Sandra Kreuzhuber, BSc

A Flexible Cross-Platform Framework for Integrating Multi-Factor Authentication

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieurin

Master's degree programme: Software Development and Business Management

submitted to

Graz University of Technology

Tutor

O.Univ.-Prof. Dipl.-Ing. Dr.techn. Reinhard Posch
Dipl.-Ing. Dr.techn. Peter Teufl

Institute for Applied Information Processing and Communications (IAIK)

Graz, November 2015

AFFIDAVIT

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

SIGNATURINFORMATIONEN	**Unterzeichner**	Sandra Julia Kreuzhuber
	Datum/Zeit-UTC	2015-11-30T10:32:58+01:00
	Prüfinformation	Informationen zur Prüfung der elektronischen Signatur finden Sie unter: https://www.signaturpruefung.gv.at
Hinweis	Dieses mit einer qualifizierten elektronischen Signatur versehene Dokument ist gemäß § 4 Abs. 1 Signaturgesetz einem handschriftlich unterschriebenen Dokument grundsätzlich rechtlich gleichgestellt.	

Date

Signature

Abstract

During the past years, users have progressively changed their habits towards accessing online services with their mobile devices. Often, these services require the user to authenticate. Numerous incidents and studies have shown that simple authentication schemes based on the username and password paradigm do not provide adequate protection. Therefore, many online services have started to deploy multi-factor authentication. The solutions currently available range from mechanisms, such as one-time passwords delivered via SMS to the user's mobile device, to the use of smart cards or USB tokens in addition to username and password. However, many of the deployed multi-factor authentication methods are not applicable to mobile use cases. For example, the use of smart cards or USB tokens is not feasible on smartphones. These usage limitations do not suit the mobile computing paradigm.

In this thesis, we propose a flexible authentication framework that supports the easy integration of multi-factor authentication methods. The proposed framework has been implemented as authentication app for mobile devices. The implementation features different authentication plugins that leverage features present on current mobile devices to establish a proper proof-of-possession. In order to be platform-independent, we make use of a popular cross-platform development framework. The Apache Cordova framework enables developers to access native mobile device features using web technologies, such as HTML and JavaScript.

To evaluate the security of our implementation, we adhere to the well-proven structure of the Common Criteria for Information Technology Security Evaluation. Until now, little work has been done on analysing the security of applications that have been developed using cross-platform development frameworks. Therefore, the conducted evaluation particularly focuses on security aspects resulting from the use of Apache Cordova. Although some gaps comparing to native mobile applications have been identified, we conclude that by following some security guidelines, a decent level of security can be achieved with cross-platform mobile applications. Our results show that developers especially need to protect their applications against security threats inherent to the nature of web application development, such as code injection and cross-site scripting.

Summarising, we have implemented an authentication app that features three different multi-factor authentication methods. The security of the authentication app has been thoroughly evaluated. Therefore, our security evaluation gives an in-depth analysis on the security of applications developed using the Apache Cordova framework. Furthermore, the structure of the evaluation is applicable to arbitrary multi-factor authentication schemes on mobile devices. The definition of threat agents, assets, threats and security objectives thus represents a valuable contribution for future work in this area.

Keywords. Cross-Platform Development, Mobile Security, Apache Cordova, Multi-Factor Authentication

Kurzfassung

Im Laufe der letzten Jahre ist die Relevanz mobiler Endbenutzergeräte stetig gestiegen. Viele Benutzerinnen und Benutzer verwenden nunmehr ihr Mobilgerät, um auf Online-Dienste zuzugreifen. Für den Zugriff auf E-Mail Konten, Soziale Medien, e-Banking oder e-Government-Anwendungen ist eine Authentifizierung der Benutzerin bzw. des Benutzers notwendig. Zahlreiche Studien und wissenschaftliche Arbeiten bemängeln jedoch die Sicherheit von Authentifizierungsmethoden basierend auf Benutzernamen und Passwort. Aus diesem Grund bieten einige Dienste mittlerweile Methoden zur Mehr-Faktor-Authentifizierung an. Aktuell verfügbare Lösungen reichen von Einmalpasswörtern, über SMS-TAN-Verfahren bis hin zur Verwendung von Chipkarten und USB-Token. Nicht alle Lösungen sind jedoch auf mobilen Endgeräten anwendbar. Chipkarten oder USB-Token beispielsweise, sind nicht für eine Benutzung mit Smartphones ausgelegt. Aus diesem Grund werden alternative Lösungen benötigt, um Mehr-Faktor-Authentifizierung auch auf mobilen Geräten sicherzustellen.

Diese Arbeit stellt ein flexibles Framework zur Mehr-Faktor-Authentifizierung vor. Ziel dieses Frameworks ist es, den leichten Austausch von Authentifizierungsmethoden zu ermöglichen. Das vorgeschlagene Framework wurde als mobile Applikation umgesetzt. Für die Serverkomponente wurde ein bestehendes Authentifizierungsframework verwendet. Die mobile Applikation nutzt unterschiedlichste Gerätefunktionen, um den Faktor Besitz im Zuge des Authentifizierungsvorganges sicherzustellen. Für die Implementierung der Authentifizierungs-App wurde Apache Cordova verwendet. Apache Cordova erlaubt die Entwicklung von Cross-Plattform-Applikationen unter Zuhilfenahme von Web-Technologien. Native Gerätefunktionen werden dabei über JavaScript-Schnittstellen zur Verfügung gestellt.

Mehr-Faktor-Authentifizierung wird zur Sicherung von sensiblen Daten und Funktionen verwendet. Daher ist es unabdingbar, diese Funktionen vor Angriffen zu schützen. Aus diesem Grund wurde eine ausführliche Sicherheitsanalyse der entwickelten Komponenten durchgeführt. Die Sicherheitsanalyse wurde basierend auf den Gemeinsamen Kriterien für die Prüfung und Bewertung von Systemen der Informationstechnik aufgebaut. Da bisher kaum Publikationen zur Sicherheit von mit Cross-Plattform-Frameworks entwickelten Applikationen vorliegen, wurde der Fokus dieser Sicherheitsanalyse auf Aspekte der Cross-Plattform-Entwicklung gelegt. Die Resultate unserer Sicherheitsanalyse zeigen, dass unter Berücksichtigung einiger Empfehlungen ein vernünftiges Maß an Sicherheit in Cross-Plattform-Applikationen erreicht werden kann.

Im Zuge dieser Arbeit wurde eine Authentifizierungs-App entwickelt und es wurden drei unterschiedliche Authentifizierungsmethoden integriert. Die durchgeführte Sicherheitsanalyse gibt einen weitreichenden Einblick in Sicherheitsaspekte von Cross-Plattform-Applikationen. Die Struktur der Sicherheitsanalyse ist jedoch nicht auf unsere Implementierung beschränkt. Die definierten Assets, mögliche Angreifer und Angriffsszenarien wie auch die abgeleiteten Sicherheitsziele, sind auf beliebige Methoden zur Mehr-Faktor-Authentifizierung auf mobilen Endgeräten anwendbar.

Schlüsselwörter. Mobile Cross-Plattform-Entwicklung, Mobilsicherheit, Apache Cordova, Mehr-Faktor-Authentifizierung

Contents

List of Figures

List of Tables

Listings

Acronyms

API application programming interface

APNS Apple Push Notification Service

ATM automated teller machine

CLI command-line interface

CSP Content Security Policy

ECB Electronic Codebook

EGIZ E-Government Innovation Center

eID electronic identity

FIDO Fast Identity Online

GCM Google Cloud Messaging

HMAC Hashed Message Authentication Code

HOTP HMAC-based One-Time Password Algorithm

IPT Identity Protection

NFC Near Field Communication

OTP one-time password

PRNG pseudo-random number generator

SDK software development kit

SMS Short Message Service

SSID Service Set Identifier

TAN Transaction Authentication Number

TOTP Time-Based One-Time Password Algorithm

TOE Target of Evaluation

UAF Universal Authentication Framework

U2F Universal Second Factor

URL Uniform Resource Locator

UUID Universally Unique Identifier

W3C World Wide Web Consortium

XMPP Extensible Messaging and Presence Protocol

Acknowledgements

First and foremost, I would like to thank my supervisor Peter Teufl, whose passion for mobile security had motivated me to deepen my knowledge in this area. His comments and encouragement helped me in all the time of research and writing this thesis.

Besides my supervisor, I want to thank my colleagues at the Institute for Applied Information Processing and Communications at Graz University of Technology for their valuable feedback, their effort in correcting draft versions of this thesis, and for all the fun we have had in the last three years. My thanks also go to Herbert Leitold, who provided me the opportunity to join the A-SIT group.

The practical part of this thesis has considerably benefited from the authentication server developed by EGIZ. Therefore, special thanks go to Andreas Fitzek for his support during the implementation of the server-side components.

Finally, I would like to thank my partner Thomas, my friends and family for the support they have been showing the last years.

Sandra Kreuzhuber
Graz, Austria, November 2015

Chapter 1

Introduction

The hack of Wired reporter Mat Honan's digital life in 2012, among others, brought attention to the relevance of using multi-factor authentication to protect important online accounts [35]. Several of his online accounts, including his accounts at Google, Amazon, Twitter and his AppleID were hacked because they were somehow connected. The fall of one online account helped the attacker to gain access to the other accounts as well. His online accounts were protected by means of simple username and password authentication. Thereby, e-mail accounts, typically, act as a mean to regain access to online accounts in case of lost passwords. The thus resulting entanglement of online accounts enables potential attackers to gain access to a broad range of the victim's digital life. To reduce the attack surface, several online services, such as e-banking providers or services like Google Mail launched authentication methods that require for additional factors to log in successfully. These authentication mechanisms vary from mechanisms, such as one-time passwords (OTPs) sent via Short Message Service (SMS) to the user's mobile device to the use of USB tokens in addition to username and password. However, providing multi-factor authentication methods that are applicable for a wide range of users is non-trivial.

Many tasks that have required the use of a desktop computer a few years ago can now be carried out on the go, by using mobile devices. A report from Facebook for the first quarter of 2015 shows that approximately 40 percent of monthly active users access their account only from their mobile devices [22]. These figures confirm the progressive shift from traditional desktop computers to mobile devices. This development results in users accessing confidential data and online services that have higher security requirements from their mobile device as well. Unfortunately, many available multi-factor authentication mechanisms are not applicable for the use with mobile devices.

Multi-factor authentication using mobile devices faces some substantial challenges. First and foremost, it has to be ensured that an attacker cannot clone the mobile device covering the second factor. Therefore, a strong binding between the user's account and her mobile device has to be employed. This binding can, for example, be realised by storing secret authentication data on the mobile device. Whilst some tokens, e.g. smart cards that store secret key material, offer tamper-resistant hardware security, mobile devices, in general, cannot be trusted in protecting data against unauthorised access. Some mobile devices already include hardware-backed storage mechanisms. However, additional means of protection are needed. The second line of defence might present push notifications. The mobile platform's push-notification services ensure that messages are only received by the legitimate application on a particular mobile device.

Second, it has to be guaranteed that an attacker cannot use a stolen device for authentication. This can be achieved by combining the possession proof on the mobile device with supplying additional knowledge. This way, encryption using a password-derived key might provide additional protection against unauthorised access. However, long and complicated passwords present a hassle for the user and thus do not suit mobile devices well.

Sophisticated authentication methods that are applicable on mobile devices require a high degree of development effort. The main challenge is that they have to be developed for different mobile platforms separately. The development of applications for multiple mobile platforms requires the use of different programming languages and build tools. Furthermore, it takes time to learn the respective platform's architectural concepts, design paradigms and the different security mechanisms.

To ease development for multiple mobile platforms, various cross-platform frameworks have emerged. These frameworks aim to minimise platform-specific modifications and favour code-reuse. However, little is known about security mechanisms provided by state-of-the-art cross-platform frameworks. Multi-factor authentication has high requirements regarding access to security mechanisms. The developer has to ensure secure storage of sensitive authentication data and communication between the mobile device and external services has to be protected adequately. These requirements qualify multi-factor authentication as an ideal use case for demonstrating access to security features and evaluating the security of cross-platform development techniques. Summarising, this thesis pursues two goals:

(a) the development of a *flexible framework enabling multi-factor authentication on mobile devices* using a state-of-the-art cross-platform framework and

(b) a thorough *security evaluation* of the implemented authentication framework.

In the following, we provide an overview of the implemented flexible multi-factor authentication framework and shortly introduce the conducted security evaluation.

1.1 Contribution

The purpose of this thesis is (a) to design and implement a flexible cross-platform framework for integrating multi-factor authentication on current mobile devices and (b) a thorough security evaluation of the implemented components. In the following, a short overview of our contribution is given.

We have developed a framework that provides multi-factor authentication on mobile devices whilst being developed using state-of-the-art cross-platform development techniques. The focus of our framework is on providing ways to easily integrate new authentication methods and, therefore, enable a fast adaptation to new technologies.

The implementation is divided into a server part and a client part that is executed on the mobile device. For the server part, this thesis extends a newly developed authentication framework of EGIZ[1]. The server-side framework provides the interface to service providers and the ability to configure suitable authentication mechanisms based on the domain of the service provider. Authentication methods can be integrated by developing authentication plugins that interact with the framework core using OpenID Connect[2]. For this thesis, several new authentication plugins have been implemented. In addition to performing the actual authentication process, authentication plugins take care of setting up the respective plugin for a new user, including the rollout of cryptographic keys and setting up the push-notification service for the user's mobile platform.

For the client part, common modules of different authentication mechanisms have been identified and act as a basis for the implementation of a flexible authentication module. The authentication module has been implemented as a mobile application. The authentication module implements a dynamic routeing mechanism to trigger the requested authentication plugin. Thus, new authentication methods can be easily integrated. The flexibility of the module has been demonstrated by integrating several new authentication methods.

To ensure an adequate binding between the user's account and a particular mobile device covering the second factor, we employ a combination of cryptographic material stored on the device, the respective

[1] https://www.egiz.gv.at/de/projekte/156-ALAP

[2] http://openid.net/connect/

platform's push-notification service and a user-chosen password. By combining multiple technologies and device features, decent security is achieved.

The mobile application has been implemented using the cross-platform framework Apache Cordova. Apache Cordova allows for the development of mobile applications using standard web technologies. Applications are rendered within so-called WebViews, browser windows that are embedded within native mobile applications. For accessing native device APIs that are not available within the mobile browser, Apache Cordova supports the development of plugins written in the programming language of the respective mobile platform. In contrast to the code running within WebViews, plugins have to be developed for each platform separately. In order to provide a protected storage for cryptographic keys used within authentication, we have implemented an Apache Cordova plugin that accesses the underlying platform's key-storage facilities.

After implementing the proposed multi-factor authentication framework, we were able to evaluate the feasibility of cross-platform development techniques when deployed in security-critical applications. Therefore, we analyse the security features of the underlying cross-platform framework Apache Cordova and inspect how the security features of the mobile operating system can be integrated into cross-platform applications. Furthermore, the integration of available data storage mechanisms and the respective mobile platform's push-notification service is analysed.

For the security evaluation, we rely on an approved methodology that follows a systematic approach. Therefore, we chose to roughly adhere to Common Criteria for Information Technology Security Evaluation. Common Criteria defines the main components of a security evaluation. Summarising, the security evaluation specifies the concrete system under evaluation, defines various assets and identifies threats, which potentially compromise the security of the defined assets. To evaluate the security of our implementation, countermeasures offered by the mobile client and provided by the different implemented authentication methods are listed.

The conducted security evaluation is tailored to the multi-factor authentication use case. However, the structure of the evaluation, the defined assets, threats and objectives are applicable to other solutions that provide multi-factor authentication as well and thus represent a valuable contribution to future work focusing on securing multi-factor authentication on mobile devices.

1.2 Outline

The thesis is structured as follows. Chapter 2 provides the reader with necessary background knowledge on developing cross-platform applications. Subsequently, Chapter 3 discusses existing methods for multi-factor authentication. This includes a short presentation of commercial solutions for authentication methods and some thoughts about security as well. Chapter 4 extracts building blocks found in typical authentication methods. Based on the extracted building blocks and features available on mobile devices, Chapter 5 introduces new authentication methods. After having presented the authentication methods, Chapter 6 presents the implemented authentication framework. Besides describing the architecture of the implemented modules, this thesis focuses on a thorough security analysis of the components running on the mobile device. Therefore, Chapter 7 systematically analyses the security of the implemented framework and the integrated authentication methods. For the analysis, we roughly adhere to the Common Criteria for Information Technology Security Evaluation. In addition to assessing the security of the integrated authentication methods, special focus is put on the security of Apache Cordova applications. Final conclusions are drawn in Chapter 8.

Chapter 2

Background

Developing mobile applications for multiple different platforms leads to increased effort, as applications have to be developed separately for each platform. To minimize the effort required to support multiple mobile platforms, several approaches for cross-platform development have emerged. This chapter provides a general introduction to cross-platform development. First, an overview of different approaches for creating applications for multiple mobile platforms is given. We continue with a more detailed presentation of the cross-platform framework *Apache Cordova*[1]. Apache Cordova has been used within the implementation of the practical part of this thesis. As this thesis implements a multi-factor authentication framework, we conclude with the basic principles of user authentication.

This chapter provides basic knowledge about cross-platform development and user authentication using multiple factors. In particular, the detailed introduction to Apache Cordova equips the reader with the knowledge required for Section 6 and 7 that focus on the implemented authentication framework and provide a security evaluation of the implemented components.

2.1 Cross-Platform Mobile Application Development

During the past years, a range of different mobile platforms have emerged. To name a few popular examples, in 2007 Apple has introduced iOS, Google Android followed in 2008 and Microsoft has launched Windows Phone 7 by the end of 2010. Each mobile platform allows application developers to develop mobile applications that can be installed from the device vendor's application store. As the platforms offer different frameworks and application programming interfaces (APIs) for application development, applications have to be developed separately for each platform. Consequently, this leads to increased effort for both, developing and maintaining multiple versions of the same product. Therefore, new approaches for developing mobile applications have emerged. The goal of these new application development technologies is to provide mechanisms to facilitate the reuse of application code between multiple platforms. The following section describes different approaches for cross-platform development and assesses advantages and disadvantages of each approach.

2.1.1 Approaches for Cross-Platform Mobile Application Development

Native applications enable full use of the underlying platform's API. Developers use native user interface widgets and thus provide the look and feel of the respective platform. Inherently, a set of native applications provide similar and mostly consistent experience to the user. However, building native mobile applications requires a separate development process for each mobile platform. Current mobile platforms require developers to use different development tools and programming languages. Furthermore, they

[1] `https://cordova.apache.org/`

propose diverging concepts regarding the architecture of a mobile application and differ in available security functions and their usage. Both developing and maintaining multiple versions of the same product is costly, therefore, technologies trying to achieve code portability between multiple mobile platforms have emerged. The main challenge for these so-called cross-platform frameworks is to provide the user with native user experience whilst trying to run on as many platforms as possible. In general, technologies for cross-platform mobile application development can be categorised into four approaches [79]: The development of

- *mobile web applications,*
- *hybrid applications,*
- *interpreted applications* and
- *generated applications.*

Mobile Web Applications

Mobile web applications can be accessed by loading a Uniform Resource Locator (URL) in the mobile browser. Mobile web applications are built with standard web technologies (HTML, CSS and JavaScript). Multiple libraries such as JQuery Mobile[2], Sencha Touch[3] and others try to provide graphical user interface designs suitable especially for mobile clients. Basically, mobile web applications represent conventional web pages that have been adapted to the needs of mobile users. Web applications can only use features provided by the mobile web browser. With the emergence of the HTML5 standard, new APIs for storing data, playing video files and many more have been introduced. However, there are still numerous device features that cannot be used by mobile web applications. Especially web applications cannot access the file system directly to browse files stored locally on the device. Equally, security features like the secure storage of credentials are not accessible within web applications.

Hybrid Applications

Hybrid applications represent a combination of mobile web applications and native applications. Hybrid applications are mainly built using web technologies such as HTML5, JavaScript and CSS and are executed in so-called WebViews. A WebView defines a browser window embedded within a native mobile application[4]. Thus, hybrid applications provide the business logic and user interface within a browser window but are packaged as native applications that can be installed from the official application store of the respective mobile platform vendor. The most popular framework for creating hybrid applications is Apache Cordova [76]. Apache Cordova allows to extend the mobile browser with so-called *plugins*. Plugins allow the access to a mobile platform's hardware features and data storage. Plugins are written in native code and provide JavaScript interfaces that can be called from within a WebView. The business logic, the user interface, the core Cordova library and plugin code are packaged as native application. Applications that have been packaged using the Cordova tools can be installed in the same way as native applications. To provide a platform's native look and feel, user interface widgets such as buttons and menus have to be built using CSS. For each mobile platform different CSS themes have to be included. User interface frameworks such as Ionic[5] and Sencha Touch build up on Apache Cordova and provide CSS themes for different mobile platforms. Both provide native-styled user interface widgets and thus enable the development of HTML5 applications that provide a similar look-and-feel as native applications. Section 2.1.2 discusses Apache Cordova and its plugin-based approach in more detail.

[2] `http://jquerymobile.com/`

[3] `http://www.sencha.com/products/touch/`

[4] In iOS it is called UIWebView, but for simplicity, we use WebView for both, the iOS and Android operating system throughout this thesis.

[5] `http://ionicframework.com/`

Interpreted Applications

Frameworks as for example Appcelerator Titanium[6] and Xamarin[7] enable the development of *interpreted applications*. Appcelerator Titanium allows the developer to write the application code in JavaScript. The JavaScript code is interpreted at runtime. Applications are packaged as native applications and can be provided via the platform vendor's application store. The goal of Titanium is to allow the development of cross-platform compatible code for the application's business logic, networking, database and event handling code.

Mobile web applications and hybrid applications look and behave like conventional websites, where the user interface is realised by using CSS. Titanium applications, on the other hand, use native user interface controls and thus provide a native look & feel. During execution, proxy objects for accessing native APIs and user interface controls are created. These proxy objects act as a bridge between the JavaScript and the native context. According to the official documentation of Appcelerator Titanium, the framework follows a 'write once, adapt everywhere' [9] approach. Therefore, developers have to maintain branches within the application code for the different platforms. Titanium currently supports the development of iOS, Android and Blackberry applications [10]. Summarizing, Titanium allows the development of native mobile applications without knowledge of the underlying platform or the programming language, as a JavaScript API exposes native APIs to the developer. On the downside, Titanium applications can only use APIs that have been integrated within the Titanium software development kit (SDK).

Xamarin is another popular solution for building cross-platform mobile applications. Xamarin applications are developed using the C# programming language. Xamarin uses the Mono C# compiler and the Mono Runtime, an open source implementation of Microsoft's .NET framework. Native device APIs are made available through bindings in Xamarin's API. Xamarin.Forms includes a large set of user interface controls that are mapped to native controls at runtime. Ideally, user interface code only has to be written once. For Android, Xamarin applications are packaged as .NET applications with an integrated Mono runtime. Applications are interpreted and run by Mono at runtime [78]. As iOS does not allow for runtime code generation, Xamarin applications are compiled to ARM assembly language ahead-of-time. Xamarin enables developers to reuse existing third-party libraries provided in the programming language of the respective platform.

Both, Titanium and Xamarin might lead to a technology lock-in as the SDK provides a whole new set of APIs for creating the user interface and accessing device features. In contrast, Apache Cordova applications build on standardized web technologies and can be run as web applications on a desktop computer as well.

Generated Applications

The idea of model-driven software development is to specify requirements in a model and generate application code from the provided model definition. iPhonical[8] has been developing a framework for creating native iOS applications using a model-driven approach. Later, they started developing a code generator for Appcelerator Titanium which supports the creation of CRUD (Create, Report, Update, Delete) applications from models which have been defined using a Domain Specific Language. MD2 is an automatic code generation tool developed by the University of Münster [32]. Their goal is to build native applications from a model description, whereas the model can be created without considering the underlying platform. They enable the automatic creation of data models, user interfaces, event handling code and can even access device features such as GPS. Their framework has been implemented as prototype and supports the creation of native iOS and Android applications.

[6] `http://www.appcelerator.com/titanium/`

[7] `http://xamarin.com/`

[8] `https://code.google.com/p/iphonical/`

Automatic code generation frameworks are still in early stages as they are lacking access to sophisticated device APIs and wide platform support. Currently, they can be used for rapid prototyping and as extension to other cross-platform technologies.

2.1.2 Apache Cordova

This section provides an introduction to Apache Cordova. Apache Cordova has been used for the implementation of the proposed multi-factor authentication framework. We chose Apache Cordova as it allows access to all underlying device APIs and therefore, provides great flexibility with regard to the use of the mobile devices' security features.

As more and more developers continue choosing cross-platform development techniques, attack potential is increased. Thus, it is of particular interest to inspect security properties of these frameworks. Apache Cordova is the most popular cross-platform development framework [76], which qualifies Apache Cordova perfectly for a thorough security evaluation.

In the following, the anatomy of an Apache Cordova application is discussed and Cordova's plugin approach is presented. The security evaluation in Section 7 builds up on the knowledge provided in this section.

History

Cordova, formerly called Phonegap, has been started around 2009 by the Canadian company Nitobi [14][15]. In 2011, Adobe acquired Nitobi and thereby the rights on the brand Phonegap. The code base, however, has been donated to the Apache Software Foundation and since has been available under the Apache License 2.0. The new name *Apache Cordova* and the fact that Adobe continued maintaining Phonegap has created wide confusion in the developer community. Nowadays, the term Phonegap describes Cordova plus Adobe's ecosystem around Cordova. One example is Adobe's cloud service *Phonegap Build* that allows developers to compile their Cordova applications in the cloud and thus to eliminate the need for installing MacOS when developing an iOS application and installing Windows when developing for Windows Phone. Summarizing, Apache Cordova is the core technology that Phonegap builds up on. Cordova is open source and can be used for free, whereas the tools around Phonegap are commercial.

Overview

Apache Cordova is a platform for developing native mobile applications using web technologies such as HTML, JavaScript and CSS. HTML, JavaScript and CSS files are packaged as resource files within a native mobile application [18]. Basically, Apache Cordova applications are implemented as a web page. Each application contains a file named *index.html* that acts as an entry point to the web application and references JavaScript, CSS and other resource files. As illustrated in Figure 2.1, Apache Cordova bundles web application files as data within a native mobile application. The packaging process has to be executed separately for supporting different platforms. The packaged applications can then be hosted on the platform-specific application store and installed the same way as conventional native mobile applications. Besides the popular mobile platforms iOS, Android and Windows Phone, Apache Cordova currently supports Blackberry 10, Amazon FireOS, FirefoxOS and Tizen, as well as Desktop operating systems like Windows (starting from version 8) and Ubuntu.

During execution of Cordova applications, the application files are rendered within a WebView. A WebView is a browser window that is embedded within a native mobile application. In contrast to a mobile browser, WebViews do not display a location bar and can be used to load HTML files from the file system. The application's user interface is entirely made of HTML, CSS and JavaScript that is being rendered within the WebView and by default does not provide a native look and feel.

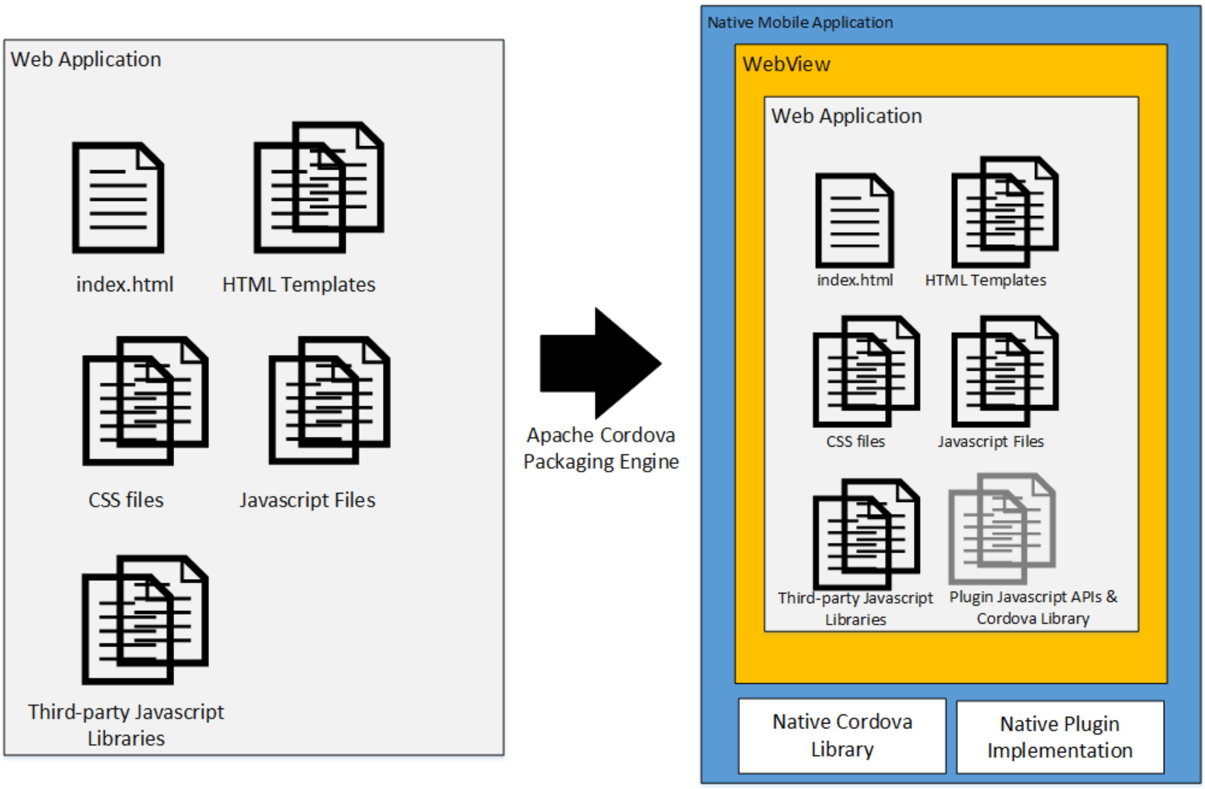

Web Application
index.html
HTML Templates
CSS files
Javascript Files
Third-party Javascript Libraries
Apache Cordova Packaging Engine
Native Mobile Application
WebView
Web Application
index.html
HTML Templates
CSS files
Javascript Files
Third-party Javascript Libraries
Plugin Javascript APIs & Cordova Library
Native Cordova Library
Native Plugin Implementation

Figure 2.1: Apache Cordova bundles web application files, such as HTML files, JavaScript, CSS and third-party libraries as data within a native mobile application. The web application is rendered in a WebView within the application.

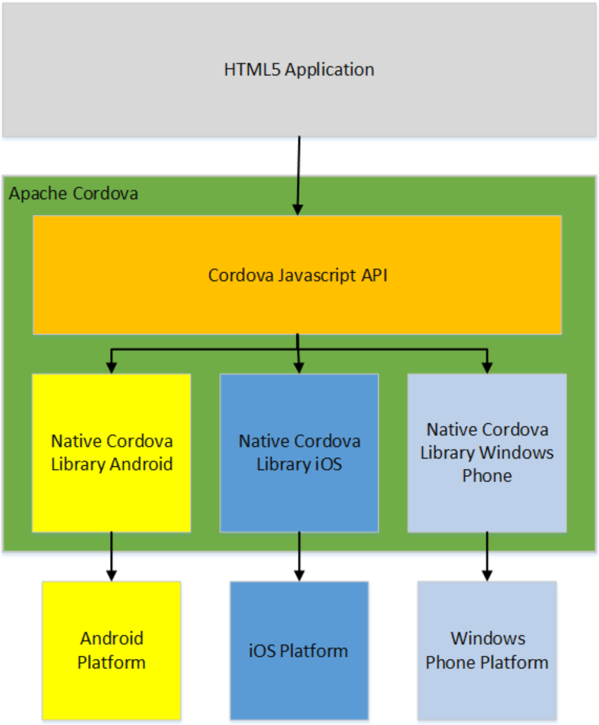


Figure 2.2: JavaScript code running inside the WebView can invoke functionality on the native side using the JavaScript API provided by Cordova. The JavaScript API acts as a bridge to the native Cordova library that provides a mapping to native platform functions.

In addition to loading the HTML files packaged within the application into the WebView, Apache Cordova allows developers to use features that are originally not available within a WebView. Therefore, JavaScript code running inside the WebView can invoke functionality on the native side using a JavaScript API provided by Cordova. This JavaScript API provides a bridge to the native Cordova library that invokes platform functions. Figure 2.2 illustrates the general architecture of Apache Cordova.

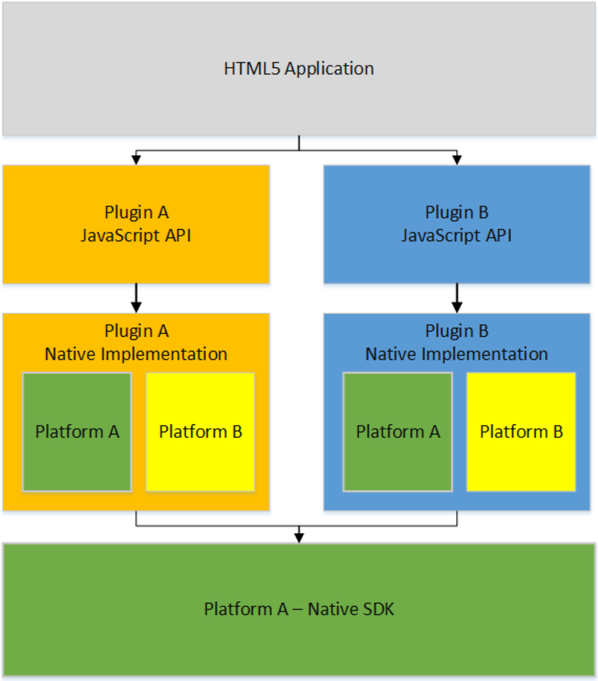


Figure 2.3: Apache Cordova uses so-called plugins for accessing device features that are not accessible from within the WebView. Plugins have to be implemented for each supported platform separately.

Functions that are not available from within a WebView can be added to the Apache Cordova framework as so-called *plugins*. Figure 2.3 illustrates the plugin concept. Cordova provides several plugins from stock, for example for accessing the logging console, camera, file system, battery status and many more. In addition, various third-party plugins[9] are available and developers can implement their own native plugins.

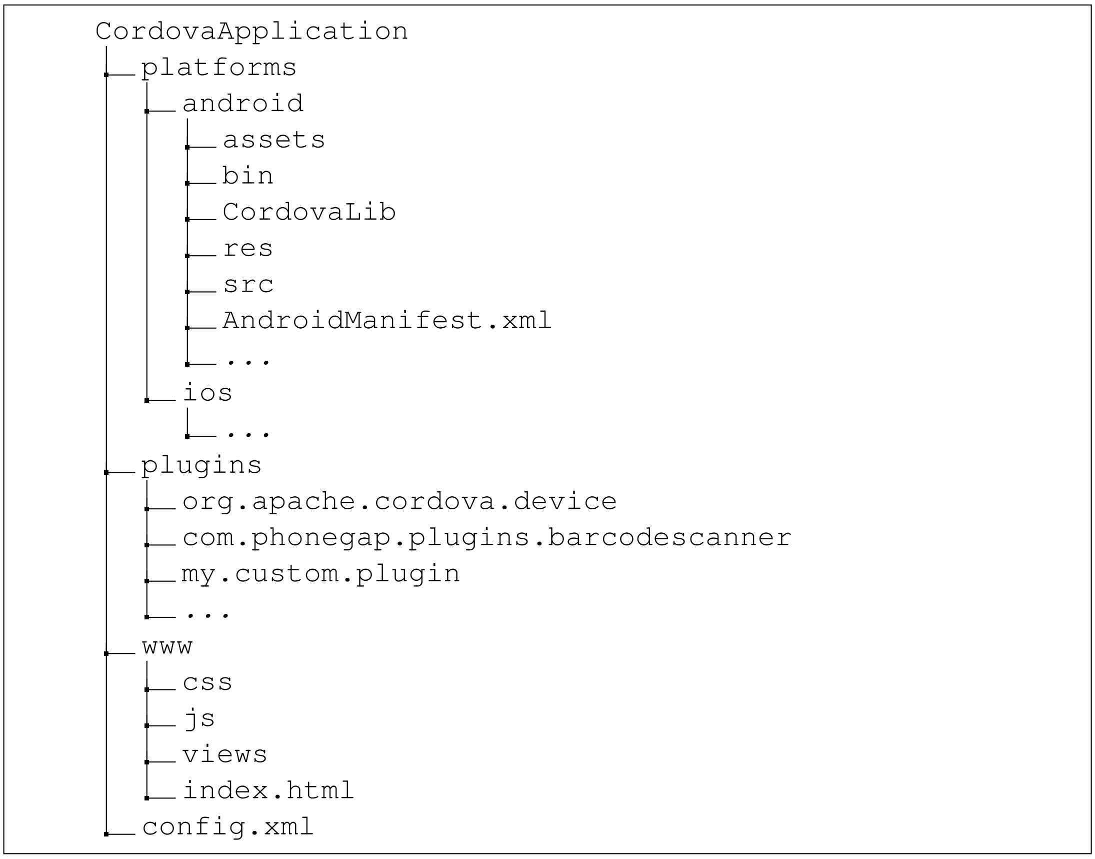
Anatomy of an Apache Cordova Application

Mobile applications using Apache Cordova are web applications packaged within native mobile applications. Applications need to follow a specified structure. Thus Figure 2.4 provides an illustration of the components within an Apache Cordova project.

