TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Raul Metsma 211774IAPM

MAN IN THE MIDDLE ATTACK PREVENTION FOR SMART-ID USING BROWSER EXTENSIONS

Master's Thesis

Supervisor: Ahto Buldas
PhD

Co-supervisor: Raul Kaidro

Tallinn 2023

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Raul Metsma 211774IAPM

SMART-ID VAHENDUSRÜNNETE VÄLTIMINE BRAUSERILAIENDUSTE ABIL

Magistritöö

Juhendaja: Ahto Buldas
PhD

Kaasjuhendaja: Raul Kaidro

Tallinn 2023

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Raul Metsma

08.05.2023

Abstract

The thesis in hand analyses the weaknesses of the current Smart-ID and proposes a new protocol to protect users against man-in-the-middle attacks. This thesis also gives proof of concept solution web browser extension and provides a conceptual description of Web Extension solution operating principles.

The thesis is written in English and is 56 pages long, including 6 chapters, 27 figures and 3 tables.

Annotatsioon

Smart-ID vahendusrünnete vältimine brauserilaienduste abil

Käesolevas lõputöös analüüsitakse praeguse Smart-ID nõrkusi ja pakutakse välja uus protokoll, mis kaitseks kasutajaid vahendusrünnete eest. See lõputöö annab ka tõestuse kontseptsioonilahenduse veebibrauseri laienduse kohta ja annab kontseptuaalse kirjelduse veebilaienduse tööpõhimõtetest.

Lõputöö on kirjutatud inglise keeles ja on 56 lehekülge pikk, sisaldab 6 peatükki, 27 joonist ja 3 tabelit.

List of Abbreviations and Terms

API	Application Programming Interface
CCA	Client Certificate Authentication
CORS	Cross-Origin Resource Sharing
CSS	Cascading Style Sheets
DNS	Domain Name System
eID	Electronic Identity
HMAC	Keyed-hash Message Authentication Code
HSM	Hardware Security Module
HTML	HyperText Markup Language
HTTP	The Hypertext Transfer Protocol
HTTPS	The Hypertext Transfer Protocol Secure
ID	Identity
IP	Internet Protocol
JWK	JSON Web Key
JWT	JSON Web Token
KID	Key Identifier
MAC	Message Authentication Code
MITM	Man In the Middle
NFC	Near-Field Communication
PIN	Personal Identificaton Number associated with an eID
QES	Qualified Electronic Signature
QR	Quick Response code
QSCD	Qualified Signature Creation Device
RIA	Estonian Information System Authority (Riigi Infosüsteemi Amet)
RP	Relying Party
RSA	(Rivest–Shamir–Adleman) is a public-key cryptosystem that is widely used for secure data transmission.
SSL	Secure Sockets Layer
SK	SK ID Solutions AS (formerly AS Sertifitseerimiskeskus)
TCP	Transmission Control Protocol
TLS	Transport Layer Security
URL	Uniform Resource Locator

List of Notations

$x \leftarrow v$	A variable x gets a value v
M	Requested message or transaction, the Indent
m	Calculated message hash value of M
cc	Control code derived from message hash value m
ds	Display string that is assigned to message M to give better context (e.g. "Login", "Accept transfer")
K	Pre-shared secret
qr	QR Code
qr^{Mac}	QR code that depends on M, URL and K
url^B	URL of a malicious site
url^G	URL of the Relying party site
s_U^{MD}	User's share of the signature created by the Mobile Device
s_U^{SP}	Server's share of the signature created by the Service Provider
h	Cryptographic hash function
f	Function to derive control code from the hash value of a message
$Sig_U(m)$	Signature value of message digest value m
$Sig_U^{MD}(m)$	Function to calculate user's share of the signature of message digest value m
$Sig_U^{SP}(m)$	Function to calculate server's share of the signature of message digest value m
$Comp(s_U^{MD}, s_U^{SP})$	Function to compose signature value from user's share s_U^{MD} and server's share s_U^{SP}
$survey(M)$	Information extracted from a message M
$Json(X)$	JSON representation of data X
$Mac_K(X)$	Message authentication code of data X computed with K
$Qr(X)$	QR encoding of data X

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Background

Smart-ID is an electronic identity solution that takes advantage of the smart capabilities of mobile devices while providing users with a high level of security and providing online service providers with a reliable and secure means of end-user electronic authentication and digital signature creation.

In the Baltic countries, Smart-ID has over 3 million users, with 680 859 of them in Estonia [06.05.2023] [1]. SK ID Solutions AS (SK) develops and operates Smart-ID in partnership with Cybernetica AS. It allows for user authentication and the creation of digital signatures for electronic services provided by both public and private entities. The commercial banks generate the majority of transactions. Smart-ID is classified as a QSCD (qualified signature creation device), meaning the signatures created are legally equivalent to handwritten signatures.

1.2 Observations

On several occasions, the Smart-ID users' behaviour and Smart-ID's technical security measures have been challenged, and there are many attempts to over-take users' accounts. Some of these attempts have already been successful [1, 2]. In the worst case scenarios, the users have lost control over their bank accounts and therefore suffered serious financial damages due to account hijacking. All of these have been possible due to the user's negligence towards the process details provided by the Smart-ID solution and the URL used to access the service.

The user always remains the weakest link in the chain of the technical solution otherwise considered to be secure [3]. The Estonian Association of Banks [4] and also the Estonian State Information Authority have issued several calls to the users to be vigilant towards possible attacks. Even though, SK ID Solutions has implemented various security measures to ensure the maximum security of Smart-ID [5], dangers are still lurking for unsuspecting users.

This work aims to identify and address potential attack vectors that can hijack Smart-ID

[1] Smart-ID `https://www.smart-id.com/`

authentication process. The focus is on the most pressing questions related to the Smart-ID man-in-the-middle (MITM) attacks: What does the MITM attack vector look like? What makes users become vulnerable to this type of attack? Why are the users not able to detect fraudulent behaviour? What are the existing mechanisms currently being implemented to protect the Smart-ID users? What solutions can be proposed to prevent these types of attacks?

There are several studies on the MITM attacks against Mobile-ID, which is, from the end-user's point of view, similar to the Smart-ID. For example, Peeter Laud and Meelis Roos completed the formal analysis of the Mobile-ID protocol in 2009 [6]. Also, a related study was completed by Cybernetica in 2019 [7]. Both the cited studies conclude that the technical protocol is secure, but the end user can be deceived and is not always able to identify the MITM of the authentication process, which makes the MITM type of attack possible.

1.3 The Problem

The current security measures are not sufficient to prevent a MITM attack. Previous news articles have reported successful attempts of these attacks, highlighting the need for further security improvements. To address this issue, there are proactive campaigns to improve the users' skillset in identifying fraudulent websites [2].

1.4 Research Questions Topics

What are the mechanisms by which the user fails to recognise a fraudulent website? What protection mechanisms have SK ID Solutions and web browser manufacturers already implemented? How can it ensure that the user is transacting through a fraudulent middleman? The purpose is to identify the shortcomings of the given solutions and approaches.

Research methods used in this work follow the usual standards of applied cryptography research. The model of the system is constructed together with the assumed action model of the attacker. The system is then modified, demonstrating that the attacker's presumable actions will not result in successful attacks in the modified model.

[2] `https://www.itvaatlik.ee/kontrolli/`

1.5 Proposed Solution

The author will propose a solution with a browser extension with additional Smart-ID changes. The extension will capture the browser's current visited URL and transfer captured info securely to relying party for additional checks.

1.6 Expected Impact

The solution presented in this work provides an alternative approach to implement additional security features in the form of a web browser extension to protect the end user from MITM type of attacks. The method is yet to be used on any Estonian eID solutions. The browser extension also significantly improves the user experience for the authentication process and provides additional assurance when identifying fraudulent websites and possible attacks.

1.7 Structure of Work

The thesis will first introduce the Smart-ID background, its operating principles, the concept and how the end-user and the relying party can use it. The thesis also describes the motivation based on how successful MITM attacks have already been carried out. Then it will raise research and development goals. The use case will be modelled during the study, and the attack will be carried out using the model.

Proposing new security measures, to prevent MITM attackers from carrying out malicious activity and relying on a party to identify users' presence on the correct site. The solution's novelty lies in the technical approach where the user's device and the service provider's website will have strong peer-to-peer authentication, making it very complicated to plant an intermediary into an authentication session.

By implementing new security measures, it is no longer possible to launch successful MITM attacks, as the user and the relying party shall be authenticated before the user authentication process takes place.

Later the research focuses on the technical implementation of these attacks and attempts to provide an additional security layer to protect the end user from malicious activity.

The model will prove that the attack is impossible to carry out and that the solution is as secure as the public key cryptography used. Additional analysis is required to ensure that

no new attack vectors are made possible by implementing the solution provided.

1.8 Contributions of Thesis

The main contributions of the thesis are:

- MITM attacks analysis
- Description of the solution based on browser extension
- Analysis of the solution
- Implementing prototype and testing the prototype

The findings and recommendations from the current work can be used to improve the security of future releases of the Smart-ID service. This will provide users with a higher level of protection against MITM attacks. Additionally, finance entities, which typically have a higher risk of these attacks, can also benefit from incorporating these improvements into their security protocols. Doing so can ensure that their clients are better protected against fraud and financial loss.

1.9 Outline of the Thesis

Section 2.1 gives a background on Smart-ID. Sections 2.2, 2.3, and 2.4 describe the MITM attack types and countermeasures implemented in the Smart-ID protocol. In Section 2.5, the working principles of browsers are described. In Sections 3.1 and 3.2, a new protocol and related key establishment are proposed. Section 3.3 analyses the new protocol. Chapter 4 describes the created proof of concept solution. In Section 4.3, the user experience with the new solution is outlined. Chapter 5 draws a conclusion about the new protocol and proof of concept solution and raises some future research topics and necessary future developments. Chapter 6 gives a summary of the current thesis. In Appendices, the code of the proof of concept solution is provided.

2. Background

The following chapter describes the current situation and covers the basic overview of Smart-ID and its operating principles. It also includes a basic introduction to the MITM attacks and explains their relevance to the Smart-ID. Later in this chapter, the attack models, the operating principles of an authentication process within browsers, the security measures currently implemented and the solutions applied (although not compelling enough to mitigate these threats) are analysed.

This chapter describes the background and the current solution. In Section 2.1 an overview of Smart-ID operating principles is provided. Section 2.2 describes Smart-ID protocol related MITM attacks. Section 2.3 shows that current measures are not effective protection MITM attacks. Section 2.4 proposes an additional possible security measure. Section 2.5 describes that browser do not have built-in measures to prevent MITM attacks and available tools to extend protocols.

2.1 Working Principles of Smart-ID

Smart-ID's classification as a QSCD requires that the private key cannot be extracted. SK ID Solutions does not store the entire private key in the Smart-ID application to meet this requirement. Instead, the key is split into two parts: one is stored in the application, and the other is stored in a hardware security module (HSM) (operated by SK ID Solutions). This approach is detailed in the paper Server-Supported RSA Signatures for Mobile Devices [8].

Using Smart-ID for authentication, the relying party (RP) requests the user's personal code, displays the calculated control code, and makes an API call to SK ID Solutions. Smart-ID application requests control code verification, and the signature is created using the two private key parts. The result is returned to the relying party, and if successful, the user is authenticated and can access the e-service.

The following parties and components are involved in the Smart-ID protocol:

1. **Browser** is User's web browser in Personal Computer (or Mobile Device) that is used to access Relying Party online services.
2. **Mobile Device** is User's Mobile Device that has Smart-ID application installed.

3. **Relying Party (RP)** is an organisation that provides online services to User, and wants to authenticate User or get a signature of User.
4. **Smart-ID App** is Smart-ID application installed in User's Mobile Device
5. **Smart-ID Server** is Smart-ID service who interacts between Relying Party and Smart-ID application.
6. **User** is Smart-ID user with Mobile Device and wants use Relying Party online services.

