Hubert Gasparitz

SkyTrust: Design and Security Analysis of a Flexible Cloud-Based Key Storage Solution

Master's Thesis

Graz University of Technology

Institute for Applied Information Processing and Communications (IAIK)
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Reinhard Posch

Evaluator: Univ.-Prof. Dipl-Ing. Dr.techn. Reinhard Posch
Supervisor: Dipl-Ing. Dr. techn. Peter Teufl

Allerheiligen bei Wildon, January 2014

Statutory Declaration

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 marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

Powered by the cloud-computing trend, central web-services have significantly gained relevance in the recent years. Several central service providers offer various services which provide the user the ability to store and access their data everywhere and anytime. Security critical applications that perform cryptographic operations represent an exception to this trend and store cryptographic keys in security tokens which are in possession of the user. Nevertheless, the demand for cloud-based cryptographic solutions has also increased and therefore several institutions have developed appropriate solutions that provide authentication and electronic signature creation. The major drawback of these solutions is that they are mostly limited to a certain scope or environment.

To remedy the limitations of these solutions and provide an alternative the *Skytrust* project was started. The Skytrust system provides a flexible and extensible central server-based key storage solution that offers all primitive cryptographic functions for arbitrary devices such as smartphones, tablets or web browsers. The Skytrust system is accessible over a platform independent interface that enables a vast amount of application scenarios.

In this master's thesis we designed and implemented a basic prototype of the *Skytrust* system, which demonstrates the integration of the central server-based key storage solution into a web application. Based on this prototype an *Application Permission System* is proposed that can be used to maintain the access to such a complex system. In order to understand the risks of the *Skytrust* system a comprehensive security analysis on the implemented prototype is conducted to determine threats and countermeasures of the system.

Key words: server-based key storage solution, cryptographic services, security analysis

Kurzfassung

Der Cloud-Computing Trend der letzten Jahre hat dazu geführt, dass zentrale Web-Services an Popularität gewinnen. Diese zentralen Webservices ermöglichen es dem Benutzer immer und überall auf seine gespeicherten Daten zuzugreifen. Sicherheitsrelevante Anwendungen sind hier eine Ausnahme und speichern kryptographische Schüssel auf Security-Tokens, welche sich im Besitz des Benutzers befinden. Nichts desto trotz steigt aber auch die Nachfrage nach serverbasierten kryptographischen Services. Zahlreiche Institutionen haben nun kryptographische Services entwickelt, die Authentifizierung und das Erstellen von Digitalen Signaturen ermöglichen. Ein großer Nachteil dieser Lösungen ist aber, dass sie meist eine eingeschränkte Funktionalität anbieten oder auf ein bestimmtes Anwendungsszenario limitiert sind.

Um diese Einschränkungen zu beseitigen wurde das *Skytrust* Projekt gestartet. Das Skytrust System bietet eine zentrale serverbasierte Schlüsselspeicherlösung, die alle kryptographischen Basisoperation für verschiedenste Geräte wie Smartphones, Tablets oder Webbrowsers zur Verfügung stellt. Das Zugang zum Skytrust System wird über ein plattformunabhängiges Interface realisiert, welches eine breite Palette von Anwendungsszenarien ermöglicht.

Hauptaufgabe dieser Masterarbeit war es, einen Prototypen des *Skytrust* Systems zu entwickeln, der die Integration des zentralen kryptographischen Schlüsselspeicherlösung in eine Webanwendung demonstriert. Basierend auf diesem Prototypen wurde ein *Application Permission System* entwickelt, welches den Zugriff auf solch ein System regeln soll. Da Webanwendungen in einer risikobehafteten Umgebung ausgeführt werden, wurde weiter eine Sicherheitsanalyse basierend auf dem Prototyp durchgeführt, um Bedrohungen und Gegenmaßnahmen des Systems zu identifizieren.

Stichwörter: Serverbasierte kryptographische Schlüsselspeicherlösung , Kryptographisches Service, Sicherheitsanalyse

Acknowledgements

At first I would like to thank my advisor Peter Teufl, who enabled me to work on the Skytrust project. Whenever any question came up during the work on the thesis, he had time for discussions and gave me new suggestions. It was a special experience to work on such a great topic, which has so much use cases in the real life. A special thank for the correction of the draft versions of this thesis. Additionally to Peter I would like to thank Florian Reimair for the support and the interesting discussions during the entire work.

Second, I would like to thank my parents Margareta and Hubert for the support during the whole study.

Third, I would like to thank my girlfriend Elisabeth for the understanding in difficult times of the study and this thesis.

Finally, a special thank to my brother Jürgen who inspired me to start a study and supported me in difficult times.

Contents

List of Figures

List of Tables

List of Abbreviations

APDU Application Protocol Data Unit
API Application Programming Interface
CA Certificate Authority
CORBA Common Object Broker Architecture
CORS Cross Origin Resource Sharing
CSRF Cross Site Request Forgery
HSM Hardware Security Module
HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
HTTPS Hypertext Transfer Protocol Secure
JCE Java Cryptography Extension
JMS Java Message Service
JSON JavaScript Object Notation
JSONP JavaScript Object Notation with Padding
MITM man-in-the-middle
MTOM Message Transmission Optimization Mechanism
NFC Near Field Communication
OTP One Time Password
PKCS Public Key Cryptography Standard
REST Representational State Transfer
RMI Remote Method Invocation
SCBE Skytrust Client Browser Environment
SE Skytrust Element
SMTP Simple Mail Transfer Protocol
SOAP Simple Object Access Protocol
SOP Same Origin Policy
SSCD Secure Signature Creation Device
SSE Skytrust Server Environment
SSL Secure Socket Layer
TLS Transport Layer Security
URI Uniform Resource Identifier
URL Uniform Resource Locator
W3C World Wide Web Consortium
XML Extensible Markup Language
XSS Cross-Site Scripting

Chapter 1

Introduction

In recent years central server-based services gained more and more reception and a further growth is predicted by Gartner [20]. Various service providers, for example, Amazon [1] or Google [2], offer *cloud services* which enables the user to store and maintain their documents and data on a web server in the Internet. An important reason for the high demand for cloud services is resulted from the platform heterogeneity. The platform heterogeneity enables the user to the access the data conveniently from various devices such as smartphones or tablets with different operating systems, for example, iOS and Android from everywhere and at any time.

However, security relevant applications that execute cryptographic operations do not entirely follow this trend. These applications store and maintain the cryptographic keys for encryption and decryption mostly in a cryptographic device which is in possession of the user. For instance, a bank card that is issued from a banking company to the user is utilized to authorize financial transactions of the user. Another example are e-identity cards such as the Austrian citizen card [35] that offers citizens the possibility to authenticate themselves or to create electronic signatures.

Although it is obvious to leave sensitive cryptographic key material directly at the user instead of storing it on a central server, this approach has a certain drawback. In case of a local smart card that stores the cryptographic key material the user has to take along the card everywhere and anytime. If the smart card is not available, broken or the smart card is incompatible to a particular system the service cannot be used. Unfortunately, the local based solution for private cryptographic key material is not flexible and platform independent enough to fulfil the requirements of modern applications.

Therefore, the leading industry has recognized that a central storage and usage of cryptographic sensitive key material is indispensable. Several companies such as *Cryptomathic* [3] offer the possibility to create electronic signatures with private signing keys that are stored on a central server storage. A solution similar to Cryptomathic, is the

[1] `http://aws.amazon.com/`
[2] `https://appengine.google.com/`
[3] `http://www.cryptomathic.com`

Austrian Mobile Phone Signature [4] that is offered by an Austrian company called *A-Trust*. This system allows the Austrian citizens to authenticate themselves and create qualified electronic signatures according to the European Signature Directive [15]. Another solution for storing cryptographic keys on a central storage and execute cryptographic operations is offered by Amazon. The *Amazon CloudHSM* [5] provides the user full access to a central stored Hardware Security Module (HSM) that can be used to perform cryptographic operations as well as to store cryptographic key material.

The main disadvantages of the above solutions are the restriction to certain cryptographic operations and the limitation to a particular infrastructure. For instance, the *Austrian Mobile Phone Signature* is limited to the creation of electronic signatures and therefore not able to perform cipher operations. On the other hand, the *Amazon CloudHSM* is restricted to a certain client that is able to communicate with the particular *Virtual Private Cloud* solution of Amazon.

To keep the limitations of the existing solutions aside, the *Skytrust system* project was started to provide a flexible, extensible, and platform independent solution that allows the central storage of cryptographic key material as well as the execution of cryptographic operations. To address these aims a core element called *Skytrust Element* is used. It breaks up the usual access of a cryptographic service into a receiving component, called *receiver*, and an acting component, called *actor*. This separation allows the flexible combination of several acting components that store keys and provide cryptographic services such as HSMs or smart cards with various receiver components such as web applications and smartphone applications.

The task of this thesis was to demonstrate the functionality of the *Skytrust system* by designing and implementing a basic prototype that enables the utilization of a server-based *Skytrust Element*, that connects a cryptographic key storage solution, from a web application. The connection between the server and the web application is established over a provided *web page*, that exposes the *API* of the server. Furthermore, a suggestion for an *Application Permission System*, that restricts the access to the *Skytrust* system, was elaborated. Finally we conducted a comprehensive security analysis to determine possible threats and countermeasures of the *Skytrust* system.

The remainder of this thesis is structured as follows:

Chapter 2 discusses the general background that is covered in the thesis. This includes basic knowledge about the OAuth framework which allows the delegation of the authentication process to another service. Then, a short introduction into web service technologies such as *Simple Object Access Protocol (SOAP)* as well as *Representational State Transfer (REST)* is given. Further, the HTML5 web messaging technology is considered. Finally, restrictions that are associated with web messaging such as *Same Origin Policy (SOP)* and *Cross Origin Resource Sharing (CORS)* are explained.

[4] `https://www.handy-signatur.at/`
[5] `http://aws.amazon.com/de/cloudhsm/`

Chapter 3 gives an overview on standard cryptographic key storage solutions such as smart cards, security tokens and Hardware Security Modules as well as related existing web-based solutions for cryptographic key storage such as the Amazon CloudHSM, Cryptomathic, SigningHub, Dictao, and the Austrian Mobile Phone Signature.

Chapter 4 introduces the basic *Skytrust* system design. A description of the *Skytrust Element*, the according protocol and the implemented prototype is given. Furthermore, the elaborated thoughts about an *Application Permission System* are presented.

Chapter 5 presents the results of the comprehensive security analysis on the implemented prototype with respect of a particular scenario. The security analysis includes the identified assets of the system, threats and countermeasures as well as an enhancement of the web messaging communication.

Finally, conclusions about the *Skytrust* system prototype are drawn and approaches for future work are given.

Chapter 2

Background

This chapter covers the general background that is needed in this master's thesis. Pushed by the *Web 2.0* development, web services became a key component of the World Wide Web. Central server-based web services that are offered over the Internet allow the user to store and process data everywhere and anytime. In order to use these web services, authentication systems to restrict the access are used. Beside the common *username / password* authentication the OAuth authentication scheme gained more and more relevance during the last years. We discuss the OAuth authentication scheme which enables the delegation of the authentication to a trusted third identity provider. Web services are designed to allow machine-to-machine communication by offering an interface. In order to interact with the web service interface different technologies such as *Simple Object Access Protocol (SOAP)* and *Representational State Transfer (REST)* are used. Since we utilize the REST technology in this master's thesis, we have a closer look on this technology. In order to use a web service in a user agent respectively a browser mainly a web interface or web application is used. The *Same Origin Policy (SOP)* of the browser ensures that an uncontrolled access to the windows or frames is not possible. In order to transfer data between the web application and the embedded frame in the browser the HTML5 web messaging standard is utilized. This standard overcomes the SOP and ensures the controlled transfer of messages between windows of different origins.

In the remainder of this section, we will discuss at first the OAuth authentication scheme in Section 2.1. Section 2.2 describes the web service technology in consideration of SOAP and REST. In Section 2.3 we focus on the HTML5 web messaging communication mechanism. Finally, Section 2.4 and Section 2.5 discuss the SOP and the CORS technology which is also used to enable cross-origin data transfer.

2.1 OAuth

In 2007 the first OAuth Core 1.0 specification [23, 42], an open standard for authentication delegation was introduced. The specification defines a protocol that allows a user to grant a requesting application limited access to a web account without showing any

sensitive data of the user.
However, the OAuth protocol is not a new technology. It combines previous developed protocols such as *Google AuthSub* [1] or *Yahoo BBAuth* [2]. The OAuth authentication standard spread out very well, so that nowadays nearly every popular provider supports the authentication standard.

Due to security issues [22] and complexity, in 2012 the successor of the OAuth core 1.0 specification standard the *The OAuth 2.0 Authorization Framework* [25] was presented by the *Hardt Auth Work Group*. The major adjustment to OAuth 1.0 is that OAuth 2.0 now supports different control flows for different deployment scenarios such as web server applications and mobile applications. Due this adjustment, the new framework is no longer backward compatible to the previous version. The ongoing development of the OAuth web authorization protocol is documented at [3].

The remainder of this section discusses the core parts of the OAuth 2.0 web authorization framework based on the *The OAuth 2.0 Authorization Framework* [25]. First, preliminaries such as roles, client registration and client profiles are discussed. Second, the basic protocol flow is considered. Third, the authorization grant types are covered and finally, a brief overview on security considerations about OAuth 2.0 is given.

2.1.1 Roles

In order to understand the OAuth protocol, the following basic roles that occur in the protocol flow have to be considered:

- **client**
  The client is an application which requests access to the protected data of the user (e.g. third-party application).
- **resource owner**
  The resource owner is a person or an application which grants the access to the protected data.
- **resource server**
  The resource server is a web server which hosts the protected data.
- **authorization server**
  The authorization server authorizes the access to the protected data. The resource server and the authorization server are in some cases the same server.

[1] `https://developers.google.com/accounts/docs/AuthForWebApps`
[2] `http://developer.yahoo.com/auth/`
[3] `http://datatracker.ietf.org/wg/oauth/`

2.1.2 Client Registration

In order to initiate the OAuth protocol flow the client application has to be registered at the authorization server to identify the requesting application on authorization server side when a request is sent. The registration is usually done by an HTML registration form where the developer of the application defines a *client type*, a *client redirection url* and *some additional information* like application name, website, logo image. As *client types* the OAuth framework distinguishes the following types:

- *confidential* and
- *public*

The difference between these two types is the ability to store the client credentials. The *confidential client* is able to store the client credentials confidential. For instance, a web server application which stores the client credentials in a safe key store.

In contrast to the *confidential client*, the *public client* is not able to store the client credentials confidential as it is usual in, for example, JavaScript or PHP applications. In such a case the *client credentials* are stored in the application code and a potential attacker is able to obtain the credentials from the code.

The *redirection url* defines the endpoint of the requesting application where the authorization server redirects the resource owner when the requesting application was granted.

After the successful registration the client credentials, consisting of a *client identifier (clientID)* and a *clientSecret*, are issued to the developer. Whereas the *client identifier* is a simple unique string that represents the client, the *clientSecret* is a secret key which should be stored in a secure environment. These two elements are used for authenticating the client application to the authorization server by the *Http-Basic Authentication Scheme* [19]. If the client application does not support the *Http-Basic Authentication*, only the clientID is sent to the authorization server as a parameter of the request.

2.1.3 Client Profiles

Supplementary to the client types the OAuth 2.0 Framework defines three client profiles which illustrate a specific application scenario.

Web application profile: The *web application profile* is used for a confidential web application which runs on a web server as shown in Figure 2.1. The resource owner interacts with a front end of the web application. The client credentials as well as the access token that are issued to the client are always maintained on a secure environment

on the server. The resource owner is not able to obtain any of this credentials or tokens.

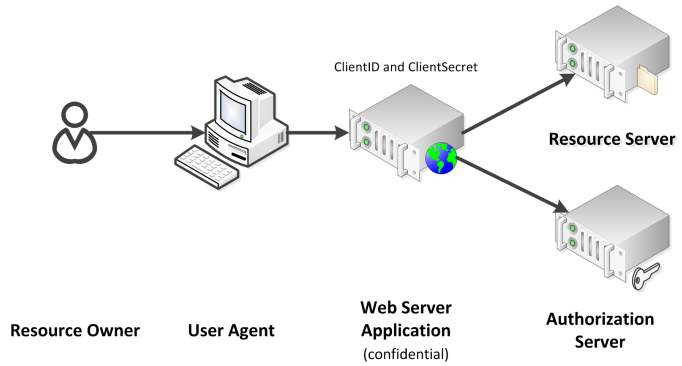

Figure 2.1: OAuth web application profile

User agent based profile: The *user agent based profile* downloads – in contrast to the web application profile – the client code from the web server as shown in Figure 2.2. The code is executed in a user agent, for example, a web browser on a device the user owns. Thus, the client credentials are embedded into the downloaded code, the *clientSecret* is omitted to prevent the access of an arbitrary attacker to the *clientSecret*.

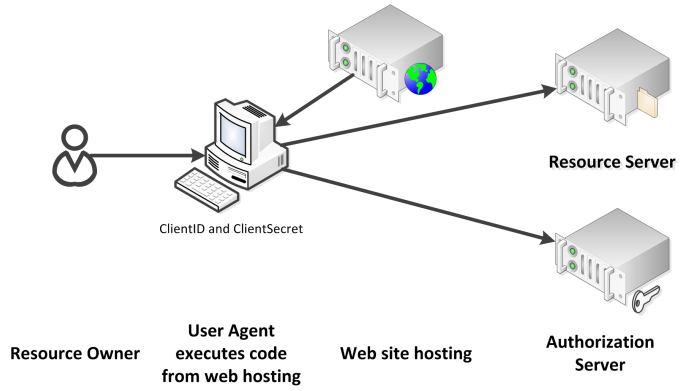

Figure 2.2: OAuth user agent profile

Native application profile: The *native application profile*, shown in Figure 2.3, is used for local installed applications on, for example, a smartphone or a computer. Thus, the code and the client credentials are executed local on the machine, it is assumed that the credentials and the issued tokens can be obtained by a potential attacker.

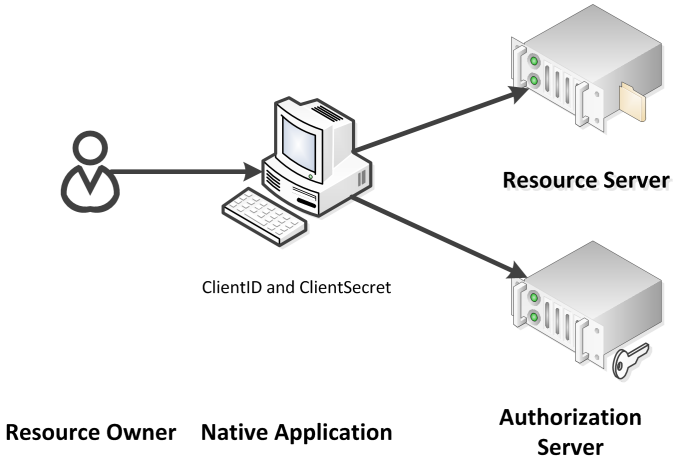

Figure 2.3: OAuth native application profile

2.1.4 Protocol Flow

After introducing the basic terms of OAuth the basic OAuth protocol flow can be discussed. The protocol flow is subdivided into six steps, as shown in Figure 2.4, and works as follows:

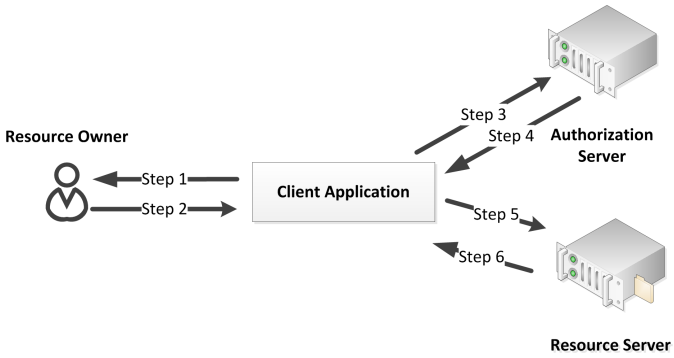

Figure 2.4: OAuth 2.0 protocol flow

Step one: The client sends an authorization request to the resource owner. This can be done directly, as shown in Figure 2.4 or via the authorization server.

Step two: After a successful authorization the client receives an authorization grant. This authorization grant represents the authorization of the resource owner to access the resource owners data. This step varies depending on different authorization grant types. These types are discussed further down in Section

2.1.5.

Step three: The client requests an access token from the authorization server by representing the authorization grant.

Step four: The authorization server validates the client as well as the authorization token, and issues – if the credentials are valid – an access token to the client.

Step five: The client is able to obtain data from the resource server by representing the access token.

Step six: The resource server validates the access token, represented by the client and serves – if the access token is valid – the requested data.

2.1.5 Authorization Grant Types

As mentioned, above at the basic protocol flow, the current version of the OAuth 2.0 framework specification distinguishes four different grant types:

- Authorization code (for web applications)
- Implicit code (for browser applications or mobile applications)
- Resource owner password credentials (for logging in with username and password)
- Client credentials (for application access)

Authorization Code Grant

The *authorization code* grant type is the common process for confidential clients to obtain an access token or a refresh token from the authorization server. An example for such an application is, for instance, a web service.

