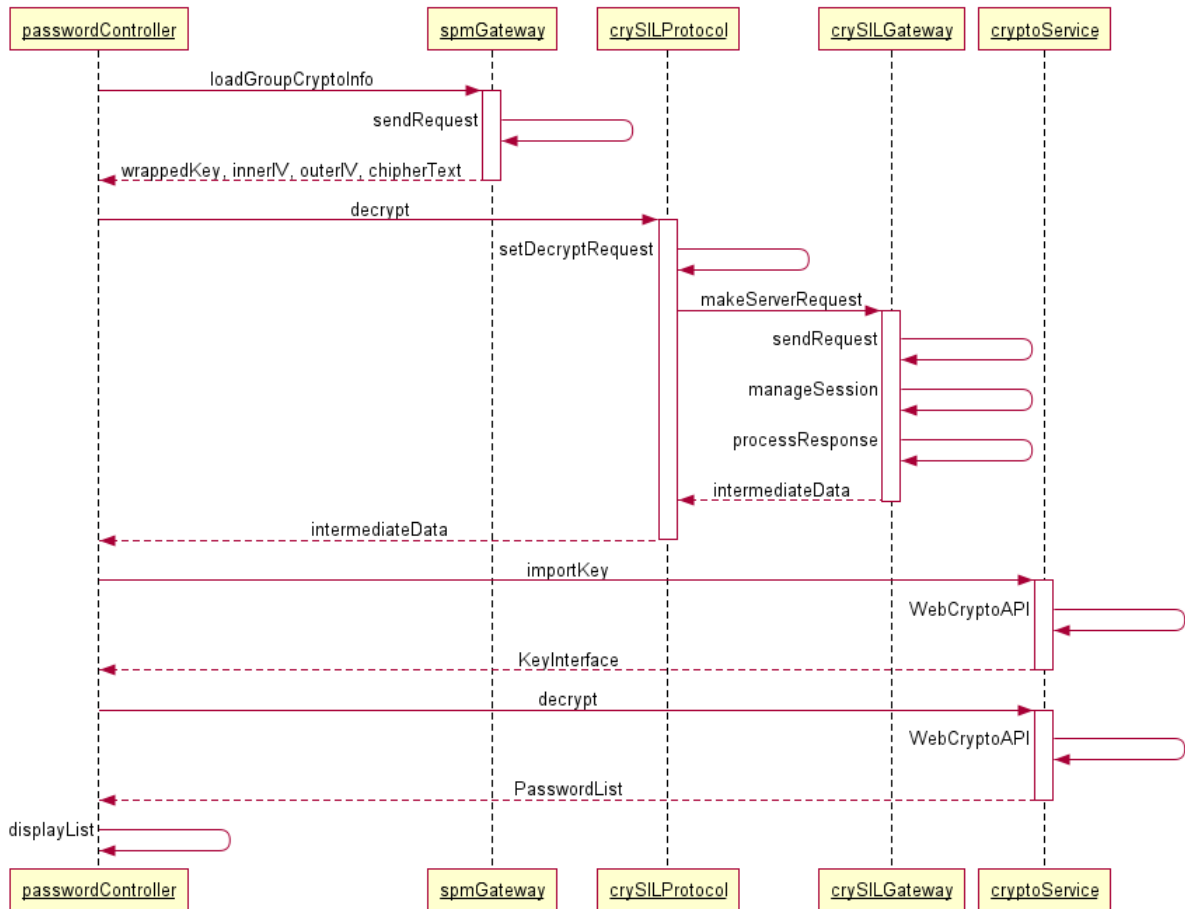


Figure 5.17: Decryption Process of a Password-Entry

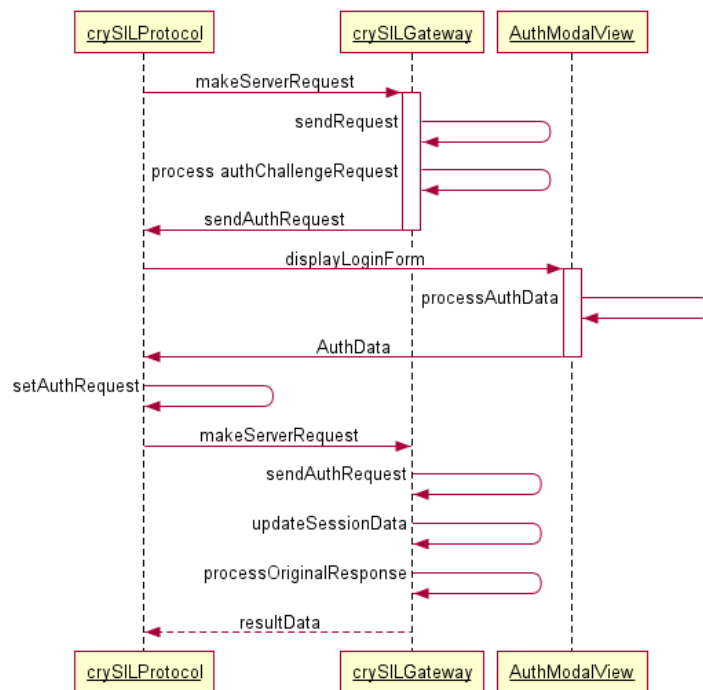


Figure 5.18: Clientside CrySIL-Authentication Process

Chapter 6

Evaluation

In this chapter, the security of the implemented SPM system is discussed. The security evaluation in this chapter is based on the requirements defined in Section 3.1. In the beginning, the different entities, that interact with each other in the SPM ecosystem are classified. Afterwards the assets, that should be protected and the assumptions on considered attacks, are viewed. At the end, different attack scenarios are discussed.

6.1 Component Classification

The major goal of SPM is to implement a password manager that can be, at least partly deployed in untrusted environments. Therefore the SPM server, where the data is stored and the business logic for distribution is located, should be robust against data loss and manipulation. On the other hand, CrySIL is viewed as a service that is carried out by a provider as well. The provider of the CrySIL service should not be able to recover any sensitive data. If one of those servers is taken over by an attacker, which is able to change the data or the logic of the software, data recovery should be denied in any case, however data loss is condoned.

As mentioned, CrySIL is designed to act like a smartcard in the SPM System. Therefore the export of key material is not possible. However, since the keys need to be stored somewhere on the server, a user with local operating system permissions, is able to recover those keys. Therefore it was decided to host CrySIL in an environment, that achieves the *Honest-but-curious* property. In other words, attackers are able to observe any data that is processed by CrySIL, however internal logic can not be manipulated as well as encryption keys can not be exported.