The authentication protocol has the following steps [9] (Figure 1):

1. User initiates authentication request in Browser.
2. Browser announces RP about an intention to continue with authentication.
3. RP prepares a "Request".
4. RP sends the "Request" to Smart-ID Server.
5. RP sends to Browser the consent screen.
6. Browser displays the consent dialogue screen that contains the Control Code (a 4-digit number computed as a deterministic function of hash value).
7. Smart-ID Server sends to Smart-ID App a "Request" message.
8. Smart-ID App displays the consent screen that contains the Control Code (Smart-ID App computes it from the "Request" hash value), RP name and Display String (metadata of transaction).
9. User verifies if the displayed control codes (by Browser and by Smart-ID App) coincide, RP name and Display String correspond to the intended action and, as a confirmation, enters PIN.
10. Smart-ID App creates User's signature share and sends it to Smart-ID Server.
11. Smart-ID Server sends to RP "Response structure" message.
12. RP verifies the "Response structure" message.
13. If successful, User is authenticated and can access the e-service, otherwise an error message is displayed on the website.
14. User is granted access to services on successful verification.

2.2 Man in the Middle Attacks

A MITM attack represents a cyberattack in which a malicious player inserts himself into a communication between two parties, impersonates both of them and gains access to the information that the two parties were trying to share. The malicious player intercepts, sends, and receives data meant for someone else – or not meant to be sent at all, without either outside party knowing until it's already too late. One may find the man-in-the-middle attack abbreviated in various ways: MITM, MitM, MiM or MIM.

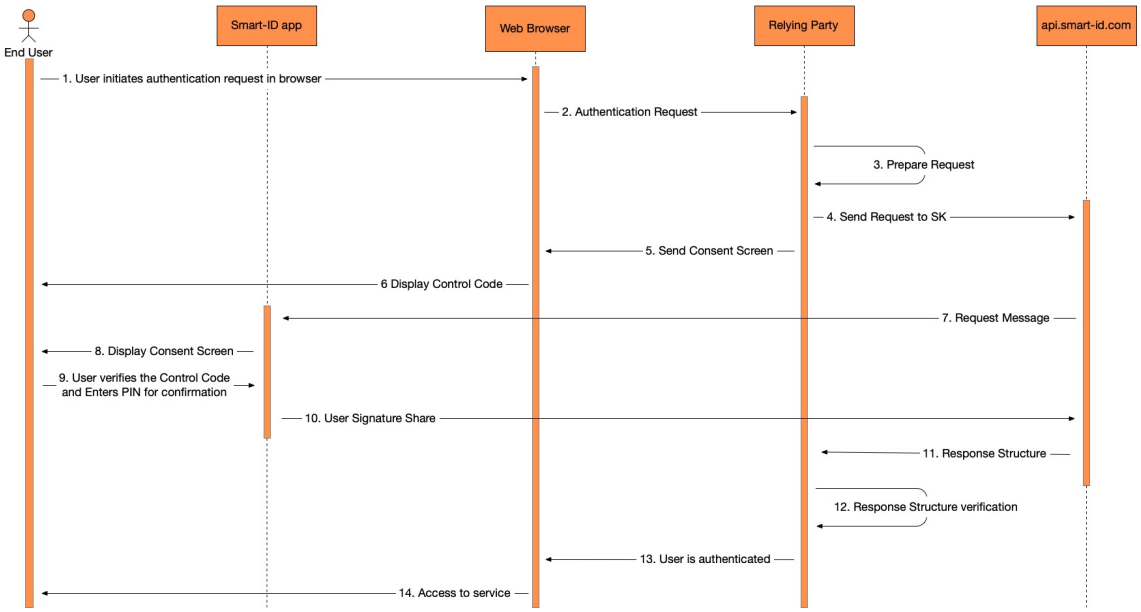

Figure 1. *Smart-ID authentication protocol.*

In the wild, there are largely four different categories of MITM attacks:

1. Spoofing - IP, HTTPS, DNS, etc. - to lure User into using a malicious server in order to gain access to data or services.
2. Hijacking - SSL, e-mail - the information shared between the victim's device and the server is intercepted by another endpoint and another server.
3. Cookie theft - by stealing cookies from your browsing sessions, criminals can obtain passwords and various other types of exclusive data.
4. Eavesdropping - whole online activity (including login credentials and payment card information) will be at the command of the middleman.

Main phishing-related security concerns are related to the following attacks:

1. Blind PIN trial, an attacker attempts to impersonate User at RP by sending a fake authentication or signing request to Smart-ID App. The attacker hopes that User will enter PIN to Smart-ID App.
2. Simple social engineering, an attacker tries to impersonate User at RP by tricking (social-engineering) User into accepting an authentication request. This can be done through various means, such as phishing emails, fake websites or other forms of social engineering.
3. Man in the middle, an attacker tricks User into clicking on a fraudulent link to intercept the communication between User and the RP and impersonates User to the RP.
4. Man in the browser, an attacker tricks User into clicking on a fraudulent link to the

adversary's website where a fraudulent script is downloaded to Browser. The script then tries to impersonate User at RP.

2.3 Insufficiency of the Existing Measures

2.3.1 General Model

We describe a general model to help analyse attacks systematically later. All communications are done through cryptographically protected channels. The model has the following steps (Figure 2):

1. At process start, request message (or transaction) M is created. RP computes a hash of the message $m \leftarrow h(M)$ and derives control code using the deterministic function from message hash $cc \leftarrow f(m)$. The display string is computed from the message: $ds \leftarrow survey(M)$. RP sends M and cc to Browser, where they are displayed to User.
2. RP sends computed hash m and display string ds to Smart-ID Server.
3. Smart-ID Server forwards the previous step hash m and display string ds to Smart-ID App. User verifies display string ds and that control code cc matches with Browser's visible control code cc.
4. As a confirmation, User enters PIN to Smart-ID App. The application creates User's share $s_U^{MD} \leftarrow Sig_U^{MD}(m)$ of the signature and sends s_U^{MD} to Smart-ID Server.
5. Smart-ID Server creates Server's share $s_U^{SP} \leftarrow Sig_U^{SP}(m)$ of the signature and composes the whole signature $Sig_U(m) \leftarrow Comp(s_U^{MD}, s_U^{SP})$. The composed signature is verified with User's public key. On successful result, $Sig_U(m)$ is sent to RP.

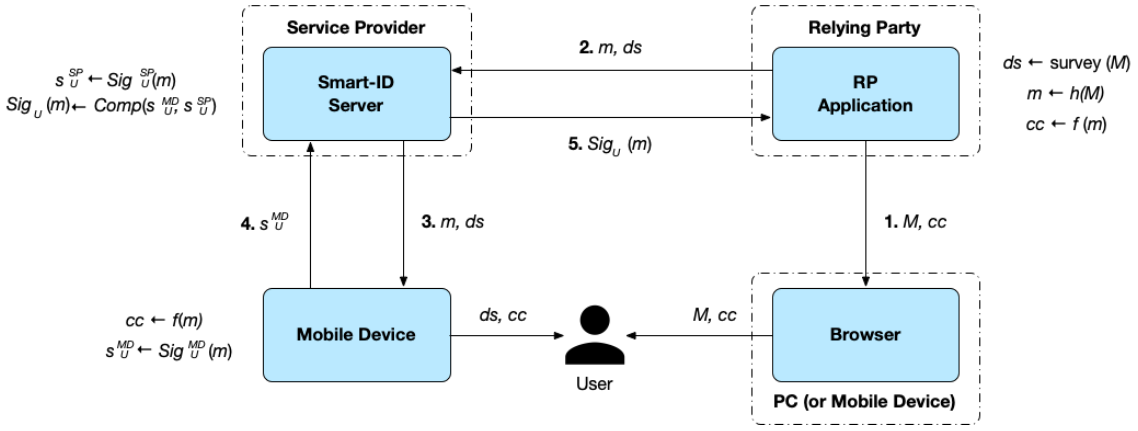

Figure 2. *General model.*

2.3.2 Blind PIN Trial Attack

Blind PIN trial attack is similar to the general model, except the process is started by a malicious party and hopes that User blindly enters PIN (Figure 3):

1. At process start, request message (or transaction) M is created. RP computes a hash of the message $m \leftarrow h(M)$ and derives control code using the deterministic function from message hash $cc \leftarrow f(m)$. The display string is computed from the message: $ds \leftarrow survey(M)$. RP sends M and cc to Attacker's Personal Computer (or Mobile Device) browser, where they are displayed to the Attacker.
2. RP sends computed hash m and display string ds to Smart-ID Server and Smart-ID Server forwards them to Smart-ID App.
3. User ignores the display string ds and control code cc and blindly enters PIN to Smart-ID App. The application creates User's share $s_U^{MD} \leftarrow Sig_U^{MD}(m)$ of the signature and sends s_U^{MD} to Smart-ID Server.
4. Smart-ID Server creates Server's share $s_U^{SP} \leftarrow Sig_U^{SP}(m)$ of the signature and composes the whole signature $Sig_U(m) \leftarrow Comp(s_U^{MD}, s_U^{SP})$. The composed signature is verified with User's public key. On the successful result, $Sig_U(m)$ is sent to RP. After that, the attacker has access to the RP service on behalf of User's account.

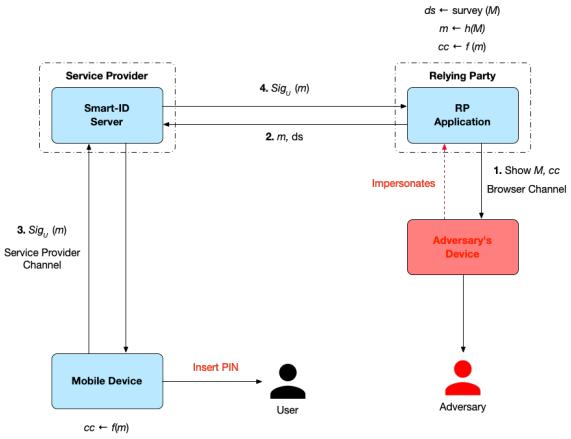

Figure 3. *Blind PIN trial attack.*

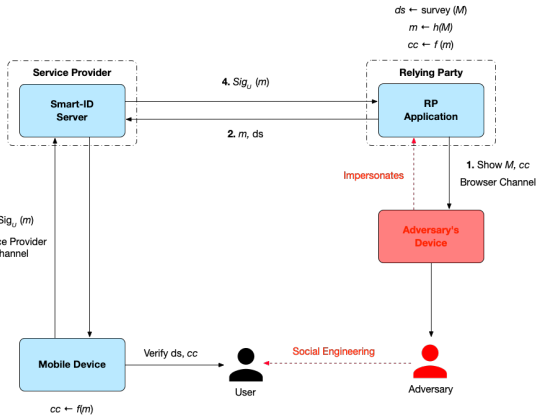

Figure 4. *Social engineering attack.*

2.3.3 Social Engineering Attack

In social engineering, the attacker initiates a request and communicates over the phone or other means indent M and control code cc to User (Figure 4):

1. At process start, request message (or transaction) M is created. RP computes a hash of the message $m \leftarrow h(M)$ and derives control code using the deterministic function

from message hash $cc \leftarrow f(m)$. The display string is computed from the message: $ds \leftarrow survey(M)$. RP sends M and cc to Attacker's Personal Computer (or Mobile Device) browser, where they are displayed to The Attacker. The Attacker forwards request M and control code cc to User over the phone or other channels.

