UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Silver Maala

Adding eMRTD authentication support to the Web eID project

Master's Thesis (30 ECTS)

Supervisor: Arnis Paršovs, PhD

Tartu 2023

Adding eMRTD authentication support to the Web eID project

Abstract:
In 2021 Estonia started issuing ID cards that contain the Electronic Machine Readable Travel Document (eMRTD) applet. This paper aims to improve the Web eID project by adding support for eMRTD authentication. The eMRTD applet stores the document owner's biometric and personal information and the data stored on the applet can be cryptographically verified. The authentication method proposed in this paper would enable a web application to authenticate the user over the web using the eMRTD applet. The eMRTD authentication method proposed in this thesis has the benefit of not needing the user to enter a PIN. In this paper we give an overview of eMRTD, its files and security features and introduce the Web eID project. The outcome of this paper are forks of the Web eID projects that support authenticating using eMRTD.

Keywords:
Electronic Machine Readable Travel Document (eMRTDs), Web eID, remote attestation, ID card

CERCS: P170 – Computer science, numerical analysis, systems, control

eMRTD autentimise toe lisamine Web eID projekti

Lühikokkuvõte:

Eesti hakkas väljastama ID kaarte mille peal on Elektrooniliste Masinloetavate Reisidokumentide (eMRTD) applet aastal 2021. Selle lõputöö eesmärk on täiustada Web eID projekti sinna lisades eMRTD autentimise võimaluse. eMRTD applet sisaldab kaardi omaniku biomeetrilisi ja isiklikke andmeid ja neid andmeid saab krütograafiliselt kontrollida, et neid pole muudetud. Selles töös loodud autentimise meetod annab võimaluse veebirakendusel autentida kasutajat üle võrgu kasutades selleks ID kaardil olevat eMRTD appleti. Selle meetodi eelis on, et kasutaja ei pea sisestama enda PIN koodi. Selles töös anname ülevaata eMRTD-st, selles olevates failidest ja turvameetmetest, ja Web eID projektist. Selle töö tulemus on Web eID projekti Git harud, milles on toetatud autentimine eMRTD-ga.

Võtmesõnad:

Elektroonilised Masinloetav Reisidokument, Web eID, internetis autentimine, ID kaart

CERCS: P170 – Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1 Introduction

From 1st of August 2021 Estonia started issuing ID cards that contain the Electronic Machine Readable Travel Document (eMRTD) applet in addition to the eID applet [1].

The aim of this thesis is to add eMRTD authentication support to the Web eID project. The Web eID project is used for secure authentication and signing on the web using digital documents [2].

The authentication mechanism proposed in this thesis allows a user to authenticate themselves without having to enter their PIN, which improves the user experience. Our proposed authentication mechanism uses the eMRTD applet and the user's facial image for authentication instead of the eID applet and PIN.

To achieve our goal we designed a new eMRTD authentication token that is created by the client and is used by the web application to authenticate the user. The authentication token contains the signature created with the private key stored in the eMRTD applet and the following files from the applet: person's facial image, public key that is used for the signature verification, machine-readable zone (MRZ) and the Document Security Object (SO_D) [3]. The web application uses the SO_D to verify the files from the eMRTD applet and the public key to verify the signature [4]. If the signature and the files are successfully validated then the web application knows the card used by the user is valid and the MRZ and facial image have been issued by the State.

After doing the token validation the web application can do further authentication of the user using the facial image included in the token. For example, a human or machine can perform facial recognition using a video feed to compare the facial image in the token with the face of the person in the video feed. This step allows a web application to make sure that the card belongs to the person presenting it. If the user's face and facial image from the eMRTD applet match then the web server can be sure of the user's identity.

Figure 1 shows the high-level overview of how our solution works.