In contrast, the web server delivering the SPM client has to be placed inside a trusted environment, since overtaking this component would result in possible information disclosure. The reason why the client part is vulnerable is, that the information is finally decrypted inside the JavaScript context.

6.2 Assets

The overall goal of the SPM system is to protect sensitive information from unauthorized access. Assets describe the data that need to be protected. The password information consists of the following parts:

[**Name:** The name of the password entry, **Description:** General description of the password, **Account:** User-name/Account name of the associated password, **URL:** The URL on which the service associated with the password can be reached, **Password:** The password itself]

Furthermore, the keys used for encryption and decryption need to be protected as well in order to prevent information reveal.

There are two different keys for two layers of encryption, that are applied to the data:

$$E_{CrySIL}(E_{spm}(PasswordEntry))$$

E_{CrySIL} is the symmetric encryption, performed by CrySIL and E_{spm} is the encryption carried out by the SPM client. The SPM Server stores two IVs used for the two encryption layers and the so called *wrapped key*. The wrapped key is the symmetric CrySIL key, used for the second layer encryption. It is protected by a hybrid encryption scheme explained in Section 5.5.3. The wrapped key also includes a policy file that contains permission information. In the following paragraph the data, that is observed by the different components of the system, is listed.

SPM Client

As already mentioned, the client interacts with the user and has therefore by definition access to the password entry as it needs to display it. This is the main reason why the SPM Client has to be deployed in trusted environments.