2. RP sends computed hash m and display string ds to Smart-ID Server and Smart-ID Server forwards them to Smart-ID App.
3. User verifies the display string ds and that control code cc matches with Attacker's instructed control code cc and, as a confirmation, enters PIN to Smart-ID App. The application creates User's share $s_U^{MD} \leftarrow Sig_U^{MD}(m)$ of the signature and sends s_U^{MD} to Smart-ID Server.
4. Smart-ID Server creates Server's share $s_U^{SP} \leftarrow Sig_U^{SP}(m)$ of the signature and composes the whole signature $Sig_U(m) \leftarrow Comp(s_U^{MD}, s_U^{SP})$. The composed signature is verified with User's public key. On successful result, $Sig_U(m)$ is sent to RP. After that, the attacker has access to the RP service on behalf of User's account.

2.3.4 Man in the Middle Attack

In a MITM attack, User is tricked into visiting the attacker's fraudulent site (e.g., https://www.bnak.eu) and the attacker hopes that User does not pay attention to the fact that the domain is fake and that the attacker is impersonating the RP (Figure 5):

1. Fraudulent RP initiates at RP process start where request message (or transaction) M is created. RP computes a hash of the message $m \leftarrow h(M)$ and derives control code using the deterministic function from message hash $cc \leftarrow f(m)$. The display string is computed from the message: $ds \leftarrow survey(M)$. RP sends M and cc to Fraudulent RP Service, where it is forwarded to Browser and is displayed to User.
2. RP sends computed hash m and display string ds to Smart-ID Server and Smart-ID Server forwards them to Smart-ID App.
3. User ignores the display string ds. User verifies that the control code cc matches with Browser's visible control code cc and, as a confirmation, enters PIN to Smart-ID App. The application creates User's share $s_U^{MD} \leftarrow Sig_U^{MD}(m)$ of the signature and sends s_U^{MD} to Smart-ID Server.
4. Smart-ID Server creates Server's share $s_U^{SP} \leftarrow Sig_U^{SP}(m)$ of the signature and composes the whole signature $Sig_U(m) \leftarrow Comp(s_U^{MD}, s_U^{SP})$. The composed signature is verified with User's public key. On successful result, $Sig_U(m)$ is sent to the RP. After that, the fraudulent RP has access to the RP service on behalf of User's account.

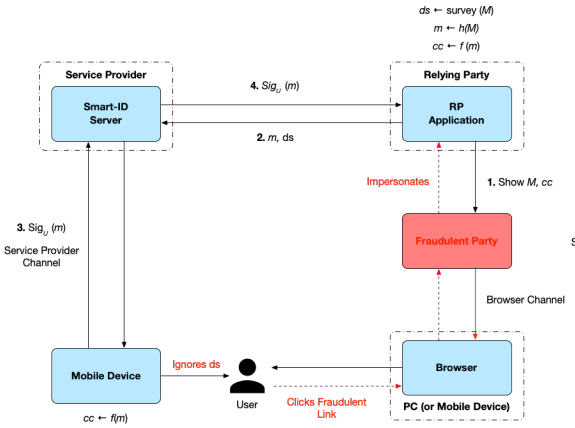

Figure 5. *Man in the middle attack.*

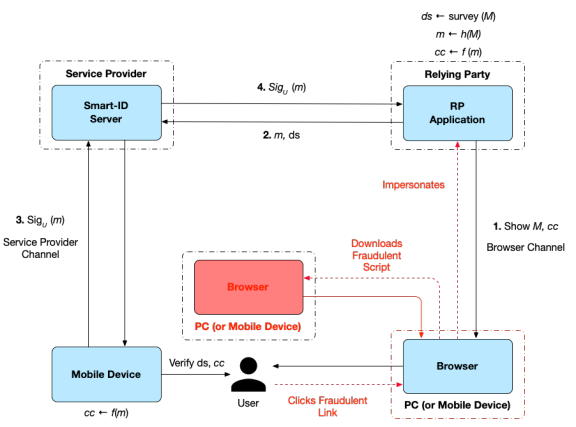

Figure 6. *Man in the browser attack.*

2.3.5 Man in the Browser Attack

In the man in the browser attack, User is tricked into clicking on a fraudulent link to the adversary's website, where a fraudulent script is downloaded to Browser. The script then tries to impersonate User at RP (Figure 6):

1. At process start, request message (or transaction) M is created. RP computes a hash of the message $m \leftarrow h(M)$ and derives control code using the deterministic function from message hash $cc \leftarrow f(m)$. The display string is computed from the message: $ds \leftarrow survey(M)$. RP sends M and cc to Browser, where they are displayed to User.
2. RP sends computed hash m and display string ds to Smart-ID Server and Smart-ID Server forwards them to Smart-ID App.
3. User verifies the display string ds and that control code cc matches with Browser's visible control code cc and, as a confirmation, enters PIN to Smart-ID App. The application creates User's share $s_U^{MD} \leftarrow Sig_U^{MD}(m)$ of the signature and sends s_U^{MD} to Smart-ID Server.
4. Smart-ID Server creates Server's share $s_U^{SP} \leftarrow Sig_U^{SP}(m)$ of the signature and composes the whole signature $Sig_U(m) \leftarrow Comp(s_U^{MD}, s_U^{SP})$. The composed signature is verified with User's public key. On the successful result, $Sig_U(m)$ is sent to RP. After that, the fraudulent script has access to the RP service on behalf of User's account.

Most current Web browsers prevent carrying out this type of attack. Browsers have a built-in mechanism called Cross-Origin Resource Sharing (CORS), which allows web browsers to make cross-domain HTTP requests in a controlled manner. When a web page served from one domain (origin) attempts to make a request to a resource on another

domain, the browser checks if the resource server explicitly allows the request by sending an HTTP response header called "Access-Control-Allow-Origin". The Access-Control-Allow-Origin header specifies the allowed origins for a resource. If the server allows the request, the browser then checks if the resource server allows the request method, headers and credentials (such as cookies or authentication tokens).

The CORS mechanism works as follows:

1. Browser sends an HTTP request to the resource server.
2. The resource server responds with an HTTP response that includes the "Access-Control-Allow-Origin" header, indicating which origins are allowed to access the resource.
3. If Browser's origin is allowed, Browser checks if the resource server allows the request method, headers and credentials.
4. If the request is allowed, Browser sends the request to the resource server.
5. The resource server sends a response to Browser.
6. Browser checks the response and any headers that the response can be shared across origins.
7. If any checks fail, Browser will block the request and return an error to the JavaScript code that made the request.

2.3.6 Countermeasures

Multiple countermeasures have been implemented to prevent carrying out the attacks. The following provides a short overview of these measures:

1. Relying Party Name - The service name shown in Smart-ID App. It gives context to the current transaction.
2. Display String - Additional text is shown in Smart-ID App. This should give a hint about the intent of the current transaction. In version 1 of the Smart-ID API, the limit is up to 60 characters, in version 2, it is increased to 200 characters.
3. Control Code - Shown in the RP service and also in Smart-ID App. User must verify that they match before entering PIN for confirmation.
4. Control Code Choice - In Smart-ID API version 2, an optional choice for RP to use Control Code is added. Smart-ID App shows three Control Code options and User must find the correct code that matches the RP shown code. This helps to prevent simple Blind PIN trial attacks.
5. QR code - A suggested measure for showing a complicated code in the browser that helps to prevent Simple social engineering attacks (See 2.4).

6. CORS - the browser built-in measure to prevent injecting malicious code in RP web page.

Table 1 describes the attack-thwarting power of existing security measures described in sub-section 2.3 against various attacks. The index column describes the attack vector and the index row describes the protection measures implemented or proposed. This results in a matrix where the colour codes indicate the effectiveness of a particular security measure. The index column describes the attack vector, and the index row describes the protection measures implemented or proposed. This results in a matrix where the colour codes indicate the effectiveness of a particular security measure:

- Thwarts attack completely (green cell with Yes).
- Provides a user-verifiable element during the process (yellow cell with Observable).
- Does not have any influence on the attack (red cell with No).

	Relying Party Name	**Display String**	**Control Code**	**Control Code Choice**	**QR code**	**CORS**
Blind PIN trial	Observable	Observable	No	Yes	Yes	No
Social engineering	Observable	Observable	No	No	Yes	No
Man in the middle	Observable	Observable	No	No	No	No
Man in the browser	Observable	Observable	No	No	No	Yes

Table 1. *Attacks and countermeasures*

2.4 Measure: QR Code

The Control Code Choice measure helps to avoid simple PIN trial attacks but does not help with social engineering attacks. One possible solution is to use a more complicated Control Code alternative, such as a QR Code. This makes it harder for attackers to replay the code to User over the phone or other channels. The idea is that instead of inserting a control code or selecting the correct code, User must scan a QR code with Smart-ID App and then the application verifies the code. The QR Code measure is still unsuccessful against MITM attacks. The fraudulent application can still intercept the QR code and show this to User on a fraudulent website (Figure 7).

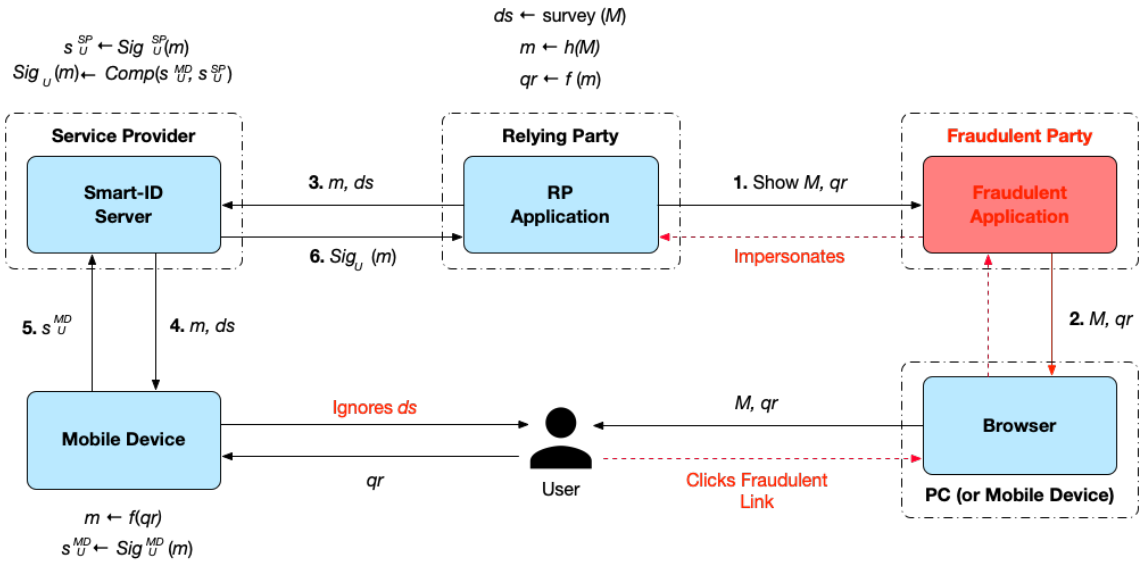

Figure 7. *Man in the middle attack with QR code.*

1. Fraudulent RP initiates at RP process start where request message (or transaction) M is created. RP computes a hash of the message $m \leftarrow h(M)$ and generates a QR code using the deterministic function from message hash $qr \leftarrow f(m)$. The display string is computed from the message: $ds \leftarrow survey(M)$. RP sends M and qr to the Fraudulent RP Service.
2. Fraudulent RP Service forwards M and qr to Browser and is displayed to User.
3. RP sends the computed hash m and the display string ds to Smart-ID Server.
4. Smart-ID Server forwards the previous step hash m and the display string ds to Smart-ID App. User verifies the display string ds and scans the QR code qr with Smart-ID App from Browser. Smart-ID App verifies that derived $m \leftarrow f(qr)$ from the QR code matches with received m from Smart-ID Server.
5. As a confirmation, User enters PIN to Smart-ID App. The application creates User's share $s_U^{MD} \leftarrow Sig_U^{MD}(m)$ of the signature and sends s_U^{MD} to Smart-ID Server.
6. Smart-ID Server creates the Server's share $s_U^{SP} \leftarrow Sig_U^{SP}(m)$ of the signature and composes the whole signature $Sig_U(m) \leftarrow Comp(s_U^{MD}, s_U^{SP})$. The composed signature is verified with User's public key. On the successful result, $Sig_U(m)$ is sent to RP. After that, the fraudulent RP has access to the RP service on behalf of User's account.