Each project has to provide a *config.xml* file that provides general information about the application. This file includes the name of the application, information about the author, a short description and the path to *index.hml* file.

The *www* folder represents the core of the project and includes all necessary HTML, JavaScript and CSS files of the application. If the application should be usable whilst being offline as well, external libraries and resources have to be downloaded and included in that folder. The *www* folder also contains the *index.html* file, which acts as a starting point for the web application that should be rendered within the WebView. Most importantly, the *index.html* file specifies the required dependencies to JavaScript li-

[9] http://plugins.cordova.io/



```
CordovaApplication
 ├─platforms
 │  ├─android
 │  │  ├─assets
 │  │  ├─bin
 │  │  ├─CordovaLib
 │  │  ├─res
 │  │  ├─src
 │  │  ├─AndroidManifest.xml
 │  │  └─...
 │  └─ios
 │     └─...
 ├─plugins
 │  ├─org.apache.cordova.device
 │  ├─com.phonegap.plugins.barcodescanner
 │  ├─my.custom.plugin
 │  └─...
 ├─www
 │  ├─css
 │  ├─js
 │  ├─views
 │  └─index.html
 └─config.xml
```

Figure 2.4: The anatomy of an Apache Cordova project.

braries and includes the Cordova JavaScript API that makes Cordova features available to the application developer.

A newly generated Apache Cordova project consists of empty *platforms* and *plugins* folders. When support for a specific platform should be enabled, the respective platform has to be added using the Cordova command-line interface (CLI). Plugins can be added using the Cordova CLI as well. The *platforms* folder includes the native Cordova library and the application's native entry point. On the Android platform the native class *CordovaApp.java* represents the native entry point of the application. Listing 2.1 comprises the Android-specific code within *CordovaApp.java*. This class creates a WebView component and loads the *index.html* file that acts as an entry point for the cross-platform web application code.

Equal to native applications, Cordova applications have to contain application configuration files required for the specific platform. In case of Android, the *platforms* folder contains the *AndroidManifest.xml* file. Cordova applications have to define the required application permissions that control access to device specific features within the manifest file. However, the developer does not have to manually edit these platform-specific configuration files but can do so by providing the required settings in configuration files provided by Cordova. The Cordova CLI then processes these settings and injects the relevant sections into the platform-specific configuration files. The *plugins* folder includes platform-specific native code, possible native libraries and the platform-independent JavaScript code that acts as the interface to the plugin.

The next section will explain the development of custom Apache Cordova plugins in detail and will examine how they are included within the mobile application.

```
import android.os.Bundle;
import org.apache.cordova.*;

public class CordovaApp extends CordovaActivity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        super.init();
        // Set by <content src="index.html" /> in config.xml
        loadUrl(launchUrl);
    }
}
```

Listing 2.1: On the Android platform the native class *CordovaApp.java* represents the entry point of the mobile application. It creates the WebView and loads the starting HTML file.

Development of Native Cordova Plugins

Cordova applications are rendered within a WebView that has the same capabilities as the mobile browser. To expose native device APIs to the WebView, Apache Cordova has introduced the plugin concept. A Cordova plugin is a package of code that allows the WebView to communicate with the native platform. Plugins thus act as a bridge to the underlying platform and provide access to device features that are not available to web applications. Since version 3.0, the core APIs of Apache Cordova are provided as plugins[10]. Among other functionality, these provided plugins comprise access to the mobile camera, the file system, battery status and more. As these features have been exposed as plugins, they can be updated easily without having to update the Cordova core libraries entirely. In addition to Apache Cordova core plugins and a multitude of third-party plugins, any developer familiar with native mobile application development can implement custom plugins for Apache Cordova.

```
cordova.exec(
           function(param) {},
           function(error) {},
           "serviceName",
           "actionName",
           ["arg1", "arg2", 3]);
```

Listing 2.2: A call to *cordova.exec()* invokes the native part of the Apache Cordova plugin. In addition to the parameters *serviceName* and *actionName* that define a method within the plugin, the application has to pass a success and error handler that will be called after the plugin has finished execution.

Each plugin consists of a JavaScript interface along with native code for each supported platform. The JavaScript interface should remain the same for each platform and thus hide the platform-specific aspects from the web application. The native code of a plugin has full access to the underlying native SDK. Therefore, applications using Apache Cordova essentially do not lack any capabilities compared to native mobile applications. A plugin consists of the following components:

- *The JavaScript Interface*
  The JavaScript interface represents the most important part of the application as it exposes the

[10] http://cordova.apache.org/docs/en/4.0.0/guide_cli_index.md.html

plugin's methods to the web application. For communicating with the native part of the plugin, the `cordova.exec()` method has to be called from within the JavaScript interface. This method acts as a bridge between the JavaScript interface and the respective native implementation. It allows for passing parameters from the JavaScript interface to the native implementation and back to the web application again. Listing 2.2 provides the method signature of the `cordova.exec()` method. Several parameters are passed to this method: each call includes a *serviceName* and an *actionName*, two callback functions and an array of parameters that should be passed along to the plugin. The *serviceName* is used to identify the plugin that is being called. The *actionName* can be used to identify a special feature or method of the plugin and is used as defined by the developer. All calls to Apache Cordova plugins are asynchronous. Therefore, the two callback functions comprise a success handler and an error handler. Either of them is called after the plugin finishes execution. To use the plugin within the web application, the developer has to include the JavaScript file with the application and then the developer can call the plugin in the same way as any other JavaScript function.

- *The Native Interface*
  Each JavaScript interface associates with one or more native implementations. Depending on the underlying platform, the implementation differs. On Android, the plugin has to extend the class *org.apache.cordova.CordovaPlugin* and on iOS the class *CDVPlugin* has to be extended. These base classes provide capabilities to register listeners, allow for background processing and they can access the native APIs the same way as any native mobile application. After the plugin finishes execution, the success callback function is invoked and parameters can be passed back to the JavaScript side. The callback methods can be invoked multiple times, for example for implementing event listeners and thus notify the web application about events received in the background. For instance, push notifications are realised as a plugin. Whenever a new message is received, the native side of the plugin invokes the callback function and hands over the received message to the JavaScript side of the plugin.

- *plugin.xml*
  Each plugin features a *plugin.xml* file located in the top-level of the respective plugin's folder. This file defines the structure of a plugin and includes a list of files required for the successful execution of the plugin. For each platform, a <*feature*> tag including the name of the plugin and the starting class of the plugin has to be added. This <*feature*> tag is injected in the *config.xml* file that is located in each platform-specific folder within the *platforms* folder. Based on the feature mappings provided by the *config.xml* file, calls from the JavaScript interface can be mapped to the native plugin implementation. Additionally, the *plugin.xml* file has to include all class files and dependencies that should be included in the plugin. Based on the provided listing, these files will be copied to the application that wants to include the plugin and placed into the correct platform-specific folder within the *platforms* folder. The process of copying the required files is done automatically by the Cordova CLI when executing the command to add a new plugin.

Summarising, Apache Cordova applications are web applications that are packaged as native mobile applications. Cordova applications are rendered within a WebView. By adding plugins to an application, the WebView can communicate with the underlying native platform. A Cordova plugin is a package of code that is injected into the application that is adding the plugin. A plugin consists of a JavaScript interface that remains the same for each platform and a set of native interfaces that have to be implemented by using the native SDKs and programming languages of the supported platforms. Thus, plugins provide access to device and platform functionality that is ordinarily unavailable to web applications. All the main Cordova API features are implemented as plugins. In addition, developers can implement their own plugins. Third-party plugins can be published to the Cordova Plugin Registry so that other developers can easily install a plugin using the Apache Cordova CLI.

2.2 User Authentication Basics

IT services such as e-mail, online banking, enterprise IT systems or social media platforms build around the concept of user accounts. Each user account is associated with potentially sensitive data and legitimate users can execute actions, such as confirming money transfers or sending e-mails on their behalf. It is obvious that these services require some access control. First, the service has to determine the identity of the user and secondly, the service requires some proof that the user is, who she claims to be. The process of associating a personal identifier with an individual person is called identification [17]. In real-world scenarios, identification is often done by presenting some legal document, for example a passport or driver's license. Authentication, on the other hand, is the process of proving the association between the person and the personal identifier.

In the digital world, the username/password paradigm is ubiquitous. Identification is done by providing a username, whereas authentication is performed by providing a password that should be kept secret. The password presents a single piece of information and once it falls in the hand of an attacker, she has full access to the service. This might include access to sensitive enterprise data as well as the possibility to impersonate the victim by sending messages on her behalf. However, the username/password paradigm suffices for applications that require a low level of assurance about the identity of the user. However, for many services it is desirable to have additional measures to prove that the user is, who she claims to be. These additional measures are often referred to as multi-factor authentication.

Multi-factor authentication enables multi-dimensional access control, which requires the user to present at least two authentication factors out of the following categories:

- *factor knowledge* or "something the user knows"
- *factor possession* or "something the user has"
- *factor inherence* or "something the user is"

Requiring more than one factor increases the difficulty for an attacker to successfully complete an authentication process. For example, modern automated teller machine (ATM) cards that allow the user to withdraw money from her account, require an attacker both, to steal the ATM card and to gain knowledge of the user's secret PIN. The ATM card represents the factor possession, whereas the PIN represents the factor knowledge. Typical examples for the factor possession are smart cards and smart tokens that comprise secret cryptographic keys. Services that require a substantially high assurance about the identity of the user, often employ authentication based on physical attributes of a person. Popular examples include voice recognition systems, measuring a person's heart beat [74], retina pattern recognitions systems, iris and fingerprint scanning systems.

Based on the identified need for multi-factor authentication, we construct several authentication methods. Therefore, in Chapter 4 we start by extracting building blocks typically found in multi-factor authentication methods and introduce features available to state-of-the-art mobile devices. A presentation of the constructed multi-factor authentication methods is then given in Chapter 5.

Chapter 3

Related Work

This chapter provides an overview of currently deployed multi-factor authentication methods that are applicable to mobile end-user devices. While there are many new authentication methods available, few projects aim to provide interfaces and standards for authentication methods or a more generalized authentication framework that provides a single interface to online services and enable the easy exchange or extension of authentication methods. Therefore, a review of approaches towards unifying access to multi-factor authentication methods is given. We conclude with a review of publications that aim to evaluate the security of user authentication methods. Some of the proposed evaluation criteria will act as a basis for our security evaluation in Chapter 7.

3.1 Multi-Factor Authentication Methods

Many online services, enterprises and public institutions have realised that simple password-based authentication does not provide sufficient protection against impersonation or theft of confidential data. Therefore, many service providers require the user to present a second factor during login. Currently, deployed multi-factor authentication methods comprise:

- *the use of one-time passwords* that rely on a pre-shared cryptographic key for the computation of a character string that can only be used in a single login process,
- *Public Key Infrastructure systems* that rely on the user presenting a token that stores a secret cryptographic key to compute a challenge. These systems comprise smart cards or USB dongles that include tamper-resistant storage for private keys. Furthermore, tokens interacting over a wireless Near Field Communication (NFC) interface can be used as well.
- *the use of static Transaction Authentication Numbers (TANs)*, where a TAN is sent to the user using a different communication channel, e.g. by using SMS technology to send a TAN and
- *biometrics* such as fingerprint or iris scanning system and voice recognition systems.

In the following, we take a closer look at two-factor authentication methods that are applicable to mobile devices. First, authentication based on the computation of OTPs and TANs are briefly discussed. Although it is possible to use smart cards with some mobile devices, they rely on the user carrying an additional smart card reader[1] and thus decrease usability drastically. However, we inspect the recently published Fast Identity Online (FIDO) standard, which includes standardization for the use of NFC-enabled tokens. In addition, a short review of biometric systems that are already deployed on mobile devices is given.

[1] `http://www.thursby.com/products/pkard-android`

3.1.1 Transaction Authentication Numbers

Several online banking services within Europe use mobile TAN in addition to password-based authentication. Thereby, the user's mobile phone implements the factor possession. During the authentication process, the TAN is sent to the user's mobile phone using SMS technology. By proving reception of the TAN, the user proves possession of the SIM card within the mobile phone. The security of TANs is mainly given due to two separate end user devices being used. Whereas the password is entered at a desktop computer, the SMS is sent to the mobile phone. The nature of using two separate devices increases the difficulty for an attacker trying to compromise the authentication process.

However, in 2011, security analysts discovered Zeus-In-The-Mobile [21][52], a malware targeting mobile TAN solutions. Zeus-In-The-Mobile intercepts TAN messages sent by the bank and forwards them to the attacker. To intercept the received SMS messages, the malware tricks the user into installing a malicious application on her smartphone. By pretending to protect the user from Internet fraud, the malicious application forwards incoming SMS to the attacker's server. A malicious program installed at the user's personal computer allows for stealing the user's login credentials used for the online banking service. The attackers were able to transfer money to arbitrary accounts by using the login credentials and the dynamically generated TAN.

Modern smartphones include a broad range of third-party applications that might contain malicious code for accessing SMS messages. Due to increasing screen sizes, increased usability and software as advanced as on the desktop computer, users tend to consume online services solely using their mobile device. However, the security of TAN systems decreases in case of using the same device for entering your credentials and proving reception of the TAN. Malicious applications might capture user input and at the same time gain access to SMS messages.

3.1.2 One-time Passwords

Authentication mechanisms that rely on the computation of OTPs require the user to enter an OTP in addition to username and password. The entered OTP is computed on the user's mobile device or some token that is in control of the user. For the computation of OTPs, a cryptographic key and some dynamic factor are needed. The cryptographic key is kept secret and is shared between the server and the client that computes the OTP first. For the dynamic factor, multiple implementation variants exist. Following the Time-Based One-Time Password Algorithm (TOTP) [57], the current time is used as dynamic factor. With the HMAC-based One-Time Password Algorithm (HOTP) [56] standard, a counter, which is stored on both the client and on the server side and incremented with every OTP computation, is used as a dynamic factor. In order to be able to verify the entered OTP, the server has to perform the same computation using the shared cryptographic key and the dynamic factor. When using the current time for OTP computation, a so-called time step is used instead of the exact time. The use of time steps enables both client and server to compute the same OTP within a specified period of time. Typically, a period of 30 or 60 seconds is used for the time step. When using a shared counter instead, the server has to provide mechanisms for the synchronization of the counter, such as computing OTPs with multiple future values of the counter, to absorb inconsistencies caused by unsuccessful authentication attempts.

By requiring OTPs that are computed on the client instead of using TANs sent via SMS, no costs for the transport of the TAN incur. Furthermore, the mobile device does not have to be online to compute the OTP and thus can be used abroad without limitations regarding roaming fees. The security of authentication mechanisms that rely on the computation of OTPs mainly builds on the security of the underlying key storage mechanism. Whereas most solutions use software storage, several hardware devices such as USB or NFC tokens can be used for OTP computation as well. In the following, we describe some services that use OTP computation as the second factor during authentication.

Google Authenticator

The Google Authenticator[2] application is one of the most popular ways for computing OTPs on smartphones. Google Authenticator is not limited to Google's web services, but can be used with other services such as Dropbox, Facebook, Microsoft and even during SSH login [50]. The Google Authenticator application follows the previously mentioned TOTP and HOTP standards. Third-party applications can integrate two-factor authentication with Google Authenticator as well. There are several implementations for the corresponding authentication server available[3]. The Google Authenticator client application is available for iOS, Android and BlackBerry[4]. As basically any application that follows the TOTP and HOTP standards can be used, Microsoft has implemented their own Authenticator application for Windows Phone devices[5].

There are large differences regarding the security of the secret key required for OTP computation. The iOS application stores the key within the Keychain[6], a mechanism for the storage of user credentials and cryptographic keys. The Android version of the Google Authenticator stores the secret key in an SQLite Database[7] on the file system. The iOS Keychain prevents attackers from stealing the secret OTP key, whereas an attacker might be able to clone the key by inspecting the internal storage of the Android application.

YubiKey

The company Yubico[8] offers USB and NFC tokens that can be used as a second factor during login. Yubico's first token, the YubiKey, operates on all platforms that support USB without requiring additional drivers. When the user presses the button on top of the Yubikey, the token computes an OTP. By implementing the Human Interface Device specifications, the YubiKey acts like a conventional keyboard and thus the OTP is directly entered in the authentication form of the web service. To support mobile devices as well, Yubico has launched the YubiKey NEO. The YubiKey NEO provides support for NFC and can be used on NFC-enabled smartphones. By simply tapping the YubiKey NEO on the NFC-enabled mobile device, the OTP is transmitted to the device and added to the authentication form. Amongst a range of protocols, the YubiKey supports the generation of HOTP and TOTP conforming OTPs. The YubiKey stores cryptographic keys on a dedicated Hardware Secure Element, thus the YubiKey offers good protection against attacks that aim to extract the secret key. However, currently the YubiKey NEO can only be used on the Android platform by installing a mobile application from Yubico [20]. Although iOS 8 supports NFC, third party developers cannot access the NFC interface [80]. Due to implementation issues, the Yubikey NEO is currently not working on Windows Phone 8 devices [54].

Intel Identity Protection

Intel Identity Protection (IPT) [36] provides hardware-based authentication mechanisms integrated directly with Intel processors. Amongst other features, Intel IPT ships with an OTP generator. The OTP generator is integrated as embedded processor, the so-called Manageability Engine of the device's motherboard. OTP computation is performed in isolation from the operating system and the cryptographic key required for OTP computation is stored on a dedicated area within hardware. Users can associate their device with a particular user account at online services that support Intel IPT with OTP. Some services, such as PayPal, have already integrated login with Intel IPT with OTP [67]. In contrast to hardware

[2] https://play.google.com/store/apps/details?id=com.google.android.apps.authenticator2&hl=en

[3] https://github.com/wstrange/GoogleAuth

[4] https://github.com/google/google-authenticator

[5] https://www.windowsphone.com/en-us/store/app/authenticator/e7994dbc-2336-4950-91ba-ca22d653759b

[6] https://github.com/google/google-authenticator/blob/master/mobile/ios/

[7] https://github.com/google/google-authenticator-android/

[8] https://www.yubico.com/

tokens used for OTP computation, Intel IPT does not require the user to present an additional device and thus improving usability drastically. However, currently only five mobile devices are equipped with Intel CPUs and not all of them provide support for Intel IPT [37]. As Intel IPT is a proprietary implementation, implementation details are not publicly available. Therefore, little can be said about the security of the hardware-based OTP generator.

3.1.3 FIDO Universal Second Factor

The FIDO alliance[9] is working towards standardization for strong authentication devices. The FIDO alliance comprises popular companies such as Google, Microsoft, ARM, Lenovo, Yubico and others. Their Universal Second Factor (U2F) specification [4] provides a standardized interface for tokens that can be used as second factor during authentication. Therefore, first the user authenticates using a conventional username and password scheme and as second step she is requested to present her U2F token. Current U2F tokens are realised as USB, NFC or Bluetooth tokens. The main goal of U2F is to integrate support for U2F tokens directly into web browsers and, thus, render additional middleware that has to be installed on the user's device unnecessary. Google Chrome already provides native U2F support[10]. Google provides the possibility for users to register U2F tokens for authentication with their Google accounts. The built-in browser API, however, can be used by any web service that aims to integrate U2F for authentication. Currently, FIDO U2F tokens can be purchased starting with a price around 6 USD per token[11].

FIDO U2F uses standardized public-key cryptography for proving possession of a U2F token. During registration, the user's U2F token creates a new public and private key pair for each web service. The registration response returned to the web service includes the newly generated public key, a key handle for the corresponding private key and the attestation certificate [2]. FIDO recommends the use of hardware-backed storage mechanisms for cryptographic keys on the device. To check that a U2F device provides a certain level of security, the U2F specification introduces attestation. Therefore, the registration message is signed using a so-called attestation key. The private attestation key is stored on the U2F token by the device manufacturer. The web service can then verify the signature of the registration response using the public key included in the attestation certificate. Furthermore, the web service should verify whether the attestation certificate was issued by a trusted certification authority. For privacy reasons, a large number of U2F tokens from the same manufacturer share the same attestation key. This prevents identification and tracking of users across origins based on their attestations key.

In subsequent authentication processes the user proves possession of the private key by creating a cryptographic signature on her U2F token. Therefore, the web service provides the key handle that identifies the private key and some challenge that should be signed by the token. Due to the U2F token having its own key pair for every web service that has been registered, the token only signs challenges for the origin, the key pair was generated for. As a key pair can only be used for the origin of one web service, the user is protected against tracking across multiple web services.

3.1.4 Tiqr

In 2011, the company SURFnet has developed Tiqr [75]. Tiqr is a smartphone application that enables two-factor authentication using QR codes and a shared secret stored both on the mobile device and the server. QR codes represent two-dimensional barcodes that can be used to encode up to 4 Kilobytes of

[9] https://fidoalliance.org/

[10] https://support.google.com/accounts/answer/6103523?hl=en

[11] http://www.amazon.com/s/ref=nb_sb_ss_i_0_6?url=search-alias%3Daps&field-keywords=u2f+security+key&sprefix=u2f+se%2Caps%2C416

alphanumeric data. Currently, SURFnet provides an iOS[12] and an Android[13] version of their mobile application.

When navigating to a web service that supports authentication via Tiqr, the user is presented with a QR code that contains a challenge and information about the web service that requests authentication. After scanning the QR code with the mobile device, the Tiqr application displays the service information to the user and prompts the user to enter her PIN for approving authentication. Tiqr uses a 4-digit PIN to secure a shared secret required for computing an OTP. Therefore, Tiqr makes use of Password-Based Encryption according to PKCS#5. The authors claim that although only a simple 4-digit PIN is used, brute-force attacks on the PIN do not impair the security of Tiqr. When trying all possible PIN values to derive the encryption key, the attacker usually requires a way to verify that the decrypted content represents the original plaintext. As the plaintext itself represents a random byte string, the only way to verify its correctness is to compute an OTP with the decrypted key. To verify the OTP, the attacker has to enter the OTP at the web service. Only the server can check the correctness of the OTP and thus if the key was decrypted with the correctly derived key. By specifying a maximum number of failed authentication attempts at the server side, this issue can be mitigated.

In addition to the shared secret, a changing factor such as the current time, a counter or other means of a challenge is required for OTP computation. Tiqr uses a challenge that is encoded within the QR code. The computed OTP is sent to the authentication endpoint of the web service. Thus, Tiqr does not require the user to type the OTP at the web service.

3.1.5 Biometric Authentication on Mobile Devices

In 2013, Apple launched the first iPhone that includes a fingerprint sensor, the so-called Touch ID [13]. Data required to match a user's fingerprint is stored in encrypted memory within the Secure Enclave, a coprocessor that, amongst others, provides cryptographic operations for the iOS Data Protection system. Fingerprint data from the Touch ID sensor is processed by the Secure Enclave and never leaves the mobile device. Since iOS 8, third-party application developers can leverage the fingerprint sensor for user authentication within their applications. The application is notified whether the authentication process was successful, but it does not receive any data associated to the user's fingerprint. Currently, Touch ID offers two capabilities for third-party applications [45]. Obviously, applications can request the user to authenticate using Touch ID. Applications do not receive any signed token or cryptographic proof that the user is authenticated successfully, only a notification that authentication has been successful. Second, applications can use Touch ID to unlock the iOS Keychain that provides protected storage for credentials. However, both methods do not provide a mechanism to authenticate the user to a remote web service. Thus, third-party applications need to implement additional means for remote authentication.

Starting with its flagship mobile phone Galaxy S5 in 2014, Samsung has equipped multiple devices with fingerprint sensors as well. Developers of third-party applications can use the Samsung Pass SDK to integrate fingerprint scanning into their applications[14]. Similar to Apple's Touch ID, the API returns no additional token, only a notification whether the fingerprint of the user matches the registered fingerprint. Samsung's fingerprint system has already been subject to serious security vulnerabilities [82]. Researchers from FireEye have found that it is possible for malicious applications to directly access the fingerprint sensor to read the scanned fingerprint before it is being stored within the ARM TrustZone of the device. Zhang et al. [82] highlight the severity of this leak, as fingerprints are associated with a series of records from public institutions, such as passports or criminal records.

[12] `https://itunes.apple.com/us/app/tiqr/id430838214?mt=8`

[13] `https://play.google.com/store/apps/details?id=org.tiqr.authenticator&hl=en`

[14] `http://developer.samsung.com/galaxy#pass`

3.2 Authentication Frameworks

The preceding presentation of current multi-factor authentication methods on mobile devices illustrates that there are plenty of methods available for enabling multi-factor authentication with mobile devices. However, no single authentication method has yet gained sufficient popularity among both service providers and users in order to replace the conventional username and password scheme on large scale. Based on the user's mobile device or tokens she already possesses, she might prefer a specific authentication scheme. Thus, in order to provide multi-factor authentication to all users, service providers need to support multiple different authentication schemes. However, integrating several different authentication mechanisms poses a challenge to both service providers and vendors of authentication solutions. When using commercial solutions, authentication methods from different vendors provide different interfaces, thus the integration into services is tailored to a single authentication method. Therefore, some vendors have started to provide authentication frameworks that integrate several different authentication methods. In this section, we will review some authentication frameworks from industry and provide a short review of the first standard proposing a universal authentication framework.

3.2.1 FIDO Universal Authentication Framework

Besides providing specifications for second-factor authentication tokens, the FIDO alliance has published specifications for a Universal Authentication Framework (UAF) [3]. The goal of FIDO UAF is to use so-called local authentication actions in order to allow for a password-less authentication process. Therefore, the user registers local authentication actions with a web service. These actions can, for example, consist of presenting biometrics, entering a PIN or presenting a NFC token. Each authenticator that provides a local authentication action possesses a cryptographic key pair. During authentication the authenticator validates the performed authentication action (e.g. by sampling biometric data such as a fingerprint or voice print or by verifying the entered PIN). If the verification has succeeded, the authenticator signs a provided challenge and returns the challenge to the web service.

In detail, FIDO UAF provides web services with mechanisms to choose the required action based on a policy. Within this policy, the web service can for example define the required key storage type (e.g. a secure element, software or a remote key handle), the type of user verification and if the authenticator is able to display transaction information to the user. This policy is included within the registration request from the web service. The UAF implementation on the client, for example, the user agent, matches the policy against all available authenticators and chooses a suitable authenticator. Therefore, FIDO UAF is an enhancement of the FIDO U2F standard. Whilst U2F tokens simply sign a challenge provided by the server, the UAF standard defines authenticators and requires the user to perform some action such as presenting a fingerprint before the challenge is signed and returned to the web service. In addition, UAF includes the possibility to display transaction-related data to the user. For example, this can be used for confirming an online banking transaction.

The FIDO UAF specification comprises the protocol that defines how web services can request the user to perform a local authentication action, the documents that define the format of UAF protocol messages, the client-side APIs that can be used within the web applications to utilize FIDO UAF and an authenticator API, which defines the interface for implementations of local authentication actions as plugins.

As FIDO UAF provides a common interface for authenticator modules, UAF supports the easy exchange of authentication actions. Thus UAF adapts well to the emergence of new technologies and potential new methods that can be used during authentication. However, currently no user agent supports UAF natively. In order to use UAF, additional native clients for desktop operating systems or mobile devices are required.

3.2.2 Commercial Authentication Frameworks

In the following, we will shortly enumerate commercially available authentication frameworks including the FIDO UAF implementations of early adopters. Due to most projects being closed-source, only vague information is available about their features and security properties.

RCDev offers OpenOTP[15], a server side authentication framework that includes various two-factor authentication methods. OpenOTP includes integration for standardized HOTP and TOTP tokens, FIDO U2F, SMS TAN and others. OpenOTP does not require a specific mobile application as it can be used with existing OTP applications, such as Google Authenticator. Service providers can access OpenOTP using different interfaces such as SOAP.

The company Nok Nok Labs implements the FIDO UAF standard [46]. They offer mobile applications for Android and iOS that act as FIDO UAF client implementation. For example, Apple's Touch ID can be used as FIDO authenticator. Further, they have implemented face recognition, voice recognition and a simple PIN authenticator. The Nok Nok App SDK enables application developers to integrate a FIDO UAF client into their applications.

Similar to the solution provided by Nok Nok Labs, Sure Pass ID [66] provides a UAF compliant authentication framework. In addition to UAF, they support authentication using OTPs and TAN and can be used with existing applications or hardware tokens that compute OTPs and smart cards. Their authentication server is operated as cloud service.

Besides offering FIDO U2F integration, the US-based company Duo Security has launched their mobile application DuoPush [41]. DuoPush receives authentication requests via the mobile platform's push-notification service. The authentication request contains a challenge. When approving to sign in, the application uses a private key stored on the mobile device to cryptographically sign the given challenge.

When implementing these multi-factor authentication frameworks, the development has to be carried out for multiple mobile platforms separately. Thus, developers have to gain knowledge about the APIs, build tools and the security properties of each mobile platform. Furthermore, they have to provide several different implementations for the storage of secret keys, the reception of push notifications and so on. In addition, maintaining and updating several versions of the same product is costly. Therefore, this thesis provides the implementation of a mobile authentication framework using cross-platform technologies and thus minimizing the effort for customization for different mobile platforms.

3.3 Existing Security Evaluations of User Authentication Methods

Username and password schemes are still very popular for user authentication on the web. Bonneau et al. [16] criticize that although decades of research have shown that passwords do not provide sufficient security, still no authentication scheme was able to replace the traditional username and password scheme. Furthermore, they criticise the lack of standard evaluation criteria, as evaluation of authentication schemes tend to be biased depending on the domain of the respective authors. Therefore, Bonneau et al. have provided an evaluation framework for authentication methods on the web. They have derived 25 criteria that an ideal authentication scheme should meet. The proposed criteria have been grouped into the categories *Usability*, *Deployability* and *Security*. The authors have applied their framework to 35 authentication methods, including conventional username-password schemes, password managers, hardware tokens and OTP variants.

Whilst Bonneau et al. did an evaluation of user authentication methods in general, Rijswijk et al. [75] aim to provide evaluation criteria particularly targeting two-factor authentication methods. They have introduced six categories to judge two-factor authentication methods. These categories com-

[15] `http://www.rcdevs.com/products/openotp/`

prise *Hardware-independence*, *Software-independence*, *Security*, *Costs*, *Open Standards Compliance* and *Ease-of-Use*. Based on these categories, they have evaluated several two factor authentication schemes, including mobile TAN systems, mobile applications that compute OTPs and their newly proposed authentication scheme Tiqr. A short review of Tiqr has been provided in Section 3.1.4.

To evaluate the security of our proposed authentication framework and the integrated authentication methods, the defined criteria act as input for the definition of security threats and the subsequent derivation of objectives that have to be met by the integrated authentication methods. Chapter 7 includes a detailed description of the methodology used for the security evaluation.

Chapter 4

Authentication Building Blocks

Based on the authentication methods presented in the previous chapter, this chapter extracts building blocks found in multi-factor authentication methods. Furthermore, we discuss device features typically available on mobile devices. From the extracted building blocks and the presented mobile device features, authentication methods suitable for the mobile use are constructed in Chapter 5.

4.1 Building Blocks of Authentication Methods

Chapter 3 describes several methods for proving the factor possession in addition to username and password during login. In this section, we extract building blocks found in the previously described authentication methods. In general, we can observe that the discussed authentication methods typically rely on the user to provide some response to a challenge. The challenge-response schemes can be categorized into two basic types.

(a) With FIDO and with schemes relying on the use of smart cards or other cryptographic tokens, the verifier generates a challenge and transfers the challenge to the user's token or mobile device. On the mobile device, the token or authentication app performs some cryptographic operations and returns the response to the verifier. After successful verification of the response, the verifier assumes that the user is genuine and authorised to access the web service.

(b) In the case of OTPs, the challenge is often not supplied by the verifier but implemented as counter in the authentication app or the challenge is derived from the current time. Therefore, methods relying on OTP methods require a shared cryptographic key present with both, the user and the verifier.

When using asymmetric cryptography, the user needs to adequately protect the private key. Symmetric cryptography relies on some shared secret key between the user and the verifier. However, both approaches require a key-storage facility on the mobile device. So far, we have identified the need for some key-storage facility on the mobile device and a communication link from the verifier to the user in order to distribute the challenge and return the response to the verifier. This communication link does not necessarily have to present a data channel but can also require the user to manually enter some data.

For computing the response to a challenge, applications typically make use of cryptographic primitives. These comprise the generation of digital signatures or the computation of a message authentication code. Furthermore, authentication methods require mechanisms for protecting against common attacks. For example, cloning of the device that represents the factor possession allows an attacker to perform authentication without having access to the user's mobile device. In addition, authentication methods also have to protect against phishing, where the genuine device is used to compute the response to a challenge that can be used in a subsequent authentication process initiated by the attacker. Cryptographic primitives are used to meet these security requirements.

As additional protection against cloning or the theft of the device covering the second factor, a web service might require proof that only the legitimate user is operating the mobile device. This can be realised by requiring the user to present some knowledge. Most commonly, this is realised by prompting the user for a password or PIN. If no password is entered in the process of identification, the application can require the user to enter a password for employing a combination of knowledge and possession for authentication.

Furthermore, it is desirable to inform the user about the current authentication process. Therefore, methods for displaying information about the service she is authenticating for or the transaction she is authorising can be displayed.

Summarising, an authentication method based on the challenge-response principle ideally comprises

- a *communication link* for exchanging the challenge and the computed response,
- the implementation of *cryptographic primitives* for computing the response to the challenge,
- *key-storage* facilities that adequately protect sensitive key material,
- mechanisms for *binding the user to a single device* and thus prevent cloning of the factor possession,
- an optional *knowledge proof* to make sure that the genuine user operates the device covering the second factor
- and an optional way for *displaying transaction-related data* to the user.

Modern mobile devices offer a range of different communication technologies, sensors and considerable computational power and storage space. Hence, these devices might well cover the just defined functional requirements.

4.2 Mobile-Device Features

This section presents features and technologies available to modern mobile applications. This includes the functioning of push notifications, storage options available on the device and a short review of other promising features.

4.2.1 Service-to-Device Communication

The GSM standard defines the Short Message Service (SMS) that allows for the transmission of short text messages using the GSM network. Messages are sent to the SMS center of the mobile network operator. As soon as the recipient can be reached, the message is sent to the device and thus employing a so-called *store-and-forward mechanism*. SMS messages are broadly supported by mobile devices. They are used for advertising, news and multi-factor authentication, for example within SMS-based TAN systems. However, SMS technology has various disadvantages. First, the message size is quite limited with a maximum of 140 8-bit characters. Second, SMS messages do not satisfy the requirements of modern mobile applications. It is fundamental to many mobile applications to deliver data in near real time. Data is then processed by the service's mobile application on the device. Some platforms, for example iOS, do not allow for automatic processing of data received via SMS, as third-party applications do not have access to the content of incoming SMS messages.

To meet the requirements of many modern mobile applications, push technology has emerged. Push notifications refer to an Internet-based communication technology, which enables the transmission of messages from a centralised server to a mobile device. For this purpose, the mobile device establishes a connection with a central push-notification service and receives messages from this service over a persistent TCP/IP connection. Each platform vendor, such as Google, Microsoft, Apple and BlackBerry, provides and operates its own push-notification service. The different push-notification services differ regarding the format of notifications and the API that is provided by the platform vendor.

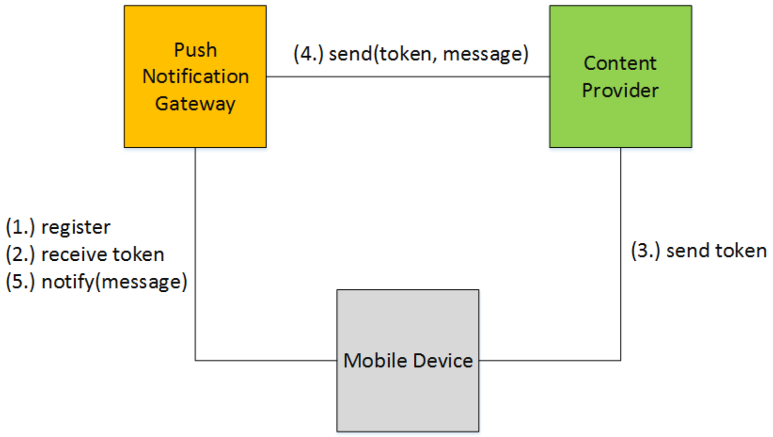


Figure 4.1: In order to use push notifications, the application on the mobile device has to register with the Push Notification Gateway. The obtained token has to be transmitted to the Content Provider. By specifying the received token the Content Provider can send messages to the mobile device via the platform specific push-notification service [77].

Figure 4.1 illustrates the general workflow for setting up push notifications for an application. If a web service wants to send push notifications to a mobile device, the mobile device has to register with the push-notification service first (Step (1.)). Only after registration, the mobile device is able to obtain notifications from a specific content provider. Upon registration, the mobile device obtains a token or registration ID (Step (2.)), which has to be transmitted to the content provider (Step (3.)). Using this token or registration ID, the content provider can unambiguously specify the receiver of the push notification. The desired message and data identifying the receiver are passed to the platform vendor's push-notification service (Step (4.)), from where it is forwarded to the mobile device (Step (5.)).

In the following, we shortly describe the push-notification systems of the two most popular mobile platforms: Google Android and Apple iOS. We discuss the general functioning, the registration workflow, available features and security implications. In particular, we illustrate how the binding to the device and a particular mobile application is realised and which mechanisms aim to protect the different communication paths.