To receive the authorization code the client must be able to communicate with the resource owners user agent – which is typically a web browser – and receiving requests from the authorization server. Figure 2.5 illustrates the control flow of the authorization code grant. In order to initiate the authorization flow the resource owner's user agent, typically a web browser, is redirected to the authorization server. This is usually done by creating a *login link* which allows the redirection. By clicking the link a prepared request with a *response type*, the *clientID*, the *redirect-uri*, an optional *scope*, and an optional *state* is sent to the authorization server.
The *response type* defines the request type. In case of the authorization code grant the type is *code*.
The optional parameter *scope* is used to restrict the resources that should be accessible on the resource server. The available scope elements are not predefined by the OAuth framework and thereby different on each authorization and resource server implementation.
The optional *state* parameter is a random value, for example, a cryptographic nonce that is not guessable, which binds the request to the client to prevent the authorization

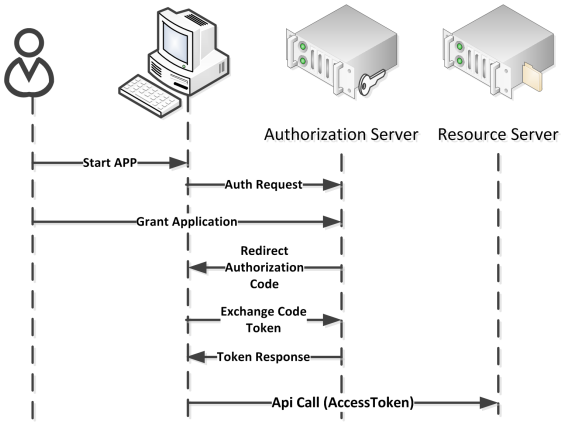

Figure 2.5: OAuth Authorization code grant

flow from cross-site forgeries. The authorization server returns the state parameter in the response that is sent from the authorization server to the client application, that is in further consequence able to verify the response by matching the state parameter.

The authorization server validates the request if the clientID is available in the database and the stated redirect URI is equal to the URI that was entered at the registration step. If the request is valid, the authorization server authenticates the resource owner and displays a consent prompt which contains information about the application, the developer and which user data should be accessed. The resource owner is able to grant or deny the client application the access to the sensitive data. If the user grants the application, the authorization server redirects the user agent back to the client web application with an *authorization code* as parameter. If not, an error message is redirected as parameter to the client web application.

More details about the error message can be found in the OAuth 2.0 Authorization Framework [25] in the Sections *Error Response*.

With the authorization code the client web application is able to request an *access token* by representing the *authorization code* to the authorization server. The request includes the *grant type*, the received *authorization code*, the *redirect-uri*, the *clientID* and the *clientSecret*. The authorization server authenticates the client by the given *clientID* and the *clientSecret*. Supplementary to the verification of *clientID* and *clientSecret* the *authorization code* and the *redirect-uri* are validated to ensure the equality to the issued credentials from the previous step. If the the credentials are valid the authorization server issues an *access token* to the client. Otherwise, an error message is returned. With this *access token* the client is able to access the resource server to request the desired data as described at the basic protocol flow above in Section 2.1.4.

Implicit Grant

Unlike to the *authorisation code grant* the *implicit grant* is designed for browser applications that are implemented in a scripting language such as JavaScript. Since browser applications are downloaded from a web server and executed inside a web browser the confidentiality of the client credentials cannot be ensured. Therefore, the *clientSecret* is not used in the implicit code grant flow. To obtain the *access token* from the authorization server the client sends a similar request as at the *authorization code grant flow*, but with the type *token*. Again the authorization server validates the *clientID* and the *redirect-uri* and displays a consent prompt to the user. If the user grants the browser application the access to the requested private data, the authorization server directly returns an *access token*. No further steps are needed in this case. With a JavaScript application the access token can be obtained from the URI.

Resource Owner Password Credentials Grant

The *resource owner password credentials grant* is designed for legacy applications. These applications requests an *access token* by showing the credentials of the user. Since the application is responsible for collecting the credentials this grant type should be only used by trusted clients. For instance, an arbitrary device operating system. This resource owner password credentials grant should be used only if the authorization code grant or the implicit grant are not realizable. The request type for the password grant is *password*. As parameters for the request the *username* and the *password* are attached to the *clientID* and the *type*. Thus, the secure storage of the client credentials cannot be ensured, the *clientSecret* is omitted.

Credentials Grant

The *credentials grant* is a special grant type and used for the application itself. With this grant type the application is able to update the website url or the application icon. The request contains only the grant type *client_credentials*, the *clientID* and the *clientSecret*. Since the *clientSecret* is attached this request must be only used by confidential clients.

2.1.6 Security Considerations

The OAuth 2.0 framework provide several security mechanisms such as limited access token lifetime or scope limitation to mitigate the impact of possible attacks. Furthermore the *OAuth 2.0 Threat Model and Security Considerations* by Lodderstedt, McGloin, and Hunt [36] provides a comprehensive analysis of the OAuth 2.0 framework and gives guidelines for the implementation of a OAuth provider.

However, several analyses showed that there are security issues.
Slack and Frostig [54] discussed security issues of the implicit grant flow and provided a solution to prevent a possible attack.
Sun and Beznosov [60] analysed the three major identity providers (Facebook, Google and Microsoft) for implementation weaknesses and found that the weaknesses result from identity provider implementations and the simplicity features of the OAuth protocol.

To account these considerations a brief excerpt on the recommended security features is given.

Authorization Code Grant: The first recommendation is to use the authorization code flow for gaining access to the requested resource. The authorization code represents the user authorization that can be used to obtain the access or the refresh token. The code is sent to the specified redirect URI to exchange the token by establishing a secure connection to the authorization server. The *Implicit Grant* is not recommended, due to simple access to the *access token*.

Redirect Uri: The *redirect_uri* parameter ensures that the authorization code is sent to the appropriate endpoint of the application. Therefore, the redirection endpoint has to be a full URI, to prevent the attacker to gain the authorization code. Incomplete *redirect_uris* can be abused by an attacker. Furthermore, the authorization server has to verify the enclosed *redirect_uri* parameter of the authorization step with the redirect endpoint that was entered at the registration.

Transmission Security: The core OAuth 2.0 specification defines that the client as well as the authorization server must guarantee the support of Transport Layer Security (TLS) [10].

State parameter: The *state* parameter helps to prevent *Cross-Site Request Forgeries(CSRF)*. By linking the request to the callback from the authorization server a forgery should be prevented.

2.1.7 Summary

In this section the OAuth 2 web authorization framework was discussed. First, the preliminaries such as roles, the client registration and the different client profiles were covered. Second, the basic protocol flow was introduced. Then, the different grant types where focused on. Especially the recommended authorization code flow. Finally, some

security considerations showed that there are vulnerabilities in connection with the *Implicit Grant*.

2.2 Web Service

A web service [21] is a software system that supports the network communication between machines by means of a machine processable format. Furthermore, it allows the communication with other services by special protocols such as *Simple Object Access Protocol (SOAP)* or *Representational State Transfer (REST)*. These protocols usually use HTTP with an Extensible Markup Language (XML) [44] or a JavaScript Object Notation (JSON) [7] serialization to transmit the data. Popular representatives of such web service providers are, for example, Google [4] and Amazon [5]. Such service providers can be used, for example, to display location based data in a web site by using the Google Maps API [6].

In this section a brief overview of the main representative protocols SOAP and REST is given.

2.2.1 Simple Object Access Protocol

The *Simple Object Access Protocol* [40] is a protocol, defined by the World Wide Web Consortium (W3C) [7], for communication between applications in decentralized environments. The SOAP standard describes how the data is represented in a message by utilizing XML as describing language and basically HTTP as transport protocol. The usage of SOAP in combination of XML and HTTP has advantages and disadvantages compared to former standards such as *Remote Method Invocation (RMI)* and *Common Object Broker Architecture (CORBA)*. A major advantage is that SOAP is platform independent. The platform independence allows the communication of clients and servers of different programming languages, for example, Java clients with C# servers. A further benefit is, that it uses open standards to transport the messages. These are as mentioned the HTTP protocol and furthermore the *Simple Mail Transfer Protocol (SMTP)* and the *Java Message Service (JMS)*. A major benefit of XML are the human readable messages. However, this advantage also has a disadvantage. In contrast to competing former standards the conversation into XML messages and parsing of the XML messages is time-consuming and therefore SOAP is significant slower than RMI and CORBA. To mitigate that problem an additional *Message Transmission Optimization Mechanism (MTOM)* [39] was designed by the W3C to reduce the effort of transmitting binary data by using common techniques such as *BASE64 Encoding* [31].

[4] `https://developers.google.com/`
[5] `http://aws.amazon.com/de/`
[6] `https://developers.google.com/maps/`
[7] `http://www.w3c.org`

Message Structure

The message structure of SOAP is very simple and human readable through using XML. The root element of SOAP message is an *envelope* as illustrated in Figure 2.6. The

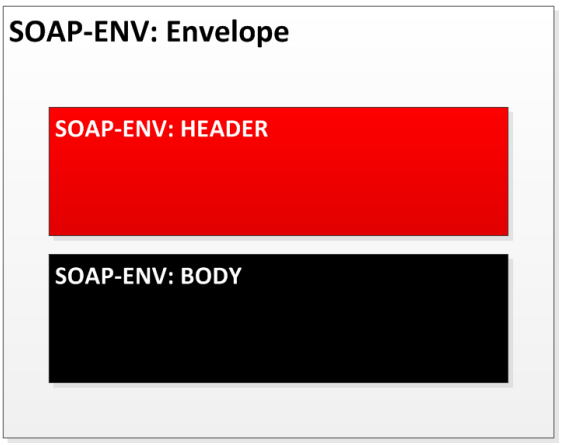

Figure 2.6: SOAP message envelope

envelope contains at least one body element. This body element contains the data that is transmitted to the receiver. Supplementary as needed an optional header is attached. The header contains meta data, routing information, authentication information and data to identify the message.

A more detailed description on SOAP can be found under *Distributed Systems: Concepts and Design* [6, Ch. 9.2.1, p. 387ff].

2.2.2 Representational State Transfer

The *Representational State Transfer (REST)* was specified by Roy Fielding in his PhD Thesis *Architectural styles and the design of network-based software architectures* [16] in 2000. REST is – in contrast to SOAP – not a protocol. It is an architecture style which does not define any protocol syntax or implementations. However, it defines four architectural principles:

Stateless Client Server Communication: This principle defines the stateless communication between client and server. Any request that is sent to the server must include every information that is needed to perform the operation respectively retrieve the data. However, it is not possible to rely on results from previous requests. The advantage of this principle is that the used system resources of the server are still available after an executed request. Any session management is directed to the client.

Resources and Resource Representations: The second principle of REST defines the usage of resources and resource representations. Resources are abstract definitions for documents, images, services or collection of resources. To identify these resources during the communication, every resource entity is linked with a unique resource identifier such as a *Uniform Resource Identifier (URI)* or a *Uniform Resource Locator (URL)*. The unique identifier enables the identification of each resource at any time. Furthermore, the principle defines resource representations. Resources are often required in various representations to be processed. Therefore, this principle defines a separation of resource and resource representation. This provides the ability to retrieve the same resource in various representations. Representation formats are, for example, XML, JSON, HTML, PDF or image formats. The separation between the representation is realized by content negotiation. A requesting client defines the representation type of the resource in the *content-type* of the request and the server retrieves the appropriate representation of the resource. Thereby, the provided resources can be consumed by various clients and not only by one specific application. Moreover, additional representations can be simply added afterwards.

Uniform Interfaces: A further principle of the REST architecture is the uniform interface. The idea behind is to generalize the component interface to simplify the whole architecture by defining a set of operations that are supported by the underlying protocol. For instance, a web service that uses the HTTP protocol supports the methods that are defined in the HTTP standard [17]. The HTTP standard defines 8 methods whereof usually only the main five are used. These operations are:

- **GET**
  The *get* method is used to retrieve a resource from the server. The method is *idempotent* and *safe*. Idempotent ensures that no side effects occur regardless how often the method is called. Safe means that the resource on the server is not modified.
- **POST**
  The *post* method is used to process an arbitrary operation at the server. Thereby, this operation is either idempotent or safe. This method is often used to create resources on the server.
- **PUT**
  The *put* method is used to insert or update a resource on the server. The put method is idempotent because a further request does not have any effect on the server.
- **DELETE**
  The *delete* method deletes a resource on the server. This method is idempotent. However, a further delete request deletes the object again, however without any result.
- **PATCH**
  The *patch* method [11] is used to update an existing resource on the server. In

case of an unknown resource on the server the method creates a new resource. Thereby, the method is neither idempotent nor safe.

Link to Resources: The final principle is the idea of links that are used to navigate between states. These states can be, for example, an additional information that can be retrieved by following a link. The advantage of a link is that any kind of resource can be linked. For instance, a list of links with associated URIs.

Security Considerations

The REST architecture itself defines no security standard. To provide a secure connection to a REST web service HTTPS is used. Due to the fact that REST uses a uniform interface HTTP can be simply replaced by HTTPS without any changes to the interface.

2.2.3 Summary

In this section web services protocols such as SOAP and REST were discussed. Both systems are widely spread and used. SOAP is a protocol in contrast to REST, which only defines architecture principles. In this master's thesis we decided to use REST. The main reason for REST was that the deployment is very simple as well as the client can be a web page which is executed in a web browser.

2.3 HTML 5 Web Messaging

In 2010 the W3C has introduced a new HTML5 web messaging standard [26]. This standard defines mechanisms to communicate between web applications from different origins in a browser context. Furthermore, the HTML5 web messaging standard is designed to prevent possible Cross-Site Scripting (XSS) attacks. The following section discusses the workflow, how the communication between two windows respectively frames can be established and how the origin of the message is validated. Finally some security considerations are drawn.

2.3.1 Workflow

In order to communicate between two windows respectively frames in a web browser environment two types of frames or windows have to be distinguished. These two different windows or frames are the *sender* and the *receiver*. Usually the sender embeds the receiver by using an *iframe*. In order to transmit a message from the *sender* to the *receiver* the *postMessage* command, as shown in Figure 2.7, is used. The *message*

parameter represents the message and the *target-origin* parameter defines the origin to which the message is sent. The message parameter is not limited to strings. Further data objects are, for example, *Files* and *ArrayBuffers*. The specified *target-origin*, for example, `http://www.test.com` has to be equivalent to the origin of the *receiver*, otherwise the message will not be dispatched to the receiver. The postMessage command supports an *all permitted (*)* symbol for the target-origin. In this case the message is dispatched to each origin. However, this can be used by an attacker to spy the messages. Hence, it is recommended to set a specific target origin.

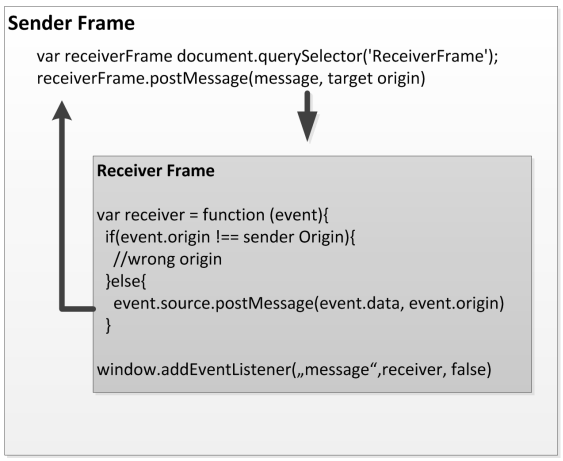

Figure 2.7: HTML 5 postMessage workflow

In order to receive an event at the receiver the window or frame has to register an event listener as shown in Figure 2.7. The first parameter defines the type of the event listener which is in this case a *message* listener. The second parameter determines which function is called if a message occurs. The third parameter is used for a special case which is not further discussed, and therefore set to false.

If a postMessage call is sent to the specific origin of the receiver window or frame the event listener is invoked and the defined function is called with an event message. The transmitted event message contains three properties. The first property is the *message* itself. The second property is the *origin* from which the message was sent from. The origin is generated by the browser and contains the *scheme*, the *port* and the *host* of the sender. The last property is the *source* which defines a reference to the source window or frame of the message. To verify the origin of the sender the receiver validates the origin as shown in Figure 2.7, by comparing for equality. It is recommended to perform this validation to prevent the malicious usage of the postMessage channel. The source property is used to sent messages back to the window or frame that is defined by the given source as shown in Figure 2.7.

2.3.2 Security Considerations

The HTML5 web communication concept provides *authenticity* and *confidentiality* for the client side communication in a web browser. The *authenticity* of the message can be verified by the origin parameter of the event message. Thereby, that the real origin of the sender is attached by the web browser the sender can be authenticated. *Confidentiality* is reached by the browser implementation of the *postMessage* command. The browser ensures that the message is exclusively dispatched to the specified target-origin.

However, Barth, Jackson, and Mitchell [4] showed a vulnerability in the confidentiality of the HTML5 web message channel by simulating a certain navigation scenario. The presented modification to ensure confidentiality was adopted by the HTML5 working group [8].
Furthermore, Hanna et al. [24] analysed the usage of the HTML5 web message channel in Facebook and Google applications and discovered that the provided origin checks are incomplete.
Son and Shmatikov [57] analysed the Alexa top 10,000 websites and discovered origin vulnerabilities which can be used for exploits.

2.3.3 Summary

In this section we discussed the HTML5 web messaging standard. We described how the HTML5 web messaging standard can be utilized to communicate between cross origin windows on client side. Furthermore, we covered security issues which result from incorrect or missing origin checks at the receiver window.

2.4 Same-Origin Policy

The *Same Origin Policy (SOP)* [3, 49] is a security concept for user agents and web related technologies. The policy limits the access of websites to that origin on which the web page is deployed. For instance, a web page which is running on `https://test.com/index.html` is allowed to call the url `https://test.com/test.html`. If the web page try to load the url `http://test.com/test.html`, the user agent bans the request due to the origin of the first page (https, test.com, 443) differs from the requested page (http, test.com, 80).

The origin itself is a combination of three URI elements. The first element is the *scheme*, which can be, for example, *http* or, *https*. The second element is the *host*, which can be, for example, *google.com*. And the last element is the *port*, which can be, for example, *80*.

[8] `http://www.w3.org/html/wg/`

In case of a missing port the port element is derived from the scheme. For instance, the *http* scheme uses the port 80.

Nevertheless, there is a necessity for loading data from cross origins such as special libraries from third party providers that are needed to operate the web service. Therefore, SOP permits several HTML tags, as illustrated in Table 2.1, to load data from different origins. However, there are further solutions to circumvent the Same Origin Policy such as *Cross Origin Resource Sharing*, discussed further down in Section 2.5, Ajax proxies and JSONP.

HTML Tag	**description**
`<script>... </script>`	script files
`<link rel="stylesheet", href="..."`	Cascading Style Sheets (CSS)
`<img>...</img>`	images
`<video>` and `<audio>`	media files
`<object><embed>` and `<applet`	plugins
`@font-face`	fonts
`<frame>, <iframe>`	frames

Table 2.1: Permitted same-origin HTML tags

Ajax Proxy: In order to circumvent the SOP restriction of the XMLHttpRequest [58] an application proxy can be used, as shown in Figure 2.8. The application proxy fetches the data from a remote server by implementing an HTTP client and provides the data for the current web page. The web page is able to load the data by using the XMLHttpRequest from the own web server. Thereby the SOP is not violated.

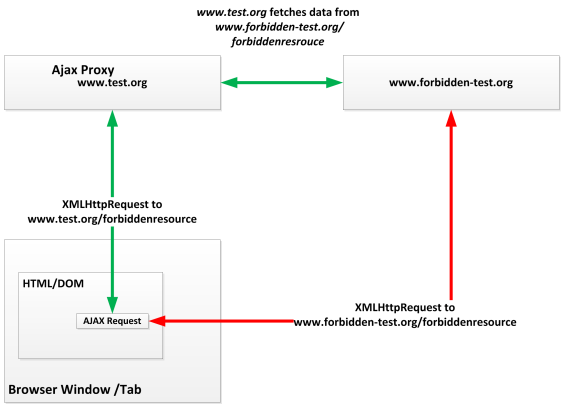

Figure 2.8: Ajax proxy

JSON with Padding: Another approach to circumvent the SOP is the usage of JavaScript Object Notation with Padding, as shown in Figure 2.9. At first, the developer of the web page defines a function that displays the cross origin data, for example, *includeForbidden* when it is fetched. Then, a request to fetch the data is inserted into the page by using the `<script>`, as shown in Figure 2.9. The response of the remote server is constructed in that way that the previous defined function to display the data is called. Thereby, the SOP can be circumvented and data from a cross origin remote server is fetched when the web page is loaded.

```
function includeForbidden(data){
  //show data
}

<script type="text/javascript" src="http://www.forbidden-test.org/
forbittenresource.php?data=all&format=json&callback=includeForbidden">

</script>
```

Figure 2.9: JSONP

2.4.1 Security Considerations

Although the *Same Origin Policy* enhances the security of user agents by restricting the origin of loadable data, the above mentioned possibilities to circumvent the SOP are abused by attackers. The most common attacks are Cross Site Request Forgery (CSRF) and Cross-Site Scripting (XSS) attacks. CSRF attacks are used to exploit the trust of a web site. For instance, a user is logged into a banking website and has an active session. An attacker sends a manipulated URL to the user. If the user clicks on the manipulated URL a manipulated transaction with the active session of the user is performed. However, there are possibilities to prevent CSRF attacks. By including secrets into requests, for example, hidden forms, using limited session lifetime or checking the referer header, CSRF attacks can be prevented.

In contrast to CSRF, XSS attacks injects code into websites in order to gain data, for example, read session information. XSS attacks can be mitigated by using frameworks for escaping and input checks.

An comprehensive analysis of SOP is presented by Saiedian and Broyles [51]. Saiedian and Broyles also discussed a protection mechanism to prevent attacks.

2.4.2 Summary

The Same Origin Policy is designed to restrict the origin of the loadable data to the origin on which a web page is deployed. However, multiple possibilities such as Ajax

Proxies, JSONP or CORS to circumvent the SOP result in vulnerabilities as discussed above. Therefore, a more promising approach such as the *Content Security Policy* [59] will be necessary. This approach supports the definition of policies for the web application. However, this technology was not scope of this work.