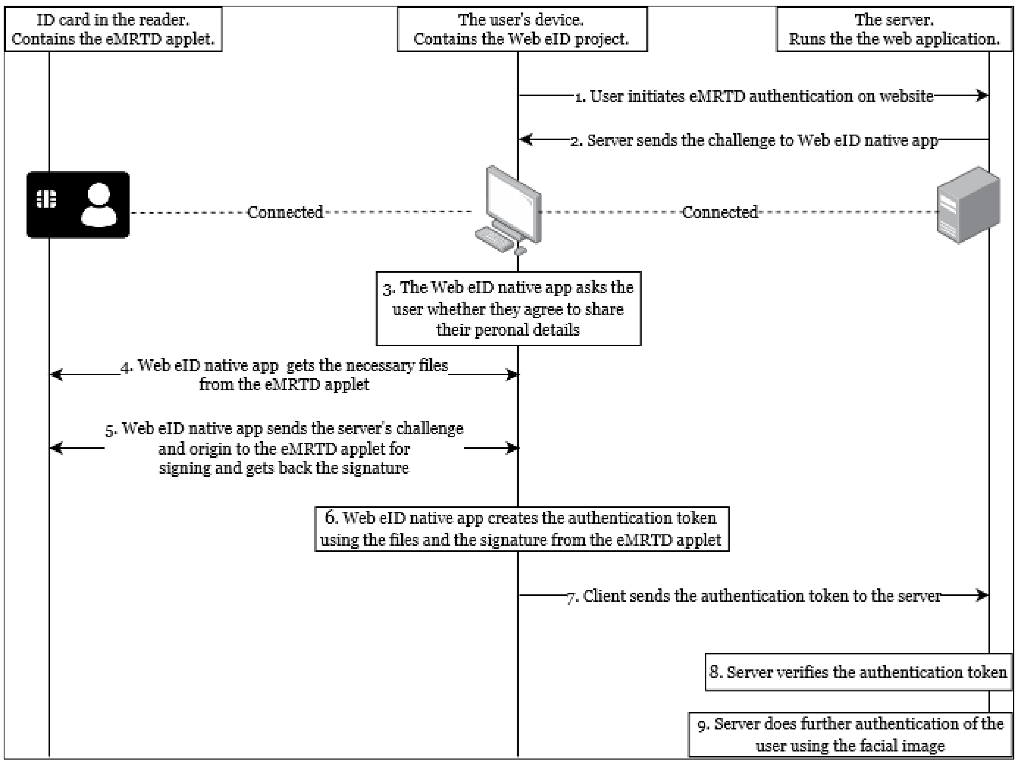

ID card in the reader.
Contains the eMRTD applet.
The user's device.
Contains the Web eID project.
The server.
Runs the the web application.
1. User initiates eMRTD authentication on website
2. Server sends the challenge to Web eID native app
Connected
Connected
3. The Web eID native app asks the user whether they agree to share their peronal details
4. Web eID native app gets the necessary files from the eMRTD applet
5. Web eID native app sends the server's challenge and origin to the eMRTD applet for signing and gets back the signature
6. Web eID native app creates the authentication token using the files and the signature from the eMRTD applet
7. Client sends the authentication token to the server
8. Server verifies the authentication token
9. Server does further authentication of the user using the facial image

Figure 1. High-level overview of our solution

This thesis is structured as follows. Background on eMRTD and Web eID is needed to understand our final implementation so Section 2 gives an overview of eMRTD, its files and security features. After that Section 3 introduces the Web eID project and its architecture. Section 4 describes our solution, how the new eMRTD authentication logic works and how we added eMRTD authentication to Web eID project. After that in Section 5 we give an overview of future work and features still needing implementing in our solution. Finally in Section 6 we give a summary of the thesis. In the Appendix we also give instructions on how to run our solution.

1.1 Related Work

This section talks about papers that were helpful with this thesis.

The thesis “Use of Electronic Identity Documents for Multi-Factor Authentication” strongly relates to the topic of our thesis [5]. The result of that thesis was an automated biometric authentication system called “eMRTD Face Access” that used the eMRTD applet on ID cards to authenticate the user.

The main difference between our solution and “eMRTD Face Access” is that the latter is a standalone application written in Python that authenticates the user locally while the former is implemented in the Web eID project using C++ and it authenticates the user over the web.

The Python code from “eMRTD Face Access” was helpful in our development process [6]. Using the Python project we were able to create test vectors to test the correctness of our C++ eMRTD cryptographic protocol implementations which sped up the development. Also it was helpful to see some of the eMRTD cryptographic protocols described in the eMRTD specifications as code instead of text.

2 Electronic Machine Readable Travel Document

A Machine Readable Travel Document (MRTD) is an official travel document issued by a State or organization that is used for travel. MRTD is designed to allow for visual and mechanical reading of the data printed on it. An example of a MRTD is a machine readable passport [7].

An Electronic Machine Readable Travel Document (eMRTD) is a MRTD which also contains a contactless integrated circuit chip with an antenna. The chip stores the document owner's biometric and personal information. eMRTD enables for automated verification of the document's authenticity and the person's identity. This allows for faster and more efficient processing at border control checkpoints and airports [7].

The specification of MRTD and eMRTD is governed by the International Civil Aviation Organization (ICAO). ICAO provides documents that set the technical specifications and standards for eMRTD and MRTD [7].

Figure 2 shows the eMRTD logo. This logo is usually printed on the document if the chip contains the eMRTD applet [8]. In the case of Estonian ID cards the logo is printed at the top left of the residence permit card shown in Figure 4 but for some reason is omitted on the regular ID card seen in Figure 3.

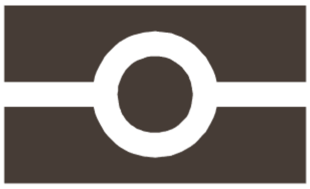
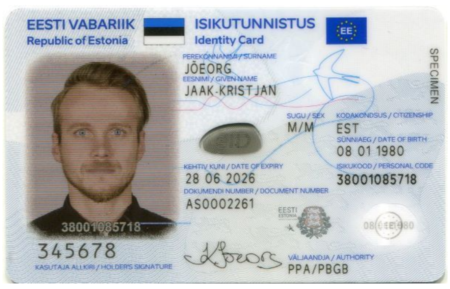
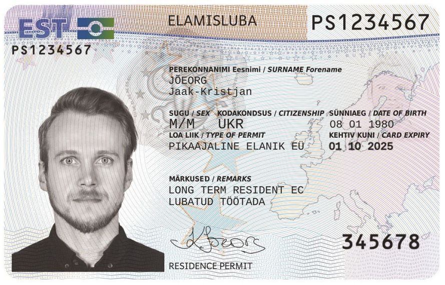


Figure 2. ICAO "chip inside" logo [8]