Google Cloud Messaging

Google Cloud Messaging (GCM) [28] offers push-notification services for Android devices. Therefore, messages are sent from the content provider, which is in control of the application developer, to the mobile application via the GCM Connection Server, which corresponds to the push-notification gateway in Figure 4.1. The GCM services can be used for free. GCM allows sending a maximum of 4 Kilobytes of data to the mobile device. Data can be represented as plain text or as a JSON-encoded data structure. Currently, two communication protocols are supported: HTTP and Extensible Messaging and Presence Protocol (XMPP) [64]. XMPP, originally named Jabber, enables upstream messages as well. This way, mobile applications can send updates to the server.

In May 2015, Google announced changes regarding APIs and features of GCM. Since then, GCM can be used for iOS devices as well and therefore, significantly reduces development effort when developing applications for multiple mobile platforms. There is a new plugin[1] for the Apache Cordova

[1] https://github.com/gonzaloaune/GCMPushPlugin

framework, which provides the new GCM features to iOS and Android devices. However, our framework still uses the former GCM API as covered by another plugin[2]. This means that the subsequent statements are valid for the old GCM system. The following remarks adhere to the GCM analysis of Li et al. [48]. However, changes to the GCM system mainly affect the naming of APIs whereas the general functioning remains untouched.

- *Device binding*

  Android uses a process running in the background, to monitor incoming messages and wake up subscribing applications. This GCM service is the same for all applications on the mobile device. When registering push notifications for an application, the application uses the APIs provided by the GCM service. The application fires an Intent[3] with the content provider's sender ID. The sender ID is uniquely assigned when registering a content provider for push notifications using the Google API console. The GCM service further sends a registration request to the GCM connection service. This registration request includes the sender ID, which identifies the content provider, the unique Android ID, which identifies the mobile device and the application ID, which is constructed from the package name and, therefore, identifies the mobile application. The GCM generates a new registration ID for the application on this specific device. The registration ID is used to locate the right application on the right mobile device and, therefore, has to be added to each push notification that should be sent. When sending a message, the content provider hands the message data and the recipient's registration ID to the GCM Connection Server. The GCM Connection Server stores the message and delivers it as soon as the device is online. On the device, the notification is delivered to the GCM service, which then forwards it to the right application.

  This makes clear that the registration ID has to be kept confidential. Manipulation of the registration ID might lead to the wrong party receiving messages intended for a legitimate user. Therefore, special care has to be taken for the protection of the registration ID during transmission from the mobile application to the content provider. Furthermore, it has to be protected when the application communicates with the GCM service on the client and during communication between content provider and the GCM Connection Servers.

- *Communication: Content Provider ↔ Push Notification Gateway*

  The GCM Connection Server provides an HTTPS endpoint only. When sending a push notification, the content provider has to supply its API key within the HTTP Authorization request-header field [23]. Therefore, it is important that the API key is kept confidential on the server.

- *Communication: Push Notification Gateway ↔ Mobile Device*

  During registration, the GCM service on the device contacts the GCM Connection Server. Li et al. [48] have inspected the sent data packages and conclude that registration requests for obtaining the registration ID use SSL/TLS.

- *Communication: Mobile Device ↔ Content Provider*

  The communication path between device and content provider is needed to send the newly obtained registration ID to the content provider. Application developers have to take care of adequate protection. This involves the use of SSL/TLS and techniques for detecting man-in-the-middle attacks, e.g. certificate pinning [73].

[2] `https://github.com/phonegap/phonegap-plugin-push`

[3] Google Android uses so-called *Intents* for inter-process communication. An Intent is the description of an operation to be performed.

Apple Push Notification Service

Apple Push Notification Service (APNS) is the equivalent of GCM but for iOS devices. In order to set up the push-notification service, Apple requires a paid Apple Developer Account. With APNS, the content provider can send up to 2 Kilobytes per message to the mobile device. The payload has to be JSON-encoded. Apple has put various mechanisms in place for providing security to its push-notification system [12]. In the following, deployed security mechanisms at each communication link are listed.

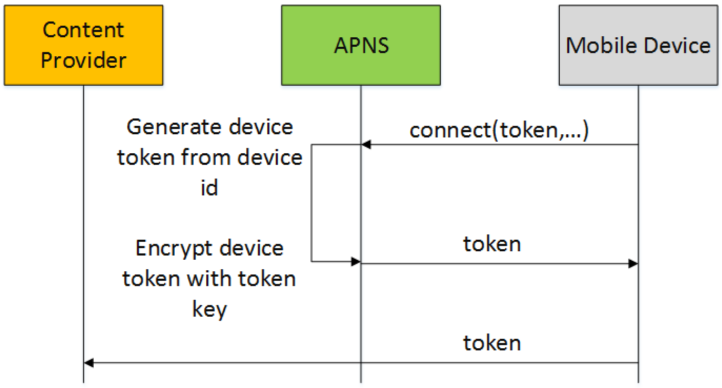

Content Provider
APNS
Mobile Device
Generate device token from device id
connect(token,...)
Encrypt device token with token key
token
token

Figure 4.2: APNS generates a device token based on data extracted from the unique device certificate. The device token is encrypted with the token key, which is only available to APNS. The device receives the encrypted device token and forwards it to the content provider. When receiving a message from the content provider, APNS decrypts the device token to identify the recipient of the message. (Figure adheres to [12])

- *Device binding*

  Each application needs to register before being able to receive push notifications. This is typically done right after the application is installed on the device. The registration request is performed by calling an API. The first time an application calls the respective API, iOS presents a dialog that asks for the user's permission to present the types of notifications the app registered for. The system forwards the registration request to the APNS.

  Each iOS device features a device certificate, used for setting up a mutually authenticated SSL/TLS connection to the APNS. Based on data extracted from the device certificate, a device token is generated. The device token, therefore, contains a unique device identifier. The device token is encrypted with a token key only available to APNS. The encrypted device token is then returned to the device. As the device token is constructed using the unique device identifier, the APNS uses it for routing notifications to the right application on a particular mobile device. Furthermore, the token key ensures that only the APNS can issue device tokens. When receiving notifications from the content provider, the APNS can check the validity of the device token by decrypting it. The notification is then forwarded to the device uniquely identified by the decrypted device token. Figure 4.2 illustrates the handling of the device token.

- *Communication: Content Provider ↔ Push Notification Gateway*

  When registering a content provider with APNS, the developer obtains a cryptographic key pair and the corresponding public key certificate. The certificate includes the bundle ID of the application (i.e. the package name). For that reason, the provider certificate is only valid for one application. When communicating with APNS, a mutually authenticated SSL/TLS connection is required. This way, both, the content provider and APNS, can validate each other's identity.

- *Communication: Push Notification Gateway ↔ Mobile Device*

As previously mentioned, each iOS device possesses a unique key pair and a certificate. The connection between APNS and the mobile device is protected using SSL/TLS with client and server authentication as well. The required key and certificate are obtained when activating the mobile device. According to Apple, the obtained credentials are stored using the device's Keychain, a hardware-backed key store available to iOS devices. When exchanging heartbeat messages with APNS, the mobile device initiates an SSL/TLS connection and receives the server's certificate. The device validates the server certificate and returns the device certificate to APNS. After validation of the device certificate, the SSL/TLS connection has been established. Figure 4.3 illustrates this procedure.

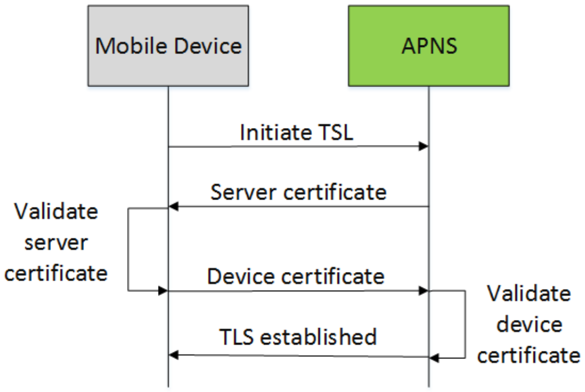


Figure 4.3: In order to set up a secure connection between the mobile device and APNS, the mobile device and APNS exchange their respective certificate (figure adheres to [12]).

- *Communication: Mobile Device ↔ Content Provider*

  The application has to transfer the obtained device token to the content provider to enable the content provider to send messages to the device. The application developer needs to ensure adequate protection of this communication path.

Although GCM and APNS provide protection mechanisms for the various communication links, payload data is available in plaintext at their premises. This can present a serious security risk if potentially confidential data is transmitted to the device. Furthermore, push-notification services lack mechanisms for providing integrity and proving that a device has received a specific push notification. These security mechanisms have to be put in place from the developer of the specific application and the content provider associated with this application.

When transferring sensitive data, the so-called *poke-and-poll model* should be preferred [77]. With poke-and-poll, the mobile application only receives the information that new data is available. This way, the application can retrieve the data over a direct and assumably secure connection with the content provider. Hence, the push notifications sent do not include sensitive data and do not carry application state.

4.2.2 Storage

Typically, mobile applications require facilities for storing application data, user preferences and other data locally on the device. Special considerations apply to applications that store sensitive data. The developed authentication framework, for example, represents an application that requires the storage of sensitive data. Authentication methods relying on the challenge-response principle require cryptographic primitives for computing the response to a given challenge. In order to protect the used cryptographic keys against various threats, such as malware or theft of the device, the authentication app requires storage facilities appropriate for storing key material.

In the following, storage options available on mobile devices are discussed. We focus on storage options that can be used by applications developed using the cross-platform framework Apache Cordova. There are four different ways to store data locally in Apache Cordova applications: on the one hand, the WebView itself provides WebStorage and IndexedDB, which conform to specifications from the World Wide Web Consortium (W3C) and the now outdated WebSQL technology. On the other hand, developers can use the Cordova File Plugin, for accessing the local file system. As we put our focus on storing sensitive data, such as cryptographic keys, methods for accessing the platform-specific key store are discussed as well.

WebStorage

WebStorage as defined within a W3C Candidate Recommendation [34] provides persistent storage facilities with an API integrated into the web browser. WebStorage is also often referred to as HTML5 storage. WebStorage allows developers to store key-value pairs where both, key and value items, are string values. Traditionally, small amounts of data were mainly stored within session cookies. WebStorage aims to provide additional storage facilities beyond session cookies. When using cookies for storing data, the stored data is transmitted with every request. Therefore, bigger amounts of data cause unnecessary network traffic and data might get leaked during transmission. WebStorage eliminates the need for transferring the data with every request.

The WebStorage standard defines two implementations of the WebStorage interface. A session-based store similar to cookies, the so-called *Session Storage* and a persistent store spawning over multiple sessions, the *Local Storage*.

The Session Storage is available for each browsing context. Therefore, data items stored in the Session Storage are accessible to any page from the same site opened within the same browser window. In contrary to conventional session cookies, data is not accessible to the same site opened in a different browser window.

Local Storage, on the other hand, provides a storage object for each origin (protocol, domain and port). Items stored in Local Storage are shared across every window or tab running within the same origin. Thus, the stored items are also accessible when a user opens a site from a specific origin within another browser window. The Local Storage is cleared either when the user requests the browser to do so (e.g. by clearing the browsing data) or in the case of limited storage space. Due to its ease of use, the Local Storage is a popular choice for web application developers wanting to locally store data. However, Local Storage only allows string values, therefore, complex objects that cannot be serialised to string objects cannot be stored in Local Storage. In addition, Local Storage only supports data retrieval based on the provided key, thus Local Storage is not suitable for use cases that require more complex data queries.

When using Apache Cordova for developing mobile applications, the application is rendered within a WebView, an embedded browser window. The WebStorage API is provided by all mobile browsers that provide a WebView component[4]. Each application is assigned its own cache, cookie store and Local Storage area [25]. This is particularly important, as Apache Cordova applications do not support the notion of origins. Each application is executed in the origin *file://*, without specification of any host name or port number. By having their own storage areas, applications are not able to access other application's data items in Local Storage. The Local Storage of Apache Cordova applications is cleared when uninstalling the application from the device, manually calling the `clear()` method or the user clearing application data in the application settings[5].

[4] `http://caniuse.com/#search=webstorage`

[5] `http://stackoverflow.com/questions/15184567/is-local-storage-for-a-phonegap-app-on-an-android-device-separate-from-the-built`

WebSQL

WebStorage might be sufficient for locally storing simple keys and their respective values. However, WebStorage does not provide means for searching over values or retrieving keys in a pre-defined order. That is where classical database systems come in. The WebSQL API, supported by WebViews on Android and iOS devices, offers support for relational databases [33]. WebSQL represents a client-side database that supports classical SQL queries. Before using a WebSQL database, the developer has to define a schema specifying the layout of the database. Currently, WebSQL is broadly supported on mobile platforms[6]. However, in 2010 the W3C Working Group announced that the draft for the WebSQL specification is no longer maintained.

IndexedDB

The W3C has started working on an evolution of WebSQL, the so-called IndexedDB [53]. The IndexedDB specification has been finished and is available as W3C Recommendation since the beginning of 2015. IndexedDB represents a large SQL table filled with key-value pairs. When creating a new database, no schema has to be provided. Developers can store unstructured data, such as complex objects in the IndexedDB database. To enable more complex queries, IndexedDB allows developers to specify values as indexes in addition to the key. Therefore, IndexedDB supports queries over values as well. Currently, all major browsers support the IndexedDB API[7].

Summarising, each Apache Cordova application includes separate files that store the IndexedDB database and files for the Local Storage. Although these files are stored within the internal storage of the application and thus protected against access from other applications, attackers with root access or attackers with full access to the hard drive of the device may be able to retrieve the therein stored data. The presented storage options provide no specific data protection mechanisms.

In the following, storage options that are implemented on top of native device APIs and thus provided to Apache Cordova applications through plugins are discussed. All statements regarding the security of stored data equally apply to the previously described storage mechanisms, as these data stores are implemented as conventional files in the file system.

File System

Apache Cordova applications can access the mobile device's filesystem by using a plugin. The most popular plugin[8] is based on the interfaces defined by the W3C Working Draft File API [63]. The plugin allows developers to choose the desired storage location. Among others, for example, the application's internal storage, external storage (if available) or a cache folder.

In order to protect all data stored on the mobile device, mobile platforms offer file-system encryption. On iOS devices and the newest generation of Android devices, the encryption of the file system is enabled by default [13][30]. When using devices running Android version prior 5 or migrating from a lower Android version to version 5 or later, encryption has to be explicitly enabled. However, differences regarding the implementation of file-system encryption exist. In the following, the iOS and Android encryption system is described.

Apple iOS devices include a chip—further referred to as hardware element—that features a unique device key, the UID key. A key derived from the UID key is used to encrypt the data partition of the device. By including a hardware element in the encryption and decryption process of the file system, attempts to decrypt the file system have to be carried out on the device itself. It is not possible to access

[6] `http://caniuse.com/#search=websql`

[7] `http://caniuse.com/#search=indexeddb`

[8] `https://github.com/apache/cordova-plugin-file/blob/master/README.md`

data within file-system images that are constructed by cloning or extracting the device storage. This way, brute-force attacks on the key cannot be parallelised and thus are slowed down considerably. Apple's file-system encryption features a substantial weakness: the encryption system does not include the user's passcode. So, when applying a jailbreak, the attacker can bypass the device lock and, therefore, the lock screen password and access the decrypted file system.

To counterfeit this weakness, Apple has introduced an additional file-based encryption system called Data Protection. The Data Protection system is used on top of file-system encryption and provides additional protection for single files. The Data Protection system should be used by developers to protect sensitive data. The encryption keys are derived from the key in the hardware element and the user's lock screen password. When jailbreaking a device, the attacker gains access to the file system. However, files that have been protected using the Data Protection system can only be decrypted, if the attacker gains knowledge of the correct passcode. The Data Protection system is activated by setting a so-called Protection Class. The Protection Class specifies when to encrypt and decrypt a specific file. This is useful for scenarios where an application requires file access even when the device is locked.

In contrast to iOS, the Android file-system encryption solely relies on a key derived from the user's password. The user has to enter her password when booting the Android device. The password and a random salt value are used to derive a symmetric key. This derived key is then used to protect the file encryption master key. Android does not depend on a hardware element for file-system encryption. This way, attackers can attempt brute-force attacks on the password even off-device. When, for example, using several cloud instances, finding the correct password is sped up considerably. The security of Android's file-system encryption, therefore, heavily depends on the complexity of the chosen password. Teufl et al. have conducted a detailed analysis of brute-force times required to find the correct decryption key. For this listing, as well as for further information regarding the Android and iOS encryption systems we refer to the work of Teufl et al. [69][70][71].

For storing cryptographic key material and therefore arguably the most sensitive data, both platforms provide dedicated key-storage mechanisms. The following paragraphs shortly describe the mechanisms available on the Android and iOS platform.

Dedicated Key Storage

The iOS Data Protection system is also used to protect the entries within the iOS Keychain, a dedicated data store for cryptographic keys, passwords and other credentials [71]. There exist two main differences to Data Protection for files: Data Protection for the Keychain varies in the set of available Protection Classes. Furthermore, it introduces the possibility to specify whether the Keychain entry can be transferred to another device or is included in the backup. The security implications are equal to Data Protection for files. Hence, Keychain entries are protected in the case of a jailbreak and the security of credentials stored in the Keychain then mainly depends on the complexity of the chosen password.

The Android platform allows applications to use the so-called Keystore to store cryptographic keys and certificates [26]. Until recently, the Keystore could only handle private keys as used for asymmetric cryptography. Since Android version 6.0, the Keystore can be used for symmetric key material as well. Keys stored in the Keystore can be used for cryptographic operations, but the raw cryptographic key cannot be exported from the Keystore. The keys stored in the Keystore are only accessible to the application itself, and other applications cannot access those Keystore entries.

Different variants of the Android Keystore exist: depending on the device, the implementation can either be software- or hardware-based [70]. Android provides an API call in order to enable developers to check whether the mobile device features a secure hardware Keystore. For the software-based implementation, an AES key is derived from the user's passcode. The passcode is the same passcode as used for Android's file-system encryption and entered during the boot process. The password-derived key is used to encrypt a master key that protects all Keystore entries. For hardware-based implementations, a master key resides inside secure hardware (e.g., Trusted Execution Environment or Secure Element). The Keystore entries are stored on the file system and encrypted with the master key from the hardware element. The master key in the hardware element is additionally protected with the password-derived key.

Depending on the implementation, a successful brute-force attack on the user's passcode has different implications: for software-based implementations, the attacker can decrypt the keys, whilst for hardware-based implementations, the master key cannot be extracted from the hardware element and thus the attacker cannot decrypt the Keystore entries. However, the attacker may still be able to use the keys stored in the Keystore on the specific mobile device.

Summary

Summarising, developers of Apache Cordova applications have various options for storing data locally on the mobile device. For small amounts of simple data structures, Local Storage seems to be a good choice. Local Storage is easy to use for developers, as it requires no knowledge about the underlying mobile platform and omits the overhead for setting up a database scheme. It might be feasible to protect sensitive data by deploying encryption using a key derived from a user-supplied password. However, the implementation of such protection mechanisms can be quite erroneous (e.g., using too few iterations for the password-based key derivation or relying on a fixed or low-entropy salt value). Thus, Local Storage should not be used for storing sensitive data. When storing sensitive data, it is best to rely on the platform features and use the provided key-storage mechanisms. Apache Cordova allows the use of plugins that make native device APIs available to the application running in the WebView. For our authentication framework, we have implemented a plugin that accesses the Android Keystore and the iOS Keychain. For more information, we refer to Chapter 6.

4.2.3 Sensors

In addition to push notifications and the storage of data locally on the mobile device, devices offer a multitude of features and sensors. Modern mobile devices feature communication technologies such as NFC and Bluetooth. These communication technologies can, for example, be used to communicate with a hardware token such as the YubiKey NEO. Such tokens offer key-storage facilities and act as a second factor during authentication.

Modern mobile devices feature a camera. The camera of the mobile device can be used to scan a QR code or any other type of barcode. The use of QR codes has emerged to a popular way of transferring small amounts of data and therefore, represents an easy-to-use communication link. The Google Authenticator application, for example, uses QR codes for transferring the secret key to the mobile device.

Among other sensors, mobile devices include GPS sensors that allow to determine the device's current position. Inspired by Geo Control and Geo Blocking features offered by banks and credit card companies[9], these sensors might be used to restrict the acceptance of authentication on a geographical basis and discover anomalies in the usage habits of the user.

[9] https://www.six-payment-services.com/financial-institutions/en/shared/success-stories/geoblocking.html

Chapter 5

Authentication Methods

Mobile devices offer a multitude of different features and sensors. Based on the building blocks presented in Chapter 4 this chapter presents authentication methods that have been constructed in the course of this thesis. The constructed authentication methods have been integrated into the developed authentication framework, which will be presented in the next chapter.

5.1 Terminology

For the sake of clarity, we first introduce terms that are heavily used throughout this chapter. The *prover* denotes a user that wants to authenticate. Therefore, the prover uses the *authentication app* that implements different authentication methods. The *authentication process* describes the set of *authentication methods* that is required to state whether the prover is genuine. The *verifier* describes the web service or identity provider the prover is authenticating at.

Furthermore, we are operating on the assumption that each successful authentication process relies on a combination of proving possession and proving knowledge to the verifier. Therefore, our analysis assumes that the user first identifies to the web service. This can, for example, be realised by prompting the prover for her username or some other identification. As a next step, the prover proves the factors knowledge and possession. The factor knowledge can already be added to the identification process if the web service prompts the prover for her username and password or can be provided during the possession proof. Biometrics are not considered, as secure biometric verification is not yet widely deployed on mobile devices. In addition, attacks on biometric systems deployed on current mobile devices are known [82]. However, we do not rule out biometric verification for future implementations of authentication methods.

5.2 Overview

After having extracted the building blocks of authentication methods and having presented some features available to modern mobile devices, we continue with the presentation of authentication methods that have been implemented using our authentication framework.

So far, the focus is put on the implementation of authentication methods using OTPs. Therefore, we reuse the algorithms defined by [56] and [57]. However, we have added new variations regarding the protection of the used key and changing challenges. We have chosen to start with variants of authentication methods using OTPs, as those are already widely deployed by industry. In addition, these methods require only a few adaptations to integrate them with existing SMS-based TAN systems.

For each implemented authentication method, its usage scenario and requirements towards the authentication method are stated. We proceed with the registration workflow and steps involved in a typical authentication process. From this overview, we assess how the defined requirements have been met and give a short disclaimer on security implications of the implemented authentication method. The detailed security evaluation of all authentication method follows in Chapter 7.

5.3 Triple Key AES OTP

The first of the implemented authentication methods requires three different cryptographic keys to compute the correct OTP. It relies on a key sent by push notification, a key stored on the mobile device and a key encoded in a QR code that has to be scanned during the authentication process.

5.3.1 Usage Scenario

This authentication method uses the mobile device to prove the factor possession during authentication. The prover supplies username and password at a desktop computer and uses her device to scan a QR code and receive a push notification to compute the result to the challenge. In total, three cryptographic keys are required to correctly compute the response. This authentication method is intended as a replacement for SMS-based TAN systems.

5.3.2 Requirements

For this authentication method the following requirements should be considered:

(a) The authentication method should protect against cloning. Hence, an attacker should not be able to equip a second mobile device with the capabilities to compute a correct response to the challenge.

(b) The authentication method should employ mechanisms for device binding. Thus, even if all data is cloned to a second device or an attacker extracts the device storage, it is not possible to complete authentication.

(c) This authentication method should not require additional password input on the mobile device, as the password (factor knowledge) is already entered on the desktop system.

(d) The authentication process must not be performed in the background. Thus, it is required that the prover triggers the authentication method or sets some action to complete authentication.

5.3.3 Workflow: Registration

The following paragraph summarises the necessary steps for setting up the Triple Key AES OTP authentication method. We assume that the mobile application has already registered with the platform's push-notification service and hence locally stores the obtained registration ID or device token. We will further refer to the device token as used by iOS devices and the registration ID as used by Android as *Push ID*.

1. The server-side application randomly generates a 256-bit symmetric key. This cryptographic key and some metadata are encoded in the form of a QR code and displayed to the prover.
2. The prover opens the authentication app on the mobile device and adds a new so-called account[1] by scanning the provided QR code.

[1] The authentication app on the mobile device allows for adding multiple accounts. An account denotes an authentication method for a specific web service.

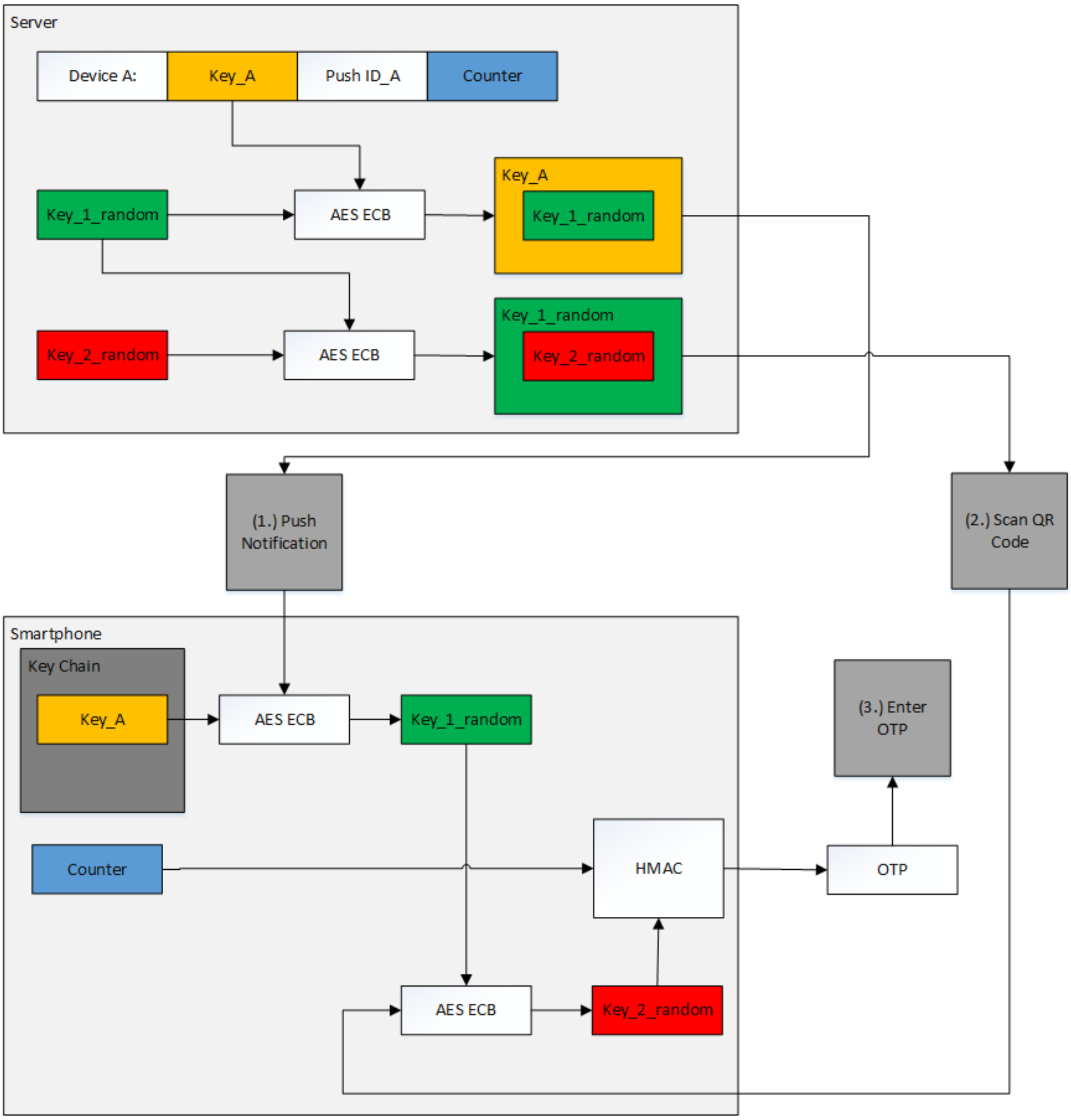


Figure 5.1: Workflows and cryptographic keys involved in the authentication method Triple Key AES OTP.

3. The authentication app on the mobile device decodes the data from the QR code. The symmetric key encoded in the QR code is stored persistently on the mobile device using the device's key-storage facilities.

4. The metadata transferred in the QR code includes a response URL, to which the mobile authentication app sends the *Push ID* obtained by the platform's push-notification service. The *Push ID* has been obtained by the mobile application before and stored securely using the key-storage facilities.

5. After receiving the *Push ID*, the server is able to send push notifications to the mobile device.

5.3.4 Workflow: Authentication

In the following, we describe a typical authentication workflow. We assume that the authentication method has been set up correctly. A 256-bit secret key (*Key_A*) is stored in the device's key storage and the push-notification system has been set up for this application. Hence, the *Push ID* has been transmitted to the server. Figure 5.1 illustrates the workflow.

1. The application on the server side generates two random keys with 256-bit length: *Key_1_random* and *Key_2_random*. For the key generation a secure pseudo-random number generator (PRNG) is used.

2. *Key_1_random* is encrypted with *Key_A* using the AES algorithm in Electronic Codebook (ECB) mode. ECB mode lacks an initialisation vector, therefore, identical plaintext result in the same ciphertext. Furthermore, ECB mode allows attackers to detect patterns in the ciphertext. However, as we encrypt random sequences of bytes without any reoccurring patterns, it is justifiable to use AES in ECB mode.

3. *Key_2_random* is encrypted using *Key_1_random* with AES in ECB mode.

4. The encrypted *Key_1_random* is transmitted via the mobile platform's push-notification system to the authentication app on the mobile device.

5. The encrypted *Key_2_random* is encoded in the form of a QR code and displayed in the prover's desktop browser.

6. The received push notification triggers the correct user interface for the authentication method, which includes user instructions how to proceed the authentication process.

7. The prover scans the provided QR code.

8. As the authentication app now obtained all three keys, the decryption process starts. *Key_A*, which is stored in the key store of the mobile device is used to decrypt the encrypted *Key_1_random*. The decrypted *Key_1_random* is used to decrypt the encrypted *Key_2_random*.

9. The authentication app computes an Hashed Message Authentication Code (HMAC) code using *Key_2_random* and the counter stored on the mobile device as input. Based on the obtained HMAC code, the required 6-digit OTP is computed according to RFC 4226 [56].

10. The prover enters the computed OTP at the web form provided by the server.

11. The server uses *Key_2_random* and the counter stored for this prover to compute the OTP. If both OTPs match, the prover has proven the factor possession.

5.3.5 Assessment

This section briefly summarises implications of the Triple Key AES OTP method and, therefore, assesses how the defined requirements have been met. For the detailed security evaluation, we refer to Chapter 7.

The authentication method Triple Key AES OTP uses the prover's mobile device as the second factor during authentication. Thereby, the user proves possession of the mobile device. It is crucial that attackers cannot clone the device covering the second factor. Therefore, this authentication method uses device binding in the form of a combination of cryptographic key material stored on the device and the platform's push-notification system.

By using the hardware-backed key-storage facilities available to some platforms, attackers cannot easily copy the cryptographic key material (*Requirement (a)*). Even if the cryptographic key falls in the hand of an attacker, the attacker needs to receive both the key from the QR code and the key sent as push notification to successfully complete the authentication method. The *Push ID* from the push-notification system (*Requirement (b)*) uniquely identifies an application on a specific mobile device. Hence, a second device does not receive notifications intended for the legitimate receiver. Summarising, push notifications can only be received by the device, whose *Push ID* is known to the server application. Special focus has to be put on assuring that the *Push ID* cannot be overwritten on the server side.

Only by getting hold of all three keys, the correct OTP can be computed. Thus, neither sniffing the push notification key nor accessing only the key stored persistently on the device allows the attacker to authenticate.

The prover does not have to enter an additional password on the mobile device (*Requirement (c)*). It is recommended, however, that the device holder protects her device against unauthorised access by setting a lock-screen password. On some platforms, the lock-screen password acts as input to platform-specific encryption mechanisms. For more information we refer to Section 4.2.2.

Although the prover does not have to express consent by providing a password, the authentication method requires explicit user action. This has been realised by requiring the prover to manually scan the QR code and further enter the computed OTP on the desktop computer (*Requirement (d)*).

5.4 Triple Key AES OTP with Knowledge Proof

We extend the Triple Key AES OTP method with a proof of knowledge. By employing password-based key derivation, we ensure that the prover has entered the correct password on the mobile device.

5.4.1 Usage Scenario

This authentication method combines a possession proof and the factor knowledge. The user proves possession similar to the Triple Key AES OTP method by using a key stored in the device's key storage and by assuring device binding by using the mobile platform's push-notification system. In addition to scanning a QR code, the prover has to enter a password on the mobile device. As a consequence, the authentication service can make sure that only legitimate persons operate the mobile device covering the second factor. This authentication method is intended as a replacement for SMS-based TAN systems. Although, entering a password in the web browser can be omitted, the prover has to identify towards the authentication service. The identification of the prover is necessary for sending the push notification to the desired mobile device.

5.4.2 Requirements

For this authentication method the following requirements should be considered:

(a) The authentication method should protect against cloning. Hence, an attacker should not be able to equip a second mobile device with the capabilities to compute a correct response to the challenge.

(b) The authentication method should employ mechanisms for device binding. Thus even if all data is cloned to a second device or an attacker extracts the device storage it is not possible to complete authentication.

(c) The authentication method should employ mechanisms to ensure that the legitimate prover is operating the device covering the second factor. Therefore, the server application has to ensure that the prover has entered a password on the mobile device.

(d) The authentication process must not be performed in the background. Thus, it is required that the prover triggers the authentication method or sets some action to complete authentication.

(e) The authentication method should allow transaction binding. Meaning, it should be possible to map the sent challenge to a specific authentication transaction.

5.4.3 Workflow: Registration

In the following, the necessary steps for setting up this authentication method are summarised. Again, we assume that the mobile application has already registered with the platform's push-notification service and hence locally stores the obtained *Push ID*.

1. The server-side application randomly generates a 256-bit symmetric key. Furthermore, a 128-bit random salt is generated. The cryptographic key, the salt and some metadata are encoded in the form of a QR code and displayed to the prover.

2. The server-side application prompts the prover to enter a password. The previously computed salt and the password act as input for the password-based key derivation function SCrypt [68]. The derived key is stored on the server-side, whereas the entered password is not stored.

3. The prover opens the authentication app on the mobile device and adds a new account by scanning the provided QR code.

4. The authentication app on the mobile device decodes the data from the QR code. The symmetric key and the salt value encoded in the QR code are both stored persistently on the mobile device using the device's key-storage facilities.

5. The metadata transferred in the QR code include a response URL, to which the mobile authentication app sends the *Push ID* obtained by the platform's push-notification service. The *Push ID* has been obtained by the mobile application before and stored securely using the key-storage facilities.

6. After receiving the *Push ID*, the server is able to send push notifications to the mobile device.

5.4.4 Workflow: Authentication

In the following, we describe a typical authentication workflow. We assume that the authentication method has been set up correctly. Hence, the 256-bit secret key (*Key_A*) and the 128-bit *salt value* are stored in the device's key storage and the push notification system has been set up for this application. Hence, the *Push ID* has been transmitted to the server). Both, the key derived from the prover's password (*Key_PW_Derived*) and the persistent device key (*Key_A*) are stored on the server side. Figure 5.2 illustrates the involved cryptographic keys and operations.