2.5 Cross-Origin Resource Sharing

Cross Origin Resource Sharing (CORS) [34, 27] defines a mechanism for user agents to retrieve data from origins which are different to the origin of the requesting web page. The following section discusses the concept of CORS as well as some security considerations.

2.5.1 Concept

Same Origin Policy (SOP) denies requests from a website located, for example, on `http://test.com` to another website located on `http://website.com`. However, by using the CORS concept such requests can be allowed.

The main concept behind CORS is to use additional custom HTTP headers and additional hidden requests to determine if a request is allowed or not. The supplementary requests are named *preflight requests* and are sent before the actual request is sent. Figure 2.10 illustrates the request process with the supplementary preflight requests. In case of a request to the server, the user agent respectively the browser applies a preflight request. This request is some kind of asking for permission for the actual request. If the request is permitted, the actual request is dispatched to the server. The preflight response is cached for further requests to reduce the traffic between the browser and the server.

Preflight Request

By sending a preflight request to the server the browser determines whether the actual request is permitted or not. To determine the access, however, some supplementary information on the server is required. The required information is added by the browser by adding the supplementary headers. The supplementary headers are added by default and cannot be influenced by the user. The preflight request contains the following headers:

- *Origin*
- *Access-Control-Request-Method*
- *Access-Control-Request-Headers*

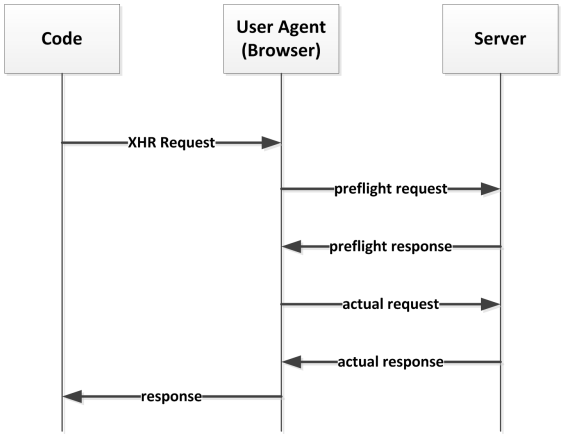

Figure 2.10: CORS preflight requests

The *Origin* header is always added by the browser and contains a triplet composed of *scheme*, *host* and *port*. The *Access-Control-Request-Method* defines the method of the actual request, which should be sent afterwards. The *Access-Control-Request-Headers* defines additional custom headers that are sent in the actual request.

Preflight Response

As the preflight request applies for permission, the server has to verify the parameters of the request, to determine whether the actual request is permitted or not. In case of a successful verification of the request headers the server issues a preflight response that contains several headers. These headers are:

- *Access-Control-Allow-Origin*
- *Access-Control-Allow-Credentials*
- *Access-Control-Expose-Headers*
- *Access-Control-Max-Age*
- *Access-Control-Allow-Methods*

The *Access-Control-Allow-Origin* header is a mandatory header when a valid CORS response is returned from the server. The given origin defines the origin from which subsequent requests are allowed. For instance,
`Access-Control-Allow-Origin: http://test.com` permits subsequent requests from *http://test.com*. Moreover, it is allowed to set an *all permitted (*)* symbol to permit every origin. However, the usage of this literal is not recommended to prohibit a malicious usage of the server.
The *Access-Control-Allow-Credentials* header is, in contrast to the *Access-Control-Allow-Origin* header, optional. The header defines if the actual request contains any HTTP

cookies or HTTP authentication information. Together with this header the *withCredentials* property of the XMLHttpRequest is used. Both parameters must be true to retrieve a valid response from the server.
The *Access-Control-Expose-Headers* is an optional header. In combination with the XMLHttpRequest2 [58] method the client is able to access simple response headers during the preflight requests. These simple response headers are *Cache-Control*, *Content-Language*, *Content-Type*, *Expires*, *Last-Modified* and *Pragma*. To access further headers the header has to be defined at the *Access-Control-Expose-Headers* header. Due to known bugs [9], it is not ensured that common browser implementations support the *Access-Control-Expose-Headers* in the same way.
The *Access-Control-Max-Age* header is also optional and defines how long a preflight response should be cached. As mentioned above the preflight is cached on client side. If the *Access-Control-Max-Age* time is expired a preflight request is sent to the server before the actual request is sent.
The *Access-Control-Allow-Methods* header is mandatory and defines which HTTP methods are allowed for further requests. It is possible to permit a comma-separated list which includes more than one method. This can be, for example `Allow-Control-Allow-Methods: POST, DELETE`.

Preflight Error

In case of a non valid origin the server returns an error message that is displayed in the browser log and looks like, for example, `XMLHttpRequest cannot load http://test.com. Origin http://test.com is not allowed by Access-Control-Allow-Origin.`

2.5.2 Security Considerations

The CORS concept is a tool to protect an *Application Programming Interface (API)* against attacks from cross origin. However, several implementation faults of CORS lead in vulnerabilities. Especially the *Access-Control-Allow-Origin* header is a central point of attacks by setting an *all permitted (**)* symbol. It is recommended to use white listing approach to restrict trusted domains. A further attack point is the *Access-Control-Max-Age* header. Due to performance reasons the preflight request are cached on client side as long as defined in the *Access-Control-Max-Age* header. However, the CORS concept is not able to provide a protection against Cross-Site Scripting (XSS) attacks.

Schmidt [52] provided a comprehensive analysis on CORS and discussed the vulnerabilities of the *Access-Control* mechanism.
Shah [53] discussed a CSRF attack on CORS which even enables the file upload.
Ryck et al. [50] presented a comprehensive analysis of web standards as well as the

[9] `http://www.html5rocks.com/en/tutorials/cors/`

CORS technology and showed the vulnerability of the *Access-Control-Allow-Origin* header.

2.5.3 Summary

The CORS concept was designed to protect APIs against misuse from malicious cross origin pages. However, the above discussed security considerations has shown that the CORS concept has several vulnerabilities and an additional protection is necessary to protect an API.

2.6 Summary

In this chapter the basic technologies which are in context with web services were discussed. At first, we covered the OAuth authentication scheme which is used to delegate the authentication to a trusted third party identity provider. Further, an overview on the web service interaction protocols SOAP and REST was given. Finally, the Same Origin Policy (SOP) and related technologies such as Cross Origin Resource Sharing (CORS) or the HTML5 web communication mechanism were covered.

Chapter 3

Related Cryptographic Hardware and Web-based Solutions

In this chapter, a broad overview on cryptographic hardware respective web-based solutions that are able to store cryptographic keys and perform cryptographic operations is given. Various cryptographic hardware solutions provide the ability to handle and store cryptographic keys as well as the ability to perform cryptographic operations. However, these hardware solutions have advantages in terms of handling but also disadvantages in terms of key distribution and compatibility. Web-based solution provide, in contrast to hardware solutions, flexible key distribution.
The presented overview of benefits and limitations of the existing web-based solutions establishes a basis for this master's thesis, where we introduce a solution that combines different hardware key storage solutions and provides them for local as well as for cloud-based applications.

This chapter is divided into three main sections. The first section discusses standard cryptographic key storage solutions such as smart cards, security tokens and Hardware Security Modules (HSMs). The second section focuses on the existing web-based solutions for cryptographic services. Finally, a comparison of the different solutions is conducted.

3.1 Standard Cryptographic Key Storage Solutions

Standard cryptographic hardware key storage solutions are well established and mainly used for authentication purposes, for example, bank cards. The application area for such key storage solutions is not limited to authentication purposes, however, the can be used for signature creation as well as to perform basic cipher operations such as encryption and decryption. Although hardware key storage solutions have advantages in terms of usability, they even have disadvantages in terms of key distribution and compatibility. In this section we discuss the the most common hardware key storage solutions such as smart cards, security tokens and Hardware Security Modules (HSMs).

In order to provide an overview we focus on the strengths and weaknesses of each solution.

3.1.1 Smart Card

Smart cards are a well known and wide spread technology which is used to provide authentication, identification, key storage and cryptographic operations. In 1950 the first type of smart card was introduced in the United States by Diners Club. Since the introduction of the first smart card the design has changed from a magnetic strip smart card to a smart card which contains a microprocessor. Through the development of the microprocessor smart card the application range was not limited to the tamper storage of data any more. The microprocessor enables the execution of cryptographic operations with the secret key that is stored on the smart card. The usage of the sensitive data is restricted to a secret (PIN) that has to be presented by the user of the card before any operation can be executed. This security mechanism established as standard for further technologies.

The application area of smart cards is widely spread. The range starts from banking cards over payment systems to health insurance cards. However, the smart card technology is not limited to authentication scenarios. With an appropriate smart card reader the owner of the smart card is able to perform cipher operations such as encryption and decryption.

The latest development in the field of smart cards are contactless smart cards that communicate with the reader via the *Near Field Communication (NFC)* technology. These smart cards can be utilized easily by moving it over a specific reader.

Strengths:

- **Usability:** A major strength of smart cards is they are widespread. Thereby, that smart cards are used for bank transactions they are utilized by everyone. Moreover, the new NFC communication standard simplifies the usage of the smart cards and enhances the usability.
- **Flexibility:** A further strength of smart cards is that they are flexible. Users are able to take along their smart cards and use them on different devices.
- **Use cases:** Although smart cards are mainly used for authentication scenarios the application area is not limited only to these scenarios. Smart cards offer the possibility to create signatures as well as the execution of cipher operations. Thereby, smart cards can be used in a broad range of applications.

Weaknesses:

- **Connectivity:** The major weakness of smart cards is, that an appropriate reader is needed to use the functionality of the smart card. Without this reader the smart card cannot be used.
- **Incompatibilities:** A further weakness is the restricted support of the smart card readers on different platforms. Driver and hardware incompatibilities limits the usage of smart cards.
- **Limited Scope:** Although smart cards are very popular for authentication purposes, for example, bank cards, they are occasionally used for signature creation or cipher operations. This results from the low dissemination of card readers.
- **Key Distribution:** A further weakness of smart cards is the key distribution. In case of an update the smart card has to be replaced.

In summary, the smart card is used for various applications. It provides a secure key storage and the microprocessor enables the execution of cryptographic operations. The strengths are the usability and the flexibility that enables the usage of smart cards everywhere. The weaknesses of smart cards are availability of appropriate reader and the platform incompatibilities. So the usage of smart cards for signature creation or basic cryptographic functions is limited.

More details about smart cards can be found in the *Smart Card Handbook* [47]. Furthermore, Rizvi, Rizvi, and Al-Baghdadi [48] provided a overview on the smart card technology and discussed typical application scenarios.

3.1.2 Security Tokens

Security tokens are similar to smart cards, however with the difference that security tokens utilize common connection technologies such as USB slots, SD card slots or SIM card slots. Thereby, security tokens can be easily connected to various types of machines. The following types are most used:

- SIM cards in smartphones
- Secure SD cards
- Special Tokens USB slot

The major field of application for security tokens are authentication scheme such as multi factor authentication with OTP.

A new development in this field are contactless security tokens, for example, *Yubikeys* [1] which utilize the USB slot technology as well as the Near Field Communication (NFC) standard. Thus many new application scenarios arise.

[1] `http://www.yubico.com/products/yubikey-hardware/yubikey/`

Strengths

- **Usability:** The strengths of security tokens are their handy size and the support of common connection technologies. This enables the easy usage of security tokens which is a major benefit, in contrast to common smart cards. Furthermore, the support of contactless communications standards such as NFC enhances the usability and handiness of security tokens.
- **Flexibility:** As well as smart cards security tokens are very flexible. They can be also used on several devices.
- **Connectivity:** Due to the support of common connection technologies such as USB, the weakness of smart cards is removed. Thereby, security tokens are not limited in their use cases.
- **Use cases:** The application area of security tokens is broad. Currently they are mainly used for authentication schemes. However, they can be even used for signature creation and basic cryptographic functions.

Weaknesses:

- **Key distribution:** The major weakness of security tokens is the key distribution. In order to update the key on the security token a key distribution center and a secure connection between the security token and the key distribution center is required. The key management leads to a high burden for the security token issuer.
- **Incompatibilities:** Although security tokens support common connection technologies platform incompatibilities can occur due to lack of drivers.

In summary, security tokens are a flexible solution for local cryptographic key storage. The major advantage – in contrast to smart cards – is the connectivity by utilizing common connection technologies. The only weakness of security tokens is that they have some operating system incompatibilities.

3.1.3 Hardware Security Module (HSM)

A Hardware Security Module (HSM) is tamper-resistant cryptographic device that is able to perform cryptographic operations and maintain cryptographic keys. The HSM provides these functions through an Application Programming Interface (API). The design of the HSM varies depending on the type from a plug-in PCI card to an external device. However, the main purpose of the HSM is to keep the sensitive key material secret whatever an attack is enforced on the device such as discussed by Anderson et al. [1]. The FIPS 140-2 standard [41] provides a validation for HSMs. This validation ensures that a FIPS proofed HSM guarantee a certain level of protection. The use cases for HSMs are key generators and key storages for Certificate Authoritys (CAs), secure

random generators for smart cards and functions wherever an accelerated cryptographic operation is needed.

Strengths:

- **Highest Security:** The major strength of HSMs is the provided security level. The tamper resistance ensure that an attack is not able to obtain any sensitive key material.

Weaknesses:

- **Restricted Scope:** The only weakness of HSMs is that they are mainly used for server applications. Thereby, they are not available for users to store keys and perform cryptographic operations.

To sum up, the HSM is the best solution to protect keys from tamper attacks. Nevertheless, the main use case scenarios for HSMs are server applications and thereby private users are not able to utilize them.

3.1.4 Summary

Smart cards, security tokens and Hardware Security Modules are the common solutions to store keys in a tamper resistant environment. Especially smart cards and security tokens have their strengths in usability and flexibility. However, the main limitations of these solutions are platform incompatibilities and key distribution. Therefore, web-based solutions were developed to mitigate these limitations. Most of the current solutions for web-based key storage and cryptographic operations rely on HSMs. An overview about the current solutions for web-based key storage is given in these following section.

3.2 Web-based Solutions

Due to the limitations of cryptographic hardware solutions in terms of key distribution and platform incompatibilities the leading industry has recognized that web-based solutions for cryptographic key storage are necessary. Therefore, several companies developed web-based cryptographic key storage solutions that mitigate the limitations and offer authentication and signature creation. In this section we introduce the existing solutions.

3.2.1 Amazon Cloud HSM

The *Amazon AWS CloudHSM service* [2] was introduced by Amazon in 2013. Amazon was the first provider that enables a user to use a Hardware Security Module (HSM) appliance that is not installed locally.

By utilizing the *Amazon Virtual Private Cloud (Amazon VPC)* the full access to the HSM can be enabled. In order to use the Amazon CloudHSM the user is responsible to set up the HSM appliance. Thereby, that the user sets up the appliance Amazon is not able to access or to obtain the cryptographic keys which are stored in the HSM.

In order to access the HSM appliance inside the Amazon VPC a mutual authenticated Secure Socket Layer (SSL) connection is used.

The Amazon CloudHSM supports the common API standards such as *PKCS11* [45], *Microsoft CAPI* [9] and *Java JCA/JCE* [29].

3.2.2 Austrian Mobile Phone Signature

The *Austrian Citizen Card* [35] defines a signature concept, that combines the ability to create qualified electronic signatures and to determine a citizen's identity. As the citizen card is technology neutral, various appearances are possible. One of these appearances is the well known *health insurance card* which was introduced in 2004. Apart from the citizen card concept a server-based solution for the signature creation and the citizens identification exists since 2009. The *Austrian Mobile Phone Signature* [3] provides an XML based interface for applications to access an HSM that stores all sensitive keys of the Austrian citizens. The access to the appropriate key is protected by a two factor authentication mechanism including a password as well as a One Time Password (OTP) that is sent to the citizens mobile phone.

The *Austrian Mobile Phone Signature* offers the functionality for identification and authentication and the ability to create qualified electronic signatures according to the EU Signature Directive [15]. This signature is, as defined in the EU Signature Directive, equivalent to the handwritten signature. Thereby, various government authorities as well as banks accept authentication and signature creation with the Austrian Mobile Phone Signature. The wide acceptance of the Austrian Mobile Phone Signature leads to a user friendly solution that ensures the equivalent signature quality without any limitations of card readers, platform incompatibilities and installation issues.

The signature creation of the Austrian Mobile Phone Signature technology is restricted to the *XMLDSig* [12] scheme. Any creation of raw signatures as well as the ability to perform basic cipher operations such as encryption and decryption is not permitted.

[2] `http://aws.amazon.com/de/cloudhsm/`
[3] `https://www.handy-signatur.at/`

More details about the Austrian Mobile Phone Signature are discussed by Orthacker, Centner, and Kittl [43].

There are similar solutions to the *Austrian Mobile Phone Signature* available in Europe such as the *Norwegian BankID* [4] and *Italian server-based qualified signature* [5] that also provide the signature creation for their citizens.

3.2.3 SigningHub

SigningHub [6] enables the user to create advanced electronic signatures under sole control of the signer. To access the cloud-based service SigningHub offers a REST-API that allows an easy integration into existing solutions. The authentication is ensured by supporting multi factor authentications solutions with OTPs, smart cards, tokens and username/password. The used key material is maintained by SigningHub and stored on smart cards or tokens. Furthermore, SigningHub proposes to provide a Secure Signature Creation Device (SSCD) to produce qualified signatures.

3.2.4 Cryptomathic

Cryptomathic [7] offers authentication as well as the creation of digital signatures. The authentication is guaranteed by supporting various authentication schemes such as OTP solutions and multi factor authentication schemes. The sensitive keys are centrally stored in an HSM. As well as SigningHub, Cryptomathic proposes to reach the SSCD level to create qualified electronic signatures. Moreover, Cryptomathic allows the integration into existing applications by an appropriate client that supports standard signature formats such as *PKCS# 1* [30], *PKCS# 7* [32], *XAdES* [8], *PAdES* [14], *CAdES* [13]. Furthermore, the integration in to web application is supported by a provided applet.

3.2.5 Dictao

Dictao [8] provides – in contrast to SigningHub and Cryptomatic – only a solution for authenticating and digital signature creation that have be integrated into an existing enterprise service. Dictao supports multi factor authentication schemes as well as OTPs. Furthermore, common digital signature formats such as *XMLDSig*, *XAdES*, *CMS/PKCS# 7*, *S/MIME* [46], *PDF* are supported.

[4] https://www.bankid.no/Dette-er-BankID/BankID-in-English/This-is-how-BankID-works/
[5] http://www.digitpa.gov.it/
[6] http://www.signinghub.com/
[7] http://www.cryptomathic.com/
[8] http://www.dictao.com/

3.3 Analysis

The above presented web-based key storage solutions provide an overview of the available solutions. Although the discussed solutions provide a similar functionality they have differences in their implementation. For instance, the Amazon CloudHSM provides full access to a HSM which is not provided by the other solutions. In order to get an overview of these differences and the characteristics of each solution we analyse them by defining several criteria.

In the remainder of this section, at first we introduce the criteria and discuss their importance. Afterwards, we classify each solution in considering of the criteria and present a comparison.

3.3.1 Criteria

Although there are several solutions of web-based key storage solution with different functionalities available they differ only in a few aspects. The following criteria show the differences of the solutions.

Key Storage Solution: The *key storage* property considers the central storage of the cryptographic key material. This criterion is essential, since the utilized key storage solution effects the storage as well as the execution of the cryptographic in a crucial way.
Possible key storage solutions are smart cards, security tokens or HSMs. Either of these solutions is able to provide a high level of security. However, for web-based solutions mostly HSMs are used.

Access Protection: In order to use a key storage solution such as a smart card or a security token the user usually has to authenticate by presenting a secret, for example, a PIN. The *access protection* criterion evaluates the sole control of the user.

Provided Functionality: Cryptographic key storage solutions offers the ability to store cryptographic keys as well as the execution of cryptographic operations. The supported cryptographic operations differ depending on the application area of the solution. Mainly they offer XML based signature formats such as XMLDSig or XAdES. Furthermore, some solutions support the raw signature creation or cipher operations such as encryption and decryption.
However, the provided functionality respectively supported operation influence the applicability of these solutions and is therefore an important criterion.

Provided Interface: Basic prerequisites for cryptographic key storage solutions are usability and flexibility to embed the solution into an existing application. However, there exist several approaches to use respectively embed a web-based key storage solution. A common way to utilize a web-based key storage solution is the usage of an appropriate client. This client is responsible for the secure connection to the web-based storage. Since a secure connection is established the client manages the data transfer. However, such clients are mostly designed for specific platforms and therefore the platform independence is not provided.
Another approach is the usage of an interface, that is accessible for web applications or even local applications.

The *provided interface* criterion is essential to compare web-based key storage solutions, however, a flexible interface is crucial for a broad application area.

3.3.2 Comparison

The above presented criteria can be used to compare the, in Section 3.2 introduced, web-based key storage solutions. The comparison is shown in Table 3.1. Due to the fact that almost every solution propose to use a central HSM to store the cryptographic key material we omit the *key storage solution* criterion in the comparison. *SigningHub* does not define the key storage solution, however, the usage of a Secure Signature Creation Device (SSCD) implies the utilization of at least a smart card. Due to the low relevance of these comparison we decided to omit this criterion in the overview to enhance clarity.

The comparison of the introduced web-based key storage solutions shows that almost every solution provide several authentication schemes from from *username/password* over different kinds of *OTPs* to *multi factor authentication* schemes. Only the *Amazon CloudHSM* needs a *Virtual Private Cloud* connection that is established by a provided client.

The comparison of the functionality indicates that the most solutions offers authentication and signature creation by supporting digital signature schemes. However, *SigningHub* offers only the ability to create digital signatures. In contrast to these solutions the *Amazon CloudHSM* offers full access to an HSM. Thus, the range of cryptographic operations is not restricted.