EESTI VABARIIK
Republic of Estonia
ISIKUTUNNISTUS
Identity Card
PEREKONNANIMI / SURNAME
JÕEORG
EESNIMI / GIVEN NAME
JAAK-KRISTJAN
SUGU / SEX
M/M
KODAKONDSUS / CITIZENSHIP
EST
SÜNNIAEG / DATE OF BIRTH
08 01 1980
KEHTIV KUNI / DATE OF EXPIRY
28 06 2026
ISIKUKOOD / PERSONAL CODE
38001085718
DOKUMENDI NUMBER / DOCUMENT NUMBER
AS0002261
38001085718
345678
KASUTAJA ALLKIRI / HOLDER'S SIGNATURE
VÄLJAANDJA / AUTHORITY
PPA/PBGB
SPECIMEN

Figure 3. Example of an Estonian ID card front [9]



EST
ELAMISLUBA
PS1234567
PS1234567
PEREKONNANIMI Eesnimi / SURNAME Forename
JÕEORG
Jaak-Kristjan
SUGU / SEX
M/M
KODAKONDSUS / CITIZENSHIP
UKR
SÜNNIAEG / DATE OF BIRTH
08 01 1980
LOA LIIK / TYPE OF PERMIT
PIKAAJALINE ELANIK EÜ
KEHTIV KUNI / CARD EXPIRY
01 10 2025
MÄRKUSED / REMARKS
LONG TERM RESIDENT EC
LUBATUD TÖÖTADA
345678
RESIDENCE PERMIT

Figure 4. Example of an Estonian residence permit card front [9]

The eMRTD specification documents specify the mandatory, optional and conditional files on the eMRTD applet [3], as well as different security mechanisms supported by the eMRTD applet [4]. The main security mechanisms the eMRTD applet can have are the following. Active Authentication is a security mechanism that gives assurance that the chip is valid by having the chip sign a challenge provided to it. The signature is created using a private key that is stored in the chip's secure memory and can only be accessed by the chip [4]. Passive Authentication is another security mechanism. Passive Authentication uses something called a Document Security Object (SO_D) to verify that the files on the eMRTD chip have not been modified [4]. The SO_D is a digitally signed file that contains the hashes of files in the eMRTD applet [4]. There are also Access Control mechanisms. An Access Control mechanism prevents unauthorized access to the data on the chip. An Access Control mechanism also encrypts the messages sent between the chip and the system that is accessing the eMRTD applet. The two supported Access Control mechanisms are Basic Access Control (BAC) and Password Authenticated Connection Establishment (PACE) [4].

In this section, we provide an overview of eMRTD technical specifications, including the data stored on the chip and the security features employed. Since our implementation uses the eMRTD applet for authenticating this background on eMRTD is important for understanding our solution described in Section 4.

2.1 Logical Data Structure

The "eMRTD specification Part 10: Logical Data Structure (LDS) for Storage of Biometrics and Other Data in the Contactless Integrated Circuit (IC)" [3] specifies the Logical Data Structure (LDS). The LDS specifies how data should be stored and formatted on the chip.

There are two versions of LDS: LDS1 and LDS2. LDS1 is the mandatory eMRTD application while LDS2 is an optional extension to LDS1 that offers additional functionality. LDS2 offers storage of travel information after document issuance (electronic visas, travel records) and additional biometrics. Figure 5 shows the high-level overview of the eMRTD applet file structure. In Figure 5 LDS1 is the leftmost box at the bottom row while all the other boxes at the bottom row belong to LDS2.

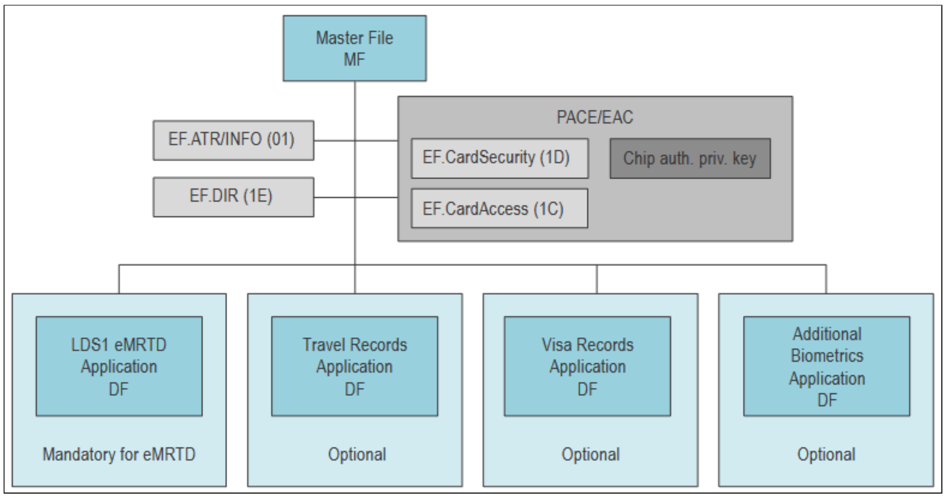


Figure 5. LDS1 and LDS2 applications of the chip [3]

2.2 File Structure

The "eMRTD specification Part 10: Logical Data Structure (LDS) for Storage of Biometrics and Other Data in the Contactless Integrated Circuit (IC)" [3] describes the mandatory, conditional and optional files on the chip.

The LDS stores a file on the chip as a Data Group (DG) which in turn is a logical grouping of related Data Elements. All files are stored as ASN.1 objects encoded with Distinguished Encoding Rule (DER). DER encodes data as tag-length-value data objects.

In this section we list all the files specified in the eMRTD specification Part 10 [3] and give a brief explanation for each file. In addition to the files described later in this section, there are also files `EF.DG6` - `EF.DG10` that are reserved for future use and are available for temporary proprietary usage until then.