2.5 Browsers

A web browser is a software application that allows users to access and interact with content on the World Wide Web. By entering a web address (URL) in a browser, several tasks are executed to display the content of the web page:

1. Web address is resolved to an IP address using the DNS system.
2. Then TCP connection is established with the web server using the IP address from the previous step. The browser sends a request for the web page to the server.
3. The server responds with the requested web page, which may include HTML, CSS, JavaScript and other resources such as images and videos and the content is rendered to the screen.
4. If the web page contains JavaScript code, the browser will execute it.
5. With the user interactions (e.g., scrolling, clicking on links), the browser may need to request and render additional resources or update parts of the page dynamically using JavaScript.
6. The browser also performs various security checks, such as verifying the TLS certificate of the webserver to ensure that the connection is secure and protecting against malicious code and phishing attacks.

2.5.1 TLS/SSL and Client Certificate Authentication

Transport Layer Security (TLS) is a protocol to have a secure connection over the internet. It supersedes the Secure Sockets Layer (SSL). TLS is used to establish a secure, encrypted connection between a **Client** (such as a web browser) and a **Server** (such as a web server). An encrypted connection is needed to protect sensitive data such as login credentials, credit card information, and other personal or financial information from being intercepted by attackers.

1. A TLS handshake starts with Client's request for a secure connection with Server where Client presents supported cipher suites.
2. Server responds to the request with its selection from the previous cipher suites list and provides a corresponding certificate including the public key and Server name.
3. Client verifies Server name and the certificate against the certificate authority.
4. Client uses Server's public key to establish a secure connection with Server and to generate a session key.
5. The session key is used for symmetric encryption during the session.

Additionally, Client can use Client Certificate Authentication (CCA) for authentication. Client uses a government-issued electronic identity (eID) card or other cryptographic tokens that contain an authentication certificate and is used during TLS handshake, to prove their identity to Server. This prevents MITM attacks and gives assurance to Server that the certificate is issued by a trusted certificate authority. Setting up TLS-CCA properly can be challenging, as described in the paper "Practical Issues with TLS Client Certificate

Authentication" [10].

1. A TLS handshake starts with Client's request for a secure connection with Server, where Client sends supported chipher suites.
2. Server responds to the request with its selection from the previous cipher suites list and provides a corresponding certificate including the public key and Server name.
3. Client verifies Server name and the certificate against certificate authority.
4. Client then sends its own certificate, which includes a public key, to Server.
5. Server verifies Client's certificate and generates a session key and encrypts it with Client's public key.
6. Client decrypts the session key using its private key.
7. The session key is used for symmetric encryption during the session.

2.5.2 Browser Extensions

Browser extensions are browser add-ons or plugins that allow users to customise their web browsers by adding or modifying new features. Web-Extensions are created using familiar web-based technologies — HTML, CSS and JavaScript. They are built using standard web technologies, but an extension has access to its own set of JavaScript API's. Extensions can be used to add functionality to web pages, change the appearance of the browser or modify the behaviour of the browser itself.

Browser extensions are supported by most modern web browsers, including Google Chrome, Mozilla Firefox, Microsoft Edge, Safari and Opera. Each browser has its own extension platform with different API's and development tools.

Browser extensions can perform a variety of tasks, such as:

- Blocking ads or other content
- Enhancing privacy and security
- Adding new features to web pages
- Modifying the appearance of the browser
- Managing bookmarks and downloads

Browser extensions are usually distributed through repositories or stores maintained by browser vendors. Users can browse and install extensions from these repositories and can manage their extensions from within the browser's settings.

2.5.3 Web-Extensions - JavaScript API

The current thesis covers only relevant Web-Extension components. These are: *background.js*, *content.js* and *option.js* as can be seen in Table 2.

Table 2. *Extension Components*

Component	**Description**
content.js	Executed within website context and this script has direct communication with the website with no communication with *background.js*.
background.js	Executed within the extension context and has the permission to store to a secure area to which the website and also content.js script have no access. Background scripts can use any of the Web-Extension API's in the script, as long as their extension has the necessary permissions.
options.js	The extension preferences window has permission to access the secure area. It has the same level of permissions as the *background.js* script.

In Table 3 are listed relevant JavaScript API's to this thesis [11, 12, 13] that are used in proof-of-concept solution in Chapter 4 and are supported by most commonly used web browsers.

Table 3. *JavaScript API Components*

JavaScript API	**Description**
browser.storage.local.get	Retrieves one or more items from the storage area.
browser.storage.local.set	Stores one or more items in the storage area or updates existing items.
crypto.subtle.generateKey	Uses the *generateKey()* method of the SubtleCrypto interface to generate a new key (for symmetric algorithms) or key pair (for public-key algorithms).
crypto.subtle.exportKey	The *exportKey()* method of the SubtleCrypto interface exports a key: that is, it takes as input a CryptoKey object and gives you the key in an external, portable format.

Continues...

Table 3 – *Continues...*

JavaScript API	**Description**
crypto.subtle.importKey	The *importKey()* method of the SubtleCrypto interface imports a key: that is, it takes as input a key in an external, portable format and gives you a CryptoKey object that you can use in the Web Crypto API.
crypto.subtle.sign	The *sign()* method of the SubtleCrypto interface generates a digital signature.
crypto.subtle.digest	The *digest()* method of the SubtleCrypto interface generates a digest of the given data. A digest is a short fixed-length value derived from some variable-length input. Cryptographic digests should exhibit collision- resistance, meaning that it's hard to come up with two different inputs that have the same digest value.
qrcode library	QR code/2d barcode generator [14].

2.5.4 Conclusion

Web browsers are facilitating a number of security mechanisms. For example, using TLS certificates provides assurance that the website is genuine, and using TLS prevents eavesdropping. These are effective against a number of attack vectors, but there is no built-in protection in browsers that prevents MITM attacks. It is relatively easy to obtain TLS certificates for fraudulent middlemen. This transfers the obligation to verify the content of TLS certificates to the user. There is the option to use TLS/CCA, but this does not apply to the Smart-ID protocol.

Widely used browsers (e.g. Chrome, Edge, Firefox, Safari) have various tools that enable the use of browser extensions, which can provide increased security to current protocols. Browser extensions can store data in a private context that is not leaked to websites, capture the visited website URL, use cryptography API to sign messages, and provide a preference page for extension configuration.

- Browser extensions can use the browser's Storage API to store data in the extension context while the data is not accessible for websites. This can be used for storing user settings and pre-shared secrets.
- Browser extensions have access to the active website URL using Tabs API. This is useful for capturing the URL of current web page and is used to prevent MITM attacks.

- Browser extensions can provide a preference page for the configuration of the extension's settings. This can be used to generate with SubtleCrypto API pre-shared secrets and sign messages.
- Browser extensions can show graphics (e.g. images and QR codes) to show additional information to the user.

3. New Measures to Prevent Man in the Middle Attack

The aim was to develop a prototype that prevents MITM attacks on the current protocol considering a list of following boundaries:

1. The authentication process must facilitate Smart-ID Server.
2. The solution must respect the limitations of the existing process flow.
3. The solution must not have the capability to connect directly to the Smart-ID API.
4. The current protocol must be improved but not replaced with OAuth [15] or any other protocol.
5. The protocol must prevent the usage of URL redirects.
6. The proposed solution must be supported by the most widely used browsers and devices.
7. The usage of QR code is permitted as all the devices compatible with Smart-ID have cameras.
8. Other possibilities include Near-Field communication (NFC), but it is not preferable due to restricted usage on Apple devices.

This chapter will introduce new measures to prevent MITM attacks. Section 3.1 will explain how new pre-shared secret is generated and distributed to Smart-ID App. Section 3.2 describes a new and improved protocol. Section 3.3 shows that the attack will not carry out on the new protocol.

3.1 Key Establishment

The process starts with User's initiation of the browser extension installation using the browser's extension store. As the extension is installed, Browser initiates the pre-shared secret generation process and generates the QR code to be scanned with Smart-ID App. User then scans the QR code by connecting Smart-ID App to the browser extension. The pre-shared secret shall be stored with Smart-ID App for future usage. The process is described in detail in Figure 8 and contains the next steps:

1. User initiates browser extension installation request using Browser.
2. Browser downloads the browser extension from the browser's extension store.
3. Browser installs the browser extension.
4. The installation status of the browser extension is displayed to User.

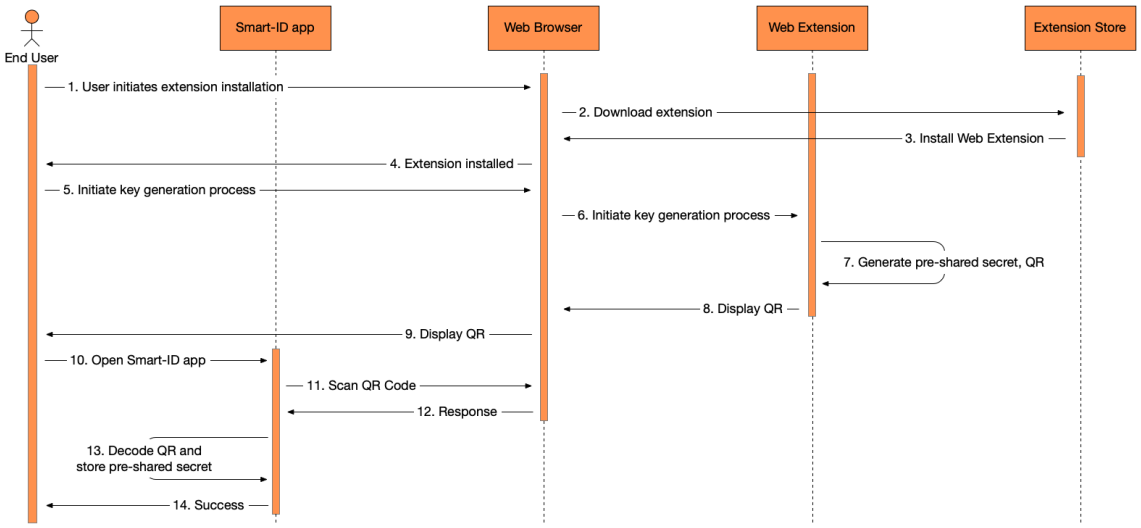

Figure 8. *Key exchange protocol with the browser extension.*

5. User initiates the generation of the pre-shared key.
6. Browser initiates the extension key generation process.
7. The extension generates pre-shared secret, stores it in secure area and the key is encoded in a QR Code.
8. Browser renders the extension-generated QR code.
9. Browser displays QR Code to User.
10. User opens Smart-ID App.
11. Using Smart-ID App, User scans the QR code displayed by Browser.
12. Smart-ID App receives the response from the QR code.
13. Smart-ID App decodes the QR code and stores the pre-shared secret in secure storage.
14. Key exchange successful, success message is displayed to User.

3.2 Modified Authentication Protocol

Using the browser extension enables the secure delivery of the URL currently used. The service provider generates a M during the authentication process which will be used to generate a QR code to be displayed on the website. Then, User, using Smart-ID App, scans the QR code displayed and if the M matches both on the website and on Smart-ID App, User is prompted for the corresponding PIN to confirm the transaction. The modified authentication protocol is described in detail in Figures 9, 10 and consists of the following steps:

1. User initiates authentication request using Browser.
2. Browser announces RP about intention to continue with the authentication process.