The analysis of the interface property shows that two solutions require a certain client to utilize the system. The *Amazon CloudHSM* is limited to the predefined *Virtual Private Network (VPN)* connection between client and CloudHSM that is maintained in the Amazon datacenter. *Cryptomathic* can be only accessed over an appropriate client or by a provided applet. *Dictao* is limited to the environment on which it is deployed and therefore cannot guarantee a flexible interface. In contrast to these solutions the

Solution	Access protection	Provided Functionality	Interface
Amazon CloudHSM	VPC over client	full HSM appliance **not restricted**	**client**
Austrian Mobile Phone Signature	MFA with OTP	**authentication**/ **signing** (XMLDSig)	XML interface
SigningHUB	MFA, OTP, SC, UP, AT	**signing** (PAdES)	REST API interface
Cryptomatic	MFA, OTP	**authentication** / **signing** (XAdES,PAdES, CAdES)	**client** / applet
Dictao	MFA, OTP	**authentication** / **signing** (XMLDSig, XAdES, CMS/PKCS#7, S/MIME)	**restricted to environment**

MFA – **M**ulti **f**actor **A**uthentication

OTP – **O**ne **T**ime **P**assword

VPC – **V**irtual **P**rivate **C**loud

SC – **S**mart **C**ard

UP – **U**sername - **P**assword

AT – **A**uthentication **T**oken

Table 3.1: Comparison web-based key storage solutions

Austrian Mobile Phone Signature and *SigningHub* provide an interface. These solutions are not limited to any certain client.

3.3.3 Summary

The current situation shows that several web-based key storage solutions, that allow authentication and signature creation, are available on the market. Especially the functionality for signature creation is offered by almost all solutions such as the Austrian Mobile Phone Signature, SigningHub, Cryptomathic and Dictao. In contrast to these solutions the *Amazon CloudHSM* provides full access to an HSM. However, almost every solution requires a client to utilize the provided functionality. Thereby the possible use cases are limited. Only the *Austrian Mobile Phone Signature* and SigningHub can be utilized without a client. A key fact of all of these solution is the availability of a network connection. Without a network connection these solution cannot be used. However, our solution, which is presented in the next section, remedy the limitations of the above discussed solutions and provides a flexible, extensible and platform independent server-based key storage.

Chapter 4

Skytrust System Design

The analysis of the existing web-based solutions for cryptographic key storage solutions has shown that almost all of them offer authentication and signature creation. Only the *Amazon CloudHSM* offer the full access to an HSM. Thereby, the range of cryptographic operations is not limited and even cipher operations can be supported. Nevertheless, the *Amazon CloudHSM* has disadvantages due to the limitation to a certain client. Without this client the functionality of the HSM cannot be utilized.
To overcome the necessity of a client and the restriction to certain operations the *Skytrust* project was started. The *Skytrust* project has the aim to provide a flexible, extensible and platform independent key storage solution. In contrast to the existing solutions the Skytrust system offers the full functionality of the cryptographic key storage solution over a flexible interface. This ensures that the Skytrust system can be utilized by arbitrary devices and platforms.

In this chapter at first, the concept behind the *Skytrust* system is discussed, by giving an overview how the limitations can be mitigated and evaded. Second, the core element of the *Skytrust* system is focused on. Third, the underlying transport protocol is covered. Fourth, possible environments are introduced which are used in the basic prototype that is presented in Section 4.5. Finally, a permission system is proposed that can be utilized to restrict the access to the *Skytrust* system.

4.1 Basic Concept

The goal of the *Skytrust* system is to provide a flexible, extensible and platform independent solution for cloud-based cryptographic services. In order to accomplish these aims, the basic concept behind the *Skytrust* system is to break up the usual process of the access to a cryptographic token or a service into a receiving component and an acting component, as illustrated in Figure 4.1

The receiving component, called *receiver*, is the access point to the Skytrust system. The receiver enables an arbitrary application to use cryptographic operations without performing any of these operations. By providing an interface, any application is able to use the Skytrust system such as a local cryptographic service. The only difference

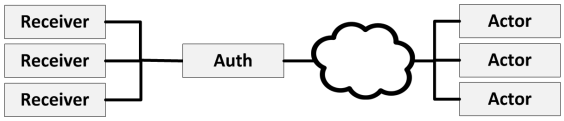

Figure 4.1: Skytrust concept

between the local and the cloud-based solution is the place where the cryptographic operation is executed or the cryptographic key material is stored. The application does not recognize, if the cryptographic operation is executed on a local connected device, for example, a smart card with smart card reader or on a cloud-based key storage solution.

The real cryptographic operation is carried out on the acting element which is called *actor*. The actor connects arbitrary key storage solution providers to the Skytrust system. The principle task of the actor is to perform the cryptographic operation by utilizing the functionality of the key storage solution. The key storage solution itself performs the cryptographic operation and the maintains the cryptographic key material such as a smart card, a Java Cryptography Extension (JCE) software key store, an HSM or a cloud key storage provider.

With the above introduced Skytrust concept the proposed goals of flexibility, extensibility and platform independence can be reached and a broad range of different deployment scenarios are conceivable. A possible deployment scenario is, for example, a local connected smart card that is accessed from an arbitrary application by a defined receiver. In order to utilize a further key storage solution, for example, only a new actor has to be attached that connects the key storage solution. The connected key storage solution can be used without further modification of the Skytrust system.
Furthermore, the Skytrust system is not limited to a certain device. Due to the separation of the cryptographic process, the receiver and the performing actor do not need to be necessarily on the same device. Thereby, the deployment scenarios can be extended by remote actors. Such a remote actor can be utilized to perform arbitrary cryptographic operations on another device, for example, a web application performs a cryptographic operation on an HSM that is placed on a server. Moreover, this functionality can be even used to offer cross platform encryption. Thereby, that various receiver types are able to utilize the same actor such as an IOS application and an Android application that access the same remote actor on a server, cross platform encryption can be established. This is a major advantage, in contrast to the already existing solutions as presented in Section 3.2.

However, the basic prerequisite for the usage of the remote actor solution is the availability of a network connection. That is the main disadvantage of the remote actor in contrast to local tokens such as smart cards and security tokens. The fact

that nowadays mobile networks are available almost everywhere this limitation is significantly reduced.

Due to the possibility to access remote actors that provide cryptographic services, the user has to be authenticated to determine which keys belongs to the user. Furthermore, some key storage solutions need a secret, for example, a PIN, to unlock the key that is used. In order to meet this requirements the Skytrust system has an *authentication* component, as illustrated in Figure 4.1. This authentication component handles the authentication process to authorize the user to the respective key. The authentication component is decoupled from the receiver in order to prevent the unauthorized access to credentials by the receiver respectively the application. This concept enhances the security of the Skytrust system. A further advantage of the decoupled authentication component is that various authentication schemes such as *username/password*, *two factor authentication methods* or OAuth authentication can be supported easily. This guarantees once again the flexibility and extensibility of the *Skytrust* system.

Summary: The *Skytrust* system is designed to provide a flexible, extensible and platform independent solution to access a cryptographic key provider. The decoupled concept enables a broad range of deployment scenarios from local elements to remote elements, which is a major advantage to current solutions. Furthermore, the receiver concept allows the easy integration of the Skytrust system into existing solutions without using a special client, which is a major benefit. Finally the authentication mechanism guarantees also a high level of security.

4.2 Skytrust Element

In order to implement the above mentioned concept of the Skytrust system a so called *Skytrust Element (SE)* is used. It is the core element and realizes the, in the previous section, mentioned *receiver* and *actor* concept. To gain flexibility and extensibility the Skytrust Element is basically constructed in a modular way. Every entity in the Skytrust Element extends the central *module* entity, as illustrated in the schematic structure in Figure 4.2. In the remainder of this section we discuss the entities of the Skytrust Element and point out their duties.

4.2.1 Receivers

The *receiver* module is the connector interface for an arbitrary application that wants to use the Skytrust system. The connector interface provides a basic set of cryptographic operations to the user. The following receiver types are planned and briefly considered:

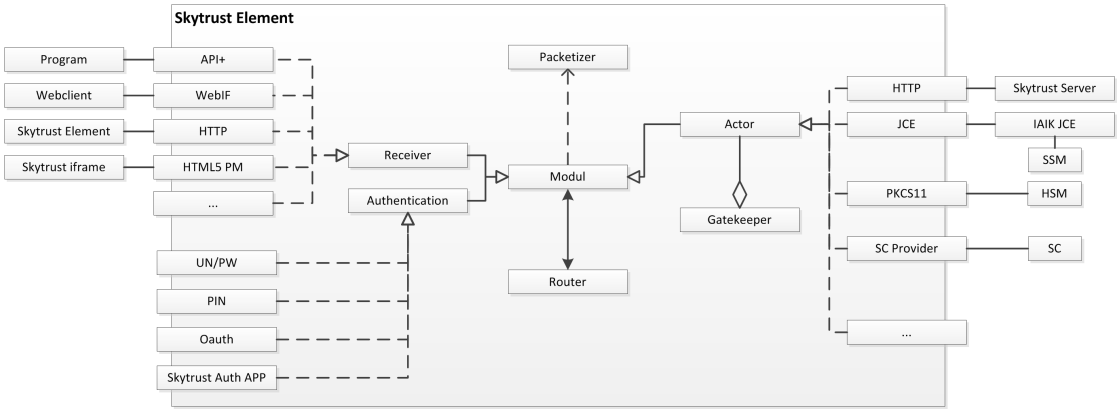

Figure 4.2: Skytrust Element

- API+ receiver
- WebIF receiver
- HTTP receiver
- HTML5 postMessage receiver

The *API+* receiver is the standard API to perform cipher operations such as *encrypt*, *decrypt* or *sign/verify*. The extension + offers additional operations to enhance the functionality of the Skytrust system. This means to provide the possibility to perform more complex operations, for example, block encryption mechanisms or to *retrieve certificates*.
The *WebIF* receiver provides support for a possible future web client.
The *HTTP* receiver is the connection point for possible other Skytrust Elements. That receiver allows, with an according actor on the other Skytrust Element, to connect two Skytrust Elements on different places and machines with each other. For instance, a cloud key storage provider with a local Skytrust Element. A describtion of possible Skytrust environments is discussed in Section 4.4.
Finally, the *HTML5 postMessage* receiver is the connection point for web applications that uses the HTML5 postMessage channel to utilize the Skytrust system. We decided, according to the principle of an extensible design, to enable the enclosure of supplementary receivers. By implementing the receiver *programming interface* a developer is able to integrate a new receiver to the Skytrust Element.

4.2.2 Actors

The *actor* module is responsible for linking an arbitrary key storage solution, which stores keys and performs cryptographic operations, to the Skytrust system. The result of the cryptographic operation is returned back to the corresponding receiver, which has sent the request. In order to support extensibility additional actor modules can

be integrated as well. To accomplish that, every actor has to provide a basic set of commands, for example, which cryptographic operations the actor is able to perform. These methods are used to establish the functionality of the actor to the Skytrust Element. We decided to support the following actors:

- *HTTP actor*
- *JCE actor*
- *PKCS11 actor*
- *Smart card provider actor*

The *HTTP actor* module connects a further Skytrust Element to the current Skytrust Element. It is the corresponding module to the, above discussed, *HTTP receiver*. The HTTP actor forwards an incoming transport packet to a further Skytrust Element. With that actor it is possible to connect several Skytrust Element instances with each other. Details on possible deployment scenarios can be found in Section 4.4.
The *JCE actor* module connects a Java Cryptography Extension (JCE) [29] software key store solution to the Skytrust Element. This actor can be used to link different JCE providers such as the *IAIK JCE* [1] provider to the Skytrust system.
A further actor is the *PKCS11 actor*. The Public Key Cryptography Standard (PKCS) [45] represents an API that connects cryptographic tokens such as HSMs to the Skytrust Element. This actor can be used to perform strong cryptographic operations in a secure environment.
The last predefined actor is the *Smart card provider actor* that connects a smart card to the Skytrust Element. That actor translates the Skytrust protocol commands, introduced further down in Section 4.3, into corresponding Application Protocol Data Unit (APDU) commands of the smart card.

4.2.3 Gatekeeper

In order to manage the access to the keys that are stored on the cryptographic key storage a component called *gatekeeper* is utilized. The gatekeeper therefore stores additional constraints for each key. These constraints determine, for example, if a user has to be authenticated to use the key. In order to achieve this the gatekeeper inspects a request that wants to perform a cryptographic operation with a restricted key and verifies whether the requesting user is authorized. If the user is not authorized to access the key, the gatekeeper enforces an authentication request. The authentication request is performed by the *authentication component*, discussed further down. If the authentication request is successful the gatekeeper unlocks the key to perform the desired operation. Otherwise, the usage of the desired key is not allowed. Beside that, the gatekeeper manages the successive use of the key. This means, that according to the additional restrictions, the key can be used for subsequent cryptographic operations without a further authentication. Therefore, the gatekeeper maintains a session identifier, which

[1] `http://jce.iaik.tugraz.at/`

is included in the transport protocol, to determine if a reuse is allowed. If the session identifier is expired, the user has to re-authenticate. More details about the session identifier can be found in the *Skytrust Transport Protocol* Section 4.3.

4.2.4 Authentication

The *authentication* component is the entity that is responsible for user authentication. As mentioned earlier, at the gatekeeper, a request is sent to authenticate the user to verify whether the user is allowed to access the key. The authentication entity can handle such a request from the gatekeeper. According to the type of authentication, for example, *PIN*, the authentication entity displays an appropriate user interface. If the authentication component is not able to handle the request type, it forwards the request to the next Skytrust Element, which may able to handle the request. If no further Skytrust Element is available the authentication component returns an error to the gatekeeper. The supported authentication methods of the Skytrust system are common methods such as *username/password*, *PIN* and *OAuth*.
A further task of the authentication component is to keep the entered user credentials or the issued *access tokens*, from the OAuth provider, away from the utilizing application. In order to meet that requirement the authentication component in the Skytrust system is decoupled from the receiver and thereby from the application. Depending on the environment conditions, the Skytrust system even allows the delegation of the authentication process to a separate application. For instance, in case of a Skytrust Element that runs on a smartphone the authentication process can be forward via internal communication methods to an especially designed authentication application.
A further purpose of the authentication component is the secure storage of the session identifier which represents the users authorization to use a selected key. As already mentioned, at the *gatekeeper*, the Skytrust system allows the successive use of the key without further authentication. In order to meet that requirement a session identifier is attached in the transport protocol. If a response transport packet contains such a session identifier, the authentication component stores it for further usage. Furthermore, the authentication component also removes the session identifier from the transport packet to prevent that the receiver respectively the utilizing application gains access to the session identifier. In case of a new request the authentication component attaches the stored session identifier to the transport protocol.

4.2.5 Packetizer

The *packetizer* component is responsible for encoding and decoding the transport packets of the *Skytrust Transport Protocol*. Every module has access to this packetizer to encode or decode incoming messages. Further details about the protocol and the encoding and decoding can be found in Section 4.3.

4.2.6 Router

The *router* is the central entity in the Skytrust Element. The main purpose of the router is to forward incoming *Skytrust Transport Protocol* packets to the corresponding module. We decided to keep the routing strategy as simple as possible. Depending on the command, every actor of the Skytrust Element is checked whether the actor is able to process the command or not. In case that no actor is able to process the command an error is returned. In order to identify the Skytrust Elements on which the transport packet was routed the router attaches the current Skytrust Element to the transport packet.
The Skytrust system allows the connection of several Skytrust Elements with each other. In order to support such complex networks the *router* has to be extended in the future.

4.2.7 Summary

The Skytrust Element itself is a complex element with different modules that communicate with each other. The *router*, as the central entity, decides to which element *Skytrust Transport Protocol* packets are directed. To maintain the platform independence the modules encode and decode the transport messages by using the *packetizer*. The *receivers* offers an interface of cryptographic operations to various platforms such as programs or web clients. The real key providers which performs the cryptographic operations and store the sensitive keys, are connected by an *actor* to the Skytrust Element. The constraints of the available keys are regulated by the *gatekeeper*. In case of needed authorization, the *gatekeeper* sends a request to authenticate the user. The authentication process is executed by the *authentication* entity which performs the authentication and returns it to the gatekeeper. To enhance security, the receivers are not able to access user credentials from the *authentication* entity. The following section the *Skytrust Transport Protocol* is discussed which is used to ensure the communication between the Skytrust Element entities.

4.3 Skytrust Transport Protocol

The *Skytrust Transport Protocol* is the underlying communication protocol of the Skytrust system to enable the transfer of messages between the Skytrust Element modules such as receivers and actors, as well as Skytrust Elements itself. The purpose of the protocol is to cover all necessary data, which is needed to carry out cryptographic operations, authentication and routing. Therefore, the structure of the transport protocol is spit into a world readable header part and a world readable or hidden payload part, as illustrated in Figure 4.3. The basic idea behind this separation is to provide the ability to encrypt respectively hide the payload for special use cases in the future such as end

to end encryption. We decided to leave the payload world readable to keep simplicity. The communication is protected by HTTPS to prevent the interception of transport protocol packets. As transport protocol format JSON [61] is used. The main reason for that decision was the easy usage, the simple structure and the availability of JSON libraries on nearly every platform [28].
The design of the presented protocol is subject to change to meet further deployment scenarios in the future.

The remainder of this section is separated into the header part and the payload part of the protocol. The header covers the information, that is needed to route the transport packets between the Skytrust Elements or Skytrust Element modules. The payload contains the data, which is needed to process the cryptographic operation, the settings for the used algorithm, key informations as well as informations about the authentication.

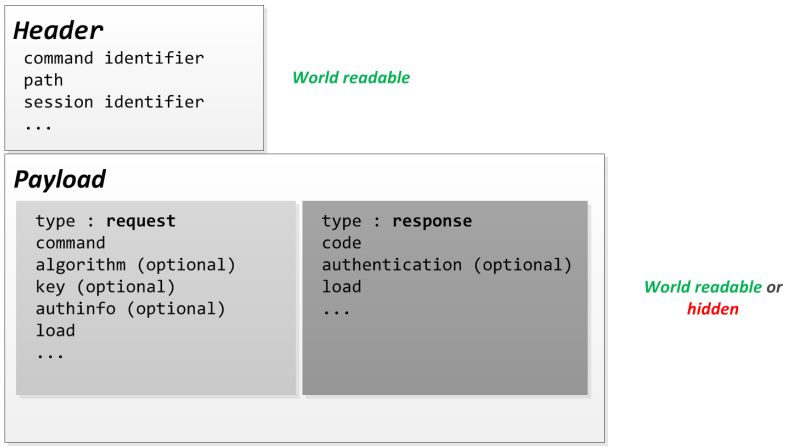

Figure 4.3: Skytrust protocol structure (Subject to changes)

4.3.1 Header

As mentioned, in the introduction, we decided to leave the header world readable. The main reason for that was that, whatever routing strategy is used in the future, any kind of information about sender, receiver or the path is needed. In the prototyping stage we decided to add only a *path* parameter into the header. The path parameter contains a list of Skytrust Elements on which the command packet was routed. The path list is constructed by the router by adding the Skytrust Element name, as described in Section 4.2.6. The constructed path list should state over which Skytrust Elements the transport packet was routed. To route a packet back from the performing actor this path is reversed. Thereby, the router is capable to return the packet back on the same

way. Additionally to the path parameter a *command identifier* is enclosed to the header. The main reason for that was to identify each packet by a uniform identifier during the transportation over several Skytrust Element entities and the ability to map command packets together. If authentication is necessary for a certain request the gatekeeper sends an authentication request which contains the command identifier of the initial request. The response of the authentication component also contains the command identifier. Thereby, the mapping between the initial request and the authentication response can be established. To keep the identifier unique a pseudorandom number is used. In the future this number can be possibly replaced by a combination of a pseudorandom number and a time stamp which is hashed by a hash function such as SHA256. A further parameter that is placed in the header is the *Skytrust Session Identifier*. The session identifier identifies the authenticated user as well as the corresponding key, which was used. This session identifier is used to perform subsequent operations without further authentication. The execution of subsequent relies on the constraints of the key. More on the subsequent execution of cryptographic operations can be found in Section 4.6. Since we decided to keep the payload world readable in the prototype stage the payload can be used to determine to which target actor the transport packet should be directed. However, as mentioned above, in the future some further parameters are necessary to enable an appropriate routing.

4.3.2 Payload

The payload contains, in contrast to the header, all data and information that is needed to perform the cryptographic operation, to handle the authentication or to perform the key discovery. The first element in the payload of the *Skytrust Transport Protocol* is the *type* which defines the type of the packet. The type distinguishes whether the transport packet is a request or a response. Based on this type the further payload elements differ, as illustrated in Figure 4.3.

The request payload contains at first a *command* that specifies which operation should be performed on the target actor. The command parameter is mandatory, otherwise no operation can be carried out. Available commands are the basic cryptographic operations such as *encrypt*, *decrypt*, *sign*, as well as the commands to retrieve public keys and user certificates from the key provider, and also commands for authentication. The first optional request payload parameter is the *algorithm* parameter. This parameter specifies which cipher algorithm is used to encrypt, decrypt or sign the enclosed data. Thereby, that not every command requires an algorithm, for example, *retrieve certificates* the parameter is optional. In order to provide flexibility the algorithm parameter is defined as a collection of entries. Which implies, that more entries are associated under the parameter. Possible sub entries of the algorithm parameter are the *name* of the algorithm and additional specifications that are needed to execute the algorithm, for example, the modulus length. The second optional parameter is the *key* parameter. This parameter is, equal to the algorithm parameter, a collection of sub entries. The key

parameter defines, which key should be used on the target actor. At the moment only the *key identifier* is fixed as sub entry of the key parameter. Any further parameters are possible. The *authinfo* collection is the next parameter, that is defined in the request protocol. The parameter is associated with the authenticate command. As mentioned, at the gatekeeper, an authentication request is sent to authenticate the user to verify, if the user is authorized to use the desired key. Therefore, the collection contains parameters which define what authentication scheme is needed as well as in case of a possible Oauth authentication, where the authentication server is reachable. Finally, the request protocol part contains the *load* parameter, which contains the plain data for encryption and signing or the encrypted data for decryption. In order to provide compatibility to a broad range of applications the data is encoded in *BASE64* [31].