CrySIL

CrySIL itself does not store data, except the asymmetric key which is used to encrypt the wrapped key. The policy nonce is a random generated number that is used to identify the version of a policy file. CrySIL processes information when it generates a key or encrypts/decrypts data. Therefore the following information can be observed by CrySIL:

- **First Layer Encryption**
The information, which is passed from the SPM Client to CrySIL, is protected (first layer encryption) before transmission. Since CrySIL has no access to this first layer encryption, the original information can not be observed.
- **Policy File**
The policy file is generated on the SPM Client and transmitted to CrySIL for the wrapped key generation.
- **Data-Encryption-Keys**
The second layer key is generated by CrySIL, protected by a hybrid encryption scheme and sent back to the client. It is used for the second layer encryption, therefore CrySIL observes this key during generation and during each encryption or decryption step. Only CrySIL is able to observe this key. The first layer key is generated on the SPM client and stored at the SPM Server. As mentioned, it is crucial that CrySIL does not observe the first layer key.

SPM Server

As described in the beginning of this section, the SPM Server stores the following information:

- Two layered encryption block
- Initialization vector for the first layer encryption
- Initialization vector for the second layer encryption
- CrySIL wrapped key
- Policy file

The focus of the security of the system lies on the SPM Server, since it is fully untrusted and therefore manipulation of the data and program logic is possible.

6.3 Threat Agents

Threat agents are defined as entities, that attack the presented system in order to reveal assets or corrupt the functionality.

Threat agents should be described by addressing aspects such as expertise, available resources, and motivation. “Common Criteria for Information Technology Security Evaluation” 1999

The threat model is defined from the perspective of an attacker. Since SPM consists of several independent components, the described attacks have different targets in order to gain information about assets. There are two general types of attackers that may target the SPM, inside attackers and outside attackers. Inside attackers own in contrast to outside attackers a valid SPM user account and have therefore more possible attack vectors. Since the SPM Server is completely untrusted, it is assumed that attackers are able to change and observe any information stored at this place. Considering attacks targeting this part of the system is of particular importance, because although passwords are stored encrypted, the SPM Server should not be able to decrypt this information.

Additional to the general categorization of inside and outside attackers, the following threat agents are defined for the security evaluation:

- *TA0 (Standard Attacker)*: attacker without server or network access
- *TA1 (SPM Server Access)*: attacker with access to the internal SPM Server permission logic
- *TA3 (SPM Database Access)*: attacker with read and write access to the SPM database
- *TA4 (SPM Network Access)*: attacker with access to the network layer, allowing inspection or manipulation of ongoing communication between SPM components

6.4 Assumptions

The following paragraph lists assumptions concerning the SPM and the environment in which it is operating. The description of attack vectors in Subsection 6.5 is based on the assumptions state here.

- The operating system of the SPM user is protected against direct attacks. (Including attacks targeting the browser)
- The deployment environment of the SPM components complies with the specification in Section 6.1
- Attackers have access to the SPM source code and insight in internal functions
- The cryptographic functions used by the SPM are correctly implemented
- All components of the SPM are connected to the internet and can be reached by users

6.5 Considered Attacks

This section describes several attack targeting different components of the SPM system. For each of the listed attacks, the threat agent, the motivation of the attack and a description is stated. The description covers how the threat agents perform the attack and in which way the SPM is effected.

SPM Server Permission Logic (AV1)

- Threat Agent: TA1
- Motivation: The aim of this attack is to gain access to assets stored in the SPM database by corrupting the permission system
- Description: It is assumed, that someone with an active SPM account managed to bypass the SPM Server permission logic. As a result, all groups and therefore all available password entries can be downloaded from the SPM Server. The information available to the attacker covers the two layered encrypted password entry, the two IVs and the wrapped key.

At this point in time the attacker has not gained any knowledge about the password entry itself nor who is permitted to decrypt it. In order to recover any information out of the given data, the next step in this attack is the decryption of the password. The outer layer of the password entry is the CrySIL encryption and therefore the attacker needs to communicate with the CrySIL component.