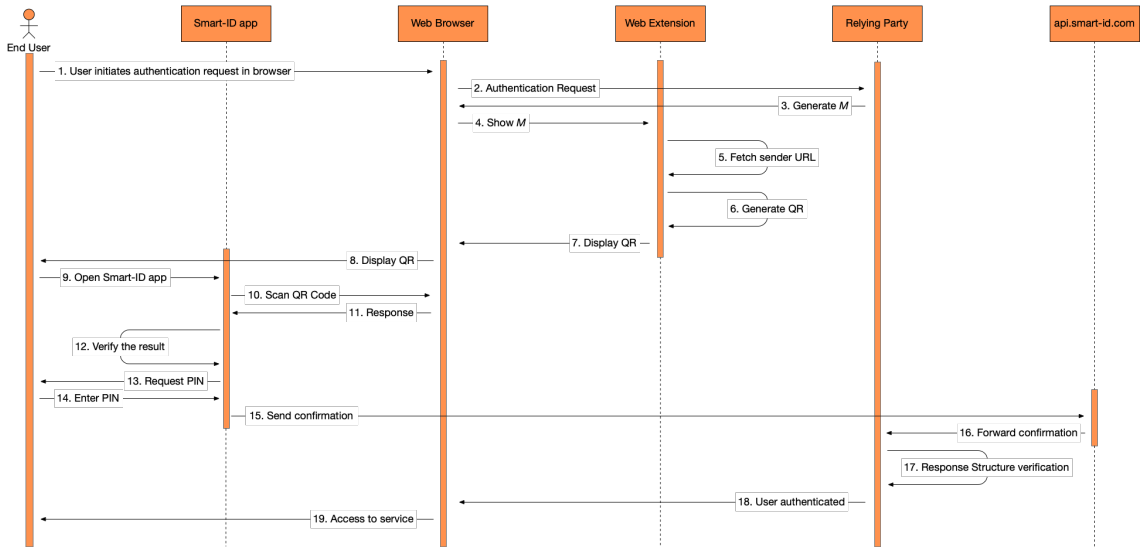

Figure 9. *Modified authentication protocol description.*

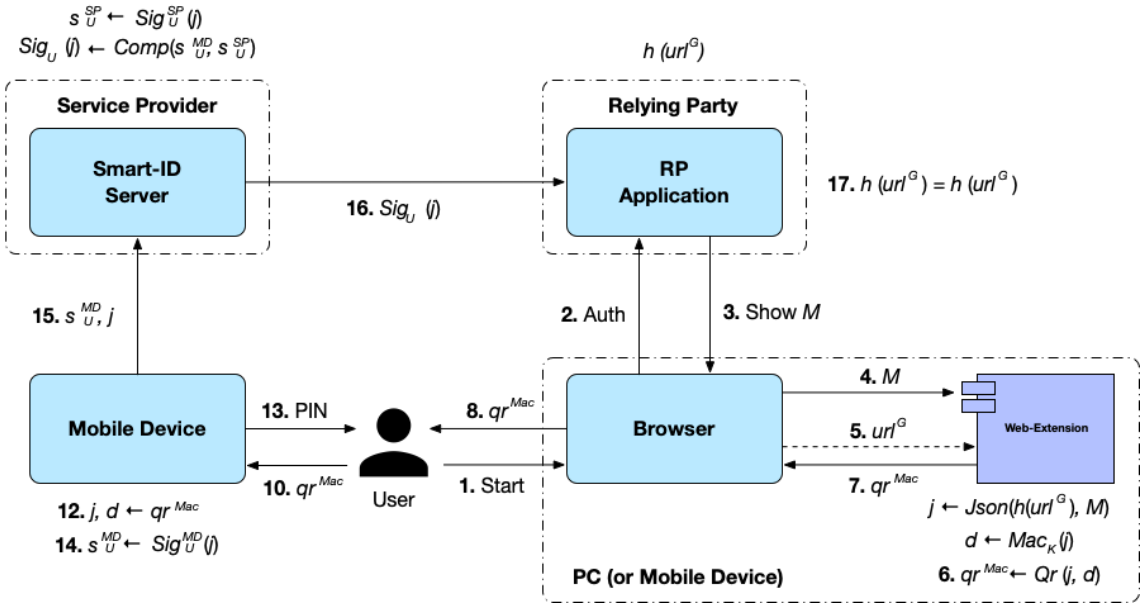

Figure 10. *Model: Authentication with the browser extension code.*

3. RP generates a M that is forwarded to Browser.
4. Browser launches the browser extension with the M as a parameter.
5. Browser extension fetches message sender url secure manner and incorporates the M to the response.
6. The response is signed with the pre-shared secret and then encoded to a QR code qr^{Mac}.
7. The generated QR Code is then forwarded to Browser.
8. Browser displays the rendered QR code to User.
9. User opens Smart-ID App.
10. User scans the rendered QR code with Smart-ID App.

11. Smart-ID App receives the QR code and decodes the message.
12. Smart-ID App finds the correct pre-shared secret referenced in the message and verifies the signature.
13. If successful, User is prompted PIN for confirmation.
14. User confirms the transaction by entering corresponding PIN.
15. Smart-ID App sends the transaction to Smart-ID Server.
16. Smart-ID Server forwards User's signed transaction to RP.
17. RP verifies the transaction content (url, M, signature).
18. On the successful result, User is authenticated.
19. User is granted access to the website.

3.3 Analysis

The following model on Figure 11 describes the process where a malicious URL is detected by the relying party and therefore this renders the MITM attack unsuccessful. The browser extension verifies the URL against RP. The verification fails because of the URL mismatch.

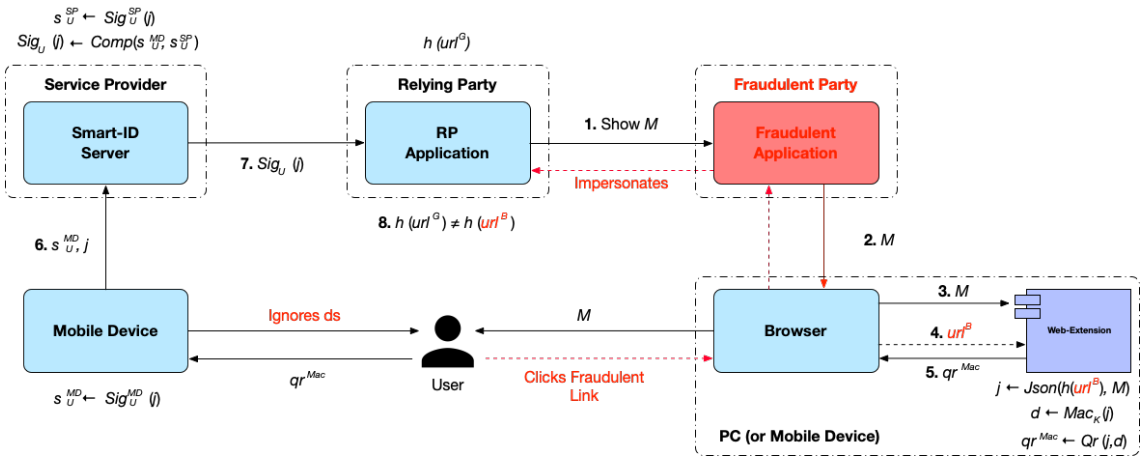

Figure 11. *Man in the middle attack with the browser extension code.*

1. Fraudulent RP initiates a RP process start where request message (or transaction) M is created. RP sends M Fraudulent RP Service.
2. Fraudulent RP Service forwards M to Browser.
3. The browser extension is executed with given parameters M.
4. The browser extension acquires current web site url^B, in this case its Fraudulent Party URL.
5. The browser extension generates qr^{Mac} from combining Fraudulent Party URL url^B and the request message M, computes $j \leftarrow Json(h(url^B), M)$, calculates MAC with pre-shared secret $d \leftarrow Mac_K(j)$, and generates QR code that includes j, d and renders it on the web page.

6. User scans the visible QR code qr^{Mac} and verifies message with the pre-shared secret K. On positive result, gives confirmation by entering PIN to Smart-ID App. The application creates User's share $s_U^{MD} \leftarrow Sig_U^{MD}(j)$ of the signature, and sends s_U^{MD}, j to the Service Provider.
7. Smart-ID Server creates Server's share $s_U^{SP} \leftarrow Sig_U^{SP}(j)$ of the signature and composes the whole signature $Sig_U(j) \leftarrow Comp(s_U^{MD}, s_U^{SP})$. The composed signature is verified with User's public key. On the successful result, $Sig_U(j)$ is sent to RP. After that, RP detects that User is directed to a malicious URL $h(url^G) \neq h(url^B)$ and therefore denies access to the service.

4. Prototype Solution

The new authentication solution is based on the idea that during the Smart-ID authentication process, the system can securely identify the website where User is currently located and link it to the authentication token. Later, the attacked service can verify whether User was on their website and did not use any intermediate attack website. Since web browsers do not have such technology today, it is necessary to use, for example, a browser extension to implement it.

This chapter will propose a proof of concept solution. Section 4.1 will explain the scope of created solution. Section 4.2 describes the details of created proof of concept browser extension. Section 4.3 shows user experience of the newly created solution.

4.1 Scope of the Solution

The currently implemented solution is a proof of concept for the Firefox browser. The author does not have access to the source code of the Smart-ID application or the Smart-ID service. The Smart-ID key exchange behaviour and QR code authentication are emulated in the JavaScript application.

The demo RP service is also created, and the Smart-ID back-end prototype is implemented to emulate the newly created protocol. The prototype RP service allows testing the whole process from the key pairing until successful authentication to the RP service. With the prototype, the effectiveness against the MITM attacks of the new protocol can be presented and verified.

4.2 Description

On the extension settings page (*options.js*), User is prompted to generate a new key. Then *crypto.subtle.generateKey* is used for generating a new HMAC [16] key with SHA-512 parameter and exported to the *crypto.subtle.exportKey* in JSON Web Key (JWK) [17] format.

Since User may have multiple browsers and multiple workstations in use, the Key Identifier (KID) is added to the JWK by creating a hash message *crypto.subtle.digest* using the K value of the key and encoded in *base64url* [18] format.

The JWK is stored in the secure area of the extension using *browser.storage.local.set*. A QR code is generated from this JWK and shown on the settings page. User captures the QR code with the Smart-ID App and stores decoded JWK from QR code in the Smart-ID App. An illustration of the process is provided in Figure 12.

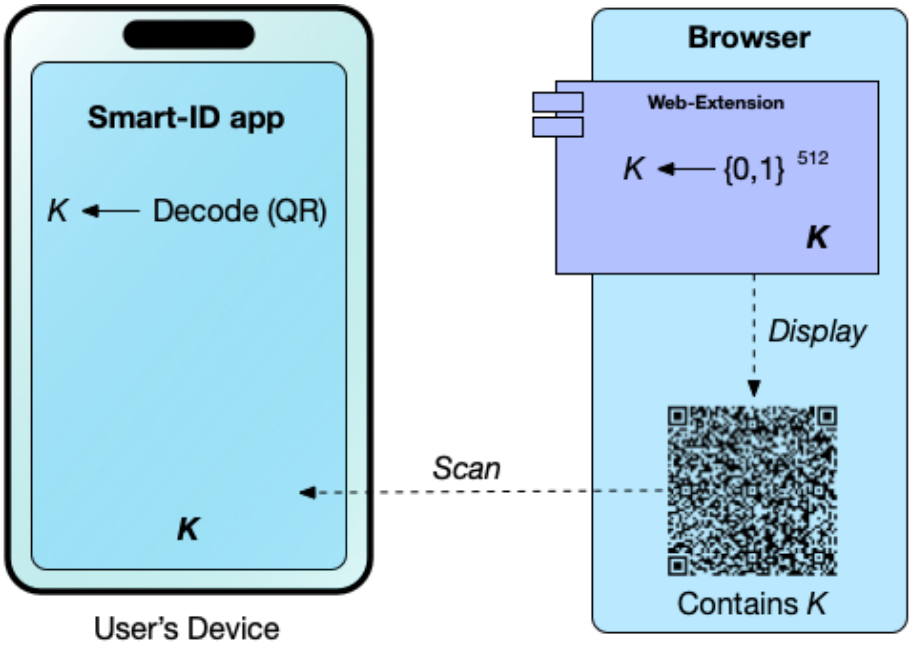

Figure 12. *Key generation process description.*

When User tries to authenticate on a page that requires the extension, the website sends a message to the extension and provides a M and an identifier (URL) using which the service provider SK ID Solutions can send a signed authentication token message back to that page.

content.js receives this message and adds the URL of the given page to the message and redirects to the *background.js* page. *background.js* securely loads the JWK key from the extension's secure area using the *browser.storage.local.get* and imports using *crypto.subtle.importKey* API to SubtleCypto key format.