The response payload contains – in contrast to the request payload – other parameters as mentioned above. The main reason for the separation is to keep the protocol as simple as possible and to keep the protocol short to enhance readability. The first parameter in the response payload is the *code* parameter. This parameter indicates, whether the request was successful or not. As return status values HTTP status codes [17] are used. The HTTP status codes are widely spread and well known. The next parameter is the optional *authentication* parameter. That parameter is a collection of several sub entries that contains the authentication *credentials*, which were collected by the authentication component. The *load* parameter contain, similar to the request payload, the data.

4.3.3 Summary

The *Skytrust Transport Protocol* is the underlying communication layer between the Skytrust Element modules and several Skytrust Elements. The availability of JSON libraries on almost every platform ensures the flexibility to distribute Skytrust Elements on various devices. In order to enable the possibility for end to end encryption in further consequence, the protocol is subdivided into a header and a payload part. The header is world readable to ensure the correct routing between elements. The payload is contains the data that is needed to perform the desired operations. In order to keep the protocol short and simple a separation between request messages and response messages is done. The compactness of the protocol enables the extension of the protocol in further consequence, to cover further deployment scenarios.

4.4 Skytrust Environments

In the recent years the demand for web services has strongly increased. Today, web services are no longer limited to the storage of data, however, even the processing of these data is possible. Commonly these web services are executed in a web browser. In

order to provide encryption and decryption of data in web applications the encryption and decryption process has to be carried out inside the browser environment. The development of the W3C Web Cryptographic API [55], a JavaScript based cryptography library, enables the ability to perform these cryptographic operations. However, the major issue of encryption in the browser environment is the secure storage of cryptographic keys which is not supported by the current web browsers.
In order to solve the secure key storage problem the above presented *Skytrust* system can be used. In order to enable the integration of the Skytrust system in to existing web applications, we introduce two basic environments. The *Skytrust Server Environment (SSE)* provides the central cloud-based key storage solution that can be utilized to store the cryptographic keys of the user. To integrate the *Skytrust Server Environment (SSE)* into the existing web application the *Skytrust Client Browser Environment (SCBE)* is used.

This section briefly focuses on the two basic environments.

4.4.1 Skytrust Server Environment

The *Skytrust Server Environment (SSE)* is the an environment on which the, in the previous section introduced, Skytrust Element (SE) can be placed. By deploying the Skytrust Element (SE) on a web server environment, the cloud-based solution for key storage can be supported by the Skytrust system. The web server, based on a REST structure, offers an API to other Skytrust Elements to perform cryptographic operations on trusted components such as an HSM or a smart card. The basic structure of this server environment is illustrated in Figure 4.4.

To ensure a secure connection between the requesting Skytrust Elements and the server environment the HTTPS protocol is used. Moreover, the server contains additional components such as a *User database* to authenticate the user which wants to access the Skytrust system and to link the appropriate keys that are stored in the cryptographic key storage solution to the user. A further component of the server is the *OAuth Authenticator*, which is responsible for requesting the user credentials at a supported OAuth provider such as *GoogleOAuth* [2]. The authenticator establishes a secure connection to the *resource server* and retrieves by presenting an *access token* the desired credentials. Furthermore, a session management is provided to enable *stateful* connections to the requesting Skytrust Elements.

All in all, the *Skytrust Server Environment* is Skytrust Element that provides the functionality of strong cryptographic key storage solution to several Skytrust Elements by presenting a REST interface.

[2] `https://developers.google.com/accounts/docs/OAuth2`

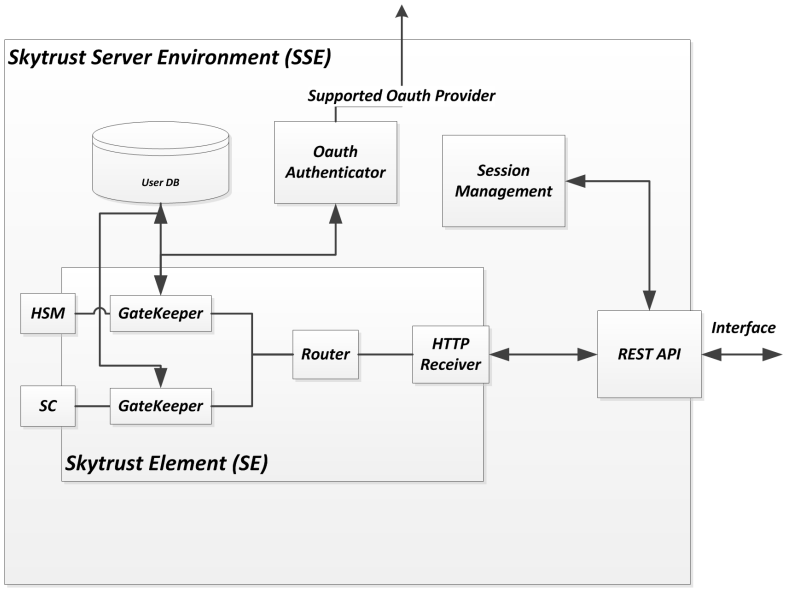

Figure 4.4: Skytrust Server Environment (SSE)

4.4.2 Skytrust Client Browser Environment

The *Skytrust Client Browser Environment (SCBE)* is an extension of the, in the previous section discussed, *Skytrust Server Environment*. It provides the functionality of the cryptographic key storage for local web applications that are executed in a user agent, for example, a web browser. An overview of this environment is illustrated in Figure 4.5.

The Skytrust Client Browser Environment (SCBE) is deployed in a web page, which can be easily embedded into an arbitrary web application by an *inline frame (iframe)*. The advantage of this solution is that the SCBE web page is deployed on a trusted web server, which uses the secure HTTPS protocol to communicate with the Skytrust Server Environment. In particular, the SCBE exposes the API of the Skytrust Server Environment, to forward or to process commands itself. Therefore, the SCBE web page includes a *Skytrust Element*. The platform independent concept of the Skytrust Element design permits to construct such a JavaScript based element. The Skytrust Element offers, by the prepared HTML5 postMessage receiver, an interface to the web application, which is able to utilize cryptographic operations that are performed on the server environment. Furthermore, the Skytrust Element of the SCBE web page is capable to integrate JavaScript based cryptographic libraries such as the W3C *Web Cryptography API*, by incorporate appropriate actors. This library can be used to provide hybrid encryption. In particular, the CryptoAPI generates a temporary symmetric key that is utilized to encapsulate the data. The temporarily generated symmetric key is

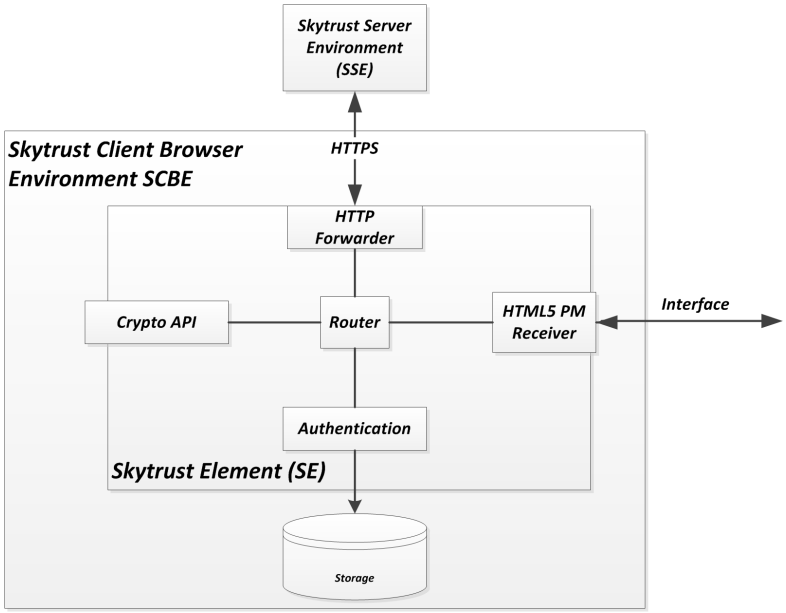

Figure 4.5: Skytrust Client Browser Environment (SCBE)

afterwards encapsulated with an asymmetric public key of the user, which is stored on the SSE. After the encryption the encapsulated symmetric key and the encrypted data are stored in a container format, for example, *S/MIME* and returned to the web application. Thereby, that the encapsulated symmetric key is stored in conjunction with the data, *no key* has to be stored in the user agent respectively the browser. In order to decrypt a container file, at first the encapsulated temporary symmetry key is separated from the encrypted data. Afterwards the encapsulated temporary symmetric key is decrypted at the SSE with the private asymmetric key of the user. In order to use the private asymmetric key, authentication is needed. This authentication is performed by the *SCBE web page* which contains an authentication component. By displaying an input form for user credentials or opening a redirection window to a supported Oauth provider, the SCBE web page is capable to perform the authentication. The decryption of the data can be also performed in the SCBE web page.

The integration of Skytrust system into an existing web application is realised by embedding the SCBE web page into a web application as an *inline frame (iframe)*. The security mechanisms of the browser such as the Same Origin Policy (SOP) ensures that an unauthorized access of the embedded SCBE web page is prevented. The communication to the SCBE web page is restricted to the HTML5 web messaging standard. This standard permits the controlled communication between frames of different origins (scheme, host and port). In order to use the *SCBE web page iframe* through the HTML5 communication channel a Skytrust Element is embedded into the *web application*. The web application is able to use the Skytrust system by utilizing a

provided receiver, for example, a W3C *Web Cryptography API* receiver. Thereby, that the web application utilizes the provided receiver the hybrid encryption scheme of the SCBE and SSE can be abstracted. The advantage of this solution is that the web application does not recognise that a hybrid encryption scheme is used. This solution enables the integration of the *Skytrust* system into various applications that utilize the W3C *Web Cryptography API*.

In order to use the SCBE and SSE environment a comprehensive security analysis has to be conducted to understand the risks of such a system. Thereby, that the data encryption is performed in the browser environment and keys are transferred between SCBE web page and the SSE an unauthorized access has to be prevented. The detailed security consideration of the presented environments is conducted in Chapter 5.

4.4.3 Summary

The *Skytrust Server Environment* and the *Skytrust Client Browser Environment* illustrate the simple integration of the Skytrust Element into different environments and deployment scenarios. Especially the *Skytrust Server Environment* is a general environment that can be used and integrated in various other scenarios. In consideration of the ability to allow cross domain key encryption the server environment has a central part. Beside that, the *Skytrust Client Browser Environment* is a very specific environment that is designed for the integration into an existing web service. Thereby, that no key has to be stored in the browser, the secure storage limitation of the browser can be removed. The flexibility of the Skytrust Element enables therefore a broad range of field of application. This addresses the opportunity for further environments such as a Skytrust Element on smartphone, which supports the local Near Field Communication (NFC) technology to perform cryptographic operations.

4.5 Basic Prototype

In order to demonstrate the fundamental functionality, flexibility and extensibility of the, in the previous section introduced *Skytrust Server Environment (SSE)* and *Skytrust Client Browser Environment (SCBE)*, we implemented a basic prototype as illustrated in Figure 4.6. This basic prototype uses both environments and accesses the cryptographic key storage solution on the server over the HTML5 postMessage interface. To check out the functionality a simple HTML web page was set up, to send appropriate commands to the interface. In this section we depict how typical cryptographic operations, as for instance *encryption*, *decryption* or *retrieve a certificate* respectively *retrieve a public key*, are performed by the Skytrust system. Security related issues, which came up on the communication between the web application and the, via an inline frame embedded, SCBE web page over HTML5 postMessage channel are covered further down in Chapter 5.

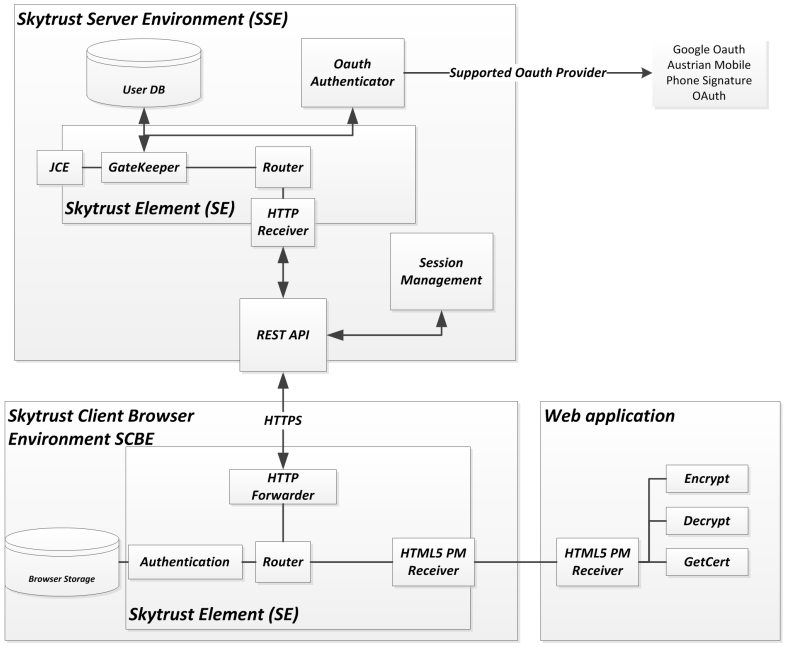

Figure 4.6: Basic prototype

4.5.1 Prototype Structure

At first we set up the *Skytrust Server Environment (SSE)* and the *Skytrust Client Browser Environment (SCBE)* web page on a web server and implemented a REST interface. Since the prototype is only constructed to demonstrate the control flow and the functionality of the Skytrust system, we reduced the components of each Skytrust Element to the essential parts. Therefore, the *Skytrust Server Environment* contains only one actor which is a JCE software key store. The JCE performs the cryptographic operation and stores the cryptographic key material. As receiver the HTTP receiver interface is used, and accessed over the *REST* interface. The *Skytrust Client Browser Environment* web page is set up on the same web server. The communication between SSE and SCBE web page is established on the server by the REST API and on the web page by the HTTP forwarder. In order to reduce the server implementation effort, we decided to deploy the SSE and the SCBE on the same server. In case of different origins, adaptations due to the *Cross Origin Resource Sharing (CORS)* have to be considered and raise the effort, which is not needed for demonstrating the functionality. The SCBE web page accommodates beside the *HTTP forwarder* only an *HTML5 postMessage receiver* – which represents the interface to the web application – and an *authentication* component. The authentication component uses the HTML5 session storage of the browser, to store the session identifier. The usage of the HTML5 session storage of the web browser to store the session identifier has security issues. However, the secure storage of the session

identifier was not scope of this work. Moreover, we set up a simple HTML web page on another web server which embedded the Skytrust system by including the SCBE web page as *inline frame (iframe)*. The main task of the simple web page was to send appropriate HTML5 postMessage commands to the SCBE web page to simulate the functionality of the Skytrust system.

4.5.2 Control Flow

In order to initiate a command such as to *retrieve a certificate*, an appropriate *Skytrust Transport Packet* is created and sent over the HTML5 postMessage channel to the SCBE web page, called *SCBE iframe*. At the *SCBE iframe* the origin check of the HTML5 postMessage channel communication is done. If the origin check is valid the packet is forwarded to the authentication component. If any session identifier is stored in the HTML5 session storage, the session identifier is added to the transport protocol by the authentication component. Afterwards the packet is sent over the *HTTP forwarder* to the server. On the server the received packet is forwarded to the *HTTP receiver*, which validates the packet structure and directs it, in case of validity to the *Router*. If the packet structure is invalid an error transport packet is returned to the *SCBE iframe* respectively the web application. In order to detect if authentication is needed, the transport packet is verified by the *gatekeeper* for a possible session identifier. If the session identifier is valid the packet is forwarded to the JCE actor. Otherwise the packet is stored in a queue and an authentication request is sent. This request is delegated back to the *SCBE iframe*, which is able to perform the authentication process. The authentication component inspects the *authinfo* parameter of the transport packet and opens a window to the appropriate OAuth provider. After the successful authentication at the OAuth provider the returned *access token* is sent back to the server. At the gatekeeper the *OAuth authenticator* is used to retrieve the retrieve the credentials at the OAuth provider. If the credentials are correct a new session identifier created by the gatekeeper and added to the stored initial packet. Then the packet is forwarded to the JCE actor. At the JCE key store the cryptographic operation is performed and the result is sent back to the *router*. In order to sent the packet back on the same way it came the *path* parameter of the protocol header is inspected and forwarded to the appropriate receiver. On the *SCBE iframe* the transport packet is directed to the authentication component to maintain the session identifier for a further request. Afterwards the packet is returned to the web application.

To sum up the control flow shows the separation of the cryptographic operation and the authentication process from the web application. Any security relevant cryptographic operation is decoupled from the insecure web application and performed on a secure environment. Moreover, the authentication is executed by the authentication component in the Skytrust Client Browser Environment web page, called *SCBE iframe* by using an external OAuth provider. This prevents the access of the web application on user

critical data or keys. A comprehensive security analysis of this prototype is provided by Chapter 5.

4.6 Application Permission System Concept

In this section we propose a general *Application Permission System Concept* for the Skytrust system. Permission systems are widely spread and used to restrict the access of applications to an API. In order to allow an application access to a specific API the user is prompted. However, the major drawback of this solution is that the user always tends to grant the application the access rights. This results from the fact that the prompted permissions are to complex and the user is not able to decide whether the permissions are correct.

Barrera et al. [2] analysed the Android Permission system by analysing over 1100 applications and showed that most of the application utilize a subset of the available permissions. A further issue they covered was that these permissions mostly inaccurate. They propose to provide a finer granularity for these frequent used permissions.

Smetters and Good [56] analysed access control features for document sharing. Smetters and Good covered that it is important to limit the flexibility of the permission system to simplify use.

Due to the fact that the *Skytrust* system performs security critical operations, a precise but even clear permission system has to be provided. To accomplish that goal, we analysed the *Skytrust* system to identify the key factors that should be considered for such a permissions system.

The remainder of this section is separated into the following subsections. The first subsection discusses the basic idea behind the application permission system as well as the influence factors. The second section describes the determined factors and how they can be used to derive appropriate permissions. Finally, an example how such a derived permission table can look like, is presented.

4.6.1 Basic Idea

The basic idea behind an *Application Permission System* is to enforce constraints under which an arbitrary application can use an interface. Therefore, many influence factors have to be considered. Such factors are the application as well as the environment on which the application is running. The application permission system joins all these factors to manages the access to a programming interface. The Skytrust system differs from a standard programming interface in that way that it allows the access to cryptographic keys. Thereby, the execution environment, from which the cryptographic keys are used, has to be considered. For instance, a web application that utilizes a

particular key has to be considered differently as a local application that is executed in a secure environment. In order to cope with the extended requirements, we decided to combine the application permissions with the still available key permissions to derive accurate constraints for each key in each environment respective on each application. In particular the existing *gatekeeper* is used to derive, on basis of the key and application permissions, appropriate constraints for an arbitrary application that uses a specific key.

In this section we deduce the basic constraints for the permission system. First, the *Key Permissions* are discussed in detail, which were introduced at the *gatekeeper*. Second, the core factors that influence the application permissions are considered.

Key Permissions

The *Key Permissions* restrict the access to a particular key that is managed by the key storage provider. In order to manage the restriction we decided that each key has to be unlocked before the cryptographic operation can be performed. That means, whenever the user wants to perform a cryptographic operation with a specific key, authentication is needed. However, this procedure is not very comfortable, in case of several successively performed operations. Therefore, the key permissions should allow successively operations and take into account the restriction of the key usage. In order to cover these restrictions we defined the key permissions that are illustrated in Table 4.1. The key permissions define for each key, identified by a *key identifier*, an *authentication type*, a *duration* and further *credentials*. The authentication type determines which authentication scheme is used to unlock the key. The duration defines the time how long respectively how often a key is usable before the user has to re-authenticate. And the credentials parameter is designated for further values that are needed to perform operations with the key.

key identifier	auth type	duration	credentials
uid	PIN, OAuth, U/P	times / minutes	-

Table 4.1: Key permissions

Derived Core Factors

In order to define accurate permissions for the *Skytrust* system we analysed the Skytrust system itself to determine the core factors which influence the cryptographic operation by an arbitrary application. The result of this analysis is presented in this section.

Operation Factor: The first factor we derived as core factor is the operation itself. The main reason for that fact is the *key type* that is used by the cryptographic operation. Depending on whether the private asymmetric key or the public asymmetric key is utilized by an operation the critical level of the operation has to be distinguished. Thereby, that the *private asymmetric key* is secret and cannot be extracted from the *Skytrust* system, operations that utilize the private key are critical. Operations that utilize the *public asymmetric key* are public and can be extracted from the *Skytrust* system are therefore less critical. The following operations, illustrated in Table 4.2, are summarized by the operation factor:

encrypt: The *encrypt* operation encrypts plain data or keys by utilizing the public asymmetric key of the user.

decrypt: The *decrypt* operation decrypts encrypted data or keys by using the private asymmetric key of the user.

sign: The *sign* operation creates a signature of the data. This is accomplished by calculating a hash value of the data which is encrypted with the private asymmetric key of the user.

Furthermore, the additional operations such as retrieve one or all certificates respectively public keys are also added to the *operation factor*. The collected elements of the operation factor are illustrated in Table 4.2.