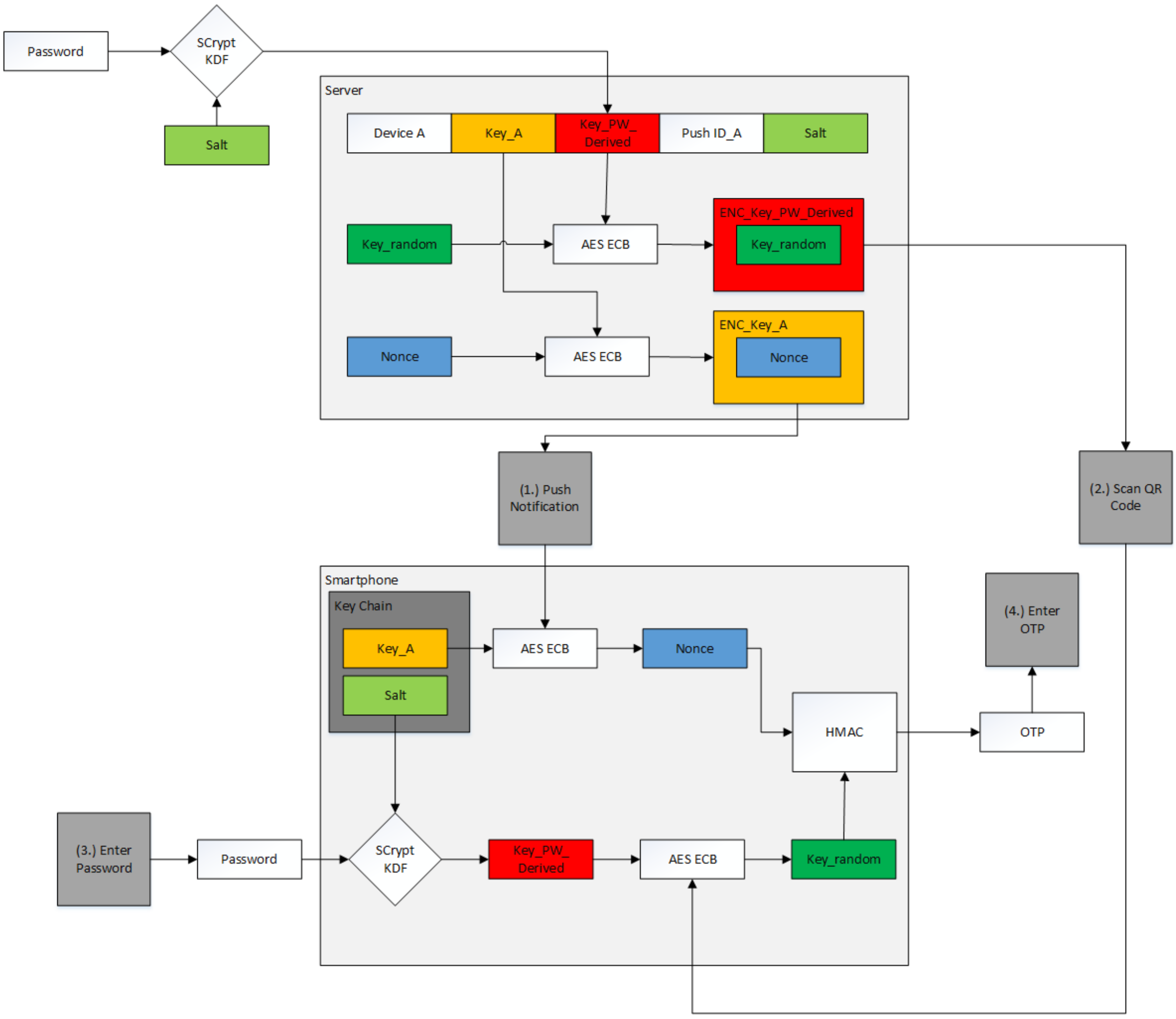


Figure 5.2: Workflows and cryptographic keys involved in the authentication method Triple Key AES OTP with Knowledge Proof.

1. The application on the server side generates the random 256-bit *Key_random* and the random number *nonce*. Therefore, a secure PRNG is used. The *nonce* represents a transaction-specific challenge.

2. *Key_random* is encrypted with *Key_PW_Derived* using the AES algorithm in ECB mode.

3. The random *nonce* is encrypted using the persistent device key *Key_A* with AES in ECB mode.

4. The encrypted *nonce* is transmitted via the mobile platform's push-notification system to the authentication app on the mobile device.

5. The encrypted *Key_random* is encoded in the form of a QR code and displayed in the user's desktop browser.

6. The received push notification triggers the correct user interface for the authentication method, which includes user instructions.

7. The prover scans the provided QR code and is prompted for her password.

8. Using the entered password and the *salt* stored on the mobile device, *Key_PW_Derived* is derived. The encrypted *Key_random* is decrypted using the password-derived key.

9. *Key_A*, which is stored in the key store on the mobile device is used to decrypt the encrypted *nonce* from the push notification.

10. The authentication app computes an HMAC code using *Key_random* and the decrypted *nonce*. Based on the obtained HMAC code, the required 6-digit OTP is computed according to RFC 4226 [56].

11. The prover enters the computed OTP at the web form provided by the server.

12. The server uses *Key_random* and the transaction-specific *nonce* to compute the OTP. If both OTPs match, the user has proven the factor possession.

5.4.5 Assessment

The authentication method Triple Key AES OTP with Knowledge Proof uses the prover's mobile device to provide a combination of knowledge and possession proof. This authentication method complies with *Requirement (a)* and *Requirement (b)* by storing sensitive data using the device's hardware-backed key-storage facilities and employing additional device binding through the mobile platform's push-notification service.

To compute the correct OTP, the prover requires the persistent key stored on the device, the nonce encoded in the push notification, the random key encoded in the QR code and a key derived from the prover's password. Only if all of these values are correct, the correct OTP can be computed. This way, the authentication service can ensure that the correct password has been entered on the mobile device (*Requirement (c)*).

By requiring a password on the mobile device and scanning a QR code, the authentication method ensures the prover's explicit consent for a transaction (*Requirement (d)*).

In order to meet *Requirement (e)*, we use a nonce provided from the server. Exchanging the supplied random value with e.g. a transaction-specific identifier, the first part of a hash value or some other token allows for a more concrete transaction binding.

Chapter 7 evaluates the security implications of this combined proof of possession and knowledge in detail.

5.5 Double Key AES OTP with Knowledge Proof

This section describes a variation of Triple Key AES with Knowledge Proof, which is designed for the mobile-only use. It requires two cryptographic keys and a random nonce for computing the correct OTP. One key is stored in the key store on the mobile device and the other key is derived from a user-supplied password. The password-derived key is used to decrypt the random nonce sent via the mobile platform's push-notification service.

5.5.1 Usage Scenario

This authentication method presents a variation of Triple Key AES with Knowledge Proof without requiring a separate desktop computer for conducting authentication. Therefore, the user proves possession by using a key stored in the device's key storage and by assuring device binding by using the mobile platform's push-notification system. As the prover has to enter a password on the mobile device, the authentication service can make sure that only legitimate persons operate the mobile device covering the second factor. This authentication method is intended as a replacement for SMS-based TAN systems. The goal is to provide an authentication method that can be triggered within the web browser on the mobile device and further allows to perform the entire authentication process on the same mobile device.

5.5.2 Requirements

For this authentication method the following requirements should be considered:

(a) The authentication method should protect against cloning. Hence, an attacker should not be able to equip a second mobile device with the capabilities to compute a correct response to the challenge.

(b) The authentication method should employ mechanisms for device binding. Thus even if all data is cloned to a second device or an attacker extracts the device storage it is not possible to complete authentication.

(c) The authentication method should employ mechanisms to ensure that the legitimate user is operating the device covering the second factor. Therefore, the server application has to ensure that the prover has entered a password on the mobile device.

(d) The authentication process must not be performed in the background. Thus, it is required that the prover triggers the authentication method or sets some action to complete authentication.

(e) The authentication method should allow transaction binding. Meaning, it should be possible to map the sent challenge to a specific authentication transaction.

(f) Furthermore, this authentication method should be suitable for mobile-only scenarios, where no web browser on a separate computer is used during the authentication process.

5.5.3 Workflow: Registration

In the following, the necessary steps for setting up this authentication method are summarised. Again, we assume that the mobile application has already registered with the platform's push-notification service and hence locally stores the obtained *Push ID*. Although this authentication method is designed for the mobile-only use case, a web browser on a separate computer is required for registration.

1. The server-side application randomly generates a 256-bit symmetric key. Furthermore, a 128-bit random salt is generated. The cryptographic key, the salt and some metadata are encoded in the form of a QR code and displayed to the prover.

2. The server-side application prompts the prover to enter a password. The previously computed salt and the password act as input for the password-based key derivation function SCrypt. The derived key is stored on the server-side, whereas the entered password is not stored.

3. The prover opens the authentication app on the mobile device and adds a new account by scanning the provided QR code.

4. The authentication app on the mobile device decodes the data from the QR code. The symmetric key and the salt value encoded in the QR code are both stored persistently on the mobile device using the device's key-storage facilities.

5. The metadata transferred in the QR code includes a response URL, to which the mobile authentication app sends the *Push ID* obtained by the platform's push-notification service. The *Push ID* has been obtained by the mobile application before and stored securely using the key-storage facilities.

6. After receiving the *Push ID*, the server is able to send push notifications to the mobile device.

5.5.4 Workflow: Authentication

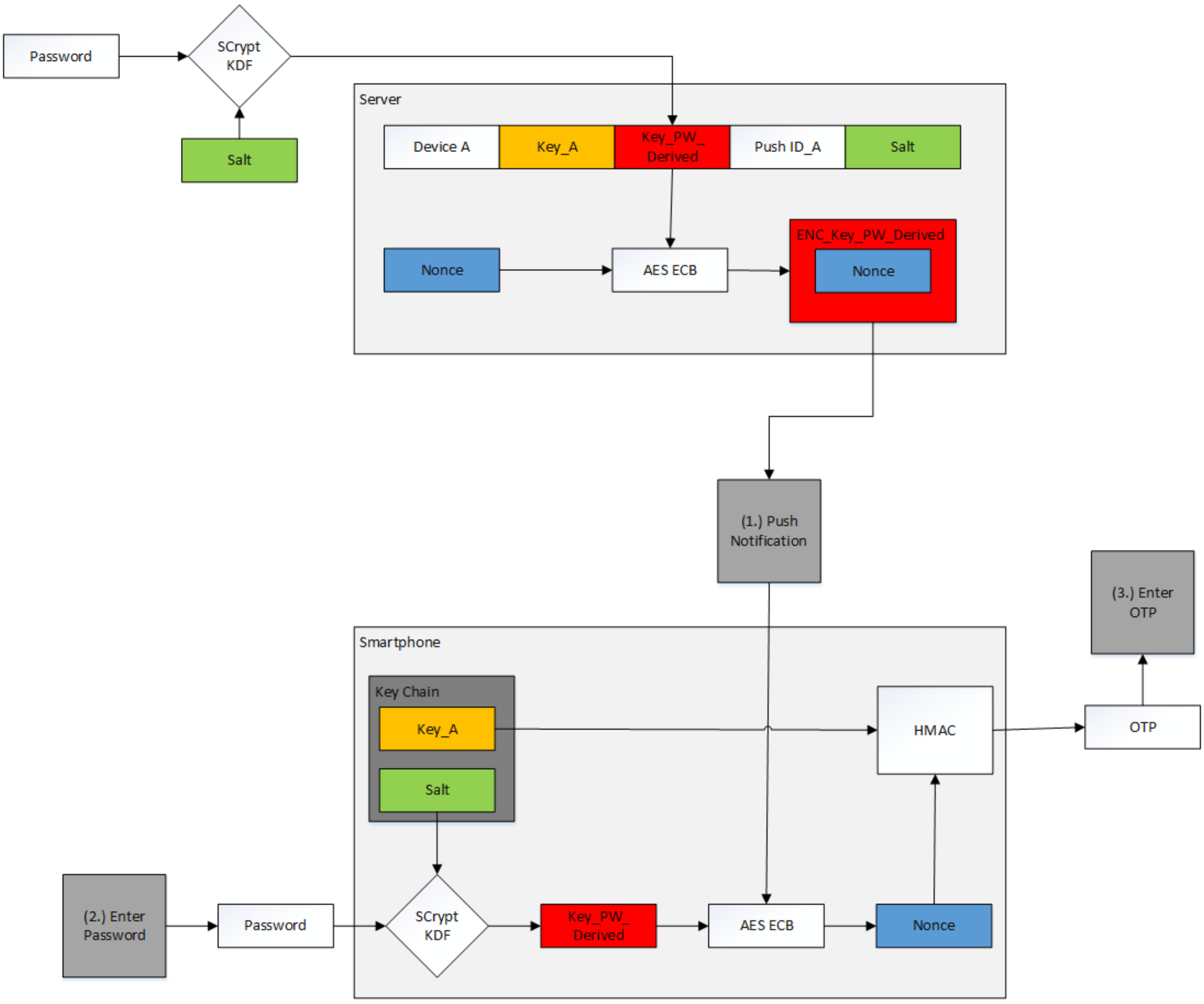


Figure 5.3: Workflows and cryptographic keys involved in the authentication method Double Key AES OTP with Knowledge Proof.

In the following, we describe a typical authentication workflow. We assume that the authentication method has been set up correctly. Hence, the 256-bit secret key (*Key_A*) and the 128-bit *salt* value are stored in the device's key storage and the push-notification system has been set up for this application. Both, the key derived from the prover's password (*Key_PW_Derived*) and the persistent device key (*Key_A*) are stored on the server side. Figure 5.3 illustrates the involved cryptographic keys and operations.

1. The application on the server side generates the random number *nonce*. Therefore, a secure PRNG is used. The *nonce* represents a transaction-specific challenge.
2. The random *nonce* is encrypted with *Key_PW_Derived* using the AES algorithm in ECB mode.
3. The encrypted *nonce* is transmitted via the mobile platform's push-notification system to the authentication application on the mobile device.
4. The received push notification triggers the correct user interface for the authentication method.
5. The application prompts the prover for her password.
6. Using the entered password and the *salt* stored on the mobile device, *Key_PW_Derived* is derived. The encrypted *nonce* is decrypted using the password-derived key.
7. The authentication app computes an HMAC code using *Key_A*, which is stored in the key store of the mobile device and the decrypted *nonce*. Based on the obtained HMAC code, the required 6-digit OTP is computed according to RFC 4226 [56].
8. The prover enters the computed OTP at the web form provided by the server.
9. The server uses the stored *Key_A* and the transaction-specific *nonce* to compute the OTP. If both OTPs match, the user has proven the factor possession.

5.5.5 Assessment

The here presented authentication method complies with *Requirement (a)* and *Requirement (b)* by storing sensitive data using the device's hardware-backed key-storage facilities and employing additional device binding through the mobile platform's push-notification service.

This authentication method requires the prover to enter a password. Therefore, the service can make sure that only the legitimate user operates the device (*Requirement (c)*) and that she expresses explicit consent for performing the transaction (*Requirement (d)*). Transaction binding (*Requirement (e)*) is achieved by using a transaction-specific nonce. As no QR code has to be scanned during authentication, this authentication method can be conducted without using a desktop computer (*Requirement (f)*). Solely for registration, a separate computer has to be used.

Chapter 6

A Flexible Cross Platform Multi-Factor Authentication Framework

As a practical part of this thesis, we have implemented an authentication framework based on cross-platform technologies. This chapter introduces the implemented framework. The framework consists of a server and a client part. For the server part, an existing framework has been used, whilst the mobile client part has been implemented from scratch. After a short presentation of the used authentication server, focus is put on the mobile client. For this purpose, the architecture of the mobile client is illustrated and the implemented components are presented. We conclude with the steps required to integrate new authentication methods into our framework.

6.1 Multi-Factor Authentication Server

The main focus of this thesis was to employ state-of-the-art cross-platform development techniques in a security-critical environment. Therefore, we have chosen to rely on existing components for the server part in order to focus rather on the mobile components. For the server part of the authentication framework, a newly-developed multi-factor authentication service provided by the E-Government Innovation Center (EGIZ)[1] has been used. This authentication server enables the use of third-party authentication plugins. These authentication plugins can be registered dynamically at the authentication server. During user authentication, the authentication server chooses suitable multi-factor authentication methods based on dynamic policies. The following enumeration lists the core features of the used authentication server.

- *Plugin-Based Approach*

  The authentication server allows the usage of external authentication plugins. These plugins handle user authentication and issue a signed JSON Web Token [42] if the user has authenticated successfully at the authentication plugin.

- *Configurable Domains*

  The authentication server supports the notion of different domains. A domain comprises, for example, a set of web services of a particular company or web services associated with a particular business segment.

- *Dynamic Authentication Policies*

  In order to find a combination of different authentication methods, the authentication server features dynamic authentication policies. These policies state requirements towards valid user-authentication methods for a specific domain. The policies are not limited to the specification of a

[1] `https://www.egiz.gv.at/`

single valid authentication method but can also enforce a combination of different authentication methods. Therefore, each authentication plugin is assigned a plugin type that states, for example, if the plugin covers the factor knowledge or the factor possession.

- *Unified Configuration Interface*

  The authentication server offers one single entry point for initialising authentication plugins for the user. The user can activate and manage her authentication plugins. In order to access this configuration platform, the user has to login using strong credentials. In practice, this has been realised by using an accredited electronic identity (eID), the Citizen Card offered by the state of Austria [47].

- *Protected Configuration Data*

  Each external authentication plugin requires configuration data for a specific user and a particular domain. This configuration data is stored centrally at the authentication server and retrieved from the authentication plugin when needed. As this data might comprise sensitive user data (e.g. a password or cryptographic key), the data is encrypted and signed before being transferred to the authentication server. Hence, configuration data is protected against unauthorised access and manipulation by unauthorised parties.

To demonstrate the authentication methods presented in Chapter 5, a server-side authentication plugin for each authentication method that has been integrated into our mobile authentication app has been developed. For the mobile multi-factor authentication framework, we use a plugin-based approach as well. The following section presents the implemented mobile multi-factor authentication framework. Section 6.3 describes the steps required to integrate a new authentication plugin both on the mobile and on the server-side.

6.2 Mobile Multi-Factor Authentication Client

This section illustrates the chosen architecture of the mobile part of the implemented multi-factor authentication framework. Moreover, a short description of the implemented components and the technologies used is given.

6.2.1 Architecture

The idea of our authentication framework is to ease the integration of newly developed authentication methods. Therefore, the framework offers the basic functionality as for example cryptographic methods, access to remote notifications and device features. The different device features and mechanisms have been split into services. A service, for example, comprises methods for deriving a cryptographic key from a password or accessing the device's key-storage facilities.

Each authentication method is implemented by an authentication plugin. An authentication plugin can make use of the offered services and only has to combine the different services to construct the authentication method. The authentication app supports multiple instances of the same authentication method, however a specific authentication method can only exist once per domain. For example, the user might have added three different domains which all provide the same authentication method but with different cryptographic key material.

In order to support different authentication plugins, we first had to extract components that differ between various authentication plugin implementations. First of all, each authentication method requires distinct configuration data that has to be stored locally on the device. In addition, when setting up a new authentication method, communication with the authentication service running on a remote server might

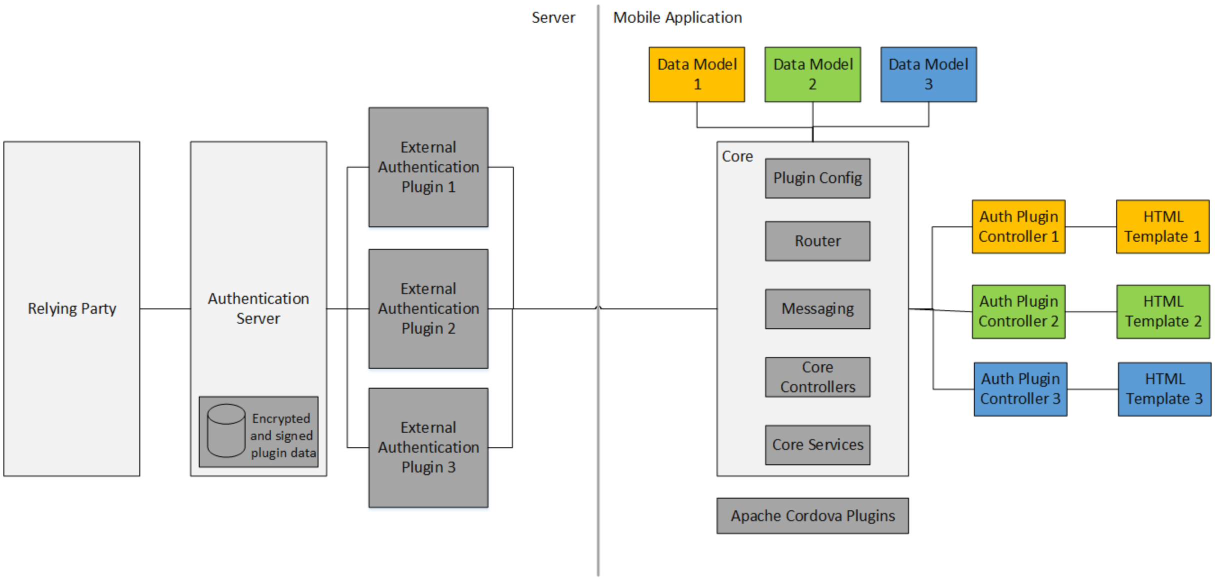


Figure 6.1: The general architecture of the implemented authentication framework. Components specific to a single authentication method have been highlighted.

be required. Therefore, each authentication plugin has to provide a data model and a corresponding configuration method that performs all steps required during the rollout of a new authentication method to the prover's mobile device.

Further differences exist during the authentication process itself. Each authentication plugin requires different steps and distinct user interaction for performing the authentication method. Thus, we had to enable authentication plugins to provide their own user interface and the corresponding business logic that aims to compute the response to a received challenge.

We tried to keep the effort for integration of new authentication methods minimal. A developer seeking to integrate a new authentication plugin has to provide a data model for her plugin, a simple HTML template, which acts as an entry point to the authentication method and the corresponding controller. Figure 6.1 shows the general architecture and highlights components specific to an authentication plugin. More information on the implementation steps required to integrate a new authentication plugin can be found later in this chapter.

Communication between the server and the mobile client is processed within the authentication app. For example, the message receiver within the authentication app handles all incoming push notifications. We have defined an own messaging format for the message exchange between the server and the mobile client. This message format includes the name of the authentication plugin and the current domain. In order to implement the routing mechanism to the correct authentication plugin, all sent messages have to comply to this format. Hence, this message format is also used for the rollout of a new authentication method to the mobile device. Currently, during rollout messages are encoded in a QR code. However, other communication paths are possible as well and it might be feasible to implement an input form for the user to manually enter the authentication plugin's initialisation data.

Communication between the mobile client and the server has to be handled by the respective implementation of the authentication plugin. This communication includes the transfer of the *Push ID* or the response to a challenge.

6.2.2 Implementation

The mobile part of the authentication framework has been implemented as a hybrid mobile application. Therefore, most parts of the application are implemented using web technologies such as HTML5, JavaScript and CSS and are run within a WebView. Features not available to the WebView are implemented using native technologies as so-called plugins. Plugins provide a JavaScript interface and thus can be interfaced from within the WebView. Apache Cordova enables this plugin mechanism and provides the packaging of web applications as native mobile applications.

When implementing hybrid mobile applications, providing a user experience similar to native mobile applications, poses the main challenge. For the user interface design, we rely on the Ionic framework[2]. Ionic is an HTML5 SDK that allows for building native-feeling mobile applications using web technologies. Therefore, Ionic bundles compiled JavaScript and CSS files and some resources such as icons. Ionic aims to fill the gap between HTML5 and native application development [1]. Ionic builds on the popular JavaScript framework AngularJS[3]. AngularJS addresses the challenges encountered when implementing single-page applications. AngularJS extends the HTML5 vocabulary to provide data binding and facilitate a slick model-view-controller architecture. For more information on AngularJS and Ionic we point to the official documentation [27][38].

Besides secure storage of cryptographic key material or connecting to the platform's push-notification service, multi-factor authentication requires a huge set of different device features and particularly cryptographic methods. In order to cover the required functionality we had to rely on various Apache Cordova plugins and even had to develop our own plugins. However, the developed or extended plugins can be easily integrated into other Apache Cordova applications as well.

In the course of this work, we have mainly developed the components required for the integration of authentication methods presented in Chapter 5. However, the developed components can easily be combined to create new authentication methods or to be used in other hybrid mobile applications with higher security demands. In general, the focus has been put on providing extensible components that can be reused in other hybrid mobile applications. This does not only apply to Apache Cordova plugins but also to the developed JavaScript components.

The mobile authentication app covers Android and iOS devices. In order to support other platforms as well, some of the used Apache Cordova plugins have to be adapted. According to Net Marketshare[4], Android holds a market share of 54 percent and iOS 39 percent. By developing for these two platforms, the vast majority of mobile devices is covered. For this reason, we decided that support for Windows Phone devices and other mobile platforms is subject of future work.

In the following, a presentation of the developed components is given. First, we describe the implemented AngularJS components. Second, the required Apache Cordova plugins are listed. The described plugins fill the gap between functionality available within the WebView and features only accessible via native device APIs.

AngularJS Components

The goal of the authentication framework was to provide components heavily used within multi-factor authentication. AngularJS relies on dependency injection to describe how the application is wired. Therefore, we have chosen to implement large parts of the framework as AngularJS services that can be easily integrated with new components. This way, the framework should simplify the integration of new authentication methods. In the following, the developed AngularJS services are presented.

- *KeychainService*

  In the course of this thesis, we have implemented an Apache Cordova plugin that provides access to the underlying platform's key-storage facilities. An AngularJS service has been implemented to

[2] `http://ionicframework.com/`

[3] `https://angularjs.org/`

[4] Net Marketshare provides statistics based on data collected by analysing the browser access statistics. The statistics can be found at the following URL: `https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1`

ease access to the developed Apache Cordova plugin. It provides methods to check the availability of the implementation on the respective mobile device and methods to store and retrieve cryptographic keys represented as Base64-encoded strings. For more information on the developed Apache Cordova plugin, we refer to the ensuing plugin description.

- *OTPService*

  All of the currently integrated authentication methods rely on the computation of OTPs. The *OTPService* provides OTP computation according to the HOTP [56] and TOTP [57] standard. Currently, this implementations uses the CryptoJS cryptography library for the underlying cryptographic operations. We chose to use CryptoJS here, as support for the different cryptographic functions of the Web Cryptography API [65] strongly depends on the particular version of the mobile platform. This choice seemed viable at the beginning of this work. However, support for the Web Cryptography API continuously evolves. Therefore, future work should prefer the use of the Web Cryptography API built in the mobile browser.

- *KeyDerivationService*

  Some authentication methods require the user to present some knowledge (e.g. a password) on the mobile device. Key derivation functions, such as SCrypt [68], are used to derive a cryptographic key from the user's password. The *KeyDerivationService* provides an interface to the SCrypt implementation offered by the JavaScript library *js-scrypt*[5].

- *CryptoService*

  The integrated authentication methods heavily rely on the entanglement of multiple cryptographic keys. In order to be able to compute the correct OTP, a set of cryptographic keys has to be obtained. These cryptographic keys are used to decrypt the key used for OTP computation. The *CryptoService* provides methods for encrypting and decrypting using AES with ECB mode. ECB mode has been used, as no additional data such as an initialisation vector or other parameters have to be exchanged between the server and the mobile authentication app. The use of the *CryptoService* is thus limited to the encryption and decryption of cryptographic keys, which represent randomly generated sequences of data. The use of ECB mode is only justifiable as the encrypted data will not show a repetitive pattern.

- *ConfigService*

  The *ConfigService* holds general configuration parameters for the mobile platform's push-notification system. It further provides methods to store and retrieve the obtained *Push ID*. The *Push ID* should be protected against disclosure to non-authorised parties, such as malicious applications on the mobile device or an attacker analysing data stored on the device. On this account, the *KeychainService* is used to store the *Push ID* by utilizing the mobile platform's key-storage facilities.

- *PluginDataService*

  With our framework, each authentication plugin requires a set of data locally stored on the device. This data includes the name of the authentication plugin, the domain it is associated with and prospective aliases used to address cryptographic key material or salt values that have been stored securely using the *KeychainService*. The *PluginDataService*, therefore, uses the Local Storage API provided by the browser to store the plugin data in the application-internal storage area.

  The authentication framework introduces `PluginData` objects that define the basic structure of the authentication plugin data. For each authentication plugin, an implementation that inherits from `PluginData` has to be provided. This customized `PluginData` is used to interface with the *PluginDataService*.

[5] `https://github.com/tonyg/js-scrypt`

- *MessageDispatcherService*

  Whenever a new remote message arrives, the *MessageDispatcherService* routes the message to the correct authentication plugin within the mobile authentication app. Currently, the *MessageDispatcherService* is only used in conjunction with the mobile platform's push-notification service. However, it is not limited to any particular technology. Each message represents a JSON message that follows some general format. For example, the message has to contain the name of the authentication plugin, as well as the domain the authentication should be performed for. Based on the name of the authentication plugin, the correct template of the authentication plugin is loaded. Furthermore, the service is used to hand the received message to the controller of the loaded template.

Even though different authentication methods require different steps to fulfil authentication and thus differ regarding their user interface, some common parts can be extracted. For these user-interface parts, we decided to offer so-called *partials*, i.e. HTML templates that are embedded in other templates. For example, our framework offers a partial that displays the computed OTP. The framework also includes the accompanying AngularJS controller implementation that performs the computation of the OTP.

Apache Cordova Plugins

Apache Cordova applications are restricted to functionality offered by the WebView component on the mobile device. The use of functionality beyond the browser APIs requires the use of plugins that provide the platform-specific implementation. The developed authentication app requires several plugins to cover the functionality of the different authentication methods. In the following, the used plugins are listed. Plugins that have been developed or adapted will be covered in more detail.

- *Keychain Plugin*

  In order to use the key-storage facilities provided by Android and iOS, we have implemented a plugin for the Apache Cordova framework. The developed plugin offers two methods, a method for storing a key `window.storeKeyInKeychain(alias, key, [protectionClass, thisDeviceOnly])` and a method for retrieving the key `window.loadKeyFromKeychain(alias)`. The parameter $alias$ denotes the identifier of the key and has to be passed to the plugin to be able to access the key later again. The parameter key comprises the key encoded as a Base64-encoded string. If a key is added with the same alias, the plugin silently overrides the stored key value.

  The developed plugin provides an interface to the iOS Keychain and the Android Keystore. In the following, platform-specific implementation details are discussed.

  Prior to Android version 6.0, the Keystore does not allow for the storage of symmetric keys. However, to still use the security provided by the Keystore for symmetric keys, we have implemented a simple key-wrapping mechanism. Hence, when adding a new entry to the Keystore, a new RSA key pair is created. The key string is encrypted with the public RSA key and stored on the file system in the internal storage area of the mobile application. The plugin uses RSA with PKCS#1 padding in ECB mode for encryption. The security of the Android Keystore highly depends on the underlying implementation. Hardware-based implementations protect the private keys from being extracted and thus being transferred to other devices. Unfortunately, not all devices offer a hardware-based implementation. Developers can check the security features of the Android Keystore. However, currently, our plugin does not support to query whether the stored key is stored inside secure hardware[6].

[6] `https://developer.android.com/reference/android/security/keystore/KeyInfo.html#isInsideSecureHardware()`

On iOS devices, the plugin stores the key values as generic password and therefore, arbitrary strings of data are supported. For the iOS version of the plugin we have added the parameters *protectionClass* and *thisDeviceOnly*. This way, developers can choose when the Keychain entry should be available to the application and thus when the system decrypts it. Apple uses multiple so-called protection classes to specify the decryption point. For each protection class there exists a *thisDeviceOnly* variant that hinders the transfer to other devices and the inclusion in the iOS backup. If this parameter is not set, items are created with *thisDeviceOnly* defaulting to false. In the following, a listing of available protection classes[7] is given:

- The protection class *AccessibleAfterFirstUnlock* implies that after a restart the Keychain item cannot be accessed until the device has been unlocked once by the user. This protection class is recommended for applications that require access to a key while running in the background.
- Regardless of the device being locked or unlocked, the application can always access Keychain items with class *AccessibleAlways*.
- *AccessibleWhenUnlocked* allows applications to access items only if the device is unlocked at that time. This is the default value if no protection class has been specified.
- The protection class *AccessibleWhenPasscodeSet* provides the same properties as *AccessibleWhenUnlocked*, but it can only be used with devices where the passcode is set. Thus when removing the passcode, the system deletes all items of this class. In addition, items cannot be migrated to other devices as this class is only available in the "ThisDeviceOnly" variant.

- *Dialog Plugin*

  Some authentication methods require the user to enter a secret PIN or password on the mobile device. In order to use the native user-interface style for dialog and notification windows, we chose to use a dedicated plugin[8]. However, this plugin does not yet feature a password field that masks the entered characters. We have extended the plugin such that the developer can specify the expected input type. Currently, the plugin supports the two parameters *alphanumeric* and *numeric* for specifying the keyboard type that should be presented to the user. If neither of both values is passed to the plugin, the input is not treated as a password.

- *Push Plugin*

  To access the mobile device's push-notification service, Apache Cordova applications require the use of a plugin. The push-notification plugin[9] currently supports Android, iOS and Windows devices. The plugin handles registration of the device with the platform vendor's push-notification service and forwards incoming notifications to the Apache Cordova application running in the WebView. The application developer can specify various options, such as, playing a sound when receiving a new notification. For Android, the GCM service is used. Although GCM is now available for iOS devices as well, this plugin uses Apple's APNS.

- *BarcodeScanner Plugin*

  The BarcodeScanner plugin[10] for Apache Cordova provides a cross-platform implementation of QR code and other barcode scanning functionality. For retrieving the QR code, the device camera is used. In our authentication app, QR codes are heavily used for the rollout of new authentication methods, as QR codes represent a simple way for transferring initial authentication data, such as cryptographic keys, from the server to the mobile device.

[7] https://developer.apple.com/library/prerelease/ios/documentation/Security/Reference/Keychainservices/#//apple_ref/doc/constant_group/Keychain_Item_Accessibility_Constants

[8] https://github.com/apache/cordova-plugin-dialogs

[9] https://github.com/phonegap/phonegap-plugin-push

[10] https://github.com/phonegap/phonegap-plugin-barcodescanner

- *Device Plugin*

  The device plugin[11] offers information on the mobile device's hardware and software capabilities. The device plugin is for example used to determine whether the authentication app is running on an Android or iOS device. Using the device plugin, the application can read the Universally Unique Identifier (UUID) that uniquely identifies the mobile device.

- *Certificate Pinning Plugin*

  The Apache Cordova framework does not support certificate pinning. However, there are third-party plugins that check the server's certificate on every connection. Cordova HTTP[12], for example, provides an AngularJS service handling HTTP communication from within the WebView. Instead of using the conventional AngularJS HTTP service[13], the developer uses the plugin's HTTP service for placing requests to the server. The plugin pins against certificate files included within the application. It has to be noted that this plugin only covers HTTP requests issued from the WebView. It does not affect requests placed from within other plugins.

- *Whitelist Plugin*

  Since Apache Cordova version 4.0, whitelisting is realised by using a Cordova plugin[14]. This plugin allows setting URL whitelists for navigation, network requests and intents. For network requests, however, it is recommended to rely on the Content Security Policy (CSP) set in the HTML templates to control which requests are allowed to be made. It has to be noted that the available whitelisting mechanisms only apply to calls originating from within the WebView. Requests placed from within Cordova plugins are not restricted.

To put the pieces together, all implemented AngularJS components and Apache Cordova plugins have been integrated into the authentication app. The authentication app allows users to add new authentication methods for different domains and manage these authentication methods, e.g. remove the authentication method from the mobile device. Annex A provides screenshots with accompanying explanation of the implemented authentication app.

6.3 Integration of New Authentication Methods

The implemented authentication framework pursues the goal to enable an easy integration of new authentication methods. Each implementation of an authentication method consists of code running as authentication plugin on the server and code running within the mobile application on the client. To add new authentication methods, both parts need to be implemented.

6.3.1 Server-side

The authentication service features external authentication plugins that handle user authentication. An external authentication plugin represents a web application that does not necessarily have to run on the same server as the authentication plugin. Each authentication plugin has to offer a configuration endpoint and an authentication endpoint, to whom the prover is being redirected by the authentication service.

In order to add a new external authentication plugin to be used by the authentication service, the administrator has to add the plugin to the plugin repository. Therefore, the authentication service offers an intuitive user interface. In addition to stating the configuration and authentication endpoint, the administrator has to provide two public-key certificates. One certificate is used during authentication for

[11] `https://github.com/apache/cordova-plugin-device`

[12] `https://github.com/wymsee/cordova-HTTP`

[13] `https://docs.angularjs.org/api/ng/service/$http`

[14] `https://github.com/apache/cordova-plugin-whitelist`

verifying the signature of the signed JSON Web Tokens, whereas the second certificate is required for the protection of the configuration data.

In the following, the main components of an external authentication plugin are introduced in more detail.

- *Configuration Pack*

  Each external authentication plugin has to include some configuration data represented in a so-called *Configuration Pack*. The Configuration Pack, for example, comprises credentials or the mobile device's *Push ID*. The Configuration Pack is stored centrally at the authentication service. The authentication plugin only has to provide the description of the used data model. When transferring the Configuration Pack to the authentication service, the data is encoded as JSON data structure and protected via means of JSON Web Encryption and JSON Web Signature.

- *Configuration Controller*

  To initialise an external authentication plugin for a specific user, the plugin has to provide a configuration endpoint. This endpoint implements the user interface and business logic for the rollout of the authentication method to the mobile authentication app.

  The authentication methods that have been integrated so far use QR codes for transferring initial data from the server to the mobile authentication app. Therefore, the configuration controller has to first generate the initialisation data, display the data within a QR code and eventually let the user enter a password. The initialisation data might include randomly generated salt values and cryptographic keys.

- *Authentication Controller*

  During authentication, the prover is redirected to the authentication endpoint of an external authentication plugin that complies with the current domain's authentication policy. The authentication endpoint provides the user interface and business logic for performing a particular authentication method. For example, this might include the generation of transaction-specific values such as a random nonce, communication with the mobile authentication app and the verification of the obtained response from the authentication app. Upon successful verification, the external authentication plugin returns a signed JSON Web Token to the authentication service, stating that the user has authenticated successfully at this authentication plugin.

6.3.2 Client-side

The Ionic framework that builds on AngularJS has been used for the implementation of the mobile part of the authentication framework. AngularJS provides a slick separation between model, view and controller code. To integrate a new authentication method, the developer has to extend the mobile app with a model, view and controller of the respective authentication methods. Thus, new authentication methods can be integrated with three steps.

- *Data Model*

  Each authentication method needs some data locally stored on the device. Depending on the authentication method, this data comprises, for example, cryptographic key material, a counter or a salt value required for key derivation. Although sensitive data, such as key material or salt values, is stored using the device's key-storage facilities, the mobile application still has to track the alias of the stored data. For the storage of this authentication data, we make use of the Local Storage provided by the WebView. We refer to Section 4.2.2 for an introduction to Local Storage.