2.2.1 Mandatory files

The "eMRTD specification Part 10 Logical Data Structure (LDS) for Storage of Biometrics and Other Data in the Contactless Integrated Circuit (IC)" [3] specifies files that are mandatory on the eMRTD applet:

- `EF.COM`: This file contains the Logical Data Structure (LDS) version information, Unicode version information and a list of the Data Groups that are present on the eMRTD applet. The Unicode version is used to identify the coding method for the characters. More details about this file can be found in the eMRTD specification Part 10 section 4.6.1 [3].

- `EF.SOD`: This file contains the Document Security Object (SO_D) that stores hash values of the DG files that are present on the eMRTD applet. These stored hash values are used to verify the integrity of the DG files. The SO_D is digitally signed by the issuing State when the document is issued and the signature is stored in the SO_D. The SO_D is implemented as a `SignedData` type which is specified in RFC-3369 [10]. The SO_D contains the Document Signer Certificate (CDS) and additional certificates in the certificate chain. The CDS is used to verify the SO_D signature. More details about the `EF.SOD` can be found in the eMRTD specification Part 10 section 4.6.2 and appendix D [3].

- `EF.DG1`: This file contains the machine-readable zone (MRZ). The MRZ stored in this file is the exact MRZ string that is printed on the back of the ID card. The Estonia ID card uses TD1 MRZ format. The TD1 format is specified in the eMRTD specification Part 5 [11]. The TD1 format construction is shown in Figure 7 and an example of TD1 usage on the Estonian ID card backside is shown in Figure 6. More details about this file can be found in the eMRTD specification Part 10 section 4.7.1 [3].

- `EF.DG2`: This file contains the document holder's facial image. This facial image is meant to be used as an input to a facial recognition system. The `EF.DG2` file contents use Biometric Information Template (BIT) group template with nested BITs specified in ISO/IEC 7816-11. The `EF.DG2` file can hold multiple entries. If there are multiple entries then the first one is the latest one. Each BIT (facial image) has extra metadata about it. For example, format owner, format type, validity period, etc. Some of these metadata entries are optional, some are required. The image itself is encoded as JFIF or JPEG 2000. ICAO also has additional recommendations for facial images, like, focus and depth of field of the camera that takes the photo, eye distance in pixels, photo background, etc. [12]. ICAO also specifies that the optimum size for the compressed facial image is in the 15kB – 20kB range [13]. This allows the photo to as little space as possible while offering good facial recognition performance. More details about this file can be found in the eMRTD specification Part 10 section 4.7.2 [3].

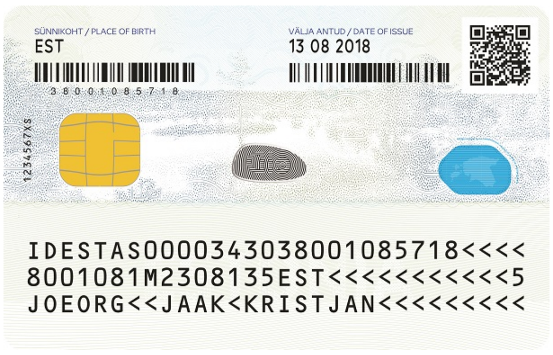

SÜNNIKOHT / PLACE OF BIRTH
EST
38001085718
VÄLJA ANTUD / DATE OF ISSUE
13 08 2018
1234567XS
IDESTAS000034303800108571 8<<<<
8001081M2308135EST<<<<<<<<<<<5
JOEORG<<JAAK<KRISTJAN<<<<<<<<<

Figure 6. The back of the ID card that has the MRZ at the bottom of the card [9]