Factor	Elements
Operation	*encrypt*
	decrypt
	sign
	getKey/Cert
	retrieveKeys/Certs

Table 4.2: Operation factor

Environment Factor: A further factor that is relevant for an application permission system is the surrounding environment in which the application is executed. The surrounding environment influences the security of an application in an extensive way. Applications which are executed in an insecure environment such as a smartphone environment, are more vulnerable than applications which are executed in a secure environment such as applications on a local machine. These differences have to be considered by an application permission system. We distinguish the following, in Table 4.3 shown, environments:

local: The *local* entry defines a local application that is executed on a common personal computer or a laptop.

Factor	Elements	Attributes
Environment	*Local*	-
	Mobile	Operating system (iOS, Android,...)
	Web	Origin

Table 4.3: Environment factor

mobile: The *mobile* entry summarizes mobile application that are executed on a mobile phone platform, such as iOS or Android.

web: The *web* entry is utilized for web application that are executed in a user agent respectively a web browser.

Furthermore, the *Attibutes* column specifies additional information that can be used to identify the environment from which a request was sent. More about the usage of this factor can be found in Chapter 5.

Authentication Factor: Another factor that is important for an application permission system is the authentication type which is used to authenticate to the application. Moreover, a developer should be able to define how long respectively how often the user is able to use application before the user has to re-authenticate. We distinguish the following, in Table 4.4 illustrated, authentication schemes:

Factor	Elements	Attributes
Authentication	*PIN*	
	OAuth	duration / time
	UN/PW	

Table 4.4: Authentication factor

PIN: The *PIN* element considers entering a pin or secret code to gain access to an application.

OAuth: The *OAuth* entry considers different OAuth schemes to authenticate to an application.

UN/PW: The *UN/PW* entry abstracts username / password authentication schemes.

Developer Factor: Finally, even the status of a developer should be included as a factor in the permission system. A trusted developer with many experience has a higher confidentiality than a developer that registers at online web from to gain access to the

Skytrust system. The difference between such developers have to be considered. The different developer status types are illustrated in Table 4.5.

Factor	Elements	Attributes
Developer	*highly trusted* *medium trusted* *low trusted*	-

Table 4.5: Developer factor

In this section, we discussed the basic idea behind an application permission system. First, we presented the in the Skytrust system still available key permission system, that is enforced by the *gatekeeper*. Second, we determined the core factors that have to be considered for an application that performs cryptographic operations on the Skytrust system.

4.6.2 Derived Permission System

In this section, the permission concept is presented that includes all determined factors of the previous section. Moreover, the derivation of the permissions is presented. Furthermore, a simplified representation of the permission concept is outlined, that is shown to the user before the Skytrust system is accessed.

In order to derive appropriate permissions for each key and application, we decided to categorize each factor into three critical levels. This permits an abstract view on every factor and enhance a simpler derivation of the permissions. These levels define how critical a key or environment is. For instance, a low critical key means that the key can be used for arbitrary operations. On the other hand a high critical key should only be used in a highly trusted environment.

Key Categories: In order to categorize the key permissions into three critical levels we use the duration parameter. The duration parameter defines the duration time or count how long or often a key is state unlocked to perform cryptographic operations. Furthermore, this parameter can be used to determine the confidentiality of the key. For instance, a key with a long duration time is less critical than a key with a short duration time. Table 4.6 shows the boundaries we defined for the critical levels. Note, that the boundaries are only guidelines to classify appropriate levels.

duration	critical level
1-5 min / 10 times	high
6-30 min / 11-50 times	medium
> 30 min / > 50 times	low

Table 4.6: Key security levels

Environment Categories: The classification of the environment depends on how trustworthy an environment is on which the application is executed. We defined that local applications that are executed on a local machine are less critical than web applications that are executed in a user agent such as a web browser. Furthermore, we categorized the smartphone environment as medium critical. However, the main reason for this decision was that smartphone operating systems not always up to date. This issue is particularly relevant for the Android operating system as discussed in Android Fragmentation Report [3]. The environment critical levels are presented in Table 4.7.

Environment	critical level
local	low
smartphone	medium
web application	high

Table 4.7: Environment critical levels

Operation Categories: In contrast to the environment and the key factor the operation factor is categorized in operations groups. Since every operation uses another type of key, for example, the *encrypt* operation uses the public key and the *decrypt* operation uses the private key, the categorization of the operations is based on the key type. The categorization of the operations is illustrated in Table 4.8. The *no* group contains operations that perform no operation with the key. Hence, we categorized the retrieve key operations into this category. Furthermore, we categorized the *encrypt* operation into the *public* group and the *decrypt* and the *sign* operation into the *private* group. Based on this groups we defined the critical levels.

[3] `http://opensignal.com/reports/fragmentation-2013/`

Group	Elements	critical level
no	getKey / getCert retrieveKeys / retrieveCerts	low
public	encrypt	medium
private	decrypt sign	high

Table 4.8: Operation groups

Permissions Derivation

In order to derive the accurate permission for a particular key, the key permissions and the application permissions are combined by the following derivation rules. However, this set of rules serves only a basis and have to be modified in future to permit more granularity depending on the particular deployment scenario and use case.

1. *Authentication type*
   If the authentication type of the key and application does not match, the key authentication has to be performed anyway. In case of an identical authentication scheme the key authentication can be omitted.
2. *Authentication duration*
   The key duration has priority over the application duration. However, if the duration of the application is shorter than the duration of the key, the application duration has priority.

Beside these rules the following suggestions can be considered:

- **Environment Influence**
  The security level of an application environment reduces the duration of the key usage. This prevents the possibility that a high security key is unlocked for a long time in an insecure environment. We propose to reduce the duration time of a medium environment application by half and the for a low environment to a privilege elevation for every request.
- **Developer Influence**
  The confidence level of the developer restricts the key usage. An application which is developed by a, for example, low trusted developer is not permitted to utilize high critical keys.
- **Low Environment Influence**
  Applications that are executed in a low security environment and utilize keys with a high security level have to be granted. Therefore, we propose to prompt a separate notification to the user to grant the execution with such a critical key.

Application Grant

In order to inform the user about the restrictions of the application the above derived security levels are used. Figure 4.7 shows a possible *Application Grant* notification. The notification obtains the security levels from the appropriate levels in the application permission system to inform the user about the properties of the application.

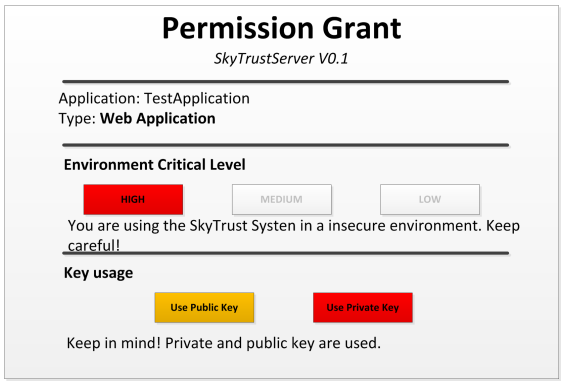

Figure 4.7: Application grant notification

The *Application Grant* notification 4.7 can be interpreted as follows:

- The web application *TestApplication* requests access to the *Skytrust Server*.
- The web application is executed in a web browser and therefore the *Environment critical level* is *high*.
- The web application utilizes operation which use the public and the private key of the user.

4.6.3 Permission Concept - Example

In this section, we illustrate possible entries for the key permissions, shown in Figure 4.8, and entries for the application permission entries, shown in Figure 4.9.

Key Permissions

key identifier	auth type			duration		credentials	critical level		
	Pin	Oauth	UN/PW	minutes	times		low	medium	high
emailkey1	x				3	-			x
emailkey2			x	11		-		x	
emailkey3		x		35			x		

Figure 4.8: Key permission table

Application Permissions

api key	Name	auth type		duration		Environment				Operations				
		Oauth	UN/PW	minutes	times	local	smart phone	os	web origin	getKey/Cert	retrieveKeys/Certs	encrypt	decrypt	sign
158x38	Test		x	10			x	iOS		x	x	x	x	
68ad82	Test2	x		30					x oes.com	x	x	x	x	

Figure 4.9: Application permission table

A derived entry in the final permission table, shown in Figure 4.10, for a particular key and a particular application can be interpreted as follows:

- *Emailkey1* is used by the smartphone application *Test*.
- The *emailkey1* has to be unlocked by entering a *PIN* and can be reused *3 times*. The derived critical level of the key is *high*.
- Application *Test* utilizes a *username / password* authentication scheme and allows the user to use the application *10 minutes* before the user has to re-authenticate. The derived environment critical level is *medium*.
- The application *Test* utilizes the *retrieve certificates / keys, encrypt and decrypt* operations. The derived operations critical level is *high*.
- Thereby, that the application *Test* is a smartphone application the duration of the *Emailkey1* is reduced by half. This results in a key duration of *one*. In particular, each operation with *Emailkey1* such as *encrypt or decrypt* has to be granted by the user.

		Derived Permission																			
		auth -type																			
		key			application		Environment						Operations						auth		
Key	App	type	duration	derived level	type	duration	local	smart phone	os	web	origin	derived Level	getKey/Cert	retrieveKeys/Certs	encrypt	decrypt	sign	derived Level	omit key auth	duration (key)	Notification
emailkey1	Test	Pin	**3 t**	high	UN/PW	10 m		x	IOS			medium	x	x	x	x		high		1 t	Usage: high critical key and operation in medium critical environment, Prompt for explicit grant
emailkey2	Test	UN/PW	11 m	medium	UN/PW	**10 m**						medium						high	x	5 m	Usage: high critical operation
emailkey3	Test	Oauth	35 m	low	UN/PW	**10 m**						medium						high		5 m	Usage: high critical operation
emailkey 1	Test2	Pin	**3 t**	high	Oauth	30 m				x	...	high	x	x	x	x		high		1 t	Usage: high critical key and operation in high critical environment, Prompt for explicit grant
emailkey2	Test2	UN/PW	**11 m**	medium	Oauth	30 m						high						high		1 t	Usage: high critical operation
emailkey3	Test2	Oauth	35 m	low	Oauth	**30 m**						high						high	x	1 t	Usage: high critical operation

t... times
m...minutes

Figure 4.10: Derived permission table

Summary: In this section an application permission concept was presented that allows the combination of key permission and application permission. Furthermore, classification categories to simplify the deduction and to provide a simple representation for the user were presented. Although the combination of the key permission and the application permission provide a more precise configuration of the access of application to appropriate keys, this advantage leads in complex deriving rules.

4.7 Summary

In this chapter we presented the Skytrust Element and its broad range of deployment scenarios. The flexible, extensible and platform independent concept was demonstrated by a prototype which contains two possible environments on which a Skytrust Element can be engaged. Furthermore, a possible *Application Permission System* which allows the precise restriction of accessing applications to keys, was presented. In the following section a comprehensive security analysis on the outlined Skytrust Element respective the presented prototype is provided.

Chapter 5

Security Analysis

In this chapter a comprehensive security analysis on the *Skytrust* system is given. Web services that offer the ability to store and process data gained high relevance in the past years. These web services are mostly utilized by web applications which are executed in a user agent respectively a web browser. Thereby web browser becomes a central element in the web application scenario. In order to enable the ability to perform cryptographic operations on the processed data of the web applications, the *Web Cryptography API* [55] was developed. However, the main problem of cryptographic operations that are executed in the web browser is that web browsers are not capable to provide a secure key storage. To solve the secure storage problem the *Skytrust* system can be utilized. The *Skytrust Server Environment* and the *Skytrust Client Browser Environment* enables the integration of the Skytrust system into existing web applications. However, the execution of cryptographic operations in the browser context as well as the transfer of data between frames and environments enables the possibility to enforce attacks.

In order to understand the risks of the *Skytrust* system in a web application scenario, a comprehensive security analysis to determine core assets, threats and countermeasures is conducted.

The foundation how to conduct the threat modelling and the risk analysis is taken from [5] and [38].

The remainder of this section is structured as follows. First, the underlying scenario of the security analysis, the work flow and assumptions on this scenario are presented. Second, the core assets are defined and categorized into three categories. Afterwards five attack scenarios are defined and discussed by pointing out the threats, the countermeasures and the residual risk. Finally a concluding summary is given.

5.1 Scenario

The underlying scenario, which is used to analyse the Skytrust system, is based on Skytrust prototype which was presented in Section 4.5. The scenario, illustrated in Figure 5.1, simulates a web application that utilizes the Skytrust system for hybrid data encryption.

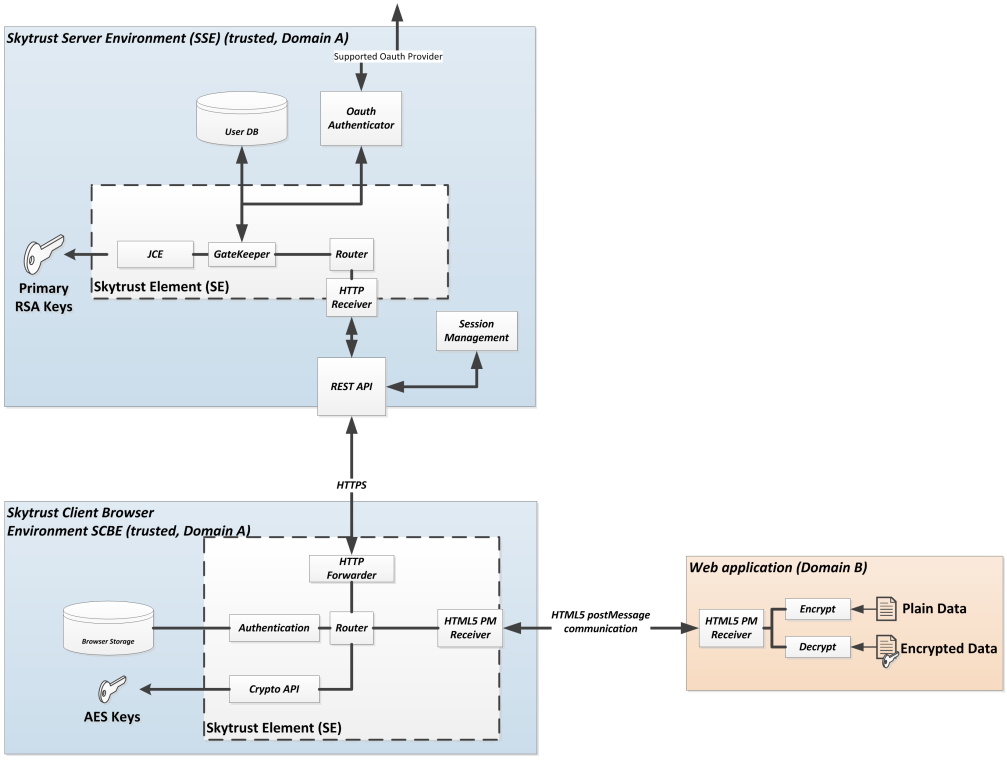

Figure 5.1: Security analysis scenario

The hybrid encryption scheme enables the execution of the data encryption in the browser and the key encryption on the server. Thereby, that the *symmetric AES key* is sealed with a secure stored key of the cloud-storage solution, the *symmetric AES key* must not stored in the browser. In order to enable this scenario the Skytrust Server Environment (SSE) is used. The SSE connects a software keystore which contains the private sensitive key material of the user. This server is provided by a trusted institution at *Domain A* and it can be reasonably assumed that only the authorized user has access to the appropriate keys. Moreover, the server provider manages all accessing web applications that are able to access the system. In order to access the server also a trusted *Skytrust Client Browser Environment (SCBE)* web page on *Domain A* is provided that can be embedded into the web application as an *inline frame (iframe)*.

Skytrust Client Browser Environment (SCBE) web page offers an HTML5 postMessage interface for the *Web application* on *Domain B*. Furthermore, the *Skytrust Client Browser Environment (SCBE)* web page ensures the symmetric data encryption and decryption by a connected JavaScript *Web Cryptography API* as well as asymmetric encryption and decryption of the symmetric AES keys by the trusted server over an HTTPS connection. As the *Skytrust Client Browser Environment (SCBE)* web page is embedded as an iframe, it is decoupled of the web application and can be used for secure user authentication.

5.1.1 Workflow

In order to get a clear picture of how the above scenario works a detailed description of the workflow is given. The first phase of interaction is the loading of the web application and the embedded SCBE web page as *iframe* into a client, respectively a browser. After that step the user is able to encrypt plain and decrypt encrypted data. Regardless of which function is requested, a transport packet is created. The transport packet encloses the function, the plain or encrypted data and additional information which is forwarded over the HTML5 postMessage channel to the iframe respectively the SCBE web page. At the SCBE web page the appropriate function and data is processed. In case of an encryption call the plain data is encrypted by a *random AES key* that is generated by the *Web Cryptography API*. Afterwards, the *AES key* is encrypted on the server with the *public RSA key* of the receiver. The encryption of the *AES key* with the *public RSA* can be also operated in the SCBE web page, however, the *public RSA* key of the user can be retrieved from the server. After the symmetric key encryption the *encrypted data* and the *encrypted AES key* are returned to the web application in a container format, for example, *S/MIME*. Thereby, that the *encrypted AES key* is stored with the encrypted data, the *AES key* can be deleted in the browser.

In order to decrypt the encrypted data from the web application the previous process is reversed. At first, the *encrypted AES key* is decrypted at the server. Contrary to the encryption process the decryption process has to be carried out on the trusted Skytrust Server Environment, as the *private RSA key* must not leave the server. Furthermore, an additional authentication is needed to authorize the decryption of the *encrypted AES key* with the *private RSA key* of the user. The authentication step is performed by the SCBE web page by redirecting the user to an OAuth provider to enter the credentials. The returned *access token* from the OAuth provider is forwarded to the server to unlock the *private RSA key*. The server itself requests with the *access token* at the OAuth provider in order to receive the authorization for unlocking the key. If the user is authorized to unlock the *private RSA key* the decryption of the *encrypted AES key* is performed and the *AES key* returned to the SCBE web page to perform the symmetric decryption of the data. After the decryption the data can be returned to the web application.

5.1.2 Assumptions

In order to provide more accuracy some further assumptions were made:

- **OAuth Provider:** As already mentioned the authentication process of the user is operated by the SCBE web page by redirecting the process to an OAuth provider. This provider can be, for example, Google OAuth or the *Austrian Mobile Phone Signature*. Regardless of the exact functionality of the provider an *access token* is returned to the SCBE web page, which can be used to authenticate the user. Security issues due to the authentication process with OAuth such as discussed by Sun and Beznosov [60] have to be considered to ensure security.
- **Communication:** Furthermore, the communication between the trusted SCBE web page and the server on *Domain A* is secured by HTTPS. HTTPS ensures the encrypted and authenticated communication between server and client. HTTPS bases on TLS which protects the HTTP requests. The usage of HTTPS only ensures the secure identification of the server. However, the authentication of the client, which is provided by TLS, is mostly not required by servers. By using a client certificate the security of the HTTPS connection can be enhanced. In order to prevent attacks it is mandatory to verify the server certificate.
  As the Skytrust Client Browser Environment web page and the Skytrust Server Environment are deployed on the same server limitations due the *Same Origin Policy (SOP)* do not have to be considered. If that is not the case and the SCBE web page is not offered by the same server security issues of the *Cross Origin Resource Sharing (CORS)*, as discussed in Ryck et al. [50], have to be taken into account.
- **AES keys:** As described in the scenario the *AES keys* are used for symmetric encryption of data. The AES keys exist only for a short time as long as the encryption or decryption of the data is processed and the AES key sealed or unsealed by the primary keys. Therefore, theses keys are called *temporary AES keys*.

5.2 Assets

The *Skytrust* system utilizes well-known cryptographic key storage solutions such as HSMs and communication technologies such as HTTPS. However, due to the distributed architecture of the Skytrust system the existing risk and security analyses can only partially reused at the Skytrust system. The core asset of the Skytrust system are the cryptographic keys that are stored in the Skytrust system. The security goal is to keep these cryptographic keys secret. Based on the core asset the *primary cryptographic keys* further assets have to be considered.

In the remainder of this section, the core asset and related assets are discussed. Figure 5.2 shows the above introduced scenario and the extracted assets. In order to get

an overview on the relevance of each of the assets we categorized them into three categories as shown in Table 5.1.

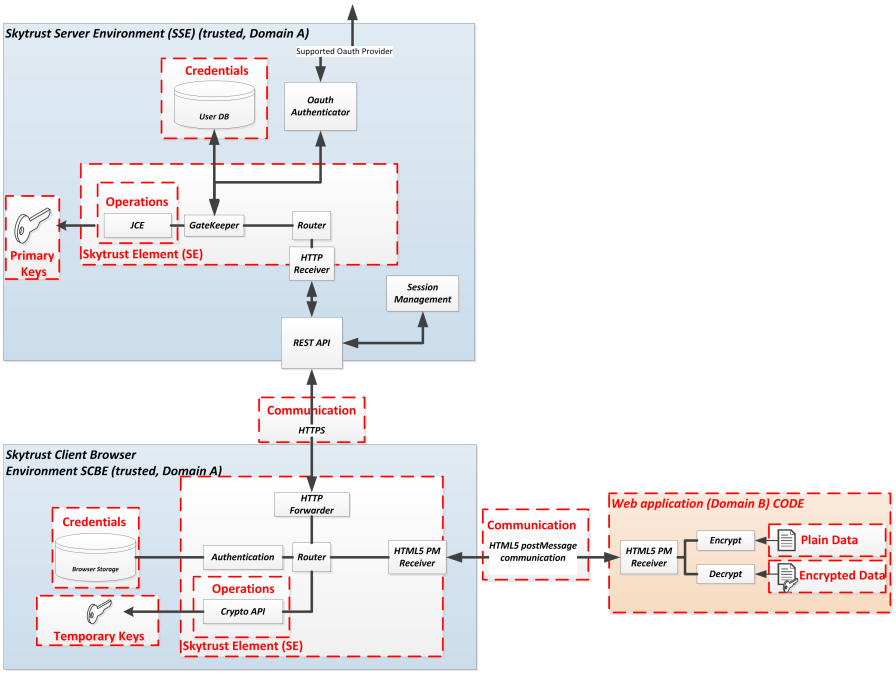

Figure 5.2: Skytrust prototype assets

5.2.1 Primary Cryptographic Keys – Core Asset

The core asset of the Skytrust system are the primary cryptographic keys. These cryptographic keys are utilized by the cryptographic functions to offer encryption, decryption and signing operations. It is essential for the Skytrust system that the primary cryptographic keys are *not extractable* of the Skytrust system. Moreover, the Skytrust system must ensure that the keys are not manipulable by an attacker.