Thus, for integrating a new authentication method, a new data model has to be created. This data model has to inherit from the `PluginData` object[15]. The `PluginData` object holds some general information on the authentication method, e.g. its name, a short description and the domain. The domain corresponds to the notion of domains as used on the server side framework. The domain is required for allowing multiple instances of the same authentication method for different services.

For each data model a corresponding method for setting up this authentication method has to be implemented. This method includes the logic for extracting the relevant data from the QR code, adding the sensitive data to the device's key-storage facilities and returning the *Push ID* required for the push-notification system to the authentication plugin on the server-side.

- *View*

  The client-side framework features dynamic routing to the correct authentication method. When receiving a push notification, a *modal*, a new pop-up window, is opened which comprises the necessary steps to complete the authentication method. This might include entering a password or some other user action. Therefore, an HTML template has to be created for each authentication method. For heavily used functionality, such as displaying the computed OTP, existing templates or so-called *partials* can be integrated in the template and thus do not need to be implemented for each authentication method separately.

- *Controller*

  For each HTML template an accompanying controller has to be created that provides the implementation of the business logic. This applies to the authentication method templates as well. The controller might use all core framework services for processing the authentication method. Hence, the controller, for example, calls the *KeyStoreService* to retrieve the secret key and some of the provided cryptography services to complete cryptographic operations.

6.4 Lessons Learned

Apache Cordova has evolved towards a powerful tool for developing hybrid mobile applications for multiple platforms. However, while using Apache Cordova and related projects, in particular external plugins, we came across multiple weaknesses and challenges compared to using the native API stack provided by the mobile platform vendors.

The plugin approach that allows accessing native device APIs from within the WebView component is one of the core features of the cross-platform framework Apache Cordova. However, this might also lead to substantial security vulnerabilities. When using important device features, such as push notifications or file-system access, developers have to rely on source code provided by external developers. There are no official review mechanisms for these publicly available Apache Cordova plugins. Anyone can release plugins to the public using the official Cordova Plugin registry[16].

One of our findings show that the Barcode Scanner plugin[17], heavily used in our mobile authentication app, silently stored the raw data of each scanned barcode to the Local Storage area of the application. For our authentication app this was fatal. During the rollout of new authentication methods to the mobile device, the user scans a QR code. This QR code contains the cryptographic key material that is then stored using the mobile device's key storage facilities. However, at the same time the presumably sensitive key material resides in the internal storage area of the application as well. This behaviour was not

[15] JavaScript is a prototype based language. Hence, each object has an internal link to another object, its *prototype*. This way, a prototype chain can be constructed until an object is reached that has no prototype. For more details, we refer to the developer documentation of Mozilla: `https://developer.mozilla.org/en/docs/Web/JavaScript/Inheritance_and_the_prototype_chain`

[16] `http://cordova.apache.org/plugins/`

[17] `https://github.com/phonegap/phonegap-plugin-barcodescanner`

documented by the developers of the plugin. However, this behaviour is no longer present in the current version of the plugin.

The Ionic framework builds on the popular model-view-controller framework AngularJS. AngularJS uses services and dependency injection to encapsulate large parts of the business logic. Therefore, it arguably makes sense to provide Apache Cordova plugins as services as well. This way, application code can remain clear and structured. The ngCordova project[18] provides a collection of AngularJS services that interface with the most commonly used Apache Cordova plugins. However, this arguably useful project adds another layer of abstraction and thus insecurity to the Apache Cordova ecosystem. We can illustrate this with an example. The beforehand hard to use API of the popular push-notification plugin[19] has been updated, but the integration via ngCordova has not yet reflected the API changes and thus cannot be used with recent versions of the plugin. This makes clear that when relying on externally provided plugins it is hard to use the Apache Cordova ecosystem in a production environment.

Similar to the well-known differences regarding supported APIs in the desktop browser environment, we face similar challenges in the mobile environment. There are different versions of the WebView components employed across devices. This way, the WebView differs regarding support for important features, such as, for example, the Web Cryptography API. Android devices that are running an operating system version prior to 5, lack support for the API altogether. Whereas on iOS devices, the API has been prefixed and thus API calls differ from Android to iOS devices. For compatibility reasons, we decided to not use the Web Crypto API within our authentication app yet. However, with the increasing market share of devices that support the Web Crypto API future developments should favour the standardized Web Crypto API over conventional JavaScript cryptography libraries. Multiple use cases would highly benefit from using the built-in Web Crypto API. Currently, some authentication methods use the SCrypt key-derivation function, which internally uses a hash function implemented purely in JavaScript. Using hash functions provided by the Web Crypto API would considerably improve the performance of this SCrypt implementation.

Although developing mobile applications with the help of Apache Cordova poses some challenges, it turned out as a decent choice for the implemented authentication framework. With the use of web technologies, the effort for adopting the application to other mobile platforms is minimized and limited to the implementation of plugins, that provide an interface to native device APIs. The next chapter provides the security evaluation of the implemented mobile components and hence analyses the security aspects of the Apache Cordova framework in detail.

[18] http://ngcordova.com/

[19] https://github.com/phonegap/phonegap-plugin-push

Chapter 7

Security Evaluation

In this chapter, the security evaluation of the implemented authentication framework and the integrated authentication methods is presented. For the evaluation, we roughly adhere to the Common Criteria for Information Technology Security Evaluation. We start with a presentation of the methodology used in this evaluation. The subsequent sections concisely define the system under evaluation, its assets and the resulting threats. Before concretely evaluating the system and the countermeasures it provides, we define security objectives that should be met by the mobile authentication app and the integrated authentication methods. For this thesis, the focus is on analysing the security of the implemented authentication methods and the authentication app on the client device with special considerations of the use of the Apache Cordova framework and web technologies on Android and iOS devices.

7.1 Methodology

Common Criteria for Information Technology Security Evaluation as standardized in ISO/IEC 15408 [39] provides a framework for the security evaluation of IT systems. Common Criteria is, for example, well-established in the security evaluation of Integrated Circuit Cards or commonly known as smart cards. Therefore, ISO/IEC 15408 provides the structure of Protection Profiles applicable for different technologies and IT systems. A Protection Profile defines an implementation-independent set of security requirements for a category of devices or systems. Besides smart cards, Common Criteria also hosts Protection Profiles for operating systems, databases and many more[1]. Based on the requirements defined in the Protection Profile, evaluators can determine the security properties of the system under evaluation and identify potential gaps. Furthermore, Protection Profiles can be used by consumers to express their IT security needs.

Each Protection Profile follows a common structure. To provide a rough overview, a Protection Profile first defines the system that should be evaluated, entities or values the system aims to protect, potential attacks on these values and security objectives aiming to counterfeit these attacks. Based on the defined security objectives, IT security requirements are derived. This means that the Protection Profile defines requirements that should be met by a particular system and its environment.

This security evaluation roughly adheres to the general structure of security evaluations based on Protection Profiles according to ISO/IEC 15408. It has to be noted that this work does not present a full-featured Protection Profile but only makes use of its well-proven structure for security evaluation. Other authors have shown that is common practice to tailor the principal components of ISO/IEC 15408 to the special characteristics of the system under evaluation [81]. This section introduces the methodology of the conducted security evaluation. First, we establish the main concepts of security evaluations according to ISO/IEC 15408. While some parts of the standardized methodology are an overkill for this thesis, the

[1] `http://www.commoncriteriaportal.org/pps/`

definition of additional components well complements our evaluation. Therefore, we provide remarks, when diverging from the structure as defined in ISO/IEC 15408.

7.1.1 ISO/IEC 15408

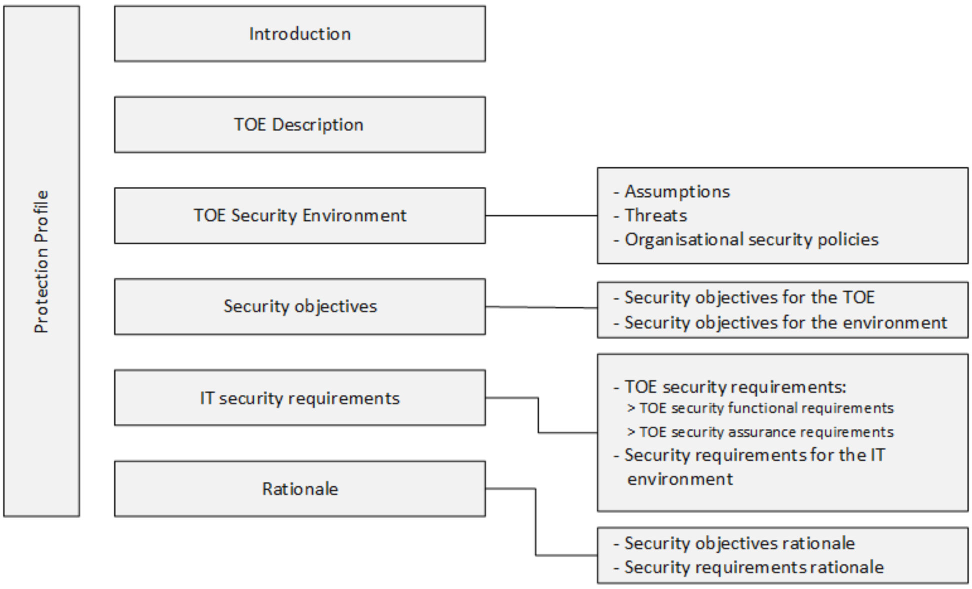

Protection Profile
Introduction
TOE Description
TOE Security Environment
Security objectives
IT security requirements
Rationale
- Assumptions
- Threats
- Organisational security policies
- Security objectives for the TOE
- Security objectives for the environment
- TOE security requirements:
> TOE security functional requirements
> TOE security assurance requirements
- Security requirements for the IT environment
- Security objectives rationale
- Security requirements rationale

Figure 7.1: The basic structure of a Protection Profile according to ISO/IEC 15408 [39].

Figure 7.1 provides an overview of the structure of ISO/IEC 15408 Protection Profiles. In the following, the different components are discussed.

- *Introduction*

  The introduction provides some general information about the Protection Profile and summarises the Protection Profile in narrative form.

- *TOE Description*

  A security evaluation according to Common Criteria starts with the definition of the system that will be analysed. The Target of Evaluation (TOE), therefore, represents the scope of the security evaluation. The TOE defines the category of software or hardware that will be evaluated. It is up to the authors of the evaluation to define the TOE as the whole IT product or only a small part or combination of parts.

- *TOE security environment*

  The TOE security environment describes security aspects of the environment the TOE is intended to be used. It includes the following:

  – *Assumptions* that provide information about the intended use of the TOE, the potential value of assets, usage limitations and information about the environment the TOE is used.
  – The Protection Profile proceeds with a description of all *threats* to the previously defined assets, against the TOE should provide adequate protection. These threats might include the leak of assets or their modification. Each threat description should include the threat agent, the asset that is under attack and the attack scenario.
  – A description of *organizational security policies* identifies rules the TOE must comply to.

- *Security objectives*

  Based on the identified threats this part defines security objectives that aim to provide countermeasures to the previously defined threats and cover the listed organisational security policies. The security objectives for the environment cover threats that cannot be countered by the TOE and, therefore, have to be met by the environment of the TOE.

- *IT security requirements*

  As the system under evaluation has been described and security objectives have been defined, the next logical step is the collection of security requirements. These can be categorised into functional requirements towards the TOE and optional security requirements for the environment the TOE is operated in.

- *Rationale*

  The rationale aims to present the completeness of the Protection Profile. Therefore, it demonstrates that all threats have corresponding security objectives aiming to counterfeit the threats. Furthermore, it includes a mapping that shows that the derived security requirements or a combination of security requirements succeed in meeting the defined security objectives.

The structure described in ISO/IEC 15408 does not fully align to the intentions of our security evaluation and the use case of our work. As the practical part of this work represents a prototype, the definition of organisational security policies seems to be an overkill. On the other hand, the definition of assets the TOE aims to protect is crucial for the secure implementation and operation of the TOE. Therefore, the definition of assets should be given more importance. The following paragraphs will explain our modified methodology.

7.1.2 Derived Methodology

Figure 7.2 summarises the methodology we have derived from ISO/IEC 15408 and that will be used for this security evaluation. Our security evaluation starts with the *definition of the TOE*. As defined by ISO/IEC 15408 we describe the system under evaluation and its environment. Section 7.2 defines the TOE of this work. As part of the environment, we define the parties that will interact with our system (the so-called *actors*, see Section 7.3). In Section 7.4 we proceed with a set of *security assumptions* about the TOE. These might include assumptions about the security of cryptographic methods or technologies.

When evaluating the security of a system, it is most relevant to analyse how well the system succeeds in protecting its assets. In order to be able to make any statement about the security of an IT system, the implementor or evaluator needs to be aware of the assets the TOE aims to protect. Therefore, as a next separate step, we *extract the assets*. Assets can, for example, include sensitive data about the system's user or key material the system aims to protect. Section 7.5 contains the assets our system seeks to protect. Based on the extracted assets, a *definition of threats* to each asset follows. The threat model in section 7.7 also contains information about the potential attacker. To model the different types of potential attackers we define so-called *threat agents*, beforehand. A threat agent describes the level of expertise, available resources and motivation of a potential attacker. Section 7.6 contains the threat agents relevant to our work.

Based on the identified threats, Section 7.8 derives *security objectives* that should be met to be able to counterfeit the defined threats. These security objectives should be met either by the TOE itself or the environment of the TOE. The simplified methodology abandons the definition of security requirements but immediately proceeds with a *description of implemented countermeasures*. A countermeasure describes how the system is protected against a threat by fulfilling a specific security objective. Each countermeasure is mapped to the security objectives it fulfills. As our system consists of the mobile authentication app that features multiple authentication methods, countermeasures are grouped into countermeasures provided by the authentication app and, therefore, valid for all authentication methods and countermeasures specific to a single authentication method.

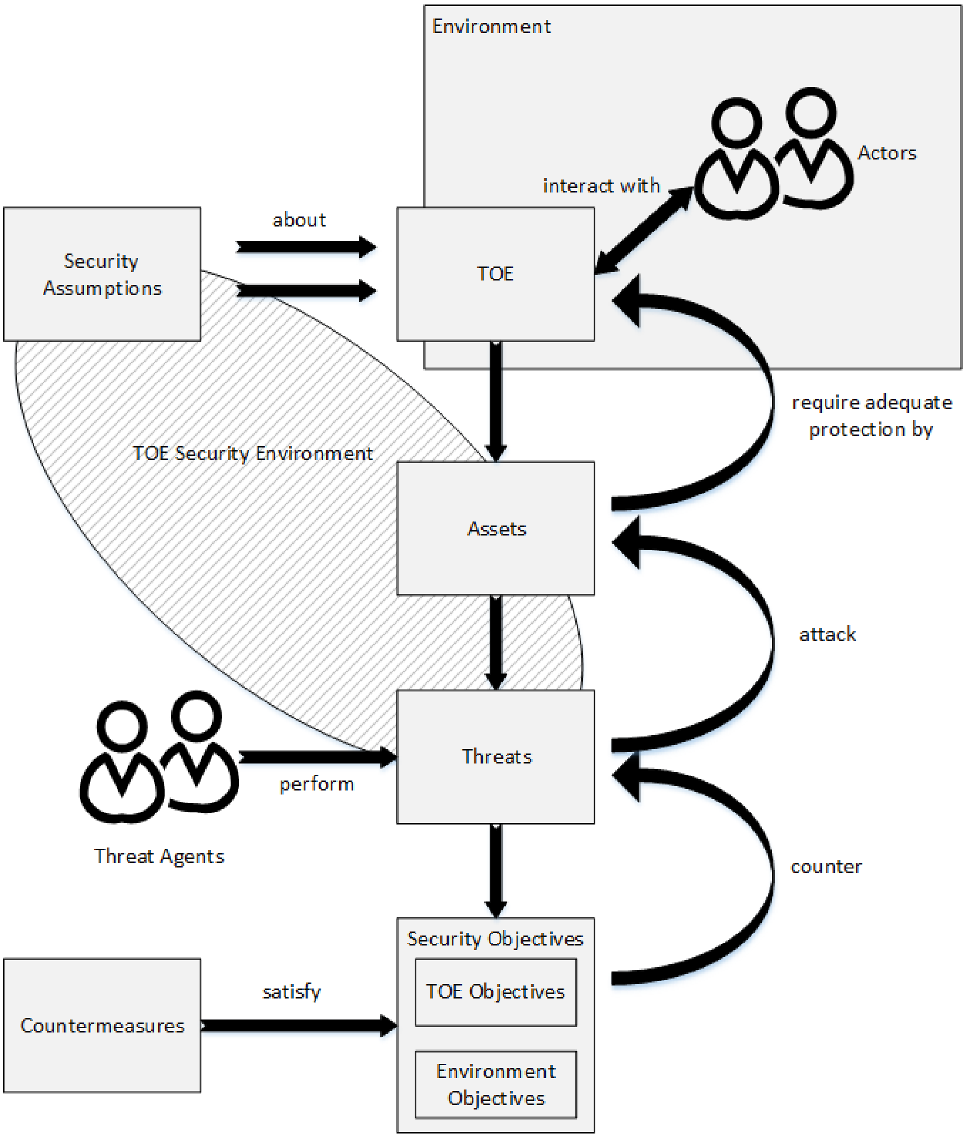


Figure 7.2: Adhering to the schema defined by ISO/IEC 15408 [39] we have derived a simplified methodology for the security evaluation of the implemented authentication framework.

Except the part on implemented countermeasures, our security evaluation is held implementation-independent. Therefore, the structure of the security evaluation including its definition of assets, threats and security objectives can be applied to other multi-factor authentication systems as well.

7.2 Target of Evaluation

The TOE defines the scope of the security evaluation. We have split the description of the TOE into two parts. The first part describes the IT system under evaluation, whereas the second part denotes the two use cases that have been considered.

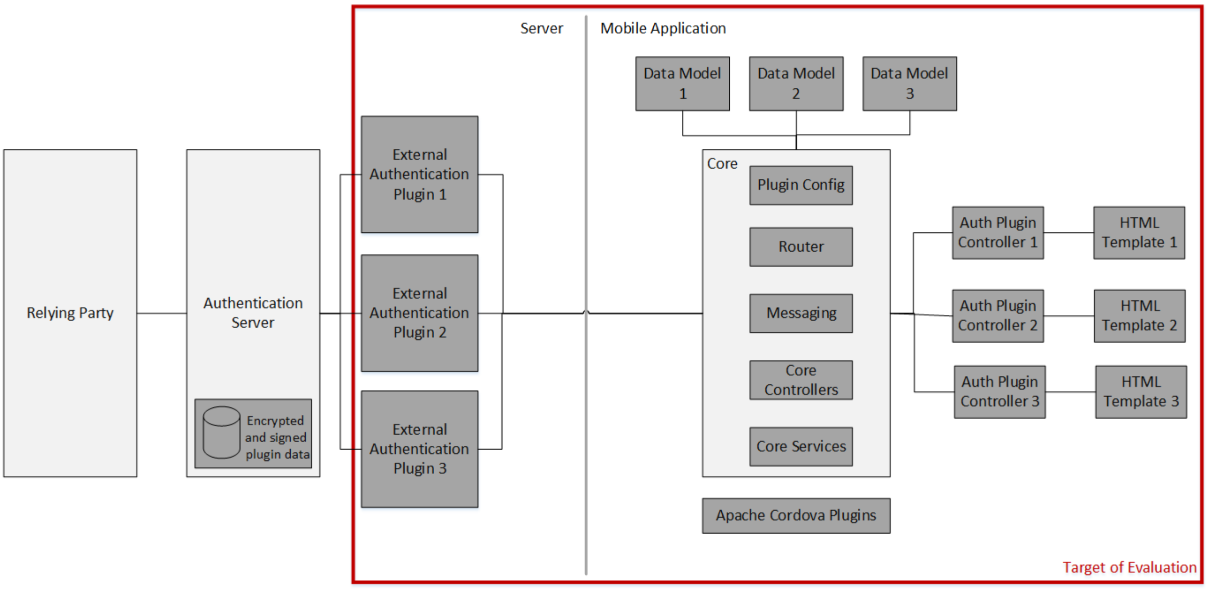


Figure 7.3: The general architecture of the implemented authentication framework. In the course of this security evaluation we define the TOE as the mobile authentication app and the plugin implementation on the server side. The relevant components have been been highlighted.

7.2.1 IT System

A first step of the security evaluation is to define the IT system under evaluation. The implemented authentication framework consists of a mobile application with the various authentication plugins and an authentication server featuring multiple external authentication plugins. Within this thesis, the mobile application and several server-side authentication plugins have been developed, whereas an existing authentication server has been used. Therefore, we define the TOE as the mobile application and the server-side authentication plugins. It is crucial to include the server-side authentication plugins within our evaluation, as the communication path between authentication plugin and mobile authentication app might present a critical path, where presumably sensitive data is exchanged. Figure 7.3 provides a graphical representation of the TOE.

Our security evaluation focuses on the two mobile platforms Google Android and Apple iOS. Hence, when stating the countermeasures to specific threats, differences regarding security properties of the two platforms are considered.

7.2.2 Use Cases

For our security evaluation we have to consider two different use cases:

- *Registration*

  The registration of a new authentication method denotes the process of exchanging initial authentication data, such as secret key material or the mobile device's *Push ID*.

- *Authentication*

  The actual authentication process requires the use of one or multiple authentication method(s). Therefore, we evaluate the security of our implemented authentication methods.

7.3 Actors

In the following, legitimate actors that communicate or influence the TOE are introduced.

- *User*

  The user or also referred to as device holder uses an authentication method within the authentication app on her mobile device to authenticate at a service (the *relying party*). The user has full access to her mobile device.

- *Authentication Plugin Provider*

  The server components provided by EGIZ support external authentication plugins that might be operated by third-parties. These plugins handle communication with the user and in our case with the user's mobile device in order to process authentication. The authentication plugin provider only has access to the user's authentication data (e.g. cryptographic key material, etc.) for a particular authentication plugin. The authentication plugin provider initialises new authentication plugins for the user and completes all steps during authentication.

- *Authentication Service Provider* (out of scope)

  The authentication service provider operates the authentication service. It receives requests from the relying party and requests the user to complete a set of authentication plugins in order to authenticate successfully. Upon successful authentication, the authentication service provider responds to the relying party's request. Furthermore, the authentication service provider manages the set of available authentication plugins.

  The defined TOE explicitely excludes the authentication service from evaluation. Therefore, we have listed the authentication service provider mainly for the sake of completeness.

- *Relying Party* (out of scope)

  The relying party operates a service that requests user authentication from the authentication service provider. The relying party belongs to a specific domain. Depending on the domain, the server components provided by EGIZ allow the specification of authentication policies. These policies define the set of available authentication plugins to the user.

  The defined TOE does not include the relying party. Therefore, the relying party will not be considered in the course of this evaluation.

In the following security evaluation, we focus on the two actors user and authentication plugin provider. The authentication service provider and the relying party are considered out of scope, as this evaluation focusses on the interaction between the server-side authentication plugins and the authentication app on the user's mobile device. We omit those two actors, as we do not intend to conduct a security evaluation of the server components provided by EGIZ, but solely focus on the server-side authentication plugin and its mobile counterpart.

7.4 Security Assumptions

The following assumptions describe properties of the TOE and the environment in which the TOE is operated.

(AS1) The server-side authentication service and the server-side authentication plugin are trustworthy. This includes that the authentication plugins on the server are authenticated towards the authentication service.

(AS2) The server computers are trusted. Only authorised personnel has access to the servers. Hence, data and software running on the server-side is protected from unauthorised modification.

(AS3) All cryptographic algorithms that are in use are implemented correctly. This also includes PRNGs.

(AS4) The authentication service and all authentication plugins feature a valid certificate issued by a verified certificate authority.

(AS5) The services comprised in the TOE are available during the registration and the authentication process.

7.5 Assets

The security of a system is mainly measured in how well it succeeds in protecting its assets. Assets describe data or components that should be protected by the TOE. This section carefully defines assets relevant for this security evaluation. For the sake of clarity, we chose to hierarchically define the assets to be protected by the TOE.

7.5.1 Primary Assets

A service that integrates multi-factor authentication typically tries to enhance the protection of sensitive data or components by increasing the barrier for potential attackers to gain access to those. The assets typically reside on the server-side and differ from service to service. Such assets typically are:

- *Sensitive data* in general. Sensitive data, for example, can include access to company internal documents and services or access to the an e-mail account (e.g. Google Mail).

- The *eID* of a person. Multiple member states of the European Union provide citizens with an eID that enables access to online governmental services such as filing a tax return or residence registration.

- The private *signature key* for creating electronic signatures. Austria, for example, provides a server-based signature solution for creating qualified electronic signatures, the so-called Mobile Phone Signature[2]. In order to access the private signature key, the user has to authenticate using TANs sent via SMS.

- Access to *online banking* services. Most banks offer services for triggering money transfers or checking the balance online.

7.5.2 Secondary Assets

The primary assets are protected by the user authentication process. Upon successful authentication, the user can access the previously defined primary assets. For authentication, the user typically provides the response to a challenge issued by the service. This response can either be a simple password or some cryptographic result (e.g. an HMAC or a digital signature). Hence, we define the response to the challenge as secondary asset.

- *Response to a challenge*

[2] https://www.handy-signatur.at/

7.5.3 Tertiary Assets

Within the context of this evaluation, we consider systems that rely on some computational steps for supplying the response to the challenge. Our TOE computes the response to the challenge on the mobile device. Therefore, the authentication app on the mobile device uses various input vectors. Amongst others, it uses data that resides locally on the device, input from the user or other features available on the device. This means that the security of the TOE depends on the protection of the different input vectors. Consequently, we define the tertiary assets as follows.

(A1) *Cryptographic key material* stored on the mobile device. The cryptographic keys must be held secret and protected against access by malicious parties.

(A2) The user-supplied *password* that is used to compute the response to a challenge. It is important to ensure that the password is not leaked and that malicious parties are not able to derive the password from the computed result.

(A3) The *Push ID* for the platform's push-notification service defines to whom data is sent. This identifier requires adequate protection when stored on the mobile device and during transfer to the application server.

(A4) The *physical device* itself. Authentication methods relying on some sort of device binding (for example when using push-notification services or SMS) require the physical device to be protected against malicious access.

(A5) The *response* to the challenge itself. This can, for example, be represented by an OTP.

Cryptographic key material stored on the server has been omitted intentionally, as the cryptographic key material is not stored within the authentication plugin on the server but transferred to the authentication service, which is not part of the TOE. The authentication service only receives the encrypted and signed plugin data packages and has no way of accessing the plain data.

The tertiary assets represent the data and components our authentication frameworks needs to protect. For that reason, we abandon the primary and secondary assets in the subsequent deliberations.

7.6 Threat Agents

Common Criteria defines a threat agent as an "entity that can adversely act on assets" [19]. Threat agents describe individual entities, but can also be used to describe a type of entities. Common Criteria lists the following examples: hackers, users, computer processes and accidents. To additionally refine the description of threat agents, a statement about their expertise, available resources, opportunity and motivation can be added.

In the course of this security evaluation, we have defined multiple threat agents. For each threat agent, the capabilities and characteristics are listed.

(TA1) *Standard Attacker*: attacker with access to standard malware on device

(TA2) *Advanced Attacker*: attacker with malware that has or gains root access on the device

(TA3) *Nearby Attacker*: attacker that has physical access to the device, e.g. a thief or someone willing to perform a targeted attack on a user

(TA4) *Network Attacker*: attacker with access to the network, who is able to inspect and/or manipulate traffic between the mobile device and the authentication plugin, i.e. a classical man-in-the-middle attacker

(TA5) *Web Attacker*: attacker that hosts malicious JavaScript code, and therefore controls one or more domains, e.g. a malicious advertiser that wants to collect sensitive data about the user

7.7 Security Threats

This section describes possible attack scenarios (the so-called *threats*) that affect the assets listed in Section 7.5. According to Common Criteria, "a threat consists of an adverse action performed by a threat agent on an asset" [19]. Hence, for each threat we state the affected asset or group of assets. Furthermore, each threat definition contains the involved threat agent and a short description of the possible attack scenario. In addition, a short statement about the implications of a successful attack is included.

7.7.1 Primary Threats

In the interests of clarity, we provide a hierarchical listing of our threats. Primary threats denote general threat scenarios whereas secondary threats discuss, how an attacker might be able to achieve this general attack.

- *Impersonate Legitimate User Once*

  The attacker is able to impersonate the user once. An attacker can achieve this, for example, by mounting a man-in-the-middle attack. This general threat requires the user to start an authentication process first. The attacker can then apply means to eavesdrop, intercept or replay the authentication process.

- *Impersonation Without Involvement of the Legitimate User*

  The attacker is able to impersonate a legitimate user without interfering with an ongoing authentication process. This might, for example, be the case when an attacker is able to clone the device covering the second factor.

- *Hinder a Legitimate User From Authenticating Successfully*

  The attacker is able to block the legitimate user from authenticating towards the authentication service. However, the attacker does not accomplish a successful authentication on behalf of the legitimate user.

7.7.2 Secondary Threats

Secondary threats describe threats that directly or indirectly enable an attacker to mount a primary threat. For example, stealing the device covering the second factor and phishing the user's credentials allow the attacker to impersonate the legitimate user. Hence, secondary threats enable primary threats. Some of the presented secondary threats are inspired by Bonneau et al.'s listing of benefits that an ideal authentication scheme should provide [16]. In addition a subset of the OWASP Top Ten Security Threats regarding Web Security [59] and Mobile Security [72] has been incorporated.

(T1) *Device Theft*

The attacker gains access to the mobile authentication app by stealing the mobile device. Usage of the authentication app enables the attacker to authenticate successfully at the authentication service and thus impersonate the user.

Depending on the deployed authentication method, the security implications of a device theft vary. If the user is required to provide the factor knowledge as well, the impact of this attack is reduced as the attacker has to additionally guess or eavesdrop the correct password in order to authenticate successfully.

Affected Asset(s): (A4) Physical Device

Involved Threat Agent(s): (TA3) Nearby Attacker

(T2) *Copy or Extract Device Storage*

The attacker gains physical access to the storage of the mobile device. This is realised by physically accessing the device and extracting the device storage. This threat does not differentiate between device storage in general and the private storage area of the application.

This scenario might enable an attacker to extract sensitive data present on the mobile device and thus clone the device covering the second factor.

Affected Asset(s): (A1) Cryptographic key material, (A3) Push ID

Involved Threat Agent(s): (TA3) Nearby Attacker

(T3) *Accessing Data On-Device*

The attacker is able to access the storage on device. Access can be gained by deploying malware with root access on the mobile device.

This scenario might enable other applications to access sensitive data present on the mobile device. This data can further be used to clone the device covering the second factor.

Affected Asset(s): (A1) Cryptographic key material, (A3) Push ID

Involved Threat Agent(s): (TA2) Advanced Attacker, (TA5) Web Attacker

(T4) *Accessing Data During Transport*

The attacker accesses data during the transport from the mobile device to the authentication plugin on the server-side. This can be realised by sniffing data traffic or employing a man-in-the-middle attack.

This scenario might enable an attacker to gain access to presumably sensitive data.

Affected Asset(s): (A1) Cryptographic key material, (A3) Push ID, (A5) Response

Involved Threat Agent(s): (TA4) Network Attacker

(T5) *Phishing*

An attacker who fakes the authentication service tricks the user into supplying her credentials and thus enables the attacker to collect valid credentials.

It has to be distinguished between the user leaking the persistent factor knowledge and a dynamically generated token, such as an OTP or TAN. The security implications differ regarding the type of the leaked credential.

Affected Asset(s): (A2) Password, (A3) Push ID, (A5) Response

Involved Threat Agent(s): (TA1) Standard Attacker, (TA5) Web Attacker

(T6) *Throttled Guessing*

Although the rate of guessing is constrained by the authentication service, the attacker manages to guess the correct response to the challenge. The attacker could, for example, try to guess the factor knowledge, or the response to the transaction-specific challenge.

This scenario might enable the attacker to authenticate successfully.

Affected Asset(s): (A2) Password, (A5) Response

Involved Threat Agent(s): (TA1) Standard Attacker

(T7) *Unthrottled Guessing*

The rate of guessing is only constrained by available computing resources. Therefore, an attacker manages to guess the correct response to a challenge. Besides trying to guess the factor knowledge or a transaction-specific response to the challenge the attacker can also apply brute-force attacks on the involved cryptographic key.

This scenario might enable the attacker to authenticate successfully.

Affected Asset(s): (A1) Cryptographic key material, (A2) Password, (A5) Response

Involved Threat Agent(s): (TA1) Standard Attacker

(T8) *Leak from Verifier*

The authentication plugin on the server side verifies the response to the challenge and informs the user about the outcome of the verification. Some information the server could possibly leak, helps an attacker to impersonate the user by giving the attacker knowledge of the expected outcome.

Affected Asset(s): (A5) Response

Involved Threat Agent(s): (TA1) Standard Attacker, (TA4) Network Attacker

(T9) *Internal Observation*

The attacker intercepts user input on the mobile device (e.g. by installing a key logger) or monitors network traffic. The hereby gained information can be used to successfully authenticate at the authentication plugin.

The implications of the attack depend on whether the attacker being able to complete a single authentication process or being able to authenticate in future sessions as well.

Affected Asset(s): (A2) Password, (A5) Response

Involved Threat Agent(s): (TA1) Standard Attacker, (TA2) Advanced Attacker, (TA5) Web Attacker

(T10) *Physical Observation*

By observing the user during authentication one or more times, the attacker is able to authenticate successfully. For example, when relying only on username and password for authentication, observing the user entering the password enables the attacker to impersonate the legitimate user.

This scenario enables an attacker to authenticate successfully by solely observing the user performing the authentication.

Affected Asset(s): (A2) Password, (A5) Response

Involved Threat Agent(s): (TA3) Nearby Attacker

(T11) *Manipulation of Authentication Data*

The attacker is able to manipulate authentication data, such as cryptographic key material, on the mobile device.

The attacker hence hinders the legitimate user from authenticating at the server-side authentication plugin.

Affected Asset(s): (A1) Cryptographic key material, (A5) Response

Involved Threat Agent(s): (TA2) Advanced Attacker, (TA5) Web Attacker

(T12) *Manipulation of Push ID*

The attacker modifies the *Push ID* and hence hinders the legitimate mobile device from receiving messages from the server.

Messages redirected to the attacker's device allow the attacker to authenticate successfully.

Affected Asset(s): (A3) Push ID

Involved Threat Agent(s): (TA2) Advanced Attacker, (TA4) Network Attacker, (TA5) Web Attacker

(T13) *Intercepting Remote Notifications On-Device*

The attacker intercepts remote notifications on the mobile device before the received data is handed to the legitimate mobile application.

Using the data encoded within the remote notification, the attacker computes the correct response and hence authenticates successfully.

Affected Asset(s): (A5) Response

Involved Threat Agent(s): (TA2) Advanced Attacker

(T14) *Intercepting Remote Notifications During Transport*

The attacker intercepts remote notifications during transport before the notification reaches the legitimate mobile device.

Using the data encoded within the remote notification, the attacker computes the correct response and hence authenticates successfully.

Affected Asset(s): (A5) Response

Involved Threat Agent(s): (TA4) Network Attacker

(T15) *Code Injection*

The attacker injects malicious code by supplying manipulated input data. Code injection occurs whenever the application takes data without validation or escaping. The most popular type of code injection is cross-site scripting, which allows an attacker to execute scripts in the user's web browser.

Input fields, QR codes or other data is prone to this attack. Injected code might steal sensitive data or perform unwanted actions.

Affected Asset(s): (A1) Cryptographic key material, (A3) Push ID, (A5) Response

Involved Threat Agent(s): (TA1) Standard Attacker, (TA5) Web Attacker

(T16) *Missing Function Level Access Control*

An attacker with network access sends a request to the server-side authentication plugin, as the plugin does not protect its API properly.

This request allows the attacker to access private functionality or manipulate user data.

Affected Asset(s): (A3) Push ID, (A4) Response

Involved Threat Agent(s): (TA1) Standard Attacker

(T17) *Forging the Mobile Application*

By means of reverse-engineering, the attacker implements a mobile application that behaves like the benign authentication app. The deployed application, however, is malicious, e.g. it forwards sensitive data to the attacker.

The attacker is in full control of the mobile authentication app. Whilst the user can authenticate successfully using the forged authentication app, the attacker is also able to use the app under his control to authenticate successfully.

Affected Asset(s): (A1) Cryptographic key material, (A2) Password, (A3) Push ID, (A5) Response

Involved Threat Agent(s): (TA1) Standard Attacker

(T18) *Automatically Process Authentication Without The User's Consent*

The authentication process can be performed without the explicit consent of the user.

The attacker triggers the authentication process and also performs the authentication method successfully without the user noticing and without her assistance.

Affected Asset(s): (A5) Response

Involved Threat Agent(s): (TA1) Standard Attacker; (TA2) Advanced Attacker, (TA5) Web Attacker

Each defined threat covers one or multiple assets. Table B.1 provides the concrete mapping between threats and assets. Furthermore, the mapping between threat agents and threats is given in Table B.2. This mapping describes which threat agent is able to mount an attack and thus to pose a concrete threat to the TOE.

7.8 Security Objectives

In order to counterfeit the previously listed threats, the following set of security objectives has to be met.