An object of type JSON Web Token (JWT) [19] is created with a reference to the type *'typ'*, the signature algorithm HS512 *'alg'*, and the key identifier *'kid'* in the header. The content of the message is the M *'jti'* and captured page URL *'sub'*. A signature is added to this object, which is calculated on the JWT header and content using *crypto.subtle.sign*. A QR code is generated from the received content and displayed on the web page.

```
{
  "alg": "HS512",
  "typ": "JWT",
  "kid": "d74399e5deb20055b0bf19 ..."
```

```
}
.
{
  "sub": "https://www.metsma.ee",
  "jti": "fc652dc4-40c8-11ec-973a-0242ac130003"
}
.
base64url(signatureValue)
```

User now captures the provided QR code with the Smart-ID application. It uses a KID provided within JWT header and tries to match with previously stored JWK keys. If it finds a key, it checks the validity of the given JWT signature. With a valid signature, a new JWT is generated. User's Smart-ID authentication certificate is added to the message header within the *'x5c'* array. The signature algorithm RS256 *'alg'* is also provided and using the previous message body, the given JWT is signed with User's Smart-ID authentication key. The JWT is forwarded to the Smart-ID Server.

```
{
  "alg": "RS256",
  "typ": "JWT",
  "x5c": ["MIIG7zCCBNegAwIBAgIQEAA..."]
}
.
{
  "sub": "https://www.metsma.ee",
  "jti": "fc652dc4-40c8-11ec-973a-0242ac130003"
}
.
base64url(signatureValue)
```

The website receives User-signed JWT in order to verify its validity. User has the appropriate access. The website checks if the M found inside the message matches the M previously issued. Website checks if the URL belongs to this web page. If the criteria are met, authentication is completed successfully. An illustration of the process is provided in Figure 13.

4.3 Sample Scenario

The sample scenario is provided to demonstrate the behaviour of the solution in the context of authentication.

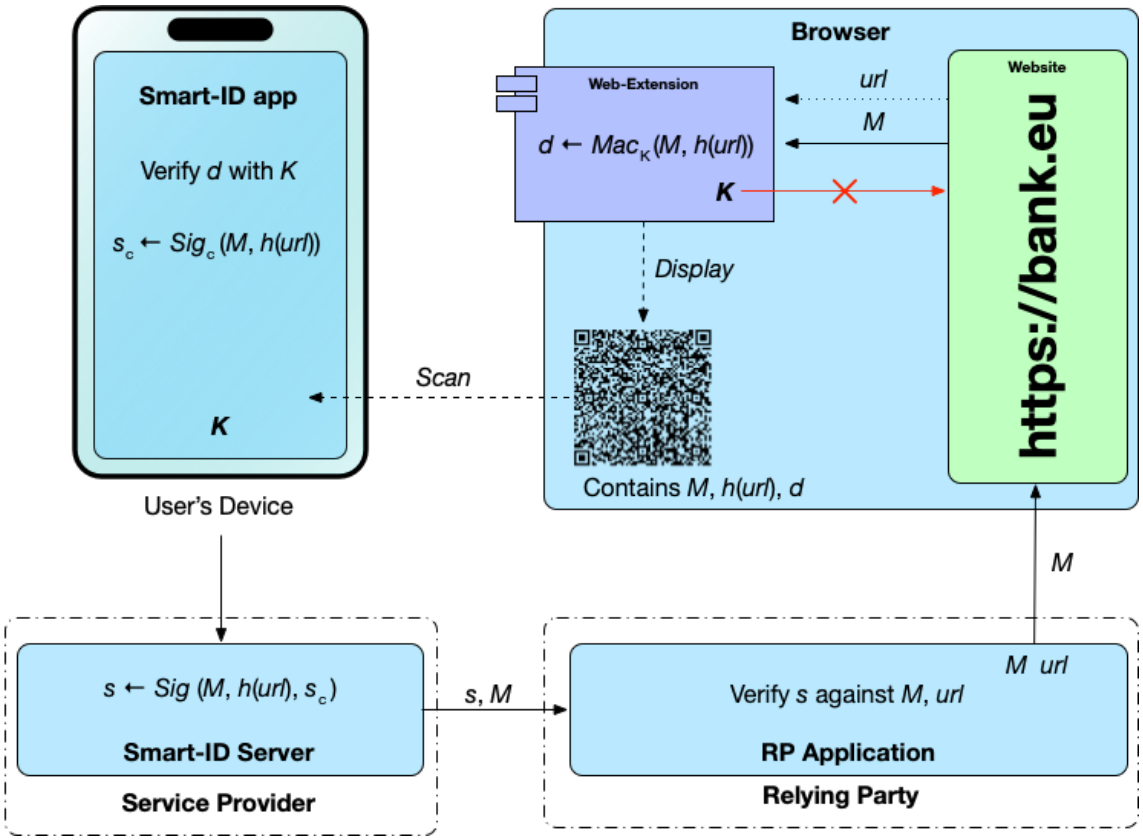

Figure 13. *Authentication process description.*

The process starts with the installation of the browser extension, followed by the key generation and pairing with the Smart-ID smartphone application. As the pairing has been completed, User can now use the extension during the authentication process.

User tries to authenticate to Relaying Party. Opens Browser and opens the RP website. The website has additional security enabled for Smart-ID protocol.

4.3.1 Browser Extension Installation

Here is visualised how User is requested to install the Smart-ID browser extension from the Firefox add-ons store when missing Figure 14, 15.

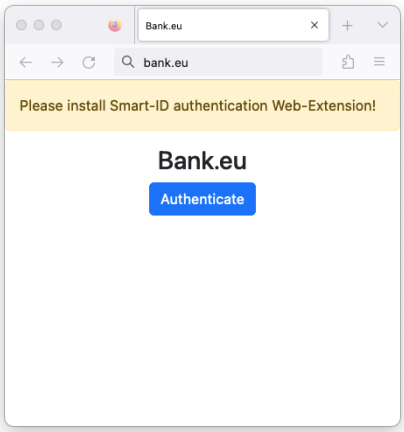

Figure 14. *Missing extension.*

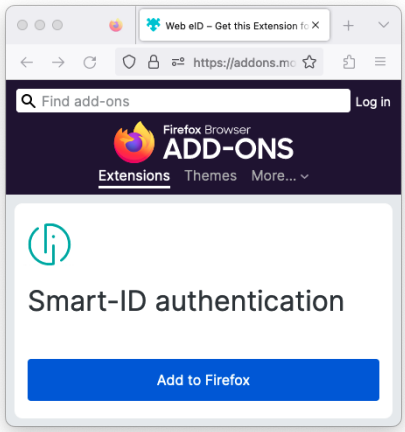

Figure 15. *Extension installation.*

4.3.2 Key Generation

On extension installation User is greeted with a welcome page and offered to generate a new pre-shared secret Figure 16. User initiates key generation. On the successful key generation, a QR code is shown to User on the extension configuration page Figure 17.

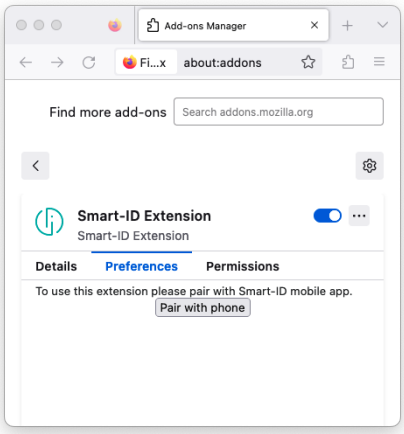

Figure 16. *Extension installed.*

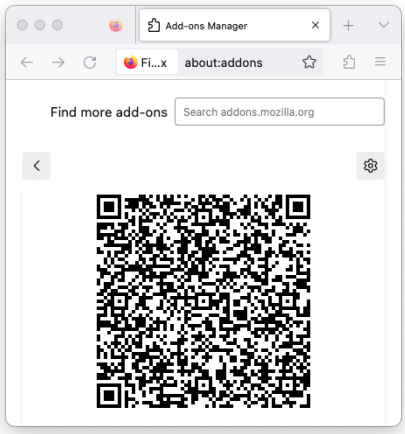

Figure 17. *Pair with Smart-ID mobile app.*

4.3.3 Pairing

User is instructed to open the Smart-ID application and activate the QR scanning feature Figure 18. The QR code is decoded on the scanning, and receives a pre-shared secret and stored in the application's private area Figure 19.

Figure 18. *Smart-ID mobile app scan button.*

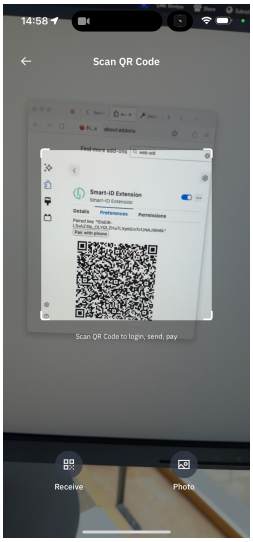

Figure 19. *Scan pre-shared secret.*

4.3.4 Authentication

Continuing with authentication on the RP website extension is activated with M parameter Figure 20. Extension captures active web page URL, uses previously generated pre-shared secret to sign the message, and shows the QR code to User Figure 21.

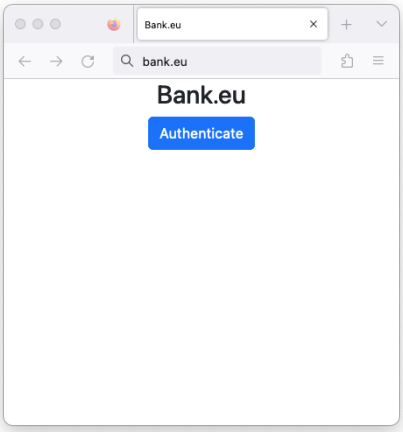

Figure 20. *Login screen.*

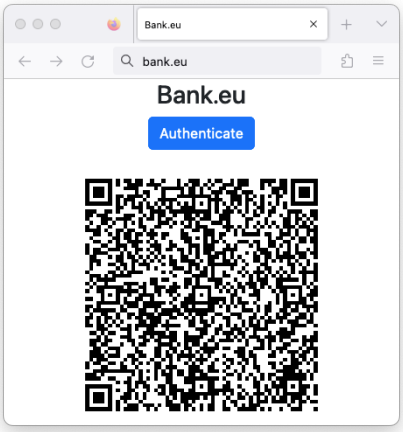

Figure 21. *Scan authentication QR.*

User is instructed to open the Smart-ID App and activate the QR scanning functionality Figure 22. On the successful QR scanning, the message signature is verified using the previously-stored pre-shared secret Figure 23.