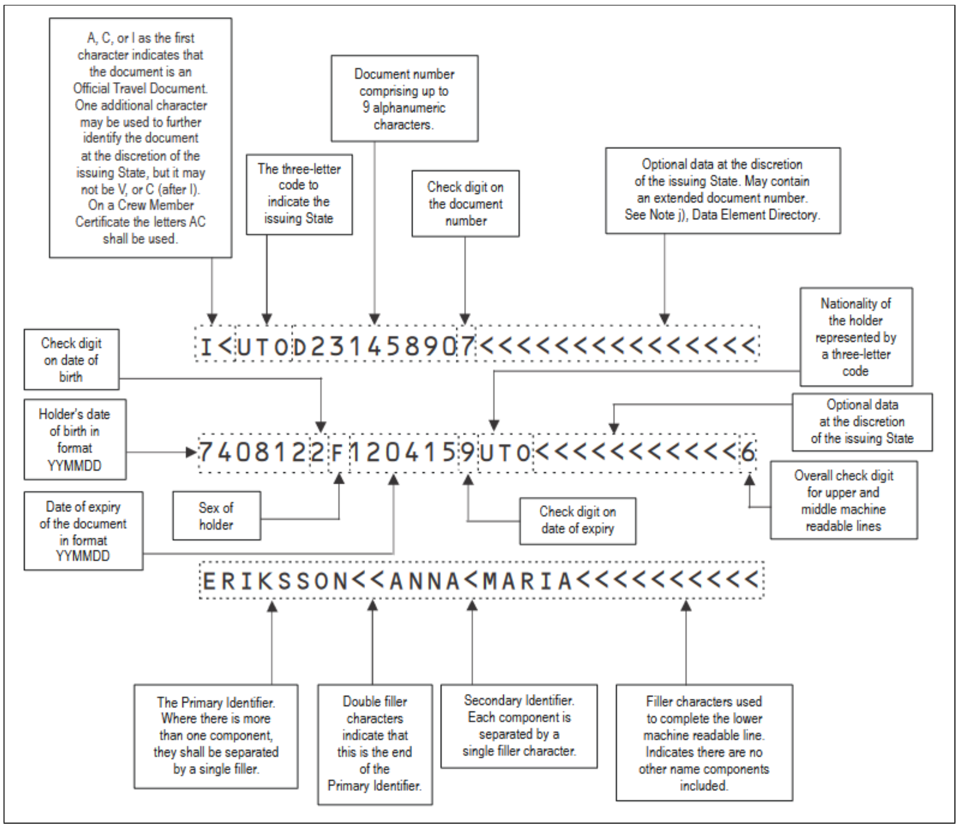

A, C, or I as the first character indicates that the document is an Official Travel Document. One additional character may be used to further identify the document at the discretion of the issuing State, but it may not be V, or C (after I). On a Crew Member Certificate the letters AC shall be used.
The three-letter code to indicate the issuing State
Document number comprising up to 9 alphanumeric characters.
Check digit on the document number
Optional data at the discretion of the issuing State. May contain an extended document number. See Note j), Data Element Directory.
I<UTOD231458907<<<<<<<<<<<<<<<
Check digit on date of birth
Nationality of the holder represented by a three-letter code
Holder's date of birth in format YYMMDD
Optional data at the discretion of the issuing State
7408122F1204159UTO<<<<<<<<<<<6
Date of expiry of the document in format YYMMDD
Sex of holder
Check digit on date of expiry
Overall check digit for upper and middle machine readable lines
ERIKSSON<<ANNA<MARIA<<<<<<<<<<
The Primary Identifier. Where there is more than one component, they shall be separated by a single filler.
Double filler characters indicate that this is the end of the Primary Identifier.
Secondary Identifier. Each component is separated by a single filler character.
Filler characters used to complete the lower machine readable line. Indicates there are no other name components included.

Figure 7. MRZ format TD1 construction [11]

2.2.2 Optional files

The "eMRTD specification Part 10: Logical Data Structure (LDS) for Storage of Biometrics and Other Data in the Contactless Integrated Circuit (IC)" [3] specifies files that are optional on the eMRTD applet:

- `EF.DG3`: This file holds information about the document holder's fingerprints. The `EF.DG3` file contents use Biometric Information Template (BIT) group template with nested BITs specified in ISO/IEC 7816-11. The fingerprints can be used as an additional authentication method. More details about this file can be found in the eMRTD specification Part 10 section 4.7.3 [3].
- `EF.DG4`: This file holds information about the document holder's irises. The `EF.DG4` file contents use Biometric Information Template (BIT) group template with nested BITs specified in ISO/IEC 7816-11. The iris can be used as an additional authentication method. More details about this file can be found in the eMRTD specification Part 10 section 4.7.4 [3].
- `EF.DG5`: This file holds the displayed portrait. More details about this file can be found in the eMRTD specification Part 10 section 4.7.5 [3].
- `EF.DG11`: This file is used to hold additional personal details about the document holder. More details about this file can be found in the eMRTD specification Part 10 section 4.7.11 [3].
- `EF.DG12`: This file is used to hold additional information about the document. More details about this file can be found in the eMRTD specification Part 10 section 4.7.12 [3].
- `EF.DG13`: The issuing State or organization can store arbitrary information into this file. More details about this file can be found in the eMRTD specification Part 10 section 4.7.13 [3].
- `EF.DG16`: This file holds a list of emergency contact information. This file can hold multiple entries. Each entry stores the name, phone number and address. More details about this file can be found in the eMRTD specification Part 10 section 4.7.16 [3].

2.2.3 Conditional files

The "eMRTD specification Part 10: Logical Data Structure (LDS) for Storage of Biometrics and Other Data in the Contactless Integrated Circuit (IC)" [3] specifies files that are conditional and required only in certain cases on the eMRTD applet:

- `EF.DG14`: This file is required when the eMRTD supports the following security mechanisms: Chip Authentication, PACE with Chip Authentication Mapping (PACE-CAM) or Terminal Authentication. This file stores the following SecurityInfos: `ChipAuthenticationInfo` required by Chip Authentication, `ChipAuthenticationPublicKeyInfo` required by PACE-CAM/Chip Authentication and `TerminalAuthenticationInfo` required by Terminal Authentication. This file also contains the `SecurityInfos` from the `EF.CardAccess` file. More details about this file can be found in the eMRTD specification Part 10 section 4.7.14 [3].

- `EF.DG15`: Holds the Active Authentication Public Key and is required when the optional Active Authentication security mechanism is supported. The public key file structure is the `SubjectPublicKeyInfo` that is specified in RFC-5280 [14].

- `EF.CardAccess`: This file is required when the eMRTD supports PACE. The file contains the `PACEInfo` and `PACEDomainParameterInfo` SecurityInfos. More details about this file can be found in the eMRTD specification Part 10 section 3.11.3 [3]. Should be noted that this file is accessible from the Master File instead of LDS1.