(O1) *Protected Storage of Sensitive Data*

The TOE ensures that sensitive data such as cryptographic key material or the Push ID is stored protected against unauthorised access and modification.

Covered Threat(s): (T2) Copy or Extract Device Storage, (T3) Accessing Data On-Device, (T11) Manipulation of Authentication Data, (T12) Manipulation of Push ID

(O2) *Avoid Side-Channel Data Leakage*

The TOE ensures that sensitive data does not end up in operating system logs or web browser caches.

Covered Threat(s): (T3) Accessing Data On-Device

(O3) *Detect Manipulation of Sensitive Data*

The TOE ensures that any alterations or manipulations of sensitive data are detected.

Covered Threat(s): (T11) Manipulation of Authentication Data, (T12) Manipulation of Push ID

(O4) *Detect Rooted and Jailbreaked Devices*

The TOE ensures that the authentication app is not started, if the mobile device is rooted or jailbreaked.

Covered Threat(s): (T3) Accessing Data On-Device, (T9) Internal Observation, (T11) Manipulation of Authentication Data, (T12) Manipulation of Push ID, (T13) Intercepting Remote Notifications On-Device

(O5) *Validate and Escape User Input*

In order to counterfeit code injection, the TOE validates and escapes all user-supplied input. This data comprises text input, as well as data retrieved from QR codes and remote notifications.

Covered Threat(s): (T15) Code Injection

(O6) *Do Not Run Code from Untrusted Sources*

The TOE does not load source code (e.g. JavaScript libraries) from external sources. Only source code packaged within the mobile authentication app or loaded from the origin of the server-side authentication plugin is executed on the device. If content from the authentication plugin is loaded, state-of-the-art transport layer security is used.

Covered Threat(s): (T15) Code Injection

(O7) *Detect Forged Mobile Applications*

The TOE employs measures to detect whether the mobile authentication app is legitimate.

Covered Threat(s): (T17) Forging the Mobile Application

(O8) *Code Obfuscation*

In order to harden reverse-engineering and thus forging the mobile application, the TOE employs code obfuscation.

Covered Threat(s): (T17) Forging the Mobile Application

(O9) *Employ State-of-the-Art Transport Security*

The TOE uses state-of-the-art transport layer security for protecting traffic between the mobile authentication app and the server-side authentication plugin.

Covered Threat(s): (T4) Accessing Data During Transport, (T12) Manipulation of Push ID

(O10) *Employ Certificate Pinning*

To provide security against man-in-the-middle attacks the TOE employs certificate or public-key pinning.

Covered Threat(s): (T4) Accessing Data During Transport, (T12) Manipulation of Push ID

(O11) *Protection of Server-Side APIs*

The TOE protects server-side APIs adequately such that attackers cannot access or manipulate another user's data.

Covered Threat(s): (T16) Missing Function Level Access Control

(O12) *Protection of Sensitive Data in Remote Notifications*

If sensitive data is sent using remote notifications, the TOE ensures that data is additionally protected by means of encryption.

Covered Threat(s): (T13) Intercepting Remote Notifications On-Device, (T14) Intercepting Remote Notifications During Transport

(O13) *High Password Entropy*

In order to increase entropy, the TOE ensures that user-chosen passwords provide sufficient entropy, e.g. by complying to a password policy.

Covered Threat(s): (T6) Throttled Guessing, (T7) Unthrottled Guessing

(O14) *High Response Entropy*

The response to the challenge shall have sufficient entropy to render an exhaustive search computationally infeasible. If the server-side authentication plugin limits the number of tries, measures have to be taken, that the computed response does not show repetitive patterns and hence remains unpredictable.

Covered Threat(s): (T6) Throttled Guessing, (T7) Unthrottled Guessing, (T9) Internal Observation

(O15) *High Cryptograhic Key Entropy*

Cryptographic keys used by the TOE shall have sufficient strength to render an exhaustive search with respect to the cryptographic key computationally infeasible. If the verification whether the used key is correct is happening at the server-side authentication plugin, the plugin has to limit the number of tries. Furthermore, measures have to be taken, that the computed response does not show repetitive patterns and hence the used cryptographic keys remain unpredictable.

Covered Threat(s): (T6) Throttled Guessing, (T7) Unthrottled Guessing

(O16) *Adequate Verification Info*

The TOE provides no hints or detailed information in case of wrong user credentials or wrong responses. This, for example, includes advoiding displaying detailed error and exception messages.

Covered Threat(s): (T6) Throttled Guessing, (T8) Leak from Verifier

(O17) *Explicit User Consent*

The TOE has to ensure that the user explicitely gives her consent for performing authentication on the mobile device.

Covered Threat(s): (T18) Automatically Process Authentication Without The User's Consent

(O18) *Transaction-Specific Response*

The TOE ensures that the response to the challenge is transaction-specific, hence, the response cannot be used for subsequent authentication processes and is only valid for one particular authentication.

Covered Threat(s): (T5) Phishing, (T7) Unthrottled Guessing

(O19) *Assure Legitimate User is Operating the Device*

The TOE employs measures to assure that the legitimate user is operating the mobile device covering the second factor. This might be realised by requiring additional knowledge or biometric verification.

Covered Threat(s): (T1) Device Theft, (T18) Automatically Process Authentication Without The User's Consent

(O20) *Assure Factor Possession is Required for Authentication*

The TOE ensures that an attacker cannot authenticate at the authentication plugin without being in possession of the device covering the factor possession.

Covered Threat(s): (T9) Internal Observation (T10) Physical Observation

Each defined threat is countered by fulfilling one or a combination of multiple security objectives. Table B.3 provides a mapping between security objectives and threats.

7.9 General Countermeasures

This section states measures aiming to meet the previously described security objectives. The listed countermeasures have been split into *(a) General Countermeasures* that have been implemented within the authentication app and are valid for all authentication methods and *(b) Specific Countermeasures* that vary between the different integrated authentication methods. Most countermeasures equally apply to iOS and Android devices. If necessary, platform-specific solutions or implications depending on the particular threat agent are added.

(O2) *Avoid Side-Channel Data Leakage*

Cordova provides the functionality to write JavaScript logging messages to the operating system logs. By removing the required Cordova plugin, no JavaScript logs and thus no potentially sensitive data gets leaked to the operating system. The official documentation recommends removing this plugin before building release packages[3].

(O3) *Detect Manipulation of Sensitive Data*

Sensitive data stored within the mobile device's key-storage facilities is currently not protected regarding integrity. Meaning, no cryptographic signatures or message authentication codes have been used.

It could be suspect of future work, to enhance the implemented Keychain Plugin with support for cryptographic signatures.

(O4) *Detect Rooted and Jailbreaked Devices*

Apple iOS devices, by default, do not allow the installation of applications from other locations as the official AppStore. In order to remove software restrictions, many users employ jailbreaks. A jailbreak allows using software not authorised by Apple. However, jailbreaks add numerous security vulnerabilities, e.g. jailbreaked devices cannot be relied on adequately protecting sensitive application data.

[3] `http://ionicframework.com/docs/guide/publishing.html`

In contrast to iOS, Android devices allow the installation of applications from external sources. Therefore, on Android, rooting is only performed for gaining privileged access to the mobile device. With root access, the entire operating system can be manipulated as well as data from other applications can be accessed.

As rooted or jailbreaked devices pose a serious security threat, many applications employ detection mechanisms and only offer full functionality if the device is not rooted or jailbreaked. Two Apache Cordova plugins offer root and jailbreak detection mechanisms[4] for Android and iOS[5] devices. Both plugins check for specific files and folders typically present on rooted or jailbreaked devices.

These checks might well work for a large number of devices, but they do not counterfeit targeted attacks. Advanced attackers can simply render the detection mechanisms useless by renaming files and folders that are most commonly checked. Although more complex detection mechanisms exist, this still constitutes an arms race between attackers and developers aiming detection.

Currently, no root or jailbreak detection mechanisms have been deployed in the authentication app.

(O5) *Validate and Escape Input Data*

Web browsers interpret strings present within a `<script>` tag as JavaScript code. Therefore, special care has to be taken when processing data received from untrusted sources or user input.

In contrast to the typical web use case, mobile devices offer a greater range of possible unsafe input vectors [40]. For example, the Service Set Identifier (SSID) of wireless networks, contact data, SMS, QR codes and many other data sources might act as a channel for code injection. A Cordova-based SMS viewer, for example, might be susceptible to cross-site scripting attacks if JavaScript code is injected into the received messages. This shows that input validation and escaping is of particular relevance to hybrid mobile applications.

To counterfeit script injection, AngularJS escapes all data that is dynamically added to HTML templates by default[6]. In the case of plain HTML code that should be inserted, the `$sanitize` service can be used to remove unsafe instructions, i.e. JavaScript code. As long as the developer does not explicitly deactivates these protection mechanisms, any source code potentially encoded within QR codes, push notifications or user input will not be executed when being added to HTML pages[7].

(O6) *Do Not Run Code from Untrusted Sources*

Conventional web applications can rely on two protection mechanisms in place by the web browser. First of all, the browser sandbox hinders applications from accessing resources such as sensors or the file system of the underlying system. Second, the same-origin-policy prohibits applications from accessing data or source code from applications of a different origin.

With hybrid applications, a WebView is embedded within a native application. Access to resources of the underlying system is governed by the respective operating system's permission model. This means that web content running within the WebView, disregarding of its origin, has full access to the underlying system's APIs via Cordova plugins. This seems particularly drastic, as applications often tend to include external JavaScript code, i.e. for advertising purposes. Therefore, the official documentation on Apache Cordova highlights that it is not recommended to use iFrames unless the server that hosts the iFrame content is under the control of the developer as well [8].

[4] `https://github.com/trykovyura/cordova-plugin-root-detection`

[5] `https://github.com/leecrossley/cordova-plugin-jailbreak-detection`

[6] `https://docs.angularjs.org/api/ngSanitize/service/$sanitize`

[7] The JavaScript `eval(...)` function should be used with care. See the following resource for more details on safely using `eval()`: `https://www.owasp.org/index.php?title=JavaScript_Closure_Within_Eval`. None of the integrated authentication methods uses this function.

Apache Cordova uses a whitelisting mechanism in order to restrict access to external domain and disable loading of untrusted content. The integrated whitelisting mechanism has changed drastically with the release of Cordova version 4.0 in April 2015[8]. Since then, whitelisting is handled by the whitelist plugin[9]. Therefore, whitelisting policies have to be added to the applications *config.xml* file. The plugin includes the following whitelisting features:

- *Navigation Whitelisting*

  The developer can whitelist the URLs the WebView can be navigated to, e.g. by using JavaScript's `location.href` property. By default, only navigations to `file`:// URLs are allowed. In order to allow other URLs, the developer has to provide a whitelist using the <`allow−navigation`> tag.

  During inspection of the source code of the Apache Cordova framework for Android, we found that the navigation whitelist also limits access to the Cordova bridge, which enables calling native Cordova plugins from within the WebView[10].

- *Intent Whitelisting*

  The intent whitelist controls, which URLs the application is allowed to ask the system to open. By default, no external URLs are allowed. However, this whitelist does not apply to Cordova plugins, only to hyperlinks and calls to `window.open`() placed the application running in the WebView.

- *Network Request Whitelisting*

  In order to whitelist network requests, the developer can choose between specifying a whitelist by extending the *config.xml* file or using a CSP within the HTML template.

  When using the *config.xml* approach, the whitelist has to be added via <`access`> tags. This whitelist covers network requests such as `XMLHttpRequest` or embedded resources via < `img`> and <`script`> tags. If no whitelist is present, only `file`:// URLs are allowed, but newly created Cordova applications include <`access origin`="∗"> already and thus by default allow access to all URLs. The *config.xml* approach, however, does not cover all protocols. For example, on Android, the whitelist does not apply to Web Sockets requests, <`video` > and <`audio`> tags[11]. Therefore, it is recommended to rely on the CSP for whitelisting. The config.xml approach is thus mainly intended for WebViews that do not support CSP.

  The CSP offers a more fine-grained whitelisting mechanism. The developer can provide different settings for network requests, <`script`> tags and media resources. On Android, the CSP is available starting with version 4.4 of the Android operating system. Apple iOS devices offerCSP support starting with iOS 7. Mozilla provides a complete documentation on all available CSP directives [55].

The authentication app makes use of a CSP. Source code is only allowed to be loaded from local files and HTTP requests are limited to servers running authentication plugins. For the communication between the WebView and Cordova plugins it is required to set the options `unsafe−eval`, whereas `unsafe−inline` is needed for certain AngularJS features [7]. The whitelist plugin's navigation is used to restrict access to local files only.

(O7) *Detect Forged Mobile Applications*

[8] `https://cordova.apache.org/announcements/2015/04/15/cordova-android-4.0.0.html`

[9] `https://github.com/apache/cordova-plugin-whitelist`

[10] The Java class `https://github.com/apache/cordova-android/blob/master/framework/src/org/apache/cordova/CordovaBridge.java` includes the whitelisting mechanisms for accessing native URLs via plugins. Therefore, the navigation whitelist also "protects against random iframes being able to talk through the bridge" in order to "trust only pages which the app would have been allowed to navigate to anyway".

[11] `http://cordova.apache.org/docs/en/4.0.0/guide/appdev/whitelist/index.html`

Currently, there is no way for the authentication server to check whether an application on the mobile device has been tampered with or forged. In general, it is not possible to reliably protect against decompilation and repackaging of an application. As long as the application behaves like the legitimate application, the server cannot detect manipulations. Cryptographic keys or other tokens that might identify the legitimate application can be extracted and included in the forged application.

Android applications have to be signed using a self-signed certificate of the developer [6]. The signature mainly influences application updates and access to application resources. Two applications signed with the same cryptographic key can share resources and data on the device. Thus, Android developers, for example, could compare the cryptographic signature of the installed application to detect any tampering. Attackers, however, might remove these checks, repackage the application and distribute a forged application to the users.

To ensure that applications have been approved by Apple and have not been modified, Apple requires all applications to be signed using a certificate issued by Apple [13]. At runtime, the signature of the application code is verified in order to detect if the application has been tampered with. For attackers it is considerably harder to submit repackaged applications to Apple's AppStore, as Apple requires a paid Developer account backed by a valid credit card and also performs some checks before releasing applications. As users cannot install applications from other sources than the AppStore, attackers cannot easily distribute forged applications.

(O8) *Code Obfuscation*

Apache Cordova applications allow developers to write most of the code using web technologies, such as JavaScript, HTML5 and CSS, which is executed within WebViews. On Android, the resulting platform-independent source code is packaged in the *assets/* folder of the application. On iOS the files are bundled as resources or added bundled within a separate *www/* folder. Thus, by unpacking application packages the source code of the packaged JavaScript files is available easily. Therefore, application developers aiming to protect their source code, have to apply additional means of protection.

Often, the term code obfuscation and tools to minify or uglify code are used interchangeably. However, code obfuscation differs from pure code minimization. Code minimization tools are heavily used for reducing the size of JavaScript libraries that have to be downloaded from remote servers to the client. Minimization mainly happens by removing unnecessary characters in the source files. Code obfuscation tools, however, transform source code and control flow to be as incomprehensible as possible. When employing code obfuscation tools for Apache Cordova applications, it is particularly important that calls to JavaScript functions within HTML code and plugin-specific code are included in the obfuscation process as well. One tool for proper JavaScript obfuscation is *jscrambler*[12]. Nevertheless, code obfuscation should not be considered as a security feature. Rather, code obfuscation protects the intellectual property of the developers and make reverse-engineering more difficult, yet not impossible.

The developed authentication app does not yet feature code obfuscation, as it only presents a prototypical implementation. It has to be noted that in order to obfuscate applications that rely on the AngularJS framework and its dependency injection capabilities, the developer is required to change the way dependencies are declared[13].

(O9) *Employ State-of-the-Art Transport Security*

In order to communicate with authentication plugins on the server side, the application uses SSL/TLS connections. The WebView accesses the same truststore as the web browser, the system-

[12] `https://jscrambler.com/`

[13] See `https://docs.angularjs.org/tutorial/step_05` for a reference of changes required before minifying or obfuscating AngularJS applications

wide truststore. The server running the authentication plugins uses a certificate from a trusted certificate authority.

By modifying the behaviour of the WebView, a developer is able overwrite the handling of SSL/TLS errors and thus disable certificate verification entirely. Some resources recommend these modifications for testing with self-signed certificates[14]. However, there is always the risk that those modifications for testing remain present in the release version of the application as well. Android has introduced the debuggable flag in the manifest file. When debugging is enabled the system ignores SSL/TLS errors such as certificate validation or the use of self-signed certificates not present in the system-wide truststore. However, this configuration should not be used in production and thus should be disabled before building the release version of the application.

(O10) *Employ Certificate-Pinning*

Certificate pinning, also known as SSL pinning, is a technique for counterfeiting man-in-the-middle attacks by checking whether the certificate returned from the server matches the expected certificate. This implies that certificate pinning can only be employed in cases where the host to connect to is known in advance.

Apache Cordova does not provide support for certificate pinning. The official documentation states that "the main barrier to this is a lack of native APIs in Android for intercepting SSL connections to perform the check of the server's certificate. (Although it is possible to do certificate pinning on Android in Java using JSSE, the webview on Android is written in C++, and server connections are handled for you by the webview, so it is not possible to use Java and JSSE there.)" [8]. However, certificate pinnning can be realised on a per-request basis by using third-party Cordova plugins.

The Cordova HTTP[15] plugin, for example, allows the developer to supply a list of trusted certificates. Whenever the plugins `get()` and `post()` methods are called, a certificate check is enforced. The plugin currently supports Android and iOS devices. However, this plugin-based approach requires the developer to make use of the plugin's methods for each single HTTP request. Usually, hybrid applications use XMLHttpRequest or any of its wrappers for sending HTTP requests. Compared to XMLHttpRequest the plugin still has a range of limitations. For example, it is not supported to access the HTTP headers returned by the server. Furthermore, the plugin does not support automatic cookie handling, which might limit applicability for many mobile applications. Neither, HTTP requests originating from other Cordova plugins are covered.

The authentication app currently uses HTTP requests only during registration for sending the *Push ID* to the authentication plugin on the server. For these requests, the certificate-pinning mechanism from the Cordova HTTP plugin is used. This requires that the certificates of servers running trusted authentication plugins are packaged within the application. The downside of this plugin-based approach is that the CSP does not cover these requests, as only requests originating from the WebView, e.g. by using XMLHttpRequest are restricted but not calls from the native side. This actually presents a dilemma for application developers, either rely on the security mechanisms in place by the web browser and thus the WebView or trust in plugins that offer sending HTTP requests. Using plugins, the developer cannot be sure whether the plugin handles certificate validation correctly, provides a modified truststore or even disables verification entirely.

[14] `http://ivancevich.me/articles/ignoring-invalid-ssl-certificates-on-cordova-android-ios/`

[15] `https://github.com/wymsee/cordova-HTTP`

7.10 Specific Countermeasures

In the following, countermeasures specific to the integrated authentication methods are described.

7.10.1 Triple Key AES OTP

The Triple Key AES OTP method relies on three different cryptographic keys in order to compute the correct OTP: (a) a key received via the mobile platform's push-notification service, (b) a key encoded within a QR code, which has to be scanned by the user and (c) a key stored on the mobile device, using dedicated key-storage facilities. The cryptographic keys encrypt each other. Thus all three keys and an incrementing counter are required to compute the correct response. In addition to the application-wide countermeasures described in Section 7.9, countermeasures specific to this authentication method are detailed.

(O1) *Protected Storage of Sensitive Data*

The authentication method Triple Key AES OTP requires two sensitive data values stored permanently on the device: *Key_A* and the *Push ID*. Both values are stored using the mobile device's key-storage facilities. In the following, the platform-specific implications are given.

Android devices: On the Android platform, the Keystore is used. Android versions prior to version 6 do not support storing symmetric key strings. Thus the Base64-encoded *Key_A* and the *Push ID* strings are encrypted using a randomly generated RSA key pair[16]. The encrypted strings are stored on the file system in the internal storage of the authentication app. For each value that is stored in the Keystore a new random key pair is generated.

The security of the stored data mostly depends on the Keystore implementation variant. Several devices offer a hardware-backed Keystore, where the private keys cannot be extracted. Software-based implementations, however, use a key derived from the user's passcode. Depending on the considered threat agent, different security implications apply.

(TA1) Standard Attacker
An attacker that deploys malware on the mobile device does not have access to another application's data. Therefore, an attacker neither has access to Keystore entries of the authentication app nor to the encrypted values in the file system.

(TA2) Advanced Attacker
An attacker able to gain root access to the mobile device, basically, might control the entire operating system and thus access the files and Keystore entries of other applications. A leak of *Key_A* and the *Push ID*, however, do not yet allow for authenticating successfully. Still *Key_1_random* encoded in the push notification key and *Key_2_random* encoded within the QR code are needed. As push notifications can only be received on the device identified by the specific *Push ID* any further attacks have to be carried out on the device of the victim or the attacker has to forward incoming push notifications. Finally, the Triple Key AES OTP method is used in conjunction with a username and password plugin. Thus, the attacker has to brute force or phish the user's password on the desktop computer.

(TA3) Nearby Attacker
Software-based Keystore implementations: When extracting the device storage, an attacker can mount a brute-force attack on the passcode, in order to retrieve the master key that decrypts all Keystore entries.

[16] This implementation uses 2048-Bit RSA keys, as the Keystore API currently does not allow to specify the desired key size [58].

Hardware-backed Keystore implementations: The master key encrypting the Keystore entries is protected by hardware and the user's passcode. Thus, with hardware variants keys in the Keystore can only be accessed on the device. In our case, however, the sensitive data is stored on the file system but can only be decrypted with the key stored in the Keystore. An attacker, however, can try to bruteforce the private RSA keys required to decrypt *Key_A* and the *Push ID*.

In order to verify whether the decrypted string represents the correct *Key_A* the attacker has to query the authentication server. Therefore, the attacker needs the keys from the QR code and the push notification to compute an OTP. As *Key_1_random* and *Key_2_random* change for every transaction and the server only accepts a limited amount of incorrect OTPs, *Key_A* can be seen as secure.

A leaked *Push ID* does not enhance the attackers capabilities, as only the legitimate device associated with that *Push ID* receives push notifications.

(TA5) Web Attacker

An attacker that succeeds in injecting malicious JavaScript into the authentication app has full access to the native Cordova bridge and thus can access the application's Keystore entries. It has to be differentiated between malicious scripts running in an iFrame (e.g. an advertising service) and malicious scripts injected in the origin of the authentication app. In the latter case, the script has unconditional access to the Keystore entries. In case of an iFrame, the authentication app can protect the native Cordova bridge by setting the whitelisting mechanisms correctly. However, this feature is currently not documented by Apache Cordova but has been observed during source code inspection. If a web attacker has succeeded in injecting malicious JavaScript, we cannot reliably protect the sensitive data. However, we can protect against code injection and untrusted code in the first place.

iOS devices: For iOS devices, the Keychain is used for storing *Key_A* and the *Push ID*. Keychain entries are protected by a key derived from a hardware key and the user's passcode. In contrast to Android, the iOS Keychain supports the storage of arbitrary strings. Thus, no extra encryption is needed.

Teufl et al. [71] have highlighted the risk connected with encrypted iTunes backups where a user-chosen password is used to derive the key that encrypts the Keychain. An attacker can thus perform brute-force attacks on the password off-device. To counterfeit this threat, the authentication app stores all values using the *thisDeviceOnly* option of the Keychain. Meaning, the sensitive data will not be included in the iTunes backup and thus is only available on the user's device.

(TA1) Standard Attacker

An attacker that deploys malware on the mobile device does not have access to another application's Keychain entries.

(TA2) Advanced Attacker

On a jailbreaked device, the attacker might have full access to the operating system and other application's data. Equally to Android, access to *Key_A* alone, will not allow the attacker to authenticate successfully.

(TA3) Nearby Attacker

The attacker cannot access Keychain entries off-device.

(TA5) Web Attacker

The same security implications as for Android devices apply.

The listed security observations do not consider the encryption of the mobile device's file system. File system encryption adds an extra layer of protection, however, at least for Android devices we cannot assume that the device is encrypted, whereas on iOS the device lock and thus the file system encryption can be circumvented with a jailbreak.

(O11) *Protection of Server-Side APIs*

During registration, the user scans a QR code holding *Key_A* and some other data such as the plugin name, the domain and the return URL where the authentication app has to send the *Push ID*. Encoded within this return URL are the user ID and the plugin ID. Based on these two values the correct configuration data of the respective user can be updated on the server-side. The authentication app can only set the value for the *Push ID* once. Thus, an attacker gaining access to the URL cannot update the *Push ID* pointing to a device under her control.

(O12) *Protection of Sensitive Data in Remote Notifications*

This authentication method uses the mobile platform's push-notification service for distributing a symmetric key *Key_1_random* to the mobile device. In addition, the push notification contains a transaction id, the current domain and the name of the authentication method to start. The symmetric key presents sensitive data, as it influences the computation of the response. *Key_1_random* is encrypted using *Key_A* which is stored using the mobile device's Keychain or Keystore respectively. Thus, the confidentiality of *Key_1_random* is achieved as long as the attacker does not get hold off *Key_A*. The transmitted transaction id is used to map the data from the QR code to the correct push notification.

Currently, the integrity of the sent key is not protected. Manipulating the value of the encrypted *Key_1_random* further results in a wrong decrypted *Key_2_random*. As this key is used for the computation of the required OTP, a manipulation of the encrypted *Key_1_random* causes the computation of a wrong OTP and thus represents a denial-of-service attack, as the legitimate user cannot finish authentication.

(O13) *High Password Entropy*

No password is required for this authentication method. However, a typical authentication process consists of various subsequent authentication methods. The authentication process has to start with an authentication method, identifying the current user. This is currently realised as simple username and password plugin, where the user has to enter her credentials at the web browser.

(O14) *High Response Entropy*

The computed OTP consists of six digits, thus 10^6 different possible values. The server checks whether the computed OTP is correct. Therefore, the server-side authentication plugin limits the amount of tries, which counterfeits brute-force attacks. For each transaction a new *Key_1_random* and *Key_2_random* is computed and the used counter is incremented. Thus, no patterns regarding the layout of the OTP can be observed.

(O15) *High Cryptographic Key Entropy*

Each of the three cryptographic keys is realised as a randomly generated sequence of 256 Bits. The cryptographic keys are created on the server-side using a secure PRNG. An attacker aiming to brute force any of the cryptographic keys faces a substantial challenge: The attacker has no way of verifying whether the guessed key is correct without consulting the server. Only if each of the three cryptographic keys is correct, the correct OTP can be computed. In the case of a wrong cryptographic key, another value for the OTP is computed. Thus, the attacker is limited by the available tries granted by the server-side authentication plugin.

(O16) *Adequate Verification Info*

When entering a wrong OTP at the server-side authentication plugin, no valuable information is leaked to the attacker. The attacker is only informed of the entered value being wrong.

For all encryption processes, AES in ECB mode without padding was used. As all cryptographic keys have 256-bit no padding is required. In the case of wrong values for the cryptographic key, decryption does not result in an error but a wrong decrypted value.

(O17) *Explicit User Consent*

To complete authentication, the user is required to retrieve the third cryptographic key by scanning a QR code. Thus, the user explicitly has to perform an action and thereby expresses her consent. Furthermore, she has to manually enter the computed OTP at the authentication service in the web browser.

(O18) *Transaction-Specific Response*

This authentication method uses three different cryptographic keys for each authentication process. *Key_A* is stored permanently in the mobile device's key-storage facilities, however, a new pair of *Key_1_random* and *Key_2_random* is randomly generated for each transaction. Therefore, an OTP can only be used for one specific transaction, which counterfeits phishing attacks where an attacker uses a maliciously retrieved OTP for future authentication.

(O19) *Assure Legitimate User is Operating the Device*

The authentication method Triple Key AES OTP provides no assurance whether the legitimate user is operating the device covering the second factor. Anyone with access to the mobile device can successfully perform this authentication method. In practice, many devices comprise a passcode for locking the device. However, we cannot assume that the device is adequately protected.

(O20) *Assure Factor Possession is Required for Authentication*

This authentication method includes two mechanisms to ensure a binding to a particular mobile device: (a) the secret key *Key_A* stored in the key-storage facilities of the device and (b) by employing the push-notification service of the respective platform. Push notifications can only be received by a particular application on a mobile device identified by its *Push ID*.

(TA1) Standard Attacker

Key_A stored permanently on the mobile device cannot be extracted. Furthermore, an attacker cannot interfere with push notifications received on the device. Summarising, an attacker cannot clone the device covering the second factor.

(TA2) Advanced Attacker

An attacker with root access to the device might gain access to the sensitive data stored in the device's key-storage facilities. Furthermore, root access allows an attacker to control the entire operating system. Therefore, received push notifications can be intercepted and forwarded to third parties. Still, if an attacker manages to extract *Key_A* and intercepts push notifications, she has to get hold of the user's password that has to be entered on the desktop system for the username and password plugin.

(TA5) Web Attacker

In the case of a web attacker, two different scenarios have to be distinguished. If malicious JavaScript code has been embedded via an iFrame (e.g. an advertising service) the attacker might have access to the native Cordova bridge and thus can retrieve *Key_A* from the key-storage facilities. However, this can be mitigated by employing the whitelisting mechanisms correctly. JavaScript injected in the application, e.g. using not validated input vectors, on the other hand, is executed in the origin of the application and has unlimited access to the native Cordova bridge. This cannot be addressed by whitelisting mechanisms and therefore, highlights the need for consequent validation of all input data.

Summary

The Triple Key AES OTP method employs device binding by using the mobile platform's push-notification service and key material stored on Android and iOS devices. The authentication app has to adequately protect these two components against attackers. The main protection mechanisms are shortly summarised. Finally, some remaining problems are discussed

The authentication app stores the secret *Key_A* by using the Keystore on Android and the iOS Keychain. These key-storage facilities provide adequate protection against a standard attacker on the mobile device. Unfortunately, the key material cannot be protected against attackers with root access that are able to extract and manipulate the stored values. In the case of an attacker extracting the storage device, we have to distinguish between iOS and Android devices. On iOS devices, by default, a hardware-element is included in the key hierarchy. Thus keys can only be accessed on the iPhone. Whereas on Android devices, different implementations of the Keystore exist. For software-based implementations, an attacker that brute forces the user-chosen lock screen password can access the Keystore entries. Hardware-based implementations offer similar security to iOS.

There is a major risk of cryptographic material getting exposed to web attackers. Attackers that are able to inject or run malicious JavaScript code in the authentication app can easily access the cryptographic key material using the native Cordova bridge. Thus, it is of particular relevance to use whitelisting and escaping mechanisms offered by the framework.

The temporary keys distributed via the mobile platform's push-notification system are encrypted with the *Key_A* stored on the mobile device. An attacker that is able to access the push notification during transport or on the device thus cannot decrypt the key encoded within the push notification.

An attacker that is not able to access the raw key material might aim for brute-force attacks on the cryptographic keys or the response to the challenge. As a security measure, an attacker cannot verify on the device, whether the guessed values are correct. Only the server can do so, by verifying the entered OTP. Furthermore, the cryptographic keys encoded within the QR code and the push notification change for each transaction, which successfully counterfeits phishing attacks. A phished OTP, thus, cannot be used in any subsequent transactions.

Although, we were able to implement various measures to counter common security risks, still some problems exist. First of all, we cannot reliably protect our assets against an attacker with root access to the mobile device. Root access allows for controlling the entire operating system, which includes accessing cryptographic key material that has been stored in the Keystore or Keychain. In this case, the username and password entered on the desktop system represents the last line of defence. However, there exist various root detection methods for mobile devices. Along with exploits that allow attacker to gain root access, root detection methods constantly change. An attacker willing to hide root access can circumvent these detection methods. For example, many root detection methods check if specific files are present on the mobile device. An attacker can bypass these checks by renaming these files.

Similar considerations apply in the case of device theft. Apart from potential passwords used to lock the mobile device, the user is not required to supply additional knowledge on the device. Meaning, a thief can successfully operate the device covering the second factor. Only the username and password entered on the desktop system might hinder the thief from authenticating successfully.

Second, we cannot reliably protect the mobile application against forging. Attackers might decompile the application, modify the application's code and repackage the app. Android devices allow users to install applications from other sources than the official application store. This sideloading of applications enables attackers to distribute forged applications via third-party stores or by directly advertising them to the user. Whereas on iOS devices, applications can only be installed from AppStore. Thus, forged applications need to pass Apple's checks to be able to make it into AppStore, which somewhat increases security.

The authentication app and the server running authentication plugins deploy state-of-the-art transport layer security. To counterfeit man-in-the-middle attacks, certificate pinning has been integrated. However, currently, the API on the server side requires no authentication from the mobile device. During registration, the user scans a QR code that includes her user id, a plugin id, *Key_A* and the URL the *Push ID* should be sent to. If an attacker gets hold of the user id, the plugin id and the URL, requests including the *Push ID* of any other device can successfully update the *Push ID* on the server side. A manipulated *Push ID* allows the attacker to receive push notifications intended for the legitimate user on any other device.

Summarising, our authentication app and mobile applications in general, cannot be reliably protected against attackers with root access to the device. However, by deploying security measures such as strict whitelisting, input validation and by using hardware-backed key storage we counterfeit a large number of threats.

7.10.2 Triple Key AES OTP with Knowledge Proof

This authentication method Triple Key AES OTP with Knowledge Proof combines a possession proof and the factor knowledge. The user proves possession similar to the Triple Key AES OTP method by using a key stored using the device's key-storage facilities and by assuring device binding by using the mobile platform's push-notification system. In addition to scanning a QR code, the user has to enter a password on the mobile device.

In total, three different cryptographic keys are needed to compute the correct OTP: (a) *Key_A* stored on the mobile device, (b) *Key_PW_Derived* derived from the user's password and a random *salt* and (c) the transaction-specific *Key_random*. *Key_PW_Derived* is used to encrypt *Key_random*, which is displayed to the user encoded in a QR code and *Key_A* encrypts the random *nonce* sent via the mobile platform's push-notification service.

In addition to the application-wide countermeasures described in Section 7.9, countermeasures specific to this authentication method are detailed.

(O1) *Protected Storage of Sensitive Data*

The authentication method Triple Key AES OTP with Knowledge Proof requires two sensitive data values stored permanently on the device: *Key_A* used to decrypt the *nonce* and the *salt* value required for password-based key derivation.

In general, similar considerations as for the authentication method Triple Key AES OTP apply. In the following, differences are given.

Android devices: Equally to Triple Key AES OTP, a simply key wrapping mechanism has been deployed. *Key_A*, the *Push ID* and the *salt* value are encrypted using a randomly generated RSA key pair.

(TA1) Standard Attacker

An attacker that deploys malware on the mobile device does not have access to another application's data. Therefore, an attacker neither has access to Keystore entries of the authentication app nor to the encrypted values in the file system.

(TA2) Advanced Attacker

An attacker able to gain root access to the mobile device, basically, might control the entire operating system and thus access the files and Keystore entries of other applications. A leak of *Key_A*, the *salt* and the *Push ID*, however, does not yet allow for authenticating successfully. Still, the *nonce* encoded in the push notification, *Key_PW_Derived* derived from the user's password and *Key_random* encoded within the QR code are needed. As push notifications can only be received on the device identified by the specific *Push ID*, any further attacks have to be carried out on the device of the victim or the attacker has to intercept incoming push notifications and forward them to the attacker. Finally, this authentication method requires the user to enter two passwords. Thus, an attacker has to brute force or phish the user's password on the desktop computer and the password that is used for key derivation on the mobile device, which further increases the complexity of the attack.

(TA3) Nearby Attacker

An attacker that has achieved to decrypt *Key_A* and the *salt*, still needs to bruteforce the password. To verify whether the *Key_PW_Derived* is correct, the attacker has to query the authentication server. Therefore, the attacker needs the *Key_random* from the QR code and the *nonce* from the push notification for computing the OTP. As *Key_random* and the *nonce* change for every transaction and the server only accepts a limited amount of incorrect OTPs, brute-force attacks do not present a realistic threat scenario.

A leaked *Push ID* does not enhance the attackers capabilities, as only the legitimate device associated with that *Push ID* receives push notifications.

(TA5) Web Attacker

An attacker that succeeds in injecting malicious JavaScript into the authentication app has full access to the native Cordova bridge and thus can access the application's Keystore entries. It has to be differentiated between malicious scripts running in an iFrame (e.g. an advertising service) and malicious scripts injected in the origin of the authentication app. In the latter case, the script has unconditional access to the Keystore entries. In the case of an iFrame, the authentication app can protect the native Cordova bridge by setting the whitelisting mechanisms correctly. If a web attacker has succeeded in injecting malicious JavaScript, we cannot reliably protect the sensitive data. However, we can protect against code injection and untrusted code in the first place.

iOS devices: For iOS devices the Keychain is used for storing *Key_A*, the *Push ID* and the *salt* value. The authentication app stores all values using the *thisDeviceOnly* option of the Keychain.

(TA1) Standard Attacker

An attacker that deploys malware on the mobile device does not have access to another application's Keychain entries.

(TA2) Advanced Attacker

On a jailbreaked device, the attacker might have full access to the operating system and other application's data. Equally to Android, access to *Key_A*, the *Push ID* and the *salt*, will not allow the attacker to authenticate successfully.

(TA3) Nearby Attacker

The attacker cannot access Keychain entries off-device.