Figure 22. *Smart-ID mobile app scan button.*

Figure 23. *Scan authentication token.*

When the message is successfully validated, the corresponding PIN is requested, and the message is signed with User's private key and sent to the RP using the Smart-ID Server Figure 24. The RP completes additional verification, and User is greeted to use authorised services on the successful result Figure 25.

Figure 24. *Smart-ID mobile app insert PIN.*

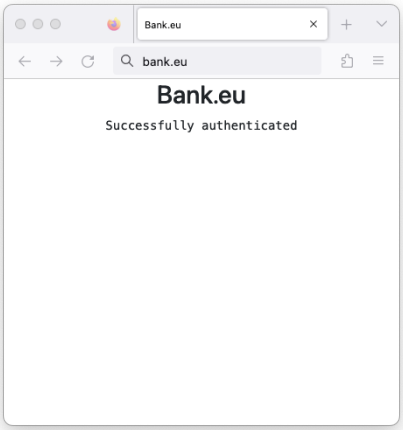

Figure 25. *RP authentication successful.*

4.3.5 Authentication Failure

When User wants to authenticate on a malicious RP website, the browser extension captures the site URL and encodes it into the QR code Figure 26. On validating, the RP checks the QR code URL parameter and responds with an authentication failure message Figure 27.

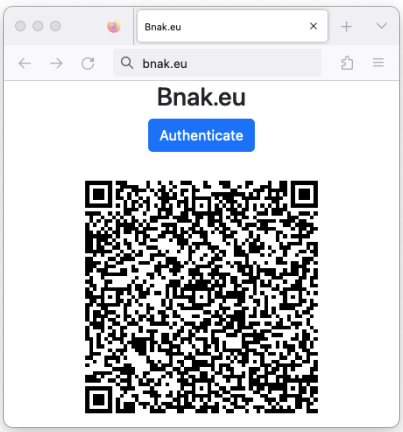

Figure 26. *Malicious RP authentication.*

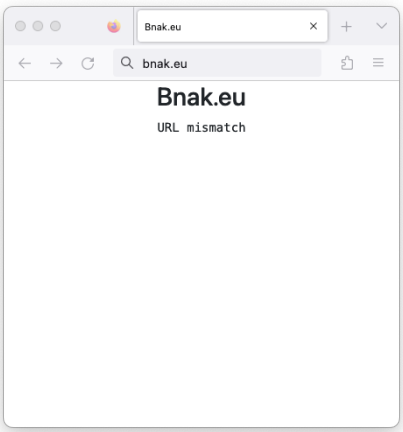

Figure 27. *RP authentication failure.*

5. Analysis, Conclusion and Future Research

Implementation of the prototype solution and testing against the prototype RP service gives strong evidence that the new measure is implementable in practice and is effective against MITM attacks. During the authentication process, the prototype service can differentiate if the request is made from a malicious or legitimate site.

While this is still a proof-of-concept solution, it needs further analysis. It provides confidence that the protocol modification with a security add-on and QR code can be efficiently implemented in practice. Future analysis is needed to determine if the message format is optimal for production usage. Feedback on the Web-eID protocol criticises using JWT in a different context than the OAuth protocol [20].

Secondly, special care must be taken in communicating with browser extensions, e.g. message passing and injecting QR codes to a website. Using secure coding best practices ensures that the pre-shared secret is not leaked to websites. Injecting a QR code directly to the website may be preferable, and using a browser built-in popup dialogue to show the QR code.

Also, the solution needs further analysis from the end-user's perspective. How complicated is installing a browser extension from the browser's repository and pairing the extension with the pre-shared secret after or during installation? Additionally, it is necessary to analyse that the new solution does not create additional attack vectors like it is hard to fake the installation process and paring in a malicious party on a website.

This solution is relatively simple to implement because most building blocks are already available in web browsers and require minimal third-party dependencies. The only external dependency is the QR encoding library in use. For industrial solutions, the browser extension needs to extend support to all major browsers and distribute browser repositories. Extension code needs to be covered with automated tests. The current Smart-ID application needs to be amended with the proposed key pairing mechanism, and also the support for the new protocol must be added to the current Smart-ID protocol.

6. Summary

This work analyses the Smart-ID authentication protocol and covers the possible man-in-the-middle attack vectors. The work proves that currently available measures are insufficient to protect the user from a list of man-in-the-middle attacks. A new protocol is proposed to prevent these types of attacks. A proof of concept prototype solution has been built that provides confidence that the new protocol prevents the given man-in-the-middle attacks. Visual reference of the user experience is also provided.

The new protocol requires a new browser extension that is acquired from the browser repository. The extension generated pre-shared secret is paired with the user Smart-ID application. Extension securely stores the relying party URL in a message and is distributed to mobile devices with a QR code. On the user's confirmation, the message is distributed to the relying party through the Smart-ID service, where it can identify the described man-in-the-middle attacks.

Additional development is needed to extend the support for Web-Extension to all major browsers and distribute the extension to browser repositories. Future analysis is required to verify that malicious actors cannot fake the same procedure on the website.

References

[1] *RIA hoiatab: mobiil-id kasutajaid püüti petta parooli andma*. [Accessed: 10-04-2023]. URL: https://www.delfi.ee/artikkel/63757670/ria-hoiatab-mobiil-id-kasutajaid-puuti-petta-parooli-andma.

[2] *Alatu skeem: kurjategijad koorisid ohvreid Smart-ID pettustega*. [Accessed: 10-04-2023]. URL: https://tehnika.postimees.ee/6682958/alatu-skeem-kurjategijad-koorisid-ohvreid-smart-id-pettustega.

[3] *Riik hindab Smart-ID-d ka pettustelaine järel turvaliseks lahenduseks*. [Accessed: 10-04-2023]. URL: https://www.err.ee/943492/riik-hindab-smart-id-d-ka-pettustelaine-jarel-turvaliseks-lahenduseks.

[4] *Pangaliit: Smart-ID pettusi aitab vältida ettevaatlikkus*. [Accessed: 10-04-2023]. URL: https://www.ituudised.ee/uudised/2019/05/23/pangaliit-smart-id-pettusi-aitab-valtida-ettevaatlikkus.

[5] *Smart-ID tegemisel on nüüd suur muudatus, mis peaks välistama võltskontode loomise*. [Accessed: 10-04-2023]. URL: https://digi.geenius.ee/rubriik/uudis/smart-id-tegemisel-on-nuud-suur-muudatus-mis-peaks-valistama-voltskontode-loomise/.

[6] *Laud, P., Roos, M.: Formal Analysis of the Estonian Mobile-ID Protocol. In: Audun Jøsang, Torleiv Maseng, and Svein J. Knapskog, (Eds.): NordSec 2009, LNCS 5838, pp.271–286 (2009)*.

[7] Cybernetica AS. *Cryptographic Algorithms Lifecycle Report*. 2016. URL: https://www.id.ee/wp-content/uploads/2020/02/cryptographic_algorithms_lifecycle_report_2016.pdf. [Accessed: 29-11-2022].

[8] *Buldas, A., Kalu, A., Laud, P., Oruaas, M.: Server-supported RSA signatures for mobile devices. In: Foley, S.N., Gollmann, D., Snekkenes, E. (Eds.): ESORICS 2017, Part I. LNCS 10492, pp. 1–19 (2017)*.

[9] SK ID Solutions. *Smart-ID Documentation*. URL: https://github.com/SK-EID/smart-id-documentation. [Accessed: 25-04-2023].

[10] Arnis Parsovs. *Practical issues with TLS client certificate authentication*. 2013.

[11] *SubtleCypto API*. [Accessed: 14-04-2023]. URL: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto.

[12] *StorageArea API*. [Accessed: 14-04-2023]. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/storage/StorageArea.

[13] *Tabs API*. [Accessed: 14-04-2023]. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/tabs/Tab.

[14] *QR Code Package*. [Accessed: 14-04-2023]. URL: https://www.npmjs.com/package/qrcode.

[15] *The OAuth 2.0 Authorization Framework*. [Accessed: 14-04-2023]. URL: https://www.rfc-editor.org/rfc/rfc6749.

[16] *Keyed-Hashing for Message Authentication*. [Accessed: 14-04-2023]. URL: https://datatracker.ietf.org/doc/html/rfc2104.

[17] *JSON Web Key*. [Accessed: 14-04-2023]. URL: https://datatracker.ietf.org/doc/html/rfc7517.

[18] *The Base16, Base32, and Base64 Data Encodings*. [Accessed: 14-04-2023]. URL: https://datatracker.ietf.org/doc/html/rfc4648.

[19] *JSON Web Token*. [Accessed: 14-04-2023]. URL: https://datatracker.ietf.org/doc/html/rfc7519.

[20] Arnis Parsovs. *On the format of the authentication proof used by RIA's Web eID solution*. 2021. URL: https://cybersec.ee/storage/webeid_auth_proof.pdf.

[21] *Html5-QRCode scanning Package*. [Accessed: 14-04-2023]. URL: https://www.npmjs.com/package/html5-qrcode.

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis[1]

I Raul Metsma

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Man in the middle attack prevention for Smart-ID using browser extensions", supervised by Ahto Buldas and Raul Kaidro
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

08.05.2023

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 - Common JavaScript Module

Listing 1. common.js

```
export const KeyParams = {name: 'HMAC', hash: 'SHA-512'};
const encoder = new TextEncoder();

export function toBase64url(bin) {
  return btoa(String.fromCharCode(...bin))
    .replace(/\+/g,'-')
    .replace(/\//g,'_')
    .replace(/\=+$/m,'');
}

export function strToUint8(str) {
  return encoder.encode(str)
}

export function arrBufToUint8(bin) {
  return new Uint8Array(bin);
}

export function fromBase64Url(data) {
  return atob(data.replace(/-/g, '+').replace(/_/g, '/'));
}

export function fromBase64UrlToBin(data) {
  return Uint8Array.from(fromBase64Url(data), c => c.charCodeAt(0))
}

export function prettyPrint(data) {
  return JSON.stringify(JSON.parse(fromBase64Url(data)), null, 2);
}
```

Appendix 3 - Extension

For extension setup, copy files *manifest.json*, *common.js*, *content.js*, *background.html*, *background.js*, *options.html*, *options.js*, *qrcode.js*, *icon.svg* to an empty folder and load *manifest.json* with Firefox Debug Add-ons option. QR Code JavaScript library *qrcode.js* is available in NPM repository [14] and *icon.svg* is available from Smart-ID site.