To do so, the decryption process has to be initiated. Since this attack is carried out by an inside attacker, a valid CrySIL account can be used for authentication. Therefore the password entry is sent together with the IV and the wrapped key to the server interface after the authentication process is successfully finished.

On the CrySIL side of the decryption process, the symmetric key is unwrapped. Inside the wrapped key, there is also a policy file which contains the permission information. Before the actual decryption takes place, this file is checked against the authenticated user. At this point, CrySIL will reject the request because the attacker did only bypass the SPM Server permission logic and has no CrySIL account that is permitted for decryption. Therefore the attacker did not learn anything from the leaked information.

SPM Server Database Leak (AV2)

- Threat Agent: TA3
- Motivation: Through direct database access, information about assets should be revealed
- Description: Because the SPM Server is untrusted, it is important to consider database leakage. In this scenario the attacker is able to gain access to the database and dump any information available.

Additional to the information which was gained in the previous attack, now there is permission information available. The database stores meta information, covering which user has the permission to read passwords or administrate a group. However this data is not viewed as secret and therefore it is not protected.

Again the next step to recover data needs decryption by CrySIL, which is, as already explained, not possible without an CrySIL account. Since this account information is not stored in the SPM database, no further steps can be taken from this point.

SPM Server Database Manipulation (AV3)

- Threat Agent: TA3

- **Motivation:** By manipulating the contents of the database, the system functions of SPM should be corrupted
- **Description:** In this kind of scenario, the attacker is able to manipulate the contents of the SPM database. There are different types of data that can be manipulated. In the one hand meta information concerning permissions and in the other hand ciphertext, IVs and crypto keys are stored at this place.

Manipulating permission information can lead to permission upgrades, which are already covered in the attack *SPM Server Permission Logic*. As already described for this kind of attack, the global system security does not rely on the SPM permission system. In other words, manipulating permission information does not threaten the confidentiality of the system.

When the manipulation of ciphertext and crypto data is examined, two different goals have to be considered. The first and obvious target is, that an attacker can not learn from the recovered data. The second and very important goal is, that any manipulation is detected and reported to the user. That means, if ciphertext, IVs or crypto keys are manipulated by an attacker, decryption should fail and no output should be generated. This requirement is achieved by using authenticated encryption [McGrew, 2008]. Authenticated encryption is a mode of operation that simultaneously provides integrity, confidentiality and authenticity.

Authenticated encryption [BN00] is a form of encryption that, in addition to providing confidentiality for the plaintext that is encrypted, provides a way to check its integrity and authenticity. McGrew, 2008

CrySIL, as well as the SPM client, use different types of symmetric encryption schemes but both use authenticated encryption. Therefore, if the data is manipulated, the integrity verification during the decryption step will fail and an error is returned. If this is the case, the attacker can not learn from the output of the decryption and a valid user who tries to decrypt will be informed about the error. As a result integrity, confidentiality and authenticity of the data can be guaranteed but data loss is possible.

Permission Upgrade (AV4)

- **Threat Agent:** TA0
- **Motivation:** An attacker that once had access to a password group, should regain password permissions
- **Description:** This attack is based on the revoke permission functionality of the SPM system. It is possible for group administrators to remove someone from a group or in other words withdraws the decryption permission. As already mentioned, information about who is permitted to decrypt a given password entry is stored in a policy file, that is placed inside the CrySIL wrapped key. Therefore when permissions are updated by the administrator, the policy file and therefore also the wrapped key is updated. When the wrapped key is updated by CrySIL, it will immediately be uploaded to the SPM Server. This workflow guarantees, that after the policy update, the correct permissions are checked for the future.

This attack scenario assumes that the attacker has access to old versions of the wrapped key. A possible sequence of actions could be:

1. A user has group access and backups the wrapped key

2. The group administrator removes the user from the group
3. The user takes the ciphertext and sends it together with the old wrapped key to CrySIL for decryption

To detect this attack, the policy file itself needs a notion of time. Therefore a nonce process was introduced in the policy file. This process guarantees, that CrySIL identifies the newest version of the wrapped key and is able to reject old versions. So if the attacker requests the decryption in step (3), CrySIL will reject the request, because old versions of policy files are detected.