(TA5) Web Attacker

The same security implications as for Android devices apply.

(O11) *Protection of Server-Side APIs*

The same considerations as for the authentication method Triple Key AES OTP apply.

(O12) *Protection of Sensitive Data in Remote Notifications*

This authentication method uses the mobile platform's push-notification service for distributing a *nonce* to the mobile device. In addition, the push notification contains a transaction id, the current domain and the name of the authentication method to start. The *nonce* presents sensitive data, as it influences the computation of the response. The *nonce* is encrypted using *Key_A* which is stored using the mobile device's Keychain or Keystore respectively. Thus, the confidentiality of the *nonce* is achieved as long as the attacker does not get hold off *Key_A*. The transmitted transaction id is used to map the data from the QR code to the correct push notification.

Currently, the integrity of the sent *nonce* is not protected. As the *nonce* is used for the computation of the required OTP, a manipulation of the encrypted *nonce* causes the computation of a wrong OTP and thus represents a denial-of-service attack, as the legitimate user cannot finish authentication.

(O13) *High Password Entropy*

Currently, no password policy is deployed within the mobile authentication app. Thus, users might choose weak passwords. In addition to guessing the chosen password, the *salt* value either has to be extracted from the key-storage facilities or guessed as well. However, for verifying whether the thus resulting *Key_PW_Derived* is correct, the attacker has to contact the authentication service. On the mobile device, no information on the correctness of the derived key is given. For the encryption of *Key_random* AES is used in ECB mode without any padding in place. Decrypting *Key_random* with an incorrect *Key_PW_Derived* results in a different value *Key_random*, but will not fail and thereby giving valuable information to the attacker.

Future work could further enhance security by enforcing a password policy.

(O14) *High Response Entropy*

The computed OTP consists of six digits, thus 10^6 different possible values. The server checks whether the computed OTP is correct. Therefore, the server-side authentication plugin limits the amount of tries, which counterfeits brute-force attacks. For each transaction a new *Key_random* and random *nonce* are generated, thus, no patterns regarding the layout of the OTP can be observed.

(O15) *High Cryptographic Key Entropy*

Cryptographic keys are randomly generated sequences of 256 Bits. The cryptographic keys are created on the server-side using a secure PRNG. *Key_PW_Derived* is derived using a secret user-chosen password and a *salt* value stored on the device. For key derivation, the SCrypt algorithm is used.

An attacker aiming to brute force any of the cryptographic keys or the password and *salt* value faces a substantial challenge: The attacker has no way of verifying whether the guessed values are correct without consulting the server. Only if each of the values are correct, the correct OTP can be computed. In the case of a wrong input value, a different OTP gets computed. Thus, the attacker is limited by the available tries granted by the server-side authentication plugin.

(O16) *Adequate Verification Info*

When entering a wrong OTP at the server-side authentication plugin, no valuable information is leaked to the attacker. The attacker is only informed about the entered value being wrong.

For all encryption processes, AES in ECB mode without padding was used. As all cryptographic keys and the *nonce* have 256-bit, no padding is required[17]. In the case of wrong values for the cryptographic key, decryption does not result in an error but in a wrong decrypted value.

(O17) *Explicit User Consent*

To complete authentication, the authentication app requires three different cryptographic keys. Both, *Key_PW_Derived* and *Key_random* can only be obtained with explicit user involvement. As the user is required to scan a QR code and enter a secret password, she thereby expresses her consent. Furthermore, she has to manually enter the computed OTP at the authentication service in the web browser.

[17] It has to be noted that although a 256-bit *nonce* is sent to the authentication app, only the last 32 bit act as input for the OTP computation. However, this does not decrease security, as still, the attacker can verify the correctness of the OTP only by querying the server.

(O18) *Transaction-Specific Response*

This authentication method uses two different cryptographic keys and a random *nonce* for each authentication. *Key_A* is stored permanently in the mobile device's key-storage facilities, however, *Key_random* and the *nonce* are randomly generated for each transaction. Therefore, an OTP can only be used for one specific transaction, which counterfeits phishing attacks where an attacker uses a maliciously retrieved OTP for future authentication.

(O19) *Assure Legitimate User is Operating the Device*

The transaction-specific *Key_random* used for OTP computation is encrypted with a key derived from the user's password. The password is never stored locally on the device and has to be entered for each authentication transaction. If the user-supplied a correct OTP to the server, the authentication plugin on the server side can assume that the legitimate user is operating the device covering the second factor.

(O20) *Assure Factor Possession is Required for Authentication*

The same considerations as for the authentication method Triple Key AES OTP apply.

Summary

Similar to the Triple Key AES OTP method this authentication method employs device binding by using the mobile platform's push-notification service and key material stored on Android and iOS devices. Furthermore, a key derived from a user-chosen password makes sure that only legitimate persons can operate the device covering the second factor. A summary on security implication of Triple Key AES OTP with Knowledge Proof is given.

The user needs to enter a password in order to decrypt the transaction-specific *Key_PW_Derived* encoded within the QR code. The password is neither stored on the mobile device nor on the server. The server only stores the derived key and the salt. Compared to the Triple Key AES OTP method, the use of an additional password somewhat increases the barrier for thieves or an attacker with root access to the device. Having access to the device or the data stored in the key-storage facilities and the operating system to be able to intercept push notifications, still does not allow for successful authentication. The attacker has to apply attacks on the password as well. For an attacker with root access to the device, this might be realised by installing a keylogger, a thief, however, is required to employ brute-force attacks on the password. Equally to brute-force attacks on the keying material, an attacker cannot check the correctness of the entered password on the mobile device. A wrong password leads to a different value for *Key_PW_Derived* and thus results in a different OTP.

Summarising, Triple Key AES OTP with Knowledge Proof slightly enhances the security compared to the authentication method Triple Key AES OTP. The additional password input on the mobile device offers valuable protection in case of theft and increases the complexity for attackers aiming to clone the factor possession or process unauthorised authentication transactions on the legitimate user's device.

7.10.3 Double Key AES OTP with Knowledge Proof

Double Key AES with Knowledge Proof is designed for the mobile-only use. Therefore, the user is not required to scan a QR code. Double Key AES with Knowledge Proof requires two cryptographic keys, *Key_A* and *Key_PW_Derived* respectively, and a random *nonce* for computing the correct OTP. *Key_A* is stored in the key store on the mobile device and the *Key_PW_Derived* is derived from a user-supplied password. The password-derived key is used to decrypt the random *nonce* sent via the mobile platform's push-notification service.

In addition to the application-wide countermeasures described in Section 7.9, countermeasures specific to this authentication method are detailed.

(O1) *Protected Storage of Sensitive Data*

The same considerations as for the authentication method Triple Key AES OTP with Knowledge Proof apply.

(O11) *Protection of Server-Side APIs*

The same considerations as for the authentication method Triple Key AES OTP apply.

(O12) *Protection of Sensitive Data in Remote Notifications*

This authentication method uses the mobile platform's push-notification service for distributing the encrypted *nonce* to the mobile device. In addition, the push notification contains a transaction id, the current domain and the name of the authentication method to start. The *nonce* presents sensitive data, as it influences the computation of the response. The *nonce* is encrypted using *Key_PW_Derived* which is only known to the user. Thus, the confidentiality of the *nonce* is achieved as long as the attacker does not get hold off *Key_PW_Derived*. The transmitted transaction id is used to map the data from the QR code to the correct push notification.

Currently, the integrity of the sent *nonce* is not protected. As the *nonce* is used for the computation of the required OTP, a manipulation of the encrypted *nonce* causes the computation of a wrong OTP and thus represents a denial-of-service attack, as the legitimate user cannot finish authentication.

(O13) *High Password Entropy*

No password policy is deployed within the mobile authentication app. Thus, users might choose weak passwords. In addition to guessing the chosen password, the *salt* value either has to be extracted from the key-storage facilities or guessed as well. However, for verifying whether the thus resulting *Key_PW_Derived* is correct, the attacker has to contact the authentication service. On the mobile device, no information on the correctness of the derived key is given, as for encrypting the *nonce* AES is used in ECB mode without any padding in place. Decrypting the *nonce* with an incorrect *Key_PW_Derived* results in a different value for the *nonce*, but will not fail and thereby giving valuable information to the attacker.

(O14) *High Response Entropy*

The computed OTP consists of six digits, thus 10^6 different possible values. The server checks whether the computed OTP is correct. Therefore, the server-side authentication plugin limits the amount of tries, which counterfeits brute-force attacks. For each transaction a new random *nonce* is computed. The *nonce* and the permanent key *Key_A* are used for the computation of the OTP.

(O15) *High Cryptographic Key Entropy*

The same considerations as for the authentication method Triple Key AES OTP with Knowledge Proof apply.

(O16) *Adequate Verification Info*

When entering a wrong OTP at the server-side authentication plugin, no valuable information is leaked to the attacker. The attacker is only informed of the entered value being wrong.

For all encryption processes, AES in ECB mode without padding was used. As all cryptographic keys and the *nonce* have 256-bit, no padding is required[18]. In the case of wrong values for the cryptographic key, decryption does not result in an error but in a wrong decrypted value.

[18] It has to be noted that although a 256-bit *nonce* is sent to the authentication app, only the last 32 bit act as input for the OTP computation. However, this does not decrease security, as still, the attacker can verify the correctness of the OTP only by querying the server.

(O17) *Explicit User Consent*

In order to complete authentication, the user is required to enter a secret password and thereby expresses her consent. Furthermore, she has to manually enter the computed OTP at the authentication service in the web browser.

(O18) *Transaction-Specific Response*

This authentication method uses a random *nonce* for each authentication process. *Key_A* is stored permanently in the mobile device's key-storage facilities, however, the *nonce* is randomly generated for each transaction. Therefore, an OTP can only be used for one specific transaction, which counterfeits phishing attacks where an attacker uses a maliciously retrieved OTP for future authentication transactions.

(O19) *Assure Legitimate User is Operating the Device*

The *nonce* used for OTP computation is encrypted with a key derived from the user's password. The password is never stored locally on the device and has to be entered for each authentication transaction. If the user has supplied a correct OTP to the server, the authentication plugin on the server side can assume that the legitimate user is operating the device covering the second factor.

(O20) *Assure Factor Possession is Required for Authentication*

This authentication method includes two mechanisms to ensure a binding to a particular mobile device: (a) the secret key *Key_A* stored in the key-storage facilities of the device and (b) by employing the push-notification service of the respective platform. Push notifications can only be received by a particular application on a mobile device identified by its *Push ID*.

(TA1) Standard Attacker

Key_A stored permanently on the mobile device cannot be extracted. Furthermore, an attacker cannot interfere with push notifications received on the device. Summarising, an attacker cannot clone the device covering the second factor.

(TA2) Advanced Attacker

An attacker with root access to the device might gain access to the sensitive data stored in the device's key-storage facilities. Furthermore, root access allows an attacker to control the entire operating system. Therefore, received push notifications can be intercepted and forwarded to third parties and keyloggers can be installed. As this authentication method is designed for the mobile-only use, username and password will be entered within the mobile browser. Thus an attacker with root acceess can collect all data required for a successful authentication.

(TA5) Web Attacker

In the case of a web attacker, two different scenarios have to be distinguished. If malicious JavaScript code has been embedded via an iFrame (e.g. an advertising service) the attacker might have access to the native Cordova bridge and thus can retrieve *Key_A* from the key-storage facilities. However, this can be mitigated by employing the whitelisting mechanisms correctly. JavaScript injected in the application, e.g. using not validated input vectors, on the other hand, is executed in the origin of the application and has unlimited access to the native Cordova bridge. This cannot be addressed by whitelisting mechanisms and therefore, highlights the need for consequent validation of all input data.

Summary

The authentication method Double Key AES OTP with Knowledge Proof is tailored to the mobile-only use case. The user enters her username and password at the mobile web browser and finishes authentication using the mobile authentication app. Enabling both steps to be carried out on the mobile devices, conceptually decreases security as an attacker only has to compromise one device.

To hinder an attacker to clone the device, the same measures as with the preceding two authentication methods are in place. Key material is protected using the mobile device's key-storage facilities. The *nonce* required for OTP computation is encrypted with a key derived from a user-chosen password and sent via the push-notification service. In the case of device theft, the attacker cannot perform authentication without gaining knowledge of the password used to derive *Key_PW_Derived* and thus decrypt the *nonce*.

However, different security implications apply for an on-device attack. Whilst key material and the password are protected against standard attackers, web attackers and attackers with root access pose a serious threat. A web attacker might be able to access the native Cordova bridge and thus the cryptographic key material and the salt value stored on the device. The attacker can also invoke the PIN dialog to phish the password used to derive *Key_PW_Derived*. To counterfeit this threat, special care has to be taken with input validation. Furthermore, by configuring the whitelisting mechanisms adequately source code from other origins is denied execution.

Similar considerations apply to attackers with root access to the mobile device. By gaining root access, all key store entries of the mobile authentication app can be accessed. As attackers with root access control the entire operating system, keyloggers can be installed for stealing the username and password in the mobile web browser and the password used for key derivation.

Although, this authentication method successfully counterfeits threat scenarios such as theft or extracting device storage, it cannot protect against attackers with root access to the device. Authentication methods that rely on the user to enter her credentials on the desktop computer implement a second line of defence, whereas this authentication method is entirely exposed to the attacker. However, this dilemma is omnipresent for all type of mobile applications and, unfortunately, could not be resolved in the course of this work.

7.11 Known Issues

The mobile device ecosystem is extremely fragmented. As developing a product for different platforms generates a tremendous development effort, more application developers make use of cross-platform frameworks. Hybrid frameworks that allow the development of web applications packaged within a native mobile application whilst still having access to native device APIs, present the most popular approach for cross-platform development [76]. Apache Cordova is currently the most popular hybrid framework. However, from a security perspective, these hybrid frameworks enclose a serious of vulnerabilities and conceptual security risks. In this section, we highlight known security issues in hybrid application development with particular reference to the Apache Cordova framework that was used for the implementation of our authentication framework.

Georgiev et al. [24] present a comprehensive work on the conceptual security problems of hybrid application development. The authors criticise that hybrid applications do not fully align with the basic principles of web security: The same-origin policy and the web browser's sandboxing mechanisms. First, conventional applications are executed within the sandbox of the browser and cannot access the underlying device features, such as the file system or sensors. Hybrid frameworks, however, expose device features to applications running within a WebView. Malicious code that has been injected into the application or loaded from untrusted sources, therefore, can break out of the sandbox and gain access to the user's data, for example, her contact list, photos, files on the file system and many more. Second, although the WebView enforces the same-origin policy, the hybrid framework does not support the notion of origins. Meaning, an iFrame that includes third-party code, for example, an advertising service can access data from the main frame that has been stored on the filesystem or even worse, stored using the device's key-storage facilities. This disregard of the same-origin policy even goes further than that.

Some platforms are prone to frame confusion, where the hybrid frameworks forwards the results of native API calls to the wrong frame within the WebView. In the following, more details on the described vulnerabilities are given. It has to be noted that the highlighted problems do not only affect Apache Cordova but other hybrid frameworks as well.

Hybrid frameworks fail to guarantee that content from untrusted domains cannot access the native APIs available through the hybrid framework. The security architecture of Apache Cordova, for example, allows the developer to whitelist trusted URLs. Thereby, only content from these domains can be loaded. The business model of many applications, however, relies on embedding advertising services from third-parties. To counterfeit this security risk, the authors have proposed their extension to Apache Cordova called *NoFrak*, which requests each origin to authenticate with a secret token before being allowed to access the native bridge. Their idea is to store this token into the Local Storage of the WebView that is only accessible by the legitimate origin. It appears that the proposed solution has been integrated into the Apache Cordova framework for Android. However, no official documentation on this mechanism exist. During inspection of the source code of the native Apache Cordova for Android devices, we stumbled across the *bridge secret*[19]. The *bridge secret* is a randomly generated string only supplied to origins that have been added to the navigation whitelist[20] and is stored as local variable within the JavaScript part of the Apache Cordova framework for Android. It has to be noted that this information is from source code inspection, no official documentation on this mechanism exist. In November 2015, Apache Cordova announced an open bug [21] targeting the lack of randomness in the generated *bridge secret* in older version of the Apache Cordova framework for Android. As this features is not documented officially, we recommend not to include content of untrusted sources. The implemented authentication app, therefore, does not include any third-party content in iFrames and does only rely on JavaScript libraries, which are already packaged within the Cordova application.

There are different ways to communicate between the JavaScript part of a hybrid framework and the native library. On Windows Phone and iOS devices, JavaScript strings constructed on the native side can be used to inject JavaScript into the WebView. For Android devices, JavaScript code can be added to the `loadURL`(...) method of the WebView. However, for all three platforms this code is executed in the main frame, which presents a problem for a legitimate iFrame wanting to access native device APIs. This behaviour has also been observed when events delivered to the main frame are used as a means of communication between the native framework and the JavaScript side of the framework. Luo et al. [51] describe this observation as the problem of "frame confusion", where WebView components incorrectly deliver their responses to the main frame. Commonly, the legitimate application is running on the main frame. However, malicious JavaScript running inside an iFrame can still call the native bridge, e.g. to delete data on the device or send SMS, but will not receive the response from the native side.

In 2014, Kaplan et al. [44] from IBM Security Systems have released a list of serious vulnerabilities in the Apache Cordova framework, which even allow for remote exploitation[22]. By combining the found vulnerabilities, they could perform the exploit remotely, by simply tricking the user to browse to a malicious website. The malicious website causes the user's web browser to download an HTML file and store the file on the device's SD card. By generating an Intent object, the website causes a vulnerable Cordova application to start. The website passes the path to the downloaded HTML file as a parameter with the Intent. The vulnerable Cordova application loads the downloaded HTML file into the WebView and thus gains access to any data or native APIs the application has access to. This exploit can even be used to forward data to the attacker or steal sensitive information, such as cookies, as the HTML file can include arbitrary JavaScript code. These vulnerabilities have been fixed with the release of Cordova

[19] `https://github.com/apache/cordova-android/blob/master/cordova-js-src/exec.js`

[20] `https://github.com/apache/cordova-android/blob/master/framework/src/org/apache/cordova/CordovaBridge.java` and `https://github.com/apache/cordova-android/blob/master/framework/src/org/apache/cordova/CordovaPlugin.java`

[21] `https://cordova.apache.org/announcements/2015/11/20/security.html`

[22] A list of all registered Apache Cordova CVE's can be found here: `http://www.cvedetails.com/vulnerability-list/vendor_id-45/product_id-27153/Apache-Cordova.html`

3.5.1, the new whitelisting plugin and support for CSP.

Rafay Baloch has found multiple vulnerabilities in the Android default browser that also affect the WebView on Android devices. Due to erroneous handling of null bytes by the URL parser of the browser, attackers can bypass the same-origin-policy and access data from other domains or take over authenticated user sessions [62]. He found a second bug that also allows attackers to bypass the same-origin-policy [60]. Whilst the bug was fixed in Google Chrome years ago, the fix was not integrated in the Android default browser. In May 2015, he discovered that the Android browser enables URL spoofing attacks [61]. This vulnerability results from improper handling of the HTTP error "No Response", error code 204 and allows an attacker to open a new tab and to visit a different website than shown in the address bar.

However, these and various other vulnerabilities remain still unfixed on a large number of Android devices. Android faces a serious problem with operating system fragmentation. Each operating system update or security patch has to be delivered from Google to the OEMs and carriers before reaching the devices. This update policy causes many devices to still run old Android versions with a significant number of security vulnerabilities. In November 2015, only 25 percent of devices run Android version 5 or higher [29]. Changing the WebView to the Chromium WebView with version 4.4 counterfeits various security problems. However, until Android version 5, the WebView component could only be updated in the course of a firmware update, causing the previously described security vulnerabilities still being present on a large number of devices. The following listing gives an overview of the used WebView components and the respective update policy [49].

- *Android 1.x - 3.x*
  Both the included web browser and the WebView are based on WebKit and can only be updated via a firmware update.

- *Android 4.x - 4.3*
  Google ships Chrome as default browser, which is updated via Google Play. However, the WebView is still based on WebKit and can only be updated via a firmware update.

- *Android 4.4 - 4.4.4*
  Chrome is shipped as default browser and the WebView is now also based on Chromium. However, to update the WebView it is still required to install a firmware update.

- *Android 5.0 and Beyond*
  Both, Chrome and the WebView based on Chromium are updated via Google Play.

In contrast to Android, Apple directly distributes firmware updates to iOS devices. In November 2015, already 67 percent of all devices are running iOS 9, which was released only two months earlier [11]. The fast adoption of new operating system versions and the direct distribution of updates allows for security vulnerabilities being closed promptly. Thus, compared to Android, exploits in the WebView components cannot be misused still years after their disclosure.

7.12 Discussion

The conducted security evaluation reveals that the TOE does not fulfill all of the derived security objectives. Table B.4 illustrates the evaluation results for each of the integrated authentication methods. This section discusses the remaining security risks and assesses the reasons for the lack of corresponding countermeasures.

The TOE fails to meet the security objectives *(O4) Detect Rooted and Jailbreaked Devices*, *(O7) Detect Forged Mobile Applications* and *(O8) Code Obfuscation*. The authentication app does not employ root detection and code obfuscation for various reasons. First, the application is still in prototype stage. In the case of the developed components being used in a productive environment, these measures should be put in place. Second and most importantly, obfuscation does not reliably counterfeit reverse-engineering and forged applications, similarly, advanced attackers can circumvent root detection mechanisms. This can be clarified with the aid of an example. In October 2014, security issues present with the pushTAN method of the German bank Sparkasse have been disclosed [43]. The involved security researchers have released a detailed analysis of the employed security measures and show why the Android version of the online banking solution of Sparkasse is prone to attacks [31]. In the following, the observations of the authors are summarised.

Since 2014, the German Sparkasse offers a mobile banking application for Android and iOS devices, which is called *Sparkasse*[23]. In order to authorise transactions, Sparkasse makes use of a separate mobile application, the S-pushTAN app[24], which is advertised as being certified by the German TÜV. Transaction processing involves the following steps: First, the user enters the transaction details at the Sparkasse app. The transaction details are transferred to the server that in turn sends an encrypted TAN to the S-pushTAN app. After that, the user verifies the transaction details displayed in the S-pushTAN app. By confirming the transaction, the TAN is transferred to the Sparkasse app using inter-application communication. Thus, the user does not need to manually enter the TAN at the online-banking application. The S-pushTAN app includes various security measures:

- When starting, both apps require the user to enter a PIN. In case a wrong PIN is entered five times, all application data gets wiped.
- A custom keyboard for counterfeiting keyloggers potentially included in third-party keyboards.
- Device fingerprinting based on hardware features and the installed Android version should counterfeit cloning of the S-pushTAN app.
- Root detection mechanisms provided by the PROMON shield[25] framework allow the app to refuse execution on rooted devices.
- The S-pushTAN app employs code obfuscation using ProGuard[26].
- Certificate pinning is used to counter man-in-the-middle attacks.

Haupert and Müller present an attack that lets them manipulate the recipient and the amount in banking transactions. The security researchers apply instrumentation to both applications. Therefore, they use the Xposed framework[27], which allows to change the behavior of the Android system and the installed applications without modifying any application packages. For installing Xposed, the Android device has to be rooted. Even if the device has not been rooted by the user intentionally, there exist malware that is able to perform root exploits. Recent discoveries have shown that malicious applications that perform root exploits can even make it into the official application store [5]. With the help of the Xposed framework, the security researchers are able to disable root detection mechanisms and manipulate the transaction data sent from the Sparkasse app to the server. Thus, the server issues a TAN to a different transaction as intended by the user. The displayed transaction data in the S-pushTAN app is manipulated to show the original transaction details. Thus the user cannot detect transaction manipulation.

Summarising, the detailed analysis of the S-pushTAN application shows that no reliable protection mechanisms against attackers with root access exist. That way, it has to be admitted that authentication cannot be securely realised in a mobile-only scenario with only a single device in place. Whereas,

[23] https://play.google.com/store/apps/details?id=com.starfinanz.smob.android.sfinanzstatus

[24] https://play.google.com/store/apps/details?id=com.starfinanz.mobile.android.pushtan

[25] http://www.promon.no/products/promon-shield/

[26] http://proguard.sourceforge.net/

[27] http://repo.xposed.info/module/de.robv.android.xposed.installer

two authentication methods require username and password to be entered on the desktop computer, the authentication method Double Key AES OTP with Knowledge Proof aims to be applicable in mobile-only scenarios where authentication is triggered within the web browser on the mobile device. However, an attacker with root access might access the key material stored on the device, install a keylogger or intercept and forward received push notifications. Therefore, the authentication method Double Key AES OTP with Knowledge Proof fails to meet the objective *(O20) Assure Factor Possession is Required for Authentication* in mobile-only scenarios.

Currently, no measures are in place to detect the manipulation of sensitive data, such as key material, stored on the mobile device. Thus, the TOE does not fulfill the objective *(O3) Detect Manipulation of Sensitive Data*. This has various reasons. Data stored using the Android Keystore and the iOS Keychain is protected against access from standard malware on the mobile device. Other applications cannot access the mobile authentication app's cryptographic key material. As they are not able to access this data, no manipulation can be performed. The situation is different for attackers with root access to the device or web attackers that have gained access to the native Cordova bridge. These attackers can access the cryptographic key in the Keystore and Keychain. However, there is no reliable way to protect against data manipulation. One possibility is to compute a cryptographic signature or message authentication code for each stored entry. This requires additional key material residing on the device. The question arises, how to protect this key material against illegitimate access. If it is stored using the key-storage facilities, the attacker can access it as well. If it is stored using, for example, the Local Storage, attackers with root access or web attackers can gain access to the key pair and only recompute the signature after data manipulation. To counterfeit data manipulation by web attackers, applications should hinder code injection and execution of code from untrusted sources. As previously assessed, there is no reliable protection against attackers with root access to the mobile device.

During registration of new authentication methods, the authentication app transmits the *Push ID* to the server-side authentication plugin. The previously scanned QR code includes the therefore used endpoint. Currently, the implementation allows the *Push ID* to be set only once. Thus, a leaked endpoint does not allow the attacker being able to overwrite the *Push ID* of the legitimate user. However, no measures are in place for protecting the user against an attacker that contacts the server before the authentication app. In the case of not being able to set the *Push ID*, an error message is returned to the authentication app. It is the responsability of the user to take action and disable the authentication plugin on the server.

We can conclude the conducted security evaluation with the résumé of Haupert and Müller [31]. Similar to online banking, multi-factor authentication using a single mobile device faces conceptional weaknesses. Mobile applications cannot be fully secured to counterfeit state-of-the-art attacks involving root exploits. Thus it is recommended to use a second independent factor. In case of the developed authentication framework, this is realised by entering username and password on a separate device, such as a desktop computer.

Chapter 8

Conclusion

During the past few years, technological advances have allowed users to progressively use their mobile devices in everyday life. Tasks that have required the use of a desktop computer a few years ago can now be carried out on the go, by using mobile devices. Many online services process sensitive data about the user or enable users to perform transactions on their behalf, e.g. online-banking services allow users to authorise money transfers. Online services thus need to employ adequate access protection mechanisms, to ensure that only legitimate users access the service. As the username and password scheme is prone to attacks, many services have introduced multi-factor authentication. These multi-factor authentication methods might include presenting a token, such as a USB token or smart card, or rely on TANs only valid for a limited period. However, the deployed multi-factor authentication mechanisms are often not applicable for the use with mobile devices.

In this thesis, we have developed a multi-factor authentication framework for mobile devices. The developed framework allows to easily add new authentication methods and thus present the user with multiple methods to choose from. We have successfully integrated three different authentication methods that leverage the security features offered by mobile devices. To support multiple mobile platforms, a state-of-the-art cross-platform development framework has been used for the implementation of the mobile components. Apache Cordova, arguably the most popular cross-platform development framework, allows developers to build applications using web technologies such as HTML, JavaScript and CSS. Applications are executed in a WebView, a browser window embedded within a native mobile application. Applications running within the WebView can access native device APIs by using plugins.

Multi-factor authentication imposes high security requirements. For example, the device covering the factor possession needs to be protected against cloning. Therefore, the implemented authentication methods employ device binding by combining the device's key-storage facilities and the mobile platform's push-notification services. Furthermore, the developed application has to provide protection against an attacker aiming to inject malicious code or intercepting the communication between the authentication server and the mobile device. If any of these components is not properly protected, an attacker might gain access to a user's sensitive data or perform actions on behalf of the user.

To evaluate the security of the developed multi-factor authentication framework, a thorough security evaluation based on Common Criteria for Information Technology Security Evaluation has been conducted. Whilst many publications analyse the security of mobile platforms and mobile application development, little work has been done in analysing the security of cross-platform development frameworks. Therefore, the conducted security evaluation especially focuses on security aspects resulting from the use of Apache Cordova.

Within the course of the security evaluations, we came across several issues related to the use of Apache Cordova. Thus, to use Apache Cordova for security-critical applications several recommendations have to be followed. As applications are implemented using web technologies they are prone to code injection attacks. Developers have to take special care for input validation. Apart from validating user input, QR codes, push notifications, contact data and even file names might include malicious code. Code injection is particularly serious as depending on the application an attacker might gain access to sensitive data of the user, including her physical location or private messages. Second, by default, application code embedded within an iFrame can fully access the device APIs. Thereby, applications should not include any iFrames where the developer is not in control of the served content. This opposes the business model of many applications that leverage advertising services. Ultimately, applications that require the use of many features from the underlying mobile platform need to rely on plugins developed by third-parties. Plugins present a serious security risk, as up-to-date there is no validation process for plugins. Everyone can publish plugins. As the plugin developer decides how to map the requested feature to the underlying device APIs, some plugins are of poor quality and simply do not provide the security they promise by using APIs in a wrong way. Therefore, we recommend inspecting the source code of each plugin before adding the plugin to an application.

The conducted evaluations show that the developed framework offers good protection against attackers with standard malware on the mobile device and attackers that gain physical access to the device. By implementing strict input validation and only running code bundled with the application at install-time we counterfeit web attackers. However, some security risks remain. We cannot reliably protect against attackers with root access to the device. This particularly poses a threat in a single device scenario where the user triggers authentication within the mobile browser and leverages the authentication framework on the same device.

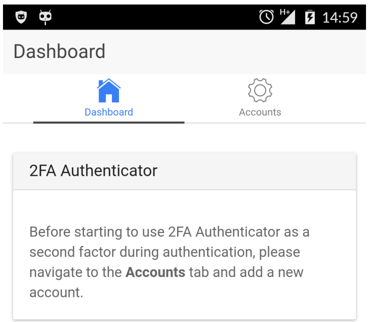
The authentication framework has been implemented as a prototype. The integrated authentication methods are limited to authentication methods that compute OTPs. To provide a broader range of authentication mechanisms, new methods implementing other challenge-response schemes might be integrated. It would be particularly interesting to include authentication leveraging NFC-enabled FIDO tokens.

Appendix A

Screenshots

A.1 Registration

In the following, a sample process of registering a new authentication method is shown. Therefore, the user has to enable the desired authentication method at the server-side authentication service provided by EGIZ and perform the pairing with the authentication app on the mobile device. The screenshots illustrate the process for the authentication method Triple Key AES OTP. The pairing is required for transferring the permanent cryptographic key *Key_A* to the device. In addition, the displayed QR code includes the URL where the authentication app has to transmit the *Push ID*.



2FA Authenticator

Before starting to use 2FA Authenticator as a second factor during authentication, please navigate to the **Accounts** tab and add a new account.

Figure A.1: The starting screen of the mobile authentication app.

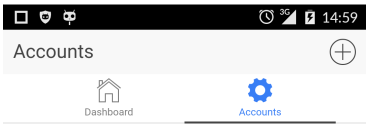
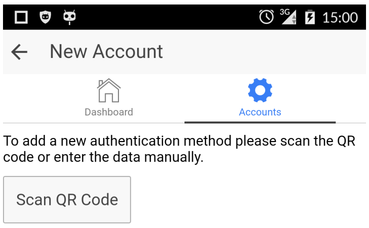
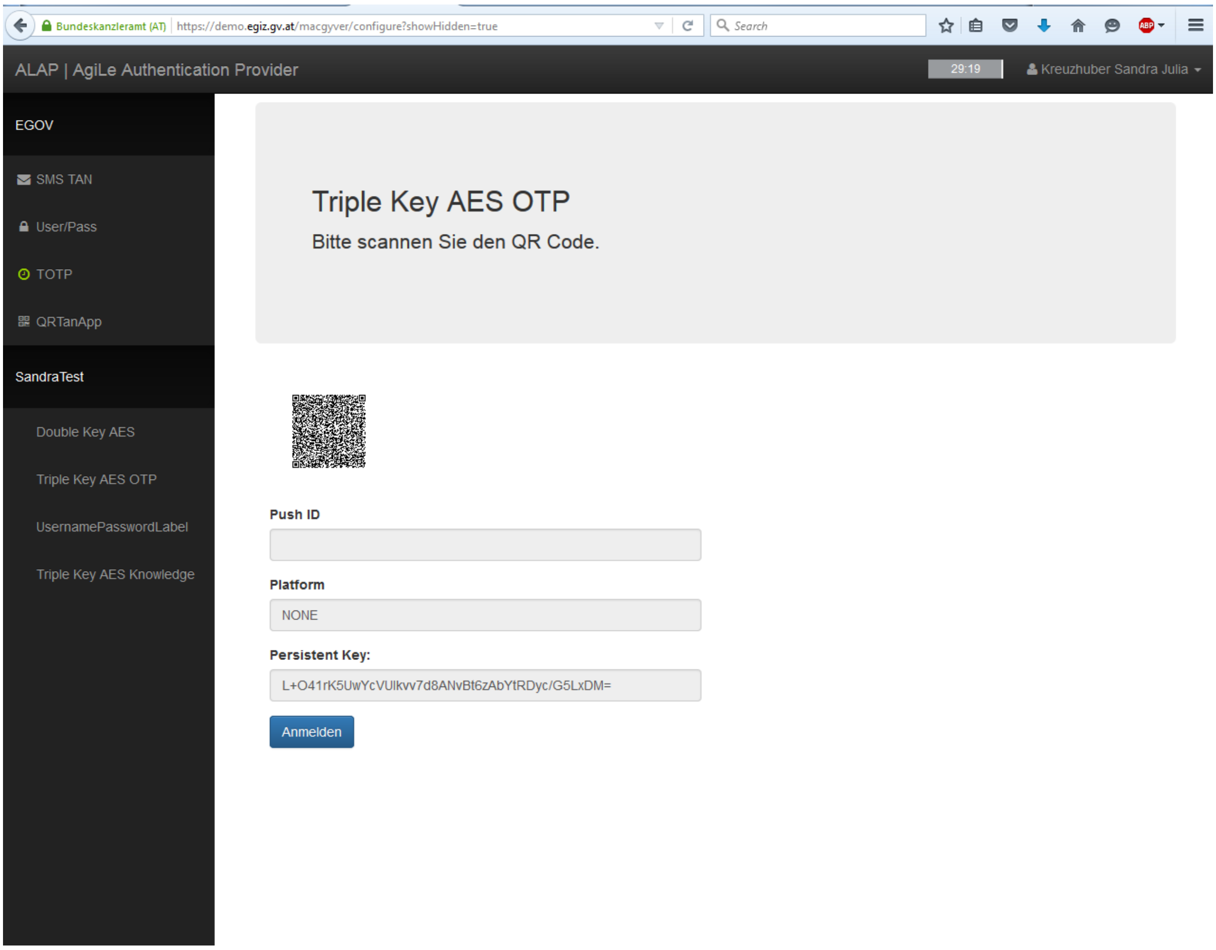


Figure A.2: The application supports multiple so-called accounts. An account represents a specific multi-factor authentication method for a particular domain. The same authentication method can be added multiple times, but only for different domains.



To add a new authentication method please scan the QR code or enter the data manually.

Figure A.3: The user can add a new account by scanning a QR code.



Triple Key AES OTP

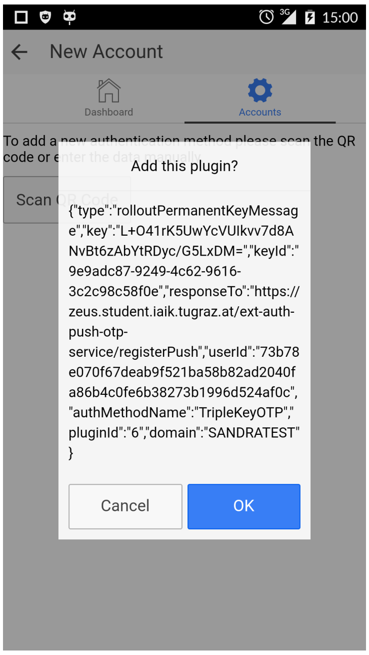
Bitte scannen Sie den QR Code.

Push ID:

Platform: NONE

Persistent Key: L+O41rK5UwYcVUlkvv7d8ANvBt6zAbYtRDyc/G5LxDM=

Figure A.4: By choosing the desired authentication method, the user is presented with the initial authentication data required for the particular authentication method.

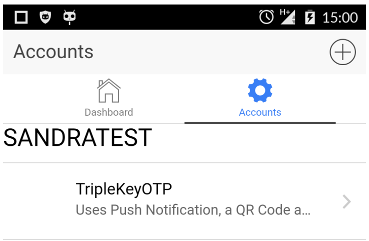


To add a new authentication method please scan the QR code or enter the data manually.

Add this plugin?