- `EF.CardSecurity`: This file is required when the eMRTD supports the following security mechanisms: Chip Authentication, PACE with Chip Authentication Mapping (PACE-CAM) or Terminal Authentication. This file stores the following SecurityInfos: `ChipAuthenticationInfo` required by Chip Authentication, `ChipAuthenticationPublicKeyInfo` required by PACE-CAM/Chip Authentication and `TerminalAuthenticationInfo` required by Terminal Authentication. This file also contains the `SecurityInfos` from the `EF.CardAccess` file. More details about this file can be found in the eMRTD specification Part 10 section 3.11.4 [3]. Should be noted that this file is accessible from the Master File instead of LDS1.

- `EF.ATR/INFO`: This file is required when the LDS2 application is present, otherwise it is optional. This file contains information about the chip itself. For example, it specifies the the maximum number of bytes that can be sent in the command and response APDUs. More details about this file can be found in the eMRTD specification Part 10 section 3.11.1 [3].

- `EF.DIR`: This file must be present when there are more applications present than just the mandatory LDS1 application. This file contains a list of applications that are present on the eMRTD applet. More details about this file can be found in the eMRTD specification Part 10 section 3.11.2 [3]. Should be noted that this file is accessible from the Master File instead of LDS1.

2.3 Security

The “eMRTD specification Part 11: Security Mechanisms for MRTDs” [4] defines multiple security measures for eMRTD. This section gives a basic overview on the most important security measures present.

2.3.1 Passive Authentication

Passive Authentication is specified in the “eMRTD specification Part 11: Security Mechanisms for MRTDs” sections 8.3 [4].

In Passive Authentication the digital signature of the Document Security Object (SO_D) file is verified using the eMRTD PKI. This ensures that the SO_D has not been altered. In turn the verified SO_D can be used to verify the DG files on the chip.

It should be noted that Passive Authentication only verifies the integrity and authenticity of the files. Passive Authentication does not prevent an attacker from using replay attacks or copying the files off the chip.

To use Passive Authentication the system that is accessing the eMRTD applet (inspection system) must have access to the Country Signing Certification Authority (C_{CSCA}) certificates or the Document Signer Certificates (CDS) for each issuing State or organization. The inspection system also needs access to the Certificate Revocation Lists (CRLs).
Passive Authentication is done as follows:

1. The inspection system reads the Document Security Object (SO_D) file which must contain the Document Signer Certificate (CDS).

2. The inspection system extracts the CDS from the SO_D.

3. The inspection system builds and validates the certification path from a Trust Anchor to the CDS. The trust anchor is the Country Signing Certification Authority (C_{CSCA}). Each State has their own C_{CSCA} that issues certificates for the eMRTD PKI [15].

4. The inspection system verifies the SO_D signature using the Document Signer Public Key from the verified Document Signer Certificate (CDS).

5. The inspection system reads the DG files listed in the SO_D from the eMRTD.

6. The inspection system hashes these read DG files. Then compare these hashes to the corresponding hashes in the SO_D. If these hashes match then this proves that these DG files have not been modified.

2.3.2 Active Authentication

Active Authentication is specified in the "eMRTD specification Part 11: Security Mechanisms for MRTDs" section 6.1 and 8.4 [4].

Active Authentication works by having the card sign a challenge with the private key ($\mathrm{KPr}_{\mathrm{AA}}$) that is on the card. If the signed challenge can be verified with the corresponding public key ($\mathrm{KPu}_{\mathrm{AA}}$) then the card is not cloned, since the private key $\mathrm{KPr}_{\mathrm{AA}}$ is stored in the card's secure memory.

Active Authentication is supported when the `EF.DG15` file is present. If this file is present then Active Authentication is supported and the card holds the Active Authentication key pair ($\mathrm{KPr}_{\mathrm{AA}}$ and $\mathrm{KPu}_{\mathrm{AA}}$). `EF.DG15` holds the Active Authentication public key ($\mathrm{KPu}_{\mathrm{AA}}$). The private key ($\mathrm{KPr}_{\mathrm{AA}}$) is held in the card's secure memory and can only be used internally.

Active Authentication can use RSA or ECDSA. Both support only `SHA-224`, `SHA-256`, `SHA-384` or `SHA-512` hashing algorithms.

If ECDSA based signature algorithm is used then the `ActiveAuthenticationInfo` SecurityInfo is present in the `EF.DG14` file. The `ActiveAuthenticationInfo` SecurityInfo holds the `signatureAlgorithm`.

Should be noted that Active Authentication requires the challenge length to be 8 bytes long. In our solution this challenge length limitation is not ideal, since we use Active Authentication to sign a hash. Even the shortest supported hash algorithm (`SHA-224`) output is 28 bytes long.

2.3.3 Chip Authentication

Chip Authentication is specified in the "eMRTD specification Part 11: Security Mechanisms for MRTDs" section 6.2 [7].

Chip Authentication works by using the Diffie-Hellman key agreement protocol to create strong session keys for secure communication and authentication. This also ensures that the chip has not been cloned - just like Active Authentication.

Main difference between Chip Authentication and Active Authentication is that Active Authentication only provides authentication while Chip Authentication also provides strong session keys.

Chip Authentication support is indicated with the presence of the `ChipAuthenticationPublicKeyInfo` SecurityInfo in the `EF.DG14` file.

The Chip Authentication Key Pair is stored in the chip. The private key is stored in the chip's secure memory and the public key is stored in the `EF.DG14` file in the `ChipAuthenticationPublicKeyInfo` SecurityInfo.

2.3.4 Terminal Authentication

Terminal Authentication is specified in the “eMRTD specification Part 11: Security Mechanisms for MRTDs” section 7.1 [7].