Further assets that are related to the primary cryptographic key are the *credentials*, the *cryptographic operation*, the *temporary keys*, the *web application code* as well as the *plain / encrypted data*.

5.2.2 Credentials – Related Asset

Accessing the Skytrust system without authentication should not be possible. Therefore the authentication scheme as well as the credentials have to be considered as an asset. The above described scenario utilizes an OAuth authentication scheme to restrict

Asset categories		
core	**related**	**utilized**
primary keys	credentials operation temporary keys web application code plain / encrypted data	communication Skytrust Element

Table 5.1: Categorized assets

the access to the user's keys. Depending on the particular deployment scenario this authentication scheme can vary. For a local Skytrust Element that, for example, utilizes a smart card to carry out cryptographic operations a *PIN* based authentication scheme is sufficient. Which is on the other hand not possible for the current scenario due to various stored keys for different users at the Skytrust server.

5.2.3 Cryptographic Operation – Related Asset

The cryptographic operation also presents a related asset, since the operation utilizes the primary or the temporary keys. Depending on the function of the operation different keys are used such as private keys for decryption and signing operations and public keys for encryption operations. However, in the distributed architecture of the Skytrust system these operations are divided into a local execution part and a remote execution part. In order to create a signature by using the signing operation the hash value is created locally and the encryption with the private key remotely at the server. This example illustrates that also additional operations such as *Hash* functions are used, which do not require a cryptographic key. These operations must also be considered in the cryptographic operation asset.

Regardless of which operation is performed, it is necessary for the Skytrust system that the execution of the operation cannot be manipulated by an attacker.

5.2.4 Temporary Cryptographic Keys – Related Asset

Depending on different encryption schemes the availability of temporary keys have to be considered as a related asset. In this context *temporary key* means that the *randomly generated symmetric AES key* for data encryption is available as long as the encryption and decryption of data is processed in the Skytrust Element. After the data encryption the symmetric AES key is sealed by an asymmetric key and attached to the encrypted

data. Afterwards the symmetric AES key is deleted in the Skytrust Element. Therefore, the randomly generated symmetric AES key are called *temporary keys*.

The symmetric AES key never leaves the Skytrust Element on which the operation is performed except in case of the decryption of the encrypted AES key.

5.2.5 Web application code – Related Asset

The code of the web application also has to be considered as related asset. A weak implementation of the web application can reveal information about plain and encrypted data.

5.2.6 Data – Related Asset

The final related asset is the data. The data is indirectly connected to the core asset as it is related to the temporary keys and the operation. By considering the data as an asset the data has to be divided into plain and encrypted data. Especially the transfer of plain data has to be taken into account, since the data can be easily read by an arbitrary attacker. On the other hand the encrypted data also have to be considered, however in another way, since the data is encrypted and thereby might not be readable by an attacker.

5.2.7 Communication – Utilized Asset

The communication between the Skytrust Elements is categorized as utilized asset. Due the large area of different deployment scenarios and the distributed architecture the communication is very important. The scope of deployment scenarios ranges from a simple Skytrust Element with a connected secure key storage such as a local connected smart card on a personal computer, over a Skytrust Element that is connected to a cloud Skytrust environment to a complex structure of Skytrust Elements with different key storage solutions. Is is essential that an arbitrary attacker is not able to gain access to data that is transferred between Skytrust Elements. Therefore, the communication is a complex asset and has to be considered depending on the particular deployment scenario.

5.2.8 Skytrust Element – Utilized Asset

A further utilized asset of the Skytrust system is the Skytrust Element itself. Thus, there are various deployment scenarios possible, the underlying platform and environment of the Skytrust Element should pay particular attention. Each platform has different

security challenges that influence the behaviour of the Skytrust Element. For instance, the JavaScript Skytrust Element which is located at the *Skytrust Client Browser Environment (SCBE)* has to consider other storage possibilities as a Java Skytrust Element on the *Skytrust Server Environment (SSE)*.

5.2.9 Summary

In this section we extracted the core assets that influence the Skytrust system by taking account of the requirements of the scenario defined in Section 5.1 and categorized them into three categories. The categorization presents the connection between the core asset and each other asset. In the next section we define the attack scenario and their effects on the assets from this section.

5.3 Attack Scenarios

In order to provide a precise security analysis we decided to define five attack scenarios which possibly occur on a real case usage of the Skytrust system. These five scenarios, illustrated in Figure 5.3, are:

- a *local attack scenario* which simulates an attack on the local machine of the user
- a *web attack scenario* which simulates an attack on the SCBE web page respectively the iframe
- a *communication attack scenario* which simulates an attack to obtain data from the communication between SCBE web page and server
- a *server attack* that simulates an attack on the server
- an *operator attack* that simulates an attack of the operator of the server

In the remainder of this section we discuss the effects of each attack scenario to extract the threats, the countermeasures and the residual risk of the Skytrust system.

5.3.1 Scenario 1 – Local Attack

The *local attack* scenario simulates an attack on the Skytrust system of an attacker which has infiltrated the local machine of the user by installing any kind of a malware application. With this malware application, for example, a key logger the attacker is able to obtain the credentials of the user. Thereby, the attacker is able to perform operations in behalf of the user. Furthermore, the attacker is able to attack the web application or the SCBE web page iframe by injecting a malicious script to receive session information to impersonate the user or to gain access to data.

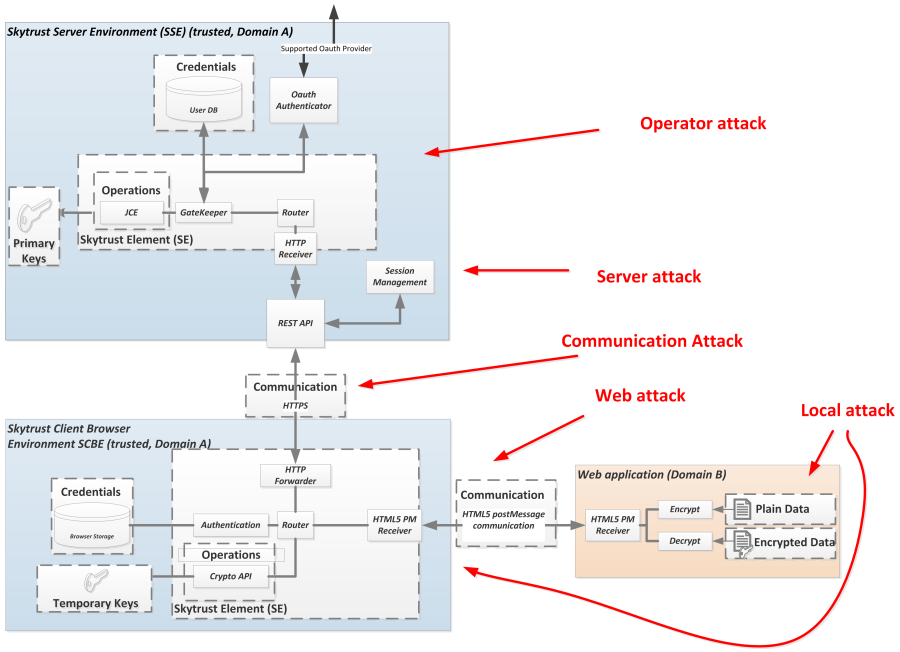

Figure 5.3: Skytrust system attack scenarios

Threats: The following threats on the *Skytrust* system occur from the local attack scenario:

- **(T): Gain Access to Data**
  An attacker who has access to the local machine of the user, is able to obtain plain and encrypted data, which is transferred from the web application to the *Skytrust Client Browser Environment (SCBE)* web page iframe. Although the attacker is not able to gain access to the core asset, the *primary key*, the attacker is also able to obtain the data of the user.
- **(T): Obtain Credentials**
  An attacker who is able to obtain the credentials of the user by using, for example, a key logger or by injection code, is able to carry out own operations with the core asset, the *primary key*.
- **(T): Obtain Session Identifier by XSS**
  An attacker who has access to the local machine of the user is able to inject code into the embedded SCBE web page iframe of the web application in the browser. Thereby, the attacker is able to gain access to the session identifier of the user. With this session identifier the attacker is able to impersonate the user by using the core asset, the primary key.
- **(T): Obtain Session Identifier by CSRF**
  An attacker who has access to the local machine of the user is able to embed

an exploit page into the web application to lure the victim to perform malicious requests.

Countermeasures: The Skytrust system provides the following countermeasures to mitigate the above mentioned threats:

- **(C): Authentication Scheme Skytrust System**
  Although the attacker is able to obtain the user's credentials on the local machine the malicious usage of the core asset, the primary key, is prevented by the Skytrust system by utilizing a strong authentication scheme with multi factor authentication, for example, two-factor authentication. By using this multi factor authentication scheme the credentials are not sufficient enough to unlock the key. Furthermore, also a *possession factor*, for example, a smartphone is necessary to permit the key usage. In order to utilize the key a verification code is sent to the smartphone. This code has to be entered to perform an operation. Thereby, that the attacker is not in possession of this device the attack can be prevented.
- **(C): Authentication Scheme Web Application**
  The access to the plain and encrypted data that is stored in the web application can be prevented by using a strong multi factor authentication scheme. However, this depends on the implementation of the web application. We recommend to use a multi factor authentication system if it is provided by the operator of the web application.
- **(C): Permission Concept**
  The Skytrust system enforces an unique session identifier to store the authentication of the user. In certain use case scenarios, for example, trusted local applications that utilize the Skytrust system, this session identifier is used to enable the subsequent operation without any further authentication. This comfortable function still has the drawback that the authentication have to be stored on client side. Since the identifier is stored in the HTML5 session storage the identifier is not protected in the same way as a cookie. To mitigate this drawback the proposed permission system allows the privilege elevation of particular operations by an additional authentication step so that the attacker is not able to reuse a critical key. The maintenance as well as the protection of the *session identifier* is out of scope of this work.
- **(C): Trusted Implementation SCBE Web Page**
  In order to prevent *Cross-Site Scripting (XSS)* attacks several protection mechanisms such as escaping frameworks or scanning tools are available. By following best practises [18] and utilizing well known libraries for executing the cryptographic operations the possibility of code injection attacks can be mitigated.

Residual Risk – Low/Medium: To determine the residual risk of the local attack the following two cases have to be distinguished:

- **Attacker is not in possession of verification device: Residual Risk – Low**
  If the attacker is not able to gain access to the verification device of the multi factor authentication scheme, the attacker is not able to use the core asset, the primary key. Therefore the residual risk is considered to be *low*.
- **Attacker is in possession of verification device: Residual Risk – Medium**
  If the attacker is able to gain access to the verification device of the multi factor authentication scheme the *Skytrust* system is not able to prevent the usage of the core asset, the primary key. Therefore, the residual risk has to be considered to be *medium*.

Furthermore, the attacker is able to obtain the plain and encrypted data which is processed by the web application. This cannot be prevented by the Skytrust system, however, only the developer of the web application is able to prevent attacks by following best practices in the communication between web application and the SCBE web page iframe.
In order to reduce the possibility of a local attack, we recommend to use common methods to avoid malware attacks such as the usage of anti-virus programs, installing operating system updates, etc.

5.3.2 Scenario 2 – Web Attack

The *web attack* scenario simulates an attack on the *Skytrust Client Browser Environment* web page to obtain any kind of information. By enforcing *Cross-Site Scripting (XSS)* attacks the attacker tries to gain access to credentials. Furthermore, a *Cross Site Request Forgery (CSRF)* attack can be used to trick the user to send a malicious request. Furthermore, the attacker attempts to manipulate the utilized libraries or to perform requests to foreign servers.

Threats: The following threats on the *Skytrust* system occur from the web attack:

- **(T): Obtain Credentials, Session Identifier, Keys**
  By injecting a malicious script the attacker is able to obtain:
  - *credentials*
  - *plain / encrypted data*
  - *temporary AES keys*

  Thereby, the attacker is able to use the core asset, the primary keys, in behalf of the user. Furthermore, also the temporary AES keys can be used to decrypt the encrypted data.

- **(T): Manipulation of Operations**
  By replaying the cryptographic library which is used in the SCBE web page an attacker is able to manipulate the cryptographic operation. Thereby, the attacker is able to gain access to the data.
- **(T): Manipulate Skytrust Element**
  By injecting a malicious script the attacker tries to manipulate the Skytrust Element to send request to a foreign server or to influence the functionality.

Countermeasures: The Skytrust system provides the following countermeasures to prevent the above mentioned threats:

- **(C): Trusted Implementation SCBE Web Page**
  In order to prevent *Cross-Site Scripting (XSS)* attacks several protection mechanisms such as escaping frameworks or scanning tools are available. By following best practises [18] and utilizing well known libraries for executing the cryptographic operations the possibility of code injection attacks can be mitigated.
- **(C): Independent Execution Environment**
  The *Skytrust Client Browser Environment (SCBE)* SCBE web page on *Domain A* is encapsulated by the Same Origin Policy (SOP) from the web application which is deployed on *Domain B*. The communication is restricted to the HTML5 postMessage channel. The HTML5 postMessage channel has still vulnerabilities as discussed in Section 2.3. To mitigate these vulnerabilities and to improve the HTML5 postMessage communication a suggestion for an improvement is presented further down in Section 5.4.
- **(C): Permission Concept**
  The *Permission Concept* countermeasure is already discussed in Section 5.3.1.
- **(C): Authentication Scheme Skytrust System**
  The *Authentication Scheme Skytrust System* countermeasure is already discussed in Section 5.3.1.

Residual Risk – Low: Due to the trusted implementation of the *Skytrust Client Browser Environment web page* the weak point of a web attack is limited to the HTML5 postMessage channel. The weaknesses of the HTML5 postMessage channel can be mitigated by implementing an optimization as discussed further down in Section 5.4.

A residual risk is the session identifier that is stored in the HTML5 session storage of the browser. The maintenance as well as the protection of the *session identifier* is out of scope of this work. However, by using the privilege elevation as described in the *Application Permission System Concept* from Section 4.6 even the session identifier storage risk can be mitigated. Therefore the residual risk of the web attack can be considered as *low*.

5.3.3 Scenario 3 – Communication Attack

The *communication attack* scenario simulates an attack on the network communication between *Skytrust Client Browser Environment (SCBE) web page* and the *Skytrust Server Environment (SSE)*. The attacker attempts to intercept the communication to perform a *man-in-the-middle (MITM)* attack. Thereby, the attacker is tries to obtain credentials to impersonate the user.

Threats: The following threats on the *Skytrust* system occur from the communication attack:

- **(T): Obtain Credentials, Data and Keys**
  By intercepting the communication, the attacker tries to obtain:
  - *credentials*
  - *plain / encrypted data*
  - *temporary AES keys*

  If the attacker has access to the authentication credentials the attacker is capable to perform cryptographic operations with the core asset, the *primary keys*. also the access to temporary AES key can be used to gain access to the plain data.

Countermeasures: The Skytrust system provides the following countermeasures to mitigate the above mentioned threats:

- **(C): Secure Communication**
  The Skytrust system utilizes well known and security proved protocols such as HTTPS for communication between the Skytrust Server Environment and the Skytrust Client Browser Environment web page. This ensures a high level of security and prevent a possible attack on the communication between *SCBE web page* and *Skytrust Server Environment*.
- **(C): Strong Authentication Scheme**
  The Skytrust system support various authentication schemes to authenticate the user. Which authentication scheme is used depends on the appropriate deployment scenario and the surrounding environment. In case of current scenario the OAuth authentication scheme is used. Furthermore, the authentication component of the *SCBE web page* maintains the appropriate token that is issued from the authorization authority. The web application never gains access to this token. However, there is a possibility to obtain the access token and perform cryptographic operations with keys that are related to the appropriate user. To mitigate this issue we emphasize, to use a short key usage time, as defined in the *Application Permission System Concept* from Section 4.6, and an authentication scheme with two factor authentication to ban the attack and reduce the impact.

Residual Risk – Low: Thereby, that the communication between *SCBE web page* and *SSE* is secured with HTTPS the residual risk of the communication attack relies on the security of HTTPS.
HTTPS uses the Transport Layer Security (TLS) protocol to secure the communication between server and client. However, several HTTPS implementations do not require a client certificate in the handshake protocol. Therefore only the server is authenticated to the client. In order to ensure security and to prevent *man-in-the-middle* attacks the server certificate has to be verified. Nevertheless, the residual risk of the communication is considered to be *low*.

5.3.4 Scenario 4 – Server attack

The *server attack* simulates an attack on the *Skytrust Server Environment (SSE)*. An arbitrary attacker uses vulnerabilities in the server implementation to gain access to the server environment. Thereby, the attacker is able to gain unauthorized access to the credentials of the users, the attacker is able to impersonate the users. Furthermore, the attacker is able to access the key storage solution of the server environment. The *server attack* simulates an attack on the provided interface. By enforcing phishing attacks the attacker tries to gain credentials.

Threats: The following threats on the *Skytrust* system occur from the server attack:

- **(T): Obtain Credentials**
  Due to the access to the server environment and spying out the Skytrust protocol the attacker is able gain unauthorized access to:
  - *credentials*
  - *temporary AES keys*
- **(T): Manipulate Skytrust Element**
  An attacker who is able to gain access to the server is able to manipulate the Skytrust Element. Thereby, the attacker is able to gain access to data.
- **(T): Obtain or Manipulate Primary Keys**
  An attacker who has access to the server also attempts to gain access to the secure storage device to retrieve or manipulate the core asset, the *primary keys*.
- **(T): Interface Attack**
  An attacker attempts to use vulnerabilities in the interface implementation to gain unauthorized access to the server or the Skytrust Element. Thereby, the attacker is able to gain access to credentials or temporary AES keys.
- **(T): Phishing Attack**
  An attacker may establish a false SCBE web page in order to execute phishing attacks.

Countermeasures: The Skytrust system provides the following countermeasures to mitigate the threats:

- **(C): Trusted Server Implementation**
  As well as for the Skytrust Element we recommend to use best practices and common technologies for the implementation of the Skytrust Server Environment. Especially for server solutions precautions against common attack such as SQL injection, Input Validation, Session Hijacking should be provided. In case of different domains of the SSE and the SCBE web page an implementation against CORS attacks should be provided.
  Moreover, also the internal execution and hypervisor access should be taken into account. Several authors have proposed solutions for ensuring security such as Kamara and Lauter, Lombardi and Pietro [33, 37].
- **(C): Secure Storage**
  The Skytrust system provide a broad range of secure key storage solutions to store cryptographic keys such as a *Hardware Security Module (HSM)*, a *smart card*, a cryptographic service provider or other cryptographic tokens that provide a high level of security. Which secure key storage solution is selected depends on the particular deployment scenario as well as on a conducted security analysis of the appropriate environment. Each of the above mentioned key storage solution prohibit the manipulation of the stored key material so that the attacker is not able to obtain the keys. However, if the attacker is in possession of the credentials that are necessary to utilize the key the attack cannot be prevented.
- **(C): Environment Identification**
  The *Skytrust Server Environment* and the *Skytrust Client Browser Environment web page* share a unique secret, which is used to on every request that is sent to the server. Furthermore the Skytrust Server Environment provides a *CORS* implementation that allows only known origins access to the server.

Residual Risk – Medium: The server attack has a strong impact on the Skytrust system, if the attacker is able to gain access to the server. By obtaining the credentials of the user the attacker is able to perform cryptographic operations in behalf of the user. Therefore, the residual risk is considered to be *medium*.

However, if the server environment is up to date and an appropriate implementation is available, the unauthorized access to the server and therefore to the *Skytrust* should be prevented.

5.3.5 Scenario 5 – Operator attack

The *operator attack* simulates an attack on the *Skytrust Element* and the secure storage solution by the operator of the server environment. In considering of this attack two sub attacks have to be distinguished:

- **Sub attack 1:** The first sub attack simulates that the secure storage solution, for example, an HSM is provided from a third party provider.
- **Sub attack 2:** The second sub attack simulates that the server provider is also the provider of the secure storage solution.

Threats: The following threats came up by taking into account the operator attack scenario. Threats and countermeasures that are related to the first sub attack scenario are marked with (1) and the second sub attack scenario with (2). If the threats and countermeasures are valid for both scenarios this is marked with (1), (2).

- **(T) (2): Obtain or Manipulate *Primary Keys***
  The attacker is in possession of the credentials to configure the secure storage solution. Thereby, the attacker is able to retrieve and to manipulate the primary keys of the user.
- **(T) (1)(2): Manipulating the Skytrust Element**
  Due the access to the server environment the attacker is able gain access to:
  - *credentials*
  - *temporary AES keys*

  Thereby, the attacker is able to modify the cryptographic operations by manipulating the *Skytrust Element*.

Countermeasures: The Skytrust system provides the following countermeasures to mitigate the threats.