CrySIL Transmission Observation (AV5)

- Threat Agent: TA4
- Motivation: An attacker wants to learn from data, which is exchanged between the system components.
- Description: In this scenario the attacker is able to observe the communication between the SPM Client and CrySIL during encryption and decryption requests. The information, which can be observed at this point can be summarised to:
 - CrySIL wrapped key
 - IV for encryption/decryption
 - input for encryption
 - output of encryption

The IV which is used for encryption, is not an asset that needs protection, therefore its observation does not harm the system security. The input for the encryption is protected by the SPM client before transmission. Therefore all observed elements are either cryptographically protected or do not threaten the security.

Cross Site Scripting (AV5)

- Threat Agent: TA3 / TA4
- Motivation: By manipulating the HTML and Javascript contents, information located at the SPM Client should be revealed
- Description: The most common attack on web applications are so called *Cross Site Scripting* attacks.

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site.

(OWASP Documentation ¹)

An example for this kind of attack applied to the SPM system could look like the following: The attacker is able to edit the contents in the SPM database and inserts JavaScript code in a property, that is displayed on the password page, for example the name of the group. When this script is executed by the browser, it collects the decrypted password data and sends it to an external server.

The SPM Client is aware of script injection and uses a white listing approach to sanitize input, before it is interpreted by the user agent. This and other web security measures of the SPM Client are described in Section 5.7.3

6.6 Residual Risks

It was shown, that as long as the trust classification is not violated, SPM is able to protect the shared information from unauthorized access. This sections is going to describe the impacts of an attack, that is mounted on SPM without the defined trust classifications. In this scenario the attacker is able to completely overtake both cloud modules, which violates the *honest but curious* constraint of CrySIL. If this is the case, both encryption keys, needed to reveal any shared protected information, are accessible and can be used by the attacker. Furthermore, the wrapped key can be unwrapped by the attacker and consequently the permission information can be updated.

This attack leads to a total breakdown of the overall system security. If therefore the trust classification is not realizable for a setup of SPM, some countermeasures have to be deployed. One possibility is to protect the key encryption key *KEK* by encrypting it based on a password of the user, using a key derivation function. The user therefore needs to enter a password, when they *KEK* is used, in order to open the wrapped key. Additionally the same protection can also be applied to the first layer encryption key at the SPM Server. If this is done, the integrity of the stored shared information can also be guaranteed, if both server modules are taken over by an attacker.

6.7 Performance

This section is going to briefly describe the run-time performance of SPM during the encryption phase. The two most relevant parts for this analysis are the two independent AES encryption rounds and the network transport. In order to perform a full encryption step of protected shared data entries, the following tasks have to be performed:

1. Loading the group meta data from the SPM Server
2. Encrypt (first layer) the local data using AES-GCM
3. Transmit the resulting chipher text to CrySIL
4. Unwrap the wrapped key (AES-GCM operation) in order retrieve the data encryption key
5. Encrypt (second layer) the data a second time with AES-CBC
6. Store resulting protected shared data entry at the SPM Server

Each AES encryption or decryption operation, that is performed during the described workflow, needs linear time with respect to the input size. When viewing the asymptotic run-time behaviour, the mode of operation does not make a difference. The unwrap key operation is applied to the wrapped key, which

¹[https://www.owasp.org/index.php/Cross-site_Scripting_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))

is independent of the input data and can therefore be viewed as constant in size. Based on this fact, also the run-time behaviour of the unwrap operation is constant with respect to the input size of the shared data. As defined in the above listing, the input data needs to be encrypted twice and therefore the overall complexity is $O(n)$.