Terminal Authentication authenticates the inspection system to make sure the inspection system is permitted to access the sensitive data. The goal is to protect sensitive data on the chip, for example, biometric data. States use a PKI called Authorization PKI [15] to manage the foreign States that are allowed to access the sensitive data.

In the context of authenticating over the web, a local application installed on the user’s device will not have access to a valid certificate in the Authorization PKI due to security issues. The certificate is not going to be protected. Plus, sending biometric data to a website is obviously a bad idea. So Terminal Authentication is not used in our solution.

2.3.5 Access Control

Usually eMRTD communication between the inspection system and the chip is contactless using near-field communication (NFC). To prevent unauthorized access to the data on the chip the chip is protected by Access Control mechanisms. An Access Control mechanism allows access to the chip only if the inspection system can provide “proof of authorization”. This “proof of authorization” is data printed on the card, for example, the MRZ [16]. An Access Control mechanism also provides secure messaging by encrypting data sent between the chip and the inspection system.

The “eMRTD specification Part 11: Security Mechanisms for MRTDs” defines two Access Control protocols: Basic Access Control (BAC) and Password Authenticated Connection Establishment (PACE) [4]. BAC and PACE are used to derive the keys used to encrypt/decrypt messages between the chip and inspection system. The message encryption can be done using `3DES` or `AES` depending which is supported by the chip.

2.3.5.1 Basic Access Control

Basic Access Control (BAC) is specified in the “eMRTD specification Part 11: Security Mechanisms for MRTDs” section 4.3 [4].

BAC uses the document number, date of birth and expiration date to negotiate the session keys which will enable secure communication with the card. The session keys (Document Basic Access Keys) K_{Enc} and K_{MAC} are used to encrypt/decrypt and authenticate the communication between the chip and the inspection system. It should be noted that these are symmetric keys.

BAC is a popular method due to its simplicity, but it has a drawback. The data from which the keys are derived have low entropy and is cryptographically weak. In the paper “E-passport: Cracking basic access control keys with copacobana” the researchers were able to test 240 million BAC keys per second [17]. This shows that BAC is vulnerable to brute force attacks.

The “eMRTD specification Part 11: Security Mechanisms for MRTDs” also gives a warning that in the future PACE may become the default access control mechanism and BAC will be deprecated [4].

2.3.5.2 Password Authenticated Connection Establishment

Password Authenticated Connection Establishment (PACE) is specified in the “eMRTD specification Part 11: Security Mechanisms for MRTDs” section 4.4 [4].

PACE is a password authenticated Diffie-Hellman key agreement protocol. It has the same purpose as BAC but was was designed to overcome the issues of BAC. PACE uses asymmetric cryptography to establish session keys. The strength of these keys is independent of the entropy of the used password and offers a stronger protection against eavesdropping. PACE uses keys derived from passwords with a key derivation function (KDF).

PACE establishes secure messaging using short and weak passwords. The passwords available for PACE are MRZ (more specifically the document number, the date of birth and the date of expiry from the MRZ) and Card Access Number (CAN). CAN is 6-digit number printed on the document and is random or pseudo-random [18]. CAN has the advantage over MRZ due to it being easier to type in manually. On the Estonian ID card the CAN is printed below the photo. The Estonian ID card with the CAN on it is shown in Figure 3.

When the `EF.CardAccess` file is present then the chip supports PACE.

2.4 Accessing Content On The Chip

To access the files in LDS1 first the eMRTD applet with the AID `A00000024710FF` needs to be selected [19]. After that the LDS1 sub-applet with the AID `A0000002471001` can be selected [3]. The LDS1 sub-applet is the "LDS1 eMRTD Application DF" in Figure 5.

Once the eMRTD application is selected and the session keys are negotiated the inspection system can start accessing the files on the chip. The inspection system can read files by sending the encrypted `SELECT` and `READ BINARY` APDU commands. There are also other commands as well, for example `INTERNAL AUTHENTICATE` to use Active Authentication [4].

Details on some of the commands supported by the eMRTD applet can be found in “eMRTD specification Part 10 Logical Data Structure (LDS) for Storage of Biometrics and Other Data in the Contactless Integrated Circuit (IC)” [3] section 3.5.

3 Web eID

The Web eID project is used for secure authentication and signing on the web using digital documents (ID-card, digital ID, e-Resident's digital ID, residence permit card, etc.). Web eID supports Estonian, Latvian, Lithuanian, Finnish, Croatian and Belgium ID-cards [2].

The Web eID goal is to fix many of the issues that are present with the current ID card authentication/signing solution. These improvements include [2]:

- With the current solution the web browsers use different cryptographic APIs across different operating systems resulting in different UIs for each browsers. With Web eID the UI is shown by a native application and is always the same.

- The current solution caches the PIN1 in the browser as not to prompt a PIN entry for each request. The browser also needs to be closed or restarted after logging out due to caching. In the Web eID project the PIN is no longer cached in the browser.

The Web eID project consists of three main components: the native application [20], the browser extension [21] and the web-eid.js [22] library. The Web eID native application is responsible for communicating with the card [20]. The Web eID webextension is a browser extension available for Chrome, Edge and Firefox that communicates with the Web eID native application [21]. Through the webextension a web application can call the Web eID native application functionality [21]. The web-eid.js library is used by the web application to use the Web eID browser extension [22]. The high level overview of Web eID architecture is shown in Figure 8.