- **(C) (1)(2): Secure Storage**
  The Skytrust system provide a broad range of secure key storage solutions to store cryptographic keys such as a *Hardware Security Module (HSM)*, a *smart card*, a cryptographic service provider or other cryptographic tokens that provide a high level of security. However, if the attacker is in possession of the credentials to manage the secure storage solution, the keys of the users can be manipulated and extracted by the attacker.
- **Trusted Server Implementation**
  The trusted server implementation countermeasure is already discussed in Section 5·3·4·

Residual Risk – Medium/High: The operator attack has the strongest impact on the Skytrust system. Depending on the sub attack scenario the residual risk has to be distinguished.

- **Sub attack 1: Residual Risk – Medium**
  Thereby, that the secure storage solution is maintained by a third party provider the attacker is not able to manipulate or retrieve the primary keys. Therefore, the core asset is secured and the residual risk is *medium*.
- **Sub attack 2: Residual Risk – High**
  If the attacker is in possession of the credentials to maintain the secure storage solution, the extraction and manipulation of the core asset, the primary keys, cannot be prevented. Therefore, the residual risk is *high*.

An approach to mitigate the residual risk of the *operator attack* is to deploy the *Skytrust Server Environment (SSE)* in a *Hardware Security Module*. If the Skytrust Server Environment is deployed in an HSM the access to the environment is restricted to the *REST API*. By utilizing an HTTPS connection direct to this environment any other access point or any manipulation of the Skytrust Element is prohibited. Only the operator who has set up and configured the HSM is able to change the configuration and the maintained keys. If the server operator is not in possession of the credentials to configure the HSM the access to the SSE is secured. This solution is an approach for a future research topic.

5.4 Securing HTML5 Communication

As already mentioned, in Section 2.3, the HTML5 communication is vulnerable against attacks that are enforced over the communication channel. In this section a possible security improvement is presented.

The main issue of the communication between frames is the validation of the origin. The origin check at the *Skytrust Client Browser Environment (SCBE)* web page can be enhanced in combination with the *Application Permission System*. The application permission system allows the specification of an origin value for an API key that is utilized by a web application. This origin value can be used to validate the API key of the web application which is sent in the Skytrust transport protocol over the HTML5 postMessage channel. By calling a service at the Skytrust Server Environment the origin can be verified. If the verification is successful the transport protocol packet is processed otherwise not. An overview of this enhanced origin check is illustrated in Figure 5.4.

In the future that enhancement is may be applicable in connection with third party SCBE web pages that wants to utilize the *Skytrust Server Environment*. By using the same approach for the *Cross Origin Resource Sharing (CORS)* white listing an improvement can be achieved. This might be also a future research topic for securing the Skytrust system.

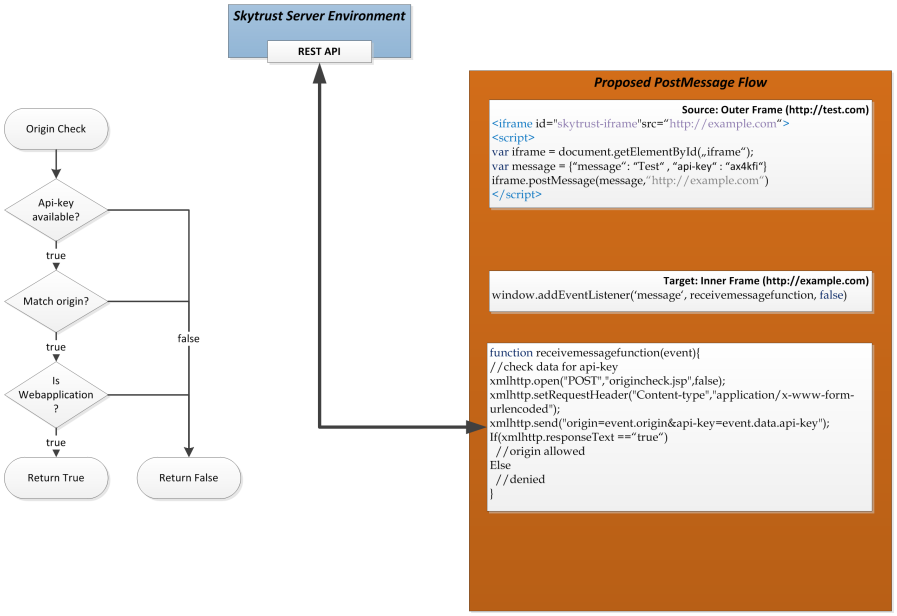

Figure 5.4: Enhanced origin check

5.5 Summary

The security analysis of the *Skytrust* system shows that the presented infrastructure provides a high level of security. The security goal of the Skytrust system to protect the cryptographic keys is ensured. Only if an attacker is in possession of the credentials to maintain the secure storage solution the primary keys can be manipulated. However, by following best practices for the implementation of the *Skytrust Client Browser Environment (SCBE) web page* as well as of the *Skytrust Server Environment (SSE)* even the malicious usage of the core asset, the primary keys can be prevented. Furthermore, even the origin check of the HTML5 web communication can be enhanced with the proposed *Application Permission System*. Moreover, the proposed privilege elevation mitigates the session identifier vulnerability. An interesting approach for future research is the deployment of the *Skytrust Server Environment* in an HSM. Thereby even the operator attack can be mitigated.

Chapter 6

Conclusion and Outlook

The goal of this thesis was to design and implement a prototype of the *Skytrust* system and to conduct a security analysis of the *Skytrust* system in a web application scenario. The major motivation to develop the Skytrust System is based on the limitation of the current available solutions such as the *Amazon Cloud HSM*, the *Austrian Mobile Phone Signature*, *SigningHub*, *Cryptomatic*, and *Dictao* to authentication and signature creation. The comparison of the existing solutions has shown that only the *Amazon Cloud HSM* is able to perform cipher operations. Nevertheless, the system has a drawback due to the restriction to a particular client. This insufficient situation motivated the development of *Skytrust* system.

However, the *Skytrust* system achieves its flexibility, extensibility and platform independence by splitting up the common access of a cryptographic service into an acting entity called *actor* and receiving entity called *receiver*. This division associated with the presented transport protocol meet the required specifications.

We demonstrated the functionality of the *Skytrust* system by implementing a prototype that simulates a cloud-based cryptographic key storage solution that can be integrated into an existing web application for data encryption. Thereby, that the asymmetric keys of the user are stored in the central cloud-based cryptographic storage, the secure key storage limitations of the browser can be eliminated.

In order to manage the access to such a cloud-based key storage solution we proposed an *Application Permission System* that ensures a precise restriction to the *Skytrust* system. Through combining key permissions and application permissions an accurate access control can be managed. The influence factors for the key permissions and the application permission were extracted by an analysis of the *Skytrust* system requirements.

Since the *Skytrust* system operates with sensitive cryptographic key material, we conducted a comprehensive security analysis based on the prototype and a particular scenario. The analysis has shown that the *Skytrust* system provides a high level of security. However, the security of the *Skytrust* system is based on the utilized technologies such as HSMs to store the cryptographic keys or HTTPS to protect the connection.

The accurate usage of these technologies enables the high security level of the *Skytrust* system.

However, the broad range of possible deployment scenarios from a single *Skytrust Element* that connects a smart card to complex configurations of several *Skytrust Elements* on various domains enables several future research topics. Moreover, the proposed deployment of the *Skytrust Server Environment* in an HSM can even be a topic for future research.

The conclusion of this work is, that the presented approach for providing a flexible, extensible, and platform independent cloud-based key storage solution that supports the execution of cryptographic operations, works as proposed and therefore presents a possible new participant in the field of cloud-based key storage solutions.

Bibliography

[1] Ross Anderson et al. *Cryptographic processors – a survey*. Tech. rep. UCAM-CL-TR-641. University of Cambridge, Computer Laboratory, Aug. 2005. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-641.pdf (cit. on p. 30).

[2] David Barrera et al. "A Methodology for Empirical Analysis of Permission-based Security Models and Its Application to Android". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. Chicago, Illinois, USA: ACM, 2010, pp. 73–84. ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866317. URL: http://doi.acm.org/10.1145/1866307.1866317 (cit. on p. 55).

[3] A. Barth. *The Web Origin Concept*. RFC 6454 (Proposed Standard). Internet Engineering Task Force, Dec. 2011. URL: http://www.ietf.org/rfc/rfc6454.txt (cit. on p. 19).

[4] Adam Barth, Collin Jackson, and ohn C. Mitchell. "Securing Frame Communication in Browsers". In: *Commun. ACM* 52.6 (June 2009), pp. 83–91. ISSN: 0001-0782. DOI: 10.1145/1516046.1516066. URL: http://doi.acm.org/10.1145/1516046.1516066 (cit. on p. 19).

[5] Jenkins B.D. *Security Risk Analysis and Management*. white paper. 1998 (cit. on p. 65).

[6] George Coulouris et al. *Distributed Systems: Concepts and Design*. 5th. USA: Addison-Wesley Publishing Company, 2011. ISBN: 0132143011, 9780132143011 (cit. on p. 15).

[7] D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 4627 (Informational). Internet Engineering Task Force, July 2006. URL: http://www.ietf.org/rfc/rfc4627.txt (cit. on p. 14).

[8] Juan Carlos Cruellas et al. *XML Advanced Electronic Signatures (XAdES)*. W3C Note. http://www.w3.org/TR/XAdES/. W3C, Feb. 2003 (cit. on p. 33).

[9] *Cryptography Reference*. Nov. 2013. URL: http://msdn.microsoft.com/en-us/library/aa380256.aspx (visited on 01/20/2014) (cit. on p. 32).

[10] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246 (Proposed Standard). Updated by RFCs 5746, 5878, 6176. Internet Engineering Task Force, Aug. 2008. URL: http://www.ietf.org/rfc/rfc5246.txt (cit. on p. 13).

[11] L. Dusseault and J. Snell. *PATCH Method for HTTP*. RFC 5789 (Proposed Standard). Internet Engineering Task Force, Mar. 2010. URL: http://www.ietf.org/rfc/rfc5789.txt (cit. on p. 16).

[12] Donald Eastlake et al. *XML Signature Syntax and Processing (Second Edition)*. W3C Recommendation. http://www.w3.org/TR/2008/REC-xmldsig-core-20080610/. W3C, June 2008 (cit. on p. 32).

[13] *ETSI TS 101 733 Electronic Signatures and Infrastructures (ESI); CMS Advanced Electronic Signatures (CAdES) v1.7.4.* Tech. rep. http://www.etsi.org. European Telecommunications Standards Institute ETSI, July 2008 (cit. on p. 33).

[14] *ETSI TS 102 778-3. Electronic Signatures and Infrastructures (ESI); PDF Advanced Electronic Signatures (PAdES); PAdES Enhanced - PadES-BES and PAdES-EPES Profiles*. Tech. rep. http://www.etsi.org. European Telecommunications Standards Institute ETSI, July 2009 (cit. on p. 33).

[15] European Union. *Directive 1999/93/EC of the European Parliament and of the Council of 13. December 1999 on a community framework for electronic signatures*. 1999 (cit. on pp. 2, 32).

[16] Roy Thomas Fielding. "Architectural styles and the design of network-based software architectures". AAI9980887. PhD thesis. University of California, 2000. ISBN: 0-599-87118-0 (cit. on p. 15).

[17] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616 (Draft Standard). Updated by RFCs 2817, 5785, 6266, 6585. Internet Engineering Task Force, June 1999. URL: http://www.ietf.org/rfc/rfc2616.txt (cit. on pp. 16, 48).

[18] OWASP Foundation. *HTML5 Security Cheat Sheet*. 2013. URL: https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet (visited on 10/16/2013) (cit. on pp. 74, 76).

[19] J. Franks et al. *HTTP Authentication: Basic and Digest Access Authentication*. RFC 2617 (Draft Standard). Internet Engineering Task Force, June 1999. URL: http://www.ietf.org/rfc/rfc2617.txt (cit. on p. 7).

[20] *Gartner Says Worldwide Public Cloud Services Market to Total £131 Billion*. Feb. 2013. URL: http://www.gartner.com/newsroom/id/2352816f (visited on 12/20/2013) (cit. on p. 1).

[21] Hugo Haas and Allen Brown. *Web Services Glossary*. W3C Note. http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/. W3C, Feb. 2004 (cit. on p. 14).

[22] Eran Hammer. *Explaining the OAuth Session Fixation Attack*. Apr. 2009. URL: http://hueniverse.com/2009/04/explaining-the-oauth-session-fixation-attack/ (visited on 10/21/2013) (cit. on p. 6).

[23] E. Hammer-Lahav. *The OAuth 1.0 Protocol*. RFC 5849 (Informational). Obsoleted by RFC 6749. Internet Engineering Task Force, Apr. 2010. URL: http://www.ietf.org/rfc/rfc5849.txt (cit. on p. 5).

[24] Steve Hanna et al. "The Emperors New APIs: On the (In)Secure Usage of New Client Side Primitives". In: *Proceedings of the 4th Web 2.0 Security and Privacy Workshop (W2SP)*. 2010 (cit. on p. 19).

[25] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749 (Proposed Standard). Internet Engineering Task Force, Oct. 2012. URL: http://www.ietf.org/rfc/rfc6749.txt (cit. on pp. 6, 11).

[26] Ian Hickson. *HTML5 Web Messaging*. Candidate Recommendation. http://www.w3.org/TR/2012/CR-webmessaging-20120501/. W3C, May 2012 (cit. on p. 17).

[27] Monsur Hossain. *Using CORS*. Oct. 2011. URL: http://www.html5rocks.com/en/tutorials/cors/ (visited on 08/19/2013) (cit. on p. 22).

[28] *Introducing JSON*. URL: http://www.json.org/ (visited on 08/22/2013) (cit. on p. 46).

[29] *Java Cryptography Architecture (JCA) Reference Guide*. URL: http://docs.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html (visited on 01/20/2014) (cit. on pp. 32, 43).

[30] J. Jonsson and B. Kaliski. *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1*. RFC 3447 (Informational). Internet Engineering Task Force, Feb. 2003. URL: http://www.ietf.org/rfc/rfc3447.txt (cit. on p. 33).

[31] S. Josefsson. *The Base16, Base32, and Base64 Data Encodings*. RFC 3548 (Informational). Obsoleted by RFC 4648. Internet Engineering Task Force, July 2003. URL: http://www.ietf.org/rfc/rfc3548.txt (cit. on pp. 14, 48).

[32] B. Kaliski. *PKCS #7: Cryptographic Message Syntax Version 1.5*. RFC 2315 (Informational). Internet Engineering Task Force, Mar. 1998. URL: http://www.ietf.org/rfc/rfc2315.txt (cit. on p. 33).

[33] Seny Kamara and Kristin Lauter. "Cryptographic cloud storage". In: *Proceedings of the 14th international conference on Financial cryptograpy and data security*. FC'10. Tenerife, Canary Islands, Spain: Springer-Verlag, 2010, pp. 136–149. ISBN: 3-642-14991-X, 978-3-642-14991-7. URL: http://dl.acm.org/citation.cfm?id=1894863.1894876 (cit. on p. 79).

[34] Anne van Kesteren. *Cross-Origin Resource Sharing*. Candidate Recommendation. http://www.w3.org/TR/2013/CR-cors-20130129/. W3C, Jan. 2013 (cit. on p. 22).

[35] Herbert Leitold, Arno Hollosi, and Reinhard Posch. "Security Architecture of the Austrian Citizen Card Concept". In: *ACSAC*. 2002, pp. 391–402 (cit. on pp. 1, 32).

[36] T. Lodderstedt, M. McGloin, and P. Hunt. *OAuth 2.0 Threat Model and Security Considerations*. RFC 6819 (Informational). Internet Engineering Task Force, Jan. 2013. URL: http://www.ietf.org/rfc/rfc6819.txt (cit. on p. 12).

[37] Flavio Lombardi and Roberto Di Pietro. "Secure virtualization for cloud computing". In: *Journal of Network and Computer Applications* 34.4 (2011). Advanced Topics in Cloud Computing, pp. 1113–1122. ISSN: 1084-8045. DOI: http://dx.doi.org/10.1016/j.jnca.2010.06.008. URL: http://www.sciencedirect.com/science/article/pii/S1084804510001062 (cit. on p. 79).

[38] J.D. Meier, Alex Mackman, and Blaine Wastell. *Threat Modeling Web Applications*. May 2005. URL: http://msdn.microsoft.com/en-us/library/ff648006.aspx (visited on 10/10/2013) (cit. on p. 65).

[39] Noah Mendelsohn et al. *SOAP Message Transmission Optimization Mechanism*. W3C Recommendation. http://www.w3.org/TR/2005/REC-soap12-mtom-20050125/. W3C, Jan. 2005 (cit. on p. 14).

[40] Noah Mendelsohn et al. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. W3C Recommendation. http://www.w3.org/TR/2007/REC-soap12-part1-20070427/. W3C, Apr. 2007 (cit. on p. 14).

[41] NIST. *Security Requirements for Cryptographic Modules (FIPS PUB 140-2)*. FIPS PUB. National Institute for Standards and Technology. Gaithersburg, MD 20899-8900, USA, May 25, 2001, pp. viii + 61. URL: http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf (cit. on p. 30).

[42] *OAuth Core 1.0*. Dec. 4, 2007. URL: http://oauth.net/core/1.0/ (visited on 07/08/2013) (cit. on p. 5).

[43] Clemens Orthacker, Martin Centner, and Christian Kittl. "Qualified Mobile Server Signature". In: *25th IFIP TC-11 International Information Security Conference, SEC 2010*. Vol. 330. AICT. Springer, 2010, pp. 103–111 (cit. on p. 33).

[44] Jean Paoli et al. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C Recommendation. http://www.w3.org/TR/2008/REC-xml-20081126/. W3C, Nov. 2008 (cit. on p. 14).

[45] *PKCS 11 v2.20: Cryptographic Token Interface Standard*. June 2004. URL: ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-20/pkcs-11v2-20.pdf (visited on 09/10/2013) (cit. on pp. 32, 43).

[46] B. Ramsdell and S. Turner. *Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification*. RFC 5751 (Proposed Standard). Internet Engineering Task Force, Jan. 2010. URL: http://www.ietf.org/rfc/rfc5751.txt (cit. on p. 33).

[47] Wolfgang Rankl and Wolfgang Effing. *Smart Card Handbook*. 4th. Wiley Publishing, 2010. ISBN: 0470743670, 9780470743676 (cit. on p. 29).

[48] S.A.M. Rizvi, Halima Sadia Rizvi, and Zaid Al-Baghdadi. "Smart Cards: The Future Gate". In: *Proceedings of The World Congress on Engineering and Computer Science 2010*. Vol. Vol. I. Oct. 2010, pp81–86. URL: http://www.iaeng.org/publication/WCECS2010/WCECS2010_pp81-86.pdf (visited on 10/22/2013) (cit. on p. 29).

[49] Jesse Ruderman. *Same Origin Policy for Javascript*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Same_origin_policy_for_JavaScript (visited on 08/19/2013) (cit. on p. 19).

[50] Philippe De Ryck et al. *A security analysis of emerging web standards - Extended version*. CW Reports CW622. partner: KUL; projects: WebSand, NESSoS; tier: NoTier. Department of Computer Science, KU Leuven, May 2012. URL: https://lirias.kuleuven.be/handle/123456789/349398 (cit. on pp. 24, 68).

[51] Hossein Saiedian and Dan S. Broyles. "Security Vulnerabilities in the Same-Origin Policy: Implications and Alternatives". In: *Computer* 44.9 (2011), pp. 29–36. ISSN: 0018-9162. DOI: http://doi.ieeecomputersociety.org/10.1109/MC.2011.226 (cit. on p. 21).

[52] Michael Schmidt. *HTML5 web security*. Dec. 2011. URL: http://media.hacking-lab.com/hlnews/HTML5_Web_Security_v1.0.pdf (cit. on p. 24).

[53] Shreeraj Shah. "HTML5 Top 10 Threats Stealth Attacks and Silent Exploits". In: *BlackHat Europe 2012*. Blackhat, 2012 (cit. on p. 24).

[54] Quinn Slack and Roy Frostig. *Murphi Analysis of of OAuth 2.0 Implicit Grant Flow*. URL: http://www.stanford.edu/class/cs259/WWW11/ (visited on 12/27/2013) (cit. on p. 13).

[55] Ryan Sleevi and David Dahl. *Web Cryptography API*. W3C Working Draft. http://www.w3.org/TR/2013/WD-WebCryptoAPI-20130625/. W3C, June 2013 (cit. on pp. 49, 65).

[56] D. K. Smetters and Nathan Good. "How Users Use Access Control". In: *Proceedings of the 5th Symposium on Usable Privacy and Security*. SOUPS '09. Mountain View, California: ACM, 2009, 15:1–15:12. ISBN: 978-1-60558-736-3. DOI: 10.1145/1572532.1572552. URL: http://doi.acm.org/10.1145/1572532.1572552 (cit. on p. 55).

[57] Sooel Son and Vitaly Shmatikov. "The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites". In: *NDSS*. The Internet Society, 2013 (cit. on p. 19).

[58] Jungkee Song, Hallvord Steen, and Julian Aubourg. *XMLHttpRequest*. W3C Working Draft. http://www.w3.org/TR/2012/WD-XMLHttpRequest-20121206/. W3C, Dec. 2012 (cit. on pp. 20, 24).

[59] Brandon Sterne and Adam Barth. *Content Security Policy 1.0*. Candidate Recommendation. http://www.w3.org/TR/2012/CR-CSP-20121115/. W3C, Nov. 2012 (cit. on p. 22).

[60] San-Tsai Sun and Konstantin Beznosov. "The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 378–390. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382238. URL: http://doi.acm.org/10.1145/2382196.2382238 (cit. on pp. 13, 68).

[61] K. Zyp and F. Galiegue, eds. *A JSON Media Type for Describing the Structure and Meaning of JSON Documents*. Jan. 2013. URL: http://tools.ietf.org/html/draft-zyp-json-schema-04 (visited on 08/22/2013) (cit. on p. 46).