Listing 2. manifest.json

```
{
  "manifest_version": 2,
  "name": "Smart-ID Extension",
  "icons": {
    "16": "icon.svg",
    "48": "icon.svg",
    "128": "icon.svg"
  },
  "description": "Smart-ID Extension",
  "version": "1.0",
  "content_scripts": [
    {
      "matches": ["<all_urls>"],
      "js": ["content.js"]
    }
  ],
  "background": { "page": "background.html" },
  "options_ui": { "page": "options.html" },
  "permissions": [ "storage" ],
  "browser_specific_settings": {
    "gecko": {
      "id": "{9b61a3fa-f8dd-11eb-9a03-0242ac130003}"
    }
  }
}
```

Listing 3. content.js

```
window.addEventListener("message", event => {
  console.log("message", event.data);
  browser.runtime.sendMessage(event.data)
    .then(message => {
      var DOMURL = window.URL || window.webkitURL || window;
      var svg = new Image(300, 300);
      svg.src = DOMURL.createObjectURL(new Blob([message], {type: 'image/svg+xml'}));
      const body = document.querySelector('#body');
      body.appendChild(svg);
    })
    .catch(console.log);
});
```

Appendix 4 - Extension Background Script

Listing 4. background.html

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <meta charset="utf-8">
    <script src="qrcode.js"></script>
    <script type="module" src="background.js"></script>
  </head>
</html>
```

Listing 5. background.js

```
import { KeyParams, strToUint8, toBase64url, arrBufToUint8 } from './common.js';

browser.runtime.onInstalled.addListener(_ =>
  browser.runtime.openOptionsPage());

browser.runtime.onMessage.addListener(async (message, sender, _) => {
  try {
    if (sender.url === undefined) {
      return '';
    }
    const jwk = await browser.storage.local.get();
    if (jwk.kid === undefined) {
      return '';
    }
    const key = await crypto.subtle.importKey('jwk', jwk, KeyParams, jwk.ext, jwk.
        key_ops);
    const header = JSON.stringify({
      typ: 'JWT',
      alg: jwk.alg,
      kid: jwk.kid
    });
    const body = JSON.stringify({
      sub: sender.url,
      jti: message
    });
    const data = toBase64url(strToUint8(header)) + '.' + toBase64url(strToUint8(body));
    const signature = await crypto.subtle.sign('HMAC', key, strToUint8(data));
    const jwt = data + '.' + toBase64url(arrBufToUint8(signature));
    return await QRCode.toString(jwt, {errorCorrectionLevel: 'L', type: 'svg'});
  } catch(error) {
    console.error(error);
    return '';
  }
});
```

Appendix 5 - Extension Options Page

Listing 6. options.html

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <meta charset="utf-8">
    <script src="qrcode.js"></script>
  </head>
  <body>
    <div>
      <div id="message"></div>
      <button type="button" id="generate">Pair with phone</button>
      <canvas id="canvas"></canvas>
    </div>
    <script type="module" src="options.js"></script>
  </body>
</html>
```

Listing 7. options.js

```
import { KeyParams, strToUint8, toBase64url, arrBufToUint8 } from './common.js';

function showKey(jwk) {
  document.querySelector('#message')
    .innerHTML = jwk.kid ? 'Paired key "' + jwk.kid + '"' : 'To use this extension
        please pair with Smart-ID mobile app.';
}

document.querySelector('#generate').addEventListener("click", async _ => {
  try {
    const key = await crypto.subtle.generateKey(KeyParams, true, ['sign', 'verify']);
    let jwk = await crypto.subtle.exportKey('jwk', key);
    const digest = await crypto.subtle.digest('SHA-256', strToUint8(jwk.k));
    jwk['kid'] = toBase64url(arrBufToUint8(digest));
    await browser.storage.local.set(jwk);
    const canvas = document.querySelector('#canvas');
    QRCode.toCanvas(canvas, JSON.stringify(jwk), { errorCorrectionLevel: 'L' });
    showKey(jwk);
  } catch(error) {
    console.error(error);
  }
});
browser.storage.local.get()
  .then(showKey)
  .catch(console.error);
```

Appendix 6 - Smart-ID Proof of Concept Project

For project setup, copy *build.gradle* file to empty folder, file *SmartIDExtensionApp.java* into *src/main/java/eu/skidsolutions* folder and files *common.js*, *index.html*, *index.js*, *bank.js*, *bank.html*, *bnak.html*, *html5-qrcode.js* into folder src/main/resources/static. QR Code scanning JavaScript library is available in NPM repository [21]. Execute *gradle* [2] tool with parameters *init* and *war* to build Web-servlet. Servlet can be executed with command *bootRun*.

Listing 8. build.gradle

```
plugins {
    id 'org.springframework.boot' version '2.7.10'
    id 'io.spring.dependency-management' version '1.1.0'
    id 'java'
    id 'war'
}

group 'eu.skidsolutions'
version '1.0-SNAPSHOT'
sourceCompatibility = JavaLanguageVersion.of(11)

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'ee.sk.smartid:smart-id-java-client:2.1.1'
    implementation 'org.glassfish.jersey.inject:jersey-hk2:2.26'
    compileOnly 'org.projectlombok:lombok:1.18.20'
    annotationProcessor 'org.projectlombok:lombok:1.18.20'
    providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'
    developmentOnly 'org.springframework.boot:spring-boot-devtools'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

[2] https://gradle.org

Appendix 7 - Smart-ID Web Application

Listing 9. SmartIDExtensionApp.java

```
package eu.skidsolutions;

import com.fasterxml.jackson.databind.ObjectMapper;
import ee.sk.smartid.*;
import ee.sk.smartid.rest.dao.*;
import lombok.*;
import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.*;

import java.util.*;

@SpringBootApplication
@Slf4j
@RestController
public class SmartIDExtensionApp {
    public static void main(String[] args) {
        SpringApplication.run(SmartIDExtensionApp.class, args);
    }

    private static final Base64.Encoder b64Url =
            Base64.getUrlEncoder().withoutPadding();
    private static final SmartIdClient client = new SmartIdClient();
    private static final ObjectMapper objectMapper = new ObjectMapper();
    private static final Map<String,String> loggedIn = new HashMap<>();
    // https://github.com/SK-EID/smart-id-java-client/wiki/SSL-configuration
    private static final String CRT = "-----BEGIN CERTIFICATE-----\n"
            + "..."
            + "-----END CERTIFICATE-----";

    static {
        try {
            client.setRelyingPartyUUID("00000000-0000-0000-0000-000000000000");
            client.setRelyingPartyName("DEMO");
            client.setHostUrl("https://sid.demo.sk.ee/smart-id-rp/v2/");
            client.setTrustedCertificates(CRT);
        } catch (Exception e) {
            log.error("SmartID init failed: {}", e.getMessage());
        }
    }

    @PostMapping("/auth")
    public String auth(@RequestParam String personalCode, @RequestBody JWTBody body)
            throws Exception {
        // Workaround to get auth certificate
        SemanticsIdentifier semanticsIdentifier = new SemanticsIdentifier(
                SemanticsIdentifier.IdentityType.PNO,
                SemanticsIdentifier.CountryCode.EE,
                personalCode);
```

```
        SmartIdAuthenticationResponse certResponse = client.createAuthentication()
                .withSemanticsIdentifier(semanticsIdentifier)
                .withAuthenticationHash(AuthenticationHash.generateRandomHash())
                .withCertificateLevel("QUALIFIED")
                .withAllowedInteractionsOrder(Collections.singletonList(
                        Interaction.displayTextAndPIN("Log␣in␣to␣self-service?")))
                .authenticate();

        JWTHeader jwtHeader = new JWTHeader("JWT", "RS256", new String[] {
                b64Url.encodeToString(certResponse.getCertificate().getEncoded()) });
        String jwtHeaderString = objectMapper.writeValueAsString(jwtHeader);
        String jwtBodyString = objectMapper.writeValueAsString(body);
        String dataToSign = b64Url.encodeToString(jwtHeaderString.getBytes())
                + "." + b64Url.encodeToString(jwtBodyString.getBytes());

        AuthenticationHash signHash = new AuthenticationHash();
        signHash.setHash(DigestCalculator.calculateDigest(
                dataToSign.getBytes(), HashType.SHA256));
        signHash.setHashType(HashType.SHA256);

        SmartIdAuthenticationResponse signResponse = client.createAuthentication()
                .withDocumentNumber(certResponse.getDocumentNumber())
                .withAuthenticationHash(signHash)
                .withCertificateLevel("QUALIFIED")
                .withAllowedInteractionsOrder(Collections.singletonList(
                        Interaction.displayTextAndPIN("Log␣in␣to␣self-service?")))
                .authenticate();

        String jwt = dataToSign + "."
                + b64Url.encodeToString(signResponse.getSignatureValue());
        loggedIn.put(body.jti, body.sub);
        return jwt;
    }

    @GetMapping("/status")
    public String status(@RequestParam String rng) {
        String sub = loggedIn.get(rng);
        if (sub == null)
            return "PENDING";
        if ("http://localhost:8080/bank.html".equals(sub))
            return "Successfully␣authenticated";
        return "URL␣mismatch";
    }

    @Data
    @AllArgsConstructor
    static class JWTHeader {
        private final String typ;
        private final String alg;
        private final String[] x5c;
    }

    @Data
    @NoArgsConstructor(force = true)
    static class JWTBody {
        private final String jti;
        private final String sub;
    }
}
```

Appendix 8 - Smart-ID Application

Listing 10. index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Smart-ID</title>
    <meta charset="utf-8">
    <script src="html5-qrcode.js"></script>
</head>
<body>
Personal code: <input type="text" id="personalCode">
<div id="reader" width="600px"></div>
JWK: <pre id="key"></pre>
JWT: <pre id="result"></pre>
MSG: <pre id="msg"></pre>
<script type="module" src="index.js"></script>
</body>
</html>
```

Listing 11. index.js

```
import { KeyParams, strToUint8, toBase64url, arrBufToUint8,
    fromBase64Url, fromBase64UrlToBin, prettyPrint } from './common.js';
let importedKey = {};
new Html5QrcodeScanner("reader", { fps: 10, qrbox: 250 }, false)
  .render(async (decodedText, decodedResult) => {
    try {
      const jwk = JSON.parse(decodedText);
      importedKey = await crypto.subtle.importKey('jwk', jwk, KeyParams, jwk.ext, jwk.
          key_ops);
      document.querySelector('#key').innerHTML = JSON.stringify(jwk, null, 2);
    } catch(error) {
      // else Not JWK
      const jwt = decodedText.split('.');
      const data = jwt[0] + '.' + jwt[1];
      const isValid = await crypto.subtle.verify('HMAC',
        importedKey, fromBase64UrlToBin(jwt[2]), strToUint8(data));
      document.querySelector('#result').innerHTML = prettyPrint(jwt[0])
        + '\n.\n' + prettyPrint(jwt[1]) + '\n.\n'  + jwt[2] + '\n'
        + (isValid ? "Valid" : "Invalid");
      const personalCode = document.querySelector('#personalCode').value;
      fetch('/auth?personalCode=' + personalCode, {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: fromBase64Url(jwt[1])
      })
        .then(response => response.text())
        .then(text => document.querySelector('#msg').innerHTML = text)
        .catch(console.warn);
    }
  }, console.warn);
```

Appendix 9 - Relying Party

Listing 12. bank.js

```
var rng = crypto.randomUUID();
var pollSmartId = () => fetch('/status?rng=' + rng)
    .then(response => response.text())
    .then(data => {
      if (data == "PENDING") {
        setTimeout(pollSmartId, 3000);
      } else {
        document.querySelector('#msg').innerHTML = data;
      }
    })
    .catch(console.warn);

document.querySelector('#auth').addEventListener("click", _ => {
    window.postMessage(rng, '*');
    pollSmartId();
});
```

Listing 13. bank.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Bank.eu</title>
  <meta charset="utf-8">
  <link rel="stylesheet" crossorigin="anonymous"
    href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css"
    integrity="sha384-rbsA2VBKQhggwzxH7pPCaAqO46MgnOM80zW1RWuH61DGLwZJEdK2Kadq2F9CUG65">
</head>
<body>
<div class="text-center" id="body">
    <h1>Bank.eu</h1>
    <button type="button" class="btn␣btn-primary" id="auth">Authenticate</button>
    <pre id="msg"></pre>
</div>
<script type="module" src="bank.js"></script>
</body>
</html>
```

Listing 14. bnak.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Bnak.eu</title>
  <meta charset="utf-8">
  <link rel="stylesheet" crossorigin="anonymous"
    href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css"
    integrity="sha384-rbsA2VBKQhggwzxH7pPCaAqO46MgnOM80zW1RWuH61DGLwZJEdK2Kadq2F9CUG65">
</head>
<body>
<div class="text-center" id="body">
    <h1>Bnak.eu</h1>
    <button type="button" class="btn␣btn-primary" id="auth">Authenticate</button>
    <pre id="msg"></pre>
</div>
<script type="module" src="bank.js"></script>
</body>
</html>
```