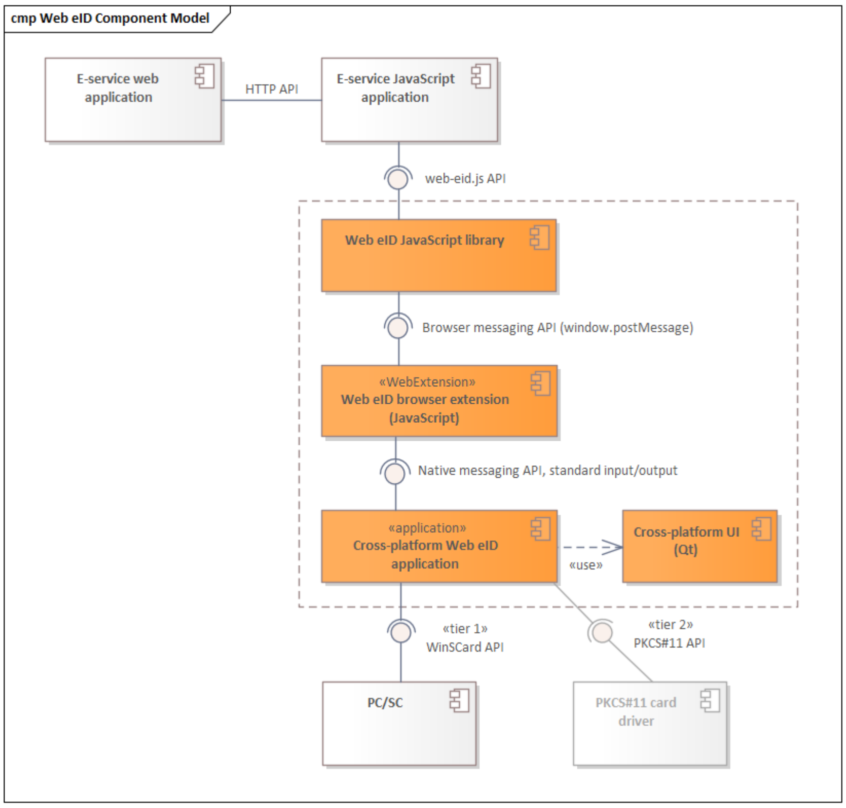

cmp Web eID Component Model
E-service web application
HTTP API
E-service JavaScript application
web-eid.js API
Web eID JavaScript library
Browser messaging API (window.postMessage)
«WebExtension» Web eID browser extension (JavaScript)
Native messaging API, standard input/output
«application» Cross-platform Web eID application
«use»
Cross-platform UI (Qt)
«tier 1» WinSCard API
«tier 2» PKCS#11 API
PC/SC
PKCS#11 card driver

Figure 8. The Web eID architecture [23]

The Web eID team has also created the Web eID webextension mock webapp [24]. The Web eID mock webapp is a web application that uses the web-eid.js library, Web eID webextension and Web eID native application to create the authentication token. The Web eID webextension mock webapp is used for testing the webextension. The Web eID mock webapp only displays the authentication token, the mock webapp does not do any validation on it other than validating the authentication token structure. In our implementation we will be adding eMRTD authentication functionality to this web application and use it to demonstrate eMRTD authentication.

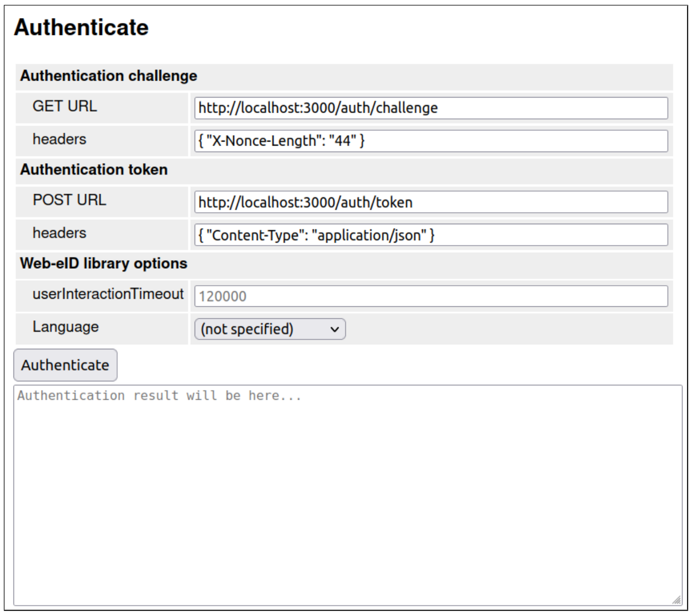
This section provides an overview of the Web eID project. This background is important for understanding our solution described in Section 4 since we add eMRTD authentication support into the Web eID project.

3.1 Usage Example

This section gives an example of authenticating with the ID card using the Web eID project.

Figure 9 shows the authentication option in the Web eID mock webapp [24]. The user presses the “Authenticate” button. The website sends the `authenticate` command and related arguments to the Web eID webextension. The webextension in turn sends it to the Web eID native application. The native application opens the PIN screen shown in Figure 10. The user enters their PIN1 and presses the “Confirm” button. The native application creates the authentication token and returns it to the webextension and the webextension in turn sends it back to the Web eID mock webapp. The authentication token is shown in Figure 11.

In an actual web application the returned token needs to be validated by the server. The Web eID mock webapp only displays it.