In order to analyse the time consumption of the transmission between the SPM system components during the encryption, a test setup with the following properties was deployed:

- Each module was deployed on a separate physical machine
- Operating System: Arch Linux (Kernel Version 4.6.4-1)
- User Agent: Chromium 52.0.2743.85 (64-bit)
- CPU of CrySIL Server: Intel Core i5-3450S (2.80GHz)
- CPU of SPM Server and SPM Client: Intel Core i5-4460 (3.20GHz)
- Tested in local network over a Ethernet cable connection

Figure 6.1 illustrates the time consumption of the individual steps during encryption. The chart shows, that the most expensive task is the second layer encryption at CrySIL. This is because, apart from the two AES operations (key unwrapping and data encryption), CrySIL authentication and handover logic need to be executed as well.

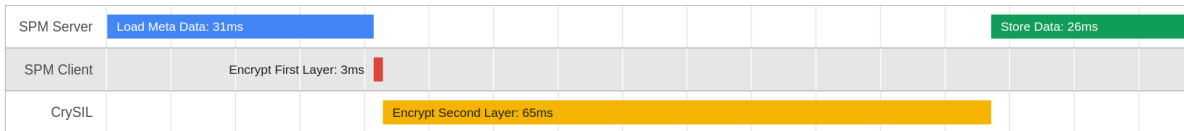


Figure 6.1: Time consumption of the Encryption Process

6.8 Conclusions

It was shown, that if the trust classification defined in Chapter 3 for each component holds, the integrity of the system can be guaranteed for many kinds of different attacks. Table 6.1 gives an overview of the discussed attacks and the impact to SPM.

Attack Vector	Possible Data Loss	Denial of Service	Information Disclosure
AV1		x	
AV2			
AV3	x	x	
AV4	x	x	
AV5			

Table 6.1: Summary of Attack Vectors and their Impact

Furthermore, the impacts of an attack targeting SPM with a violated trust environment was discussed. In this case, the integrity can be guaranteed, if a modification of the server-client protocol is applied. The presented modification could be part of future work at SPM.

Chapter 7

Conclusion

In the past years people used more services in their private and work life than ever before. Since most of these services are connected to an user account, authentication needs to be performed. According to a survey of the British government [UK, 2015], an average Briton now needs to remember 19 different passwords. More than one third of the questioned people said, that they used weak passwords, because they have hard times to remember strong passwords.

Another general security problem is, that many people use the same password for different services. This is especially dangerous, because users never know, how their passwords are stored by the services providers. Nowadays big password leaks of well known online services are happening frequently. Therefore, a high amount of password information is available on the internet. The security analyst and author Burnett [2015] has shown this, by publishing a list of ten million leaked passwords on his blog.

Weir et al. [2015] have shown, that password policies that force users to include certain chars or numbers do not increase the overall password security. This effect happens, because most of the users tend to take weaker passwords, when forced to a certain length or char set.

Possible solutions for this problem are multi factor authentication or certificate authentication mechanism. However, those authentication modes are not available for most of the services that are usually used.

Password managers, that securely encrypt the authentication information are therefore the solution for those problems. Passwords can be generated with a high amount of entropy, without forcing the user to remember them. However, those passwords often need to be shared among a group of people. Especially in companies, where different people use the same services, this is the case.

For this reason, an online password manager, designed to share information in groups was described and implemented in this work. The focus of the design, was to implement a distributed system, that is able to run in untrusted environments. This is especially important, because otherwise a leak of the application database could reveal all passwords. Furthermore, many people do not run their own servers and therefore use web-spaces, that are not fully controlled by them.

To evaluate the security of the implemented password manager, a security evaluation including different kinds of attacks was done. Web-based attacks such as cross-site-scripting are covered as well as permission manipulation or data theft. The evaluation has shown, that the designed system, is robust against many kinds of attacks.