{"type":"rolloutPermanentKeyMessage","key":"L+O41rK5UwYcVUlkvv7d8ANvBt6zAbYtRDyc/G5LxDM=","keyId":"9e9adc87-9249-4c62-9616-3c2c98c58f0e","responseTo":"https://zeus.student.iaik.tugraz.at/ext-auth-push-otp-service/registerPush","userId":"73b78e070f67deab9f521ba58b82ad2040fa86b4c0fe6b38273b1996d524af0c","authMethodName":"TripleKeyOTP","pluginId":"6","domain":"SANDRATEST"}

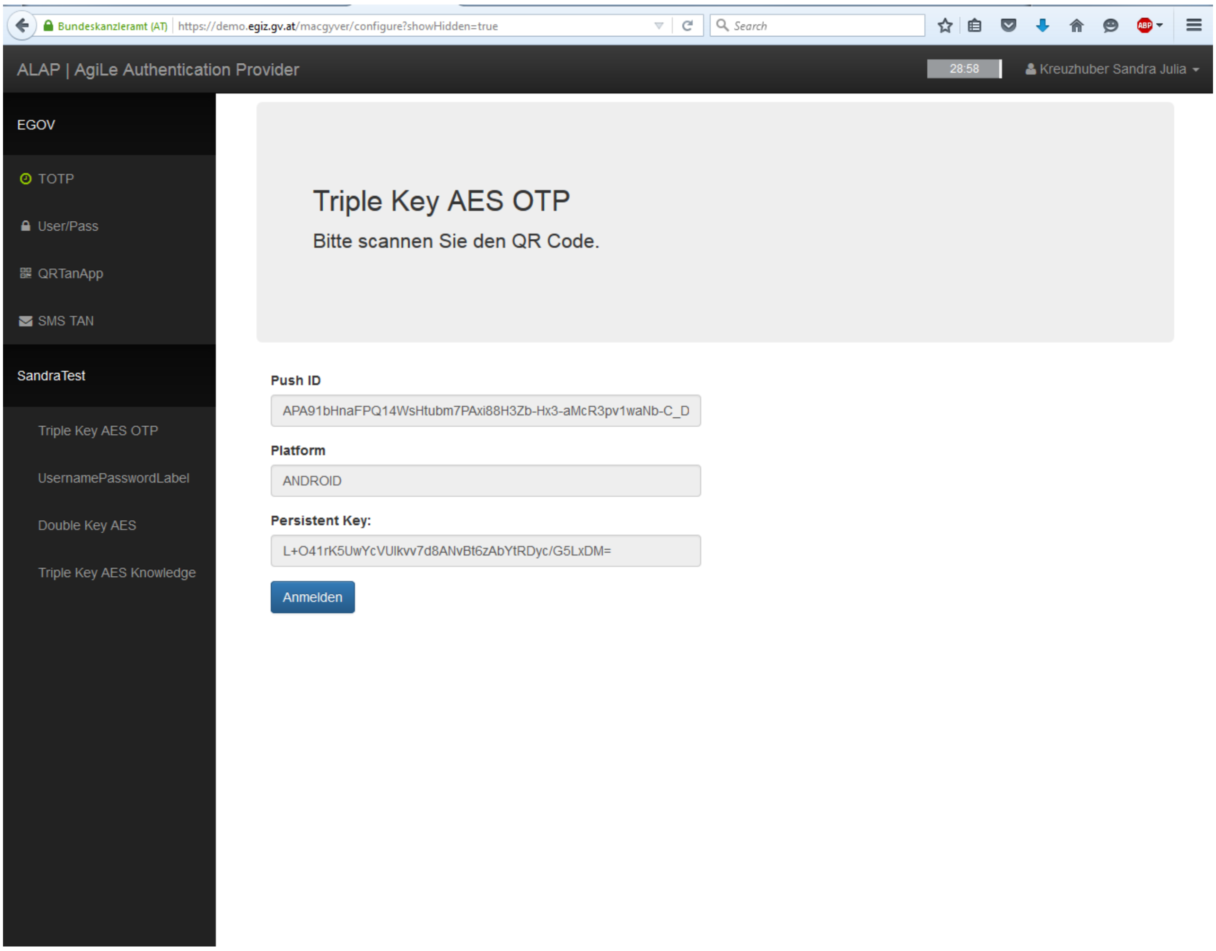
Figure A.5: The user scans the QR code and is presented with a confirmation dialog. As the authentication app still presents a prototype, all data encoded in the QR code is displayed to the user.



SANDRATEST

TripleKeyOTP
Uses Push Notification, a QR Code a...

Figure A.6: After confirming that the authentication method should be added, the authentication app transmits its *Push ID* to the authentication service and a new account is added to the account overview page.



Triple Key AES OTP

Bitte scannen Sie den QR Code.

Push ID

APA91bHnaFPQ14WsHtubm7PAxi88H3Zb-Hx3-aMcR3pv1waNb-C_D

Platform

ANDROID

Persistent Key:

L+O41rK5UwYcVUlkvv7d8ANvBt6zAbYtRDyc/G5LxDM=

Figure A.7: After refreshing the authentication service page, the transmitted *Push ID* and corresponding mobile platform are displayed.

A.2 Authentication

The following screenshots illustrate an authentication process to allow the creation of a digital signature. For the digital signature, a cryptographic key on a central server is used. The central server uses the authentication server to authenticate its users. For this authentication process, a simple authentication plugin realising the username and password scheme and the Triple Key AES OTP method are used.

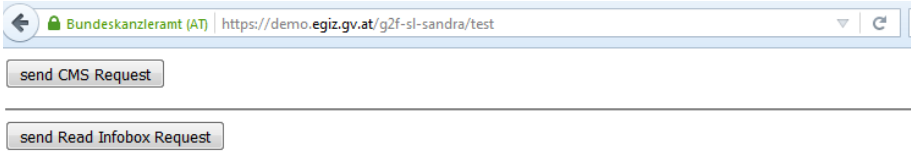
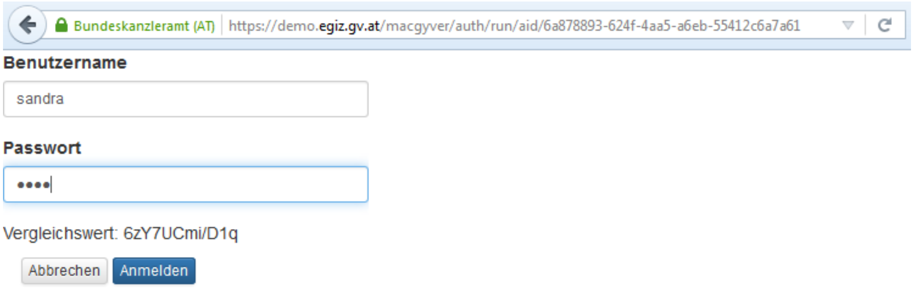

Bundeskanzleramt (AT)
https://demo.egiz.gv.at/g2f-sl-sandra/test
send CMS Request
send Read Infobox Request

Figure A.8: The user starts authentication in order to authorise her private key to be used for signature creation.


Bundeskanzleramt (AT)
https://demo.egiz.gv.at/macgyver/auth/run/aid/6a878893-624f-4aa5-a6eb-55412c6a7a61
Abbrechen
Anmelden

Benutzername

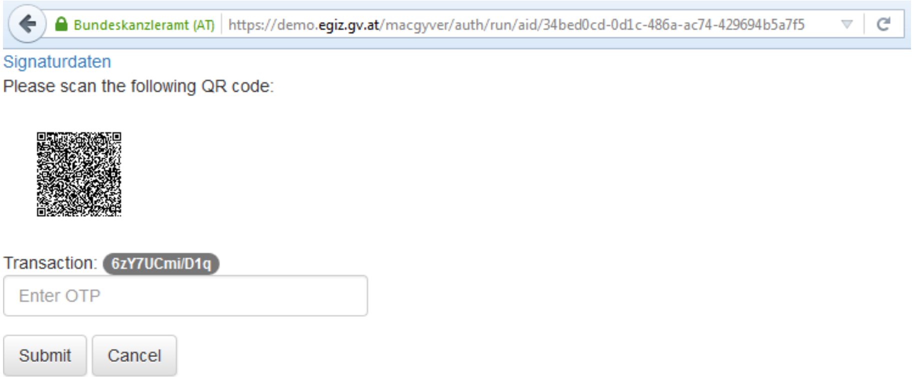
sandra

Passwort

••••

Vergleichswert: 6zY7UCmi/D1q

Figure A.9: The username and password plugin is started. The user enters her credentials.



Bundeskanzleramt (AT) https://demo.egiz.gv.at/macgyver/auth/run/aid/34bed0cd-0d1c-486a-ac74-429694b5a7f5

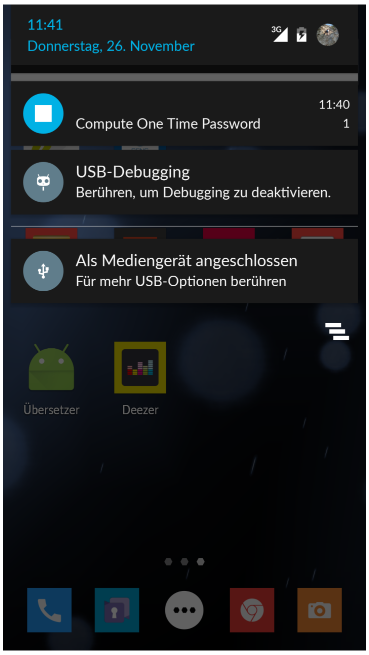
Signaturdaten

Please scan the following QR code:

Transaction: 6zY7UCmi/D1q

Enter OTP

Figure A.10: The entered credentials are correct. Hence, the user is forwarded to the second authentication method, the Triple Key AES OTP method.



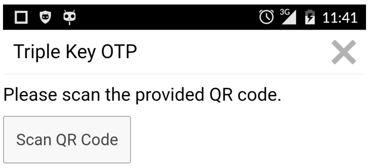
11:41
Donnerstag, 26. November

Compute One Time Password 11:40 1

USB-Debugging
Berühren, um Debugging zu deaktivieren.

Als Mediengerät angeschlossen
Für mehr USB-Optionen berühren

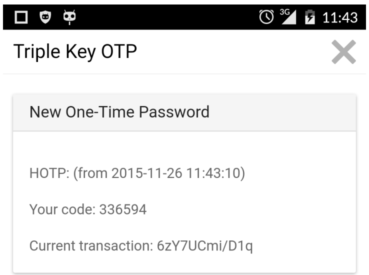
Figure A.11: A new push notification is received on the mobile device. By clicking on the push notification, the authentication app is started.



Please scan the provided QR code.

Figure A.12: The authentication app launches and immediately opens the window performing the authentication method Triple Key AES OTP. The user is requested to scan the QR code.

.



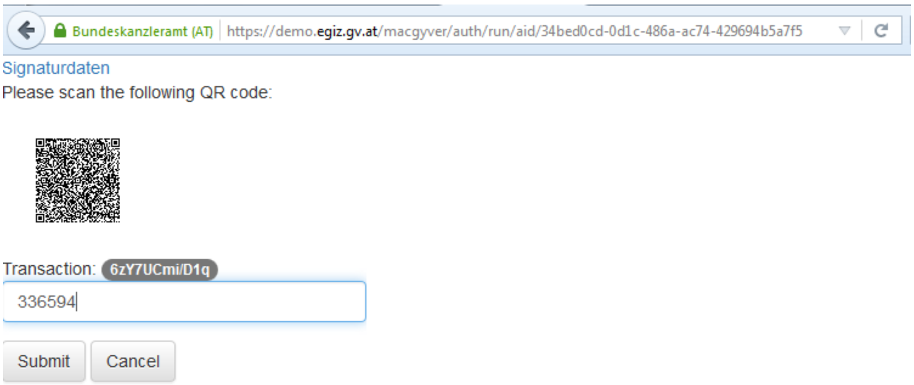
New One-Time Password

HOTP: (from 2015-11-26 11:43:10)

Your code: 336594

Current transaction: 6zY7UCmi/D1q

Figure A.13: The authentication app decrypts the keys encoded in the push notification and the QR code and subsequently computes an OTP.



Bundeskanzleramt (AT) https://demo.egiz.gv.at/macgyver/auth/run/aid/34bed0cd-0d1c-486a-ac74-429694b5a7f5

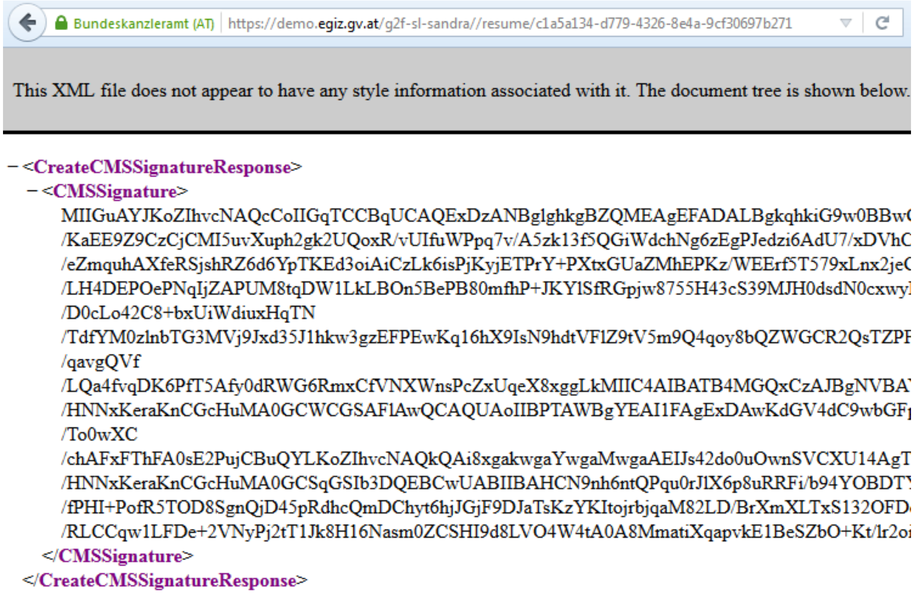
Signaturdaten

Please scan the following QR code:

Transaction: 6zY7UCmi/D1q

336594

Figure A.14: The users enters the OTP at the authentication service.



Bundeskanzleramt (AT) https://demo.egiz.gv.at/g2f-sl-sandra//resume/c1a5a134-d779-4326-8e4a-9cf30697b271

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
– <CreateCMSSignatureResponse>
  – <CMSSignature>
      MIIGuAYJKoZIhvcNAQcCoIIGqTCCBqUCAQExDzANBglghkgBZQMEAgEFADALBgkqhkiG9w0BBw
      /KaEE9Z9CzCjCMI5uvXuph2gk2UQoxR/vUIfuWPpq7v/A5zk13f5QGiWdchNg6zEgPJedzi6AdU7/xDVh
      /eZmquhAXfeRSjshRZ6d6YpTKEd3oiAiCzLk6isPjKyjETPrY+PXtxGUaZMhEPKz/WEErf5T579xLnx2je
      /LH4DEPOePNqIjZAPUM8tqDW1LkLBOn5BePB80mfhP+JKY1SfRGpjw8755H43cS39MJH0dsdN0cxwy
      /D0cLo42C8+bxUiWdiuxHqTN
      /TdfYM0zlnbTG3MVj9Jxd35J1hkw3gzEFPEwKq16hX9IsN9hdtVFlZ9tV5m9Q4qoy8bQZWGCR2QsTZP
      /qavgQVf
      /LQa4fvqDK6PfT5Afy0dRWG6RmxCfVNXWnsPcZxUqeX8xggLkMIIC4AIBATB4MGQxCzAJBgNVBA
      /HNNxKeraKnCGcHuMA0GCWCGSAFlAwQCAQUAoIIBPTAWBgYEAI1FAgExDAwKdGV4dC9wbGF
      /To0wXC
      /chAFxFThFA0sE2PujCBuQYLKoZIhvcNAQkQAi8xgakwgaYwgaMwgaAEIJs42do0uOwnSVCXU14AgT
      /HNNxKeraKnCGcHuMA0GCSqGSIb3DQEBCwUABIIBAHCN9nh6ntQPqu0rJlX6p8uRRFi/b94YOBDT
      /fPHI+PofR5TOD8SgnQjD45pRdhcQmDChyt6hjJGjF9DJaTsKzYKItojrbjqaM82LD/BrXmXLTxS132OFD
      /RLCCqw1LFDe+2VNyPj2tT1Jk8H16Nasm0ZCSHI9d8LVO4W4tA0A8MmatiXqapvkE1BeSZbO+Kt/lr2o
    </CMSSignature>
  </CreateCMSSignatureResponse>
```

Figure A.15: The entered OTP is correct. Hence, the authentication service authorises signature creation.

Appendix B

Security Evaluation Rationale

B.1 Mapping Threats to Assets

	(A1) Cryptographic Key Material	(A2) Password	(A3) Push ID	(A4) Physical Device	(A5) Response
(T1) Device Theft				X	
(T2) Copy or Extract Storage	X		X		
(T3) Accessing Data On-Device	X		X		
(T4) Accessing Data During Transport	X		X		X
(T5) Phishing		X	X		X
(T6) Throttled Guessing		X			X
(T7) Unthrottled Guessing	X	X			X
(T8) Leak from Verifier					X
(T9) Internal Observation		X			X
(T10) Physical Observation		X			X
(T11) Manipulation of Authentication Data	X				X
(T12) Manipulation of Push ID			X		
(T13) Intercepting Remote Notifications On-Device					X
(T14) Intercepting Remote Notifications During Transport					X
(T15) Code Injection	X		X		X
(T16) Missing Function Level Access Control			X		X
(T17) Forging the Mobile Application	X	X	X		X
(T18) Automatically Process Authentication Without The User’s Consent					X

Table B.1: Assets targeted by security threats.

B.2 Mapping Threat Agents to Threats

	(TA1) Standard Attacker	(TA2) Advanced Attacker	(TA3) Nearby Attacker	(TA4) Network Attacker	(TA5) Web Attacker
(T1) Device Theft			X		
(T2) Copy or Extract Storage			X		
(T3) Accessing Data On-Device		X			X
(T4) Accessing Data During Transport				X	
(T5) Phishing	X				X
(T6) Throttled Guessing	X				
(T7) Unthrottled Guessing	X				
(T8) Leak from Verifier	X			X	
(T9) Internal Observation	X	X			X
(T10) Physical Observation			X		
(T11) Manipulation of Authentication Data		X			X
(T12) Manipulation of Push ID		X		X	X
(T13) Intercepting Remote Notifications On-Device		X			
(T14) Intercepting Remote Notifications During Transport				X	
(T15) Code Injection	X				X
(T16) Missing Function Level Access Control	X				
(T17) Forging the Mobile Application	X				
(T18) Automatically Process Authentication Without The User's Consent	X	X			X

Table B.2: Threat agents that are able to pose a specific security threat.

B.3 Mapping Objectives to Threats

	(T1) Device Theft	(T2) Copy or Extract Storage	(T3) Accessing Data On-Device	(T4) Accessing Data During Transport	(T5) Phishing	(T6) Throttled Guessing	(T7) Unthrottled Guessing	(T8) Leak from Verifier	(T9) Internal Observation	(T10) Physical Observation	(T11) Manipulation of Authentication Data	(T12) Manipulation of Push ID	(T13) Intercepting Remote Notifications On-Device	(T14) Intercepting Remote Notifications During Transport	(T15) Code Injection	(T16) Missing Function Level Access Control	(T17) Forging the Mobile Application	(T18) Automatically Process Authentication Without The User's Consent
(O20) Assure Factor Possession is Required for Authentication									X	X								
(O19) Assure Legitimate User is Operating the Device	X																	X
(O18) Transaction-Specific Response					X	X												
(O17) Explicit User Consent																		X
(O16) Adequate Verification Info						X		X										
(O15) High Cryptographic Key Entropy						X	X											
(O14) High Response Entropy						X	X		X									
(O13) High Password Entropy						X	X											
(O12) Protection of Sensitive Data in Remote Notifications													X	X				
(O11) Protection of Server-Side APIs																X		
(O10) Employ Certificate-Pinning				X								X						
(O9) Employ State-of-the-Art Transport Security				X								X						
(O8) Code Obfuscation																	X	
(O7) Detect Forged Mobile Applications																	X	
(O6) Do Not Run Code from Untrusted Sources															X			
(O5) Validate and Escape User Input															X			
(O4) Detect Rooted or Jailbreaked Devices			X						X		X	X	X					
(O3) Detect Manipulation of Sensitive Data											X	X						
(O2) Avoid Side-Channel Data Leakage			X															
(O1) Protected Storage of Sensitive Data		X	X								X	X						

Table B.3: Counterfeiting a security threat by fulfilling a security objective or a combination of multiple security objectives.

B.4 Evaluating Security Measures

	Triple Key AES OTP	Triple Key AES OTP with Knowledge Proof	Double Key AES OTP with Knowledge Proof
(O1) Protected Storage on Sensitive Data	√	√	√
(O2) Avoid Side-Channel Data Leakage	√	√	√
(O3) Detect Manipulation of Sensitive Data			
(O4) Detect Rooted and Jailbreaked Devices			
(O5) Validate and Escape User Input	√	√	√
(O6) Do Not Run Code from Untrusted Sources	√	√	√
(O7) Detect Forged Mobile Applications			
(O8) Code Obfuscation			
(O9) Employ State-of-the-Art Transport Security	√	√	√
(O10) Employ Certificate-Pinning	√	√	√
(O11) Protection of Server-Side APIs	∼	∼	∼
(O12) Protection of Sensitive Data in Remote Notifications	√	√	√
(O13) High Password Entropy	N/A	∼	∼
(O14) High Response Entropy	√	√	√
(O15) High Cryptograhic Key Entropy	√	√	√
(O16) Adequate Verification Info	√	√	√
(O17) Explicit User Consent	√	√	√
(O18) Transaction-Specific Response	√	√	√
(O19) Assure Legitimate User is Operating the Device		√	√
(O20) Assure Factor Possession is Required for Authentication	√	√	∼

Table B.4: Evaluation of deployed countermeasures, which fulfil the derived security objectives.

Bibliography

[1] Adam Bradley. *Where does the Ionic Framework fit in?* 2013. http://blog.ionic.io/where-does-the-ionic-framework-fit-in/.

[2] The FIDO Alliance. *FIDO U2F Raw Message Formats*. Technical report. The FIDO Alliance, 2014.

[3] The FIDO Alliance. *FIDO UAF Architectural Overview*. Technical report. The FIDO Alliance, 2014.

[4] The FIDO Alliance. *Universal 2nd Factor (U2F) Overview*. Technical report. The FIDO Alliance, 2015.

[5] Andrey Polkovnichenko and Alon Boxiner. *BrainTest – A New Level of Sophistication in Mobile Malware*. 2015. http://blog.checkpoint.com/2015/09/21/braintest-a-new-level-of-sophistication-in-mobile-malware/.

[6] Android Developer Documentation. *Signing Your Applications*. 2015. http://developer.android.com/tools/publishing/app-signing.html.

[7] AngularJS. *API Reference ngCSP*. 2015. https://docs.angularjs.org/api/ng/directive/ngCsp.

[8] Apache Cordova. *Security Guide*. 2015. https://cordova.apache.org/docs/en/5.4.0/guide/appdev/security/index.html.

[9] Appcelerator. *Supporting Multiple Platforms in a Single Codebase*. 2015. http://docs.appcelerator.com/platform/latest/%5C#!/guide/Supporting_Multiple_Platforms_in_a_Single_Codebase.

[10] Appcelerator. *Titanium Compatibility Matrix*. 2015. http://docs.appcelerator.com/titanium/3.0/%5C#!/guide/Titanium_Compatibility_Matrix.

[11] Apple. *App Store: iOS Version Distribution*. 2015. https://developer.apple.com/support/app-store/.

[12] Apple. *Apple Push Notification Service*. 2015. https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/.

[13] Apple. *iOS Security*. 2015. https://www.apple.com/business/docs/iOS_Security_Guide.pdf.

[14] Phonegap Blog. *PhoneGap, Cordova, and what's in a name?* 2012. http://phonegap.com/2012/03/19/phonegap-cordova-and-what%C3%A2%C2%80%C2%99s-in-a-name/.

[15] The Official Ionic Blog. *The Last Word on Cordova and PhoneGap*. 2014. http://ionicframework.com/blog/what-is-cordova-phonegap/.

[16] Joseph Bonneau et al. “The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes”. In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. SP ’12. Washington, DC, USA: IEEE Computer Society, 2012, pages 553–567. ISBN 978-0-7695-4681-0. doi:10.1109/SP.2012.44. `http://dx.doi.org/10.1109/SP.2012.44`.

[17] Roger Clarke. “Human Identification in Information Systems”. In: *Information Technology & People* 7.4 (1994), pages 6–37. doi:10.1108/09593849410076799. eprint: `http://dx.doi.org/10.1108/09593849410076799`. `http://dx.doi.org/10.1108/09593849410076799`.

[18] Apache Cordova. *Apache Cordova Documentation*. 2015. `http://cordova.apache.org/docs/en/5.1.1/index.html`.

[19] Common Criteria. *Common Criteria for Information Technology Security Evaluation - Part 1: Introduction and general model*. `https://www.commoncriteriaportal.org/files/ccfiles/CCPART1V3.1R4.pdf`. 2012.

[20] Nikolay Elenkov. *Unlocking Android devices using an OTP via NFC*. 2014. `http://nelenkov.blogspot.co.at/2014/03/unlocking-android-using-otp.html`.

[21] Evild3ad. *Analysis of Android.Zitmo-Urlzone*. 2013. `http://www.evild3ad.com/3008/analysis-of-android-zitmo-urlzone/`.

[22] Facebook. *Facebook Q1 2015 Results*. 2015. `http://files.shareholder.com/downloads/AMDA-NJ5DZ/75562925x0x822961/fd718a09-c312-4605-9a17-1d6ef07bdd5a/FB_Q115EarningsSlides.pdf`.

[23] J. Franks et al. *An Extension to HTTP : Digest Access Authentication*. Technical report. RFC 2069, 1997.

[24] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. “Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks”. In: *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. 2014. `http://www.internetsociety.org/doc/breaking-and-fixing-origin-based-access-control-hybrid-webmobile-application-frameworks`.

[25] Google. *Android Documentation on WebView*. 2015. `http://developer.android.com/reference/android/webkit/WebView.html`.

[26] Google. *Android Keystore System*. 2015. `https://developer.android.com/training/articles/keystore.html`.

[27] Google. *AngularJS*. 2015. `https://angularjs.org/`.

[28] Google. *Google Cloud Messaging*. 2015. `https://developers.google.com/cloud-messaging/gcm`.

[29] Google. *Platform Versions*. 2015. `http://developer.android.com/about/dashboards/index.html`.

[30] Google. *Security*. 2015. `https://source.android.com/devices/tech/security/encryption/`.

[31] Vincent Haupert and Tilo Müller. “(Un)Sicherheit von App-basierten TAN-Verfahren im Onlinebanking”. In: 2015. `https://www1.cs.fau.de/filepool/projects/apptan/Unsicherheit-AppTAN.pdf`.

[32] Henning Heitkötter and TimA. Majchrzak. “Cross-Platform Development of Business Apps with MD2”. English. In: *Design Science at the Intersection of Physical and Virtual Design*. Edited by Jan vom Brocke et al. Volume 7939. Lecture Notes in Computer Science. Springer Berlin

Heidelberg, 2013, pages 405–411. ISBN 978-3-642-38826-2. doi:10.1007/978-3-642-38827-9_29. http://dx.doi.org/10.1007/978-3-642-38827-9_29.

[33] Ian Hickson. *Web SQL Database*. W3C Working Group Note. http://dev.w3.org/html5/webdatabase/. 2010.

[34] Ian Hickson. *Web Storage*. W3C Candidate Recommendation. http://www.w3.org/TR/webstorage/. 2015.

[35] Mat Honan. *How Apple and Amazon Security Flaws Led to My Epic Hacking*. 2012. http://www.wired.com/2012/08/apple-amazon-mat-honan-hacking/.

[36] Intel. *Intel Identity Protection Technology*. 2015. http://www.intel.in/content/www/in/en/architecture-and-technology/identity-protection/identity-protection-technology-general.html.

[37] Intel. *Intel Smartphone - A Faster Experience*. 2015. http://www.intel.com/content/www/us/en/smartphones/smartphones.html.

[38] Ionic Framework. *Ionic Documentation*. 2015. http://ionicframework.com/docs/.

[39] ISO/IEC. *Information technology - Security techniques – Evaluation criteria for IT security – Part 1: Introduction and general model*. ISO/IEC 15408-1. 1999.

[40] Xing Jin et al. "Code Injection Attacks on HTML5-based Mobile Apps". In: *Ccs* (2014), pages 66–77. doi:10.1145/2660267.2660275.

[41] Jon Oberheide. *RSA-proofing our Duo Push two-factor authentication*. 2011. https://www.duosecurity.com/blog/rsa-proofing-our-duo-push-two-factor-authentication.

[42] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT)*. Technical report. RFC 7519, 2015.

[43] Jürgen Schmidt. *Forscher demontieren App-TANs der Sparkasse*. 2015. http://www.heise.de/security/meldung/Forscher-demontieren-App-TANs-der-Sparkasse-2853492.html.

[44] David Kaplan and Roee Hay. *Remote Exploitation of the Cordova Framework*. W3C Recommendation. http://www.ibm.com/developerworks/library/se-remote-apache-cordova/index.html. 2014.

[45] Nok Nok Labs. *Leveraging Fingerprint Authentication On Mobile Devices: Apples Touch ID API and More*. 2014. https://www.noknok.com/what-they-say/blog/apple-touch-id-app-for-mobile-fingerprint-authentication.

[46] Nok Nok Labs. *Technical White Paper: Nok Nok Labs Multifactor Authentication*. Technical report. Nok Nok Labs, Inc., 2013.

[47] Posch R. Leitold H. Hollosi A. "Security Architecture of the Austrian Citizen Card Concept". In: *Proceedings of 18th Annual Computer Security Applications Conference (ACSAC'2002), Las Vegas, 9-13 December 2002. pp. 391-400, IEEE Computer Society, ISBN 0-7695-1828-1, ISSN 1063-9527.* 2002, n/a.

[48] Tongxin Li et al. "Mayhem in the Push Clouds: Understanding and Mitigating Security Hazards in Mobile Push-Messaging Services". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: ACM, 2014, pages 978–989. ISBN 978-1-4503-2957-6. doi:10.1145/2660267.2660302. http://doi.acm.org/10.1145/2660267.2660302.

[49] Liam Tung. *Lollipop stops Chromium bugs from endangering Android*. 2014. http://www.zdnet.com/article/lollipop-stops-chromium-bugs-from-endangering-android/.

[50] Linux.com. *Securing SSH with two factor authentication using Google Authenticator*. 2014. https://www.linux.com/community/blogs/133-general-linux/783135-securing-ssh-with-two-factor-authentication-using-google-authenticator.

[51] Tongbo Luo et al. "Attacks on WebView in the Android system". In: *Annual Computer Security Applications Conference (ACSAC)* (2011), page 343. doi:10.1145/2076732.2076781. http://dl.acm.org/citation.cfm?doid=2076732.2076781.

[52] Denis Maslennikov. *ZeuS-in-the-Mobile – Facts and Theories*. 2011. http://securelist.com/analysis/36424/zeus-in-the-mobile-facts-and-theories/.

[53] Nikunj Mehta et al. *Indexed Database API*. W3C Recommendation. http://www.w3.org/TR/IndexedDB/. 2015.

[54] Microsoft. *Microsoft Support Thread regarding YubiKey on Windows Phone 8 devices*. 2014. http://answers.microsoft.com/en-us/winphone/forum/wp8-wpupdate/buggy-nfc-implementation/939e7549-0170-45be-a6d5-4b8fcede7614.

[55] Mozilla Developer Network. *CSP Policy Directives*. 2015. https://developer.mozilla.org/en-US/docs/Web/Security/CSP/CSP_policy_directives.

[56] D. et al. M'Raihi. *HOTP: An HMAC-Based One-Time Password Algorithm*. Technical report. RFC 4226, 2005.

[57] D. et al. M'Raihi. *TOTP: Time-Based One-Time Password Algorithm*. Technical report. RFC 6238, 2011.

[58] Nikolay Elenkov. *Credential storage enhancements in Android 4.3*. 2013. http://nelenkov.blogspot.co.at/2013/08/credential-storage-enhancements-android-43.html.

[59] The Open Web Application Security Project. *The Ten Most Critical Web Application Security Risks*. OWASP Top 10 - 2013. https://www.owasp.org/index.php/Top10#OWASP_Top_10_for_2013. 2013.

[60] Rafay Baloch. *A Tale Of Another SOP Bypass In Android Browser < 4.4*. 2014. http://www.rafayhackingarticles.net/2014/10/a-tale-of-another-sop-bypass-in-android.html.

[61] Rafay Baloch. *Android Browser All Versions - Address Bar Spoofing Vulnerability - CVE-2015-3830*. 2015. http://www.rafayhackingarticles.net/2015/05/android-browser-address-bar-spoofing-vulnerability.html.

[62] Rafay Baloch. *Android Browser Same Origin Policy Bypass < 4.4 - CVE-2014-6041*. 2014. http://www.rafayhackingarticles.net/2014/08/android-browser-same-origin-policy.html.

[63] Arun Ranganathan and Jonas Sicking. *File API*. W3C Working Draft. http://www.w3.org/TR/FileAPI/. 2015.

[64] P. Saint-Andre. *Extensible Messaging and Presence Protocol (XMPP): Core*. Technical report. RFC 6120, 2011.

[65] Ryan Sleevi and Mark Watson. *Web Cryptography API*. W3C Candidate Recommendation. http://www.w3.org/TR/WebCryptoAPI/. 2014.

[66] Sure Pass ID. *FIDO Authentication Server Datasheet*. 2014. http://www.surepassid.com/wp/wp-content/uploads/2014/03/SurePassID-FIDO-Server-Datasheet-03-04-2014.pdf.

[67] Symantec. *Two Factor Authentication with Intel IPT and Symantec VIP*. 2011. http://www.symantec.com/connect/videos/two-factor-authentication-intel-ipt-and-symantec-vip.

[68] Tarsnap. *The scrypt key derivation function*. 2015. http://www.tarsnap.com/scrypt.html.

[69] Peter Teufl, Thomas Zefferer, and Christof Stromberger. "Mobile Device Encryption Systems". In: *28th IFIP TC-11 SEC 2013 International Information Security and Privacy Conference*. 2013, pages 203–216.

[70] Peter Teufl et al. "Android Encryption Systems". In: *International Conference on Privacy & Security in Mobile Systems*. in press. 2014.

[71] Peter Teufl et al. "iOS Encryption Systems - Deploying iOS Devices in Security-Critical Environments". In: *SECRYPT*. 2013, pages 170–182.

[72] The Open Web Application Security Project. *OWASP Mobile Security Project - Top Ten Mobile Risks*. 2015. https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks.

[73] The Open Web Application Security Project (OWASP). *Certificate and Public Key Pinning*. 2015. https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning.

[74] Lisa Vaas. *Bank tests heartbeat-encoded wristbands for online authentication*. 2015. https://nakedsecurity.sophos.com/2015/03/16/bank-tests-heartbeat-encoded-wristbands-for-online-authentication/.

[75] Roland M. Van Rijswijk and Joost Van Dijk. "Tiqr: A Novel Take on Two-factor Authentication". In: *Proceedings of the 25th International Conference on Large Installation System Administration*. LISA'11. Boston, MA: USENIX Association, 2011, pages 7–7. http://dl.acm.org/citation.cfm?id=2208488.2208495.

[76] VisionMobile. *Cross-Platform Tools 2015*. 2015. http://www.visionmobile.com/product/cross-platform-tools-2015/.

[77] I. Warren et al. "Push Notification Mechanisms for Pervasive Smartphone Applications". In: *Pervasive Computing, IEEE* 13.2 (Apr. 2014), pages 61–71. ISSN 1536-1268. doi:10.1109/MPRV.2014.34.

[78] Xamarin. *Understanding the Xamarin Mobile Platform*. 2015. http://developer.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/part_1_-_understanding_the_xamarin_mobile_platform/.

[79] Spyros Xanthopoulos and Stelios Xinogalos. "A Comparative Analysis of Cross-platform Development Approaches for Mobile Applications". In: *Proceedings of the 6th Balkan Conference in Informatics* (2013), pages 213–220. doi:10.1145/2490257.2490292. http://doi.acm.org/10.1145/2490257.2490292.

[80] Yubico. *YubiKey Neo and YubiKey Neo-N*. 2015. https://www.yubico.com/products/yubikey-hardware/yubikey-neo/.

[81] Thomas Zefferer and Bernd Zwattendorfer. "An Implementation-independent Evaluation Model for Server-based Signature Solutions". In: *WEBIST 2014 - Proceedings of the 10th International Conference on Web Information Systems and Technologies, Volume 1, Barcelona, Spain, 3-5 April, 2014*. 2014, pages 302–309. doi:10.5220/0004839603020309. http://dx.doi.org/10.5220/0004839603020309.

[82] Yulong Zhang and Tao Wei. *To Swipe or Not to Swipe: A Challenge for Your Fingers*. 2015. https://www.rsaconference.com/writable/presentations/file_upload/hta-f01-to-swipe-or-not-to-swipe-a-challenge-for-your-fingers_final.pdf.