Bibliography

- Ateniese, Giuseppe et al. [2005]. “Improved Proxy Re-Encryption Schemes with Applications to Secure Distributed Storage” (2005) (cited on page 4).
- Barth, A. [2011a]. “Request for Comments: 6265 <https://tools.ietf.org/html/rfc6265#section-4.1.2>” (2011), pages 9–10 (cited on page 57).
- Barth, A. [2011b]. “Request for Comments: 6265 Section 4.1.2 <https://tools.ietf.org/html/rfc6265#section-4.1.2>” (2011), pages 9–10 (cited on page 57).
- Barth, A. [2011c]. “Request for Comments: 6454, The Web Origin Concept <http://tools.ietf.org/html/rfc6454>” (2011) (cited on page 19).
- Barth, A. [2011d]. “Request for Comments: 6454, The Web Origin Concept Section 3 <http://tools.ietf.org/html/rfc6454#section-3>” (2011) (cited on page 18).
- Barth, A. [2011e]. “Request for Comments: 6454, The Web Origin Concept Section 4 <http://tools.ietf.org/html/rfc6454#section-4>” (2011) (cited on page 18).
- Burnett, Mark [2015]. “Cryptographic Message Syntax (CMS) <https://xato.net/today-i-am-releasing-ten-million-passwords-b6278bbe7495#.xv3mdwiyn>” (2015) (cited on page 79).
- “Common Criteria for Information Technology Security Evaluation” [1999] (1999) (cited on page 73).
- Cranor, Lorrie [2016]. “Time to rethink mandatory password changes <https://www.ftc.gov/news-events/blogs/techftc/2016/03/time-rethink-mandatory-password-changes>” (2016) (cited on page 1).
- Fahl, Sascha et al. [2013]. “Hey, You, Get Off of My Clipboard” (2013) (cited on page 5).
- Housley, R. [2009]. “Cryptographic Message Syntax (CMS) <http://tools.ietf.org/html/rfc5652#section-6.3>” (2009) (cited on page 38).
- Kaliski, B. and Eiji Okamoto [2000]. “Password-Based Cryptography Specification” (2000) (cited on page 10).
- Mambo, Masahiro and Eiji Okamoto [1997]. “Proxy Cryptosystems: Delegation of the Power to Decrypt Ciphertexts” (1997) (cited on page 4).
- McGrew, D. [2008]. “An Interface and Algorithms for Authenticated Encryption” (2008) (cited on page 75).
- Mozaffari-Kermani, Mehran and Arash Reyhani-Masoleh [2012]. “Efficient and High-Performance Parallel Hardware Architectures for the AES-GCM” (2012) (cited on pages 58, 59).
- Reimair, F., P. Teufl, C. Kollmann, et al. [2015]. “MoCrySIL – Carry Your Cryptographic Keys in Your Pocket” (2015) (cited on page 33).

- Reimair, F., P. Teufl, and T. Zefferer [2015]. “Web Cryptographic Service Interoperability Layer” (2015) (cited on page 17).
- Rescorla, E. [20000]. “Request for Comments: 2818 <https://tools.ietf.org/html/rfc2818>” (20000) (cited on page 58).
- Schneier, Bruce [2016]. “Frequent Password Changes Is a Bad Security Idea https://www.schneier.com/blog/archives/2016/08/frequent_passwo.html” (2016) (cited on page 1).
- Sleevi, Ryan and Mark Watson [2015]. “Web Cryptography API - W3C Editor’s Draft 12 <https://dvcs.w3.org/hg/webcrypto-api/raw-file/tip/spec/Overview.html>” (2015) (cited on pages 22, 61).
- UK, HM Government [2015]. “CYBER STREETWISE - Open for Business” (2015) (cited on page 79).
- Van Kesteren, Anne [2014a]. “Cross-Origin Resource Sharing <https://www.w3.org/TR/2014/REC-cors-20140116/>” (2014) (cited on page 20).
- Van Kesteren, Anne [2014b]. “Cross-Origin Resource Sharing Section 5 <https://www.w3.org/TR/2014/REC-cors-20140116/#syntax>” (2014) (cited on page 20).
- Weir, Matt et al. [2015]. “Testing Metrics for Password Creation Policies by Attacking Large Sets of Revealed Passwords” (2015) (cited on page 79).
- Zhao, Rui, Chuan Yue, and Kun Sun [2013]. “Vulnerability and Risk Analysis of Two Commercial Browser and Cloud Based Password Managers” (2013) (cited on page 3).
- Ziegler, Dominik et al. [2014]. “Do You Think Your Passwords Are Secure?” (2014) (cited on page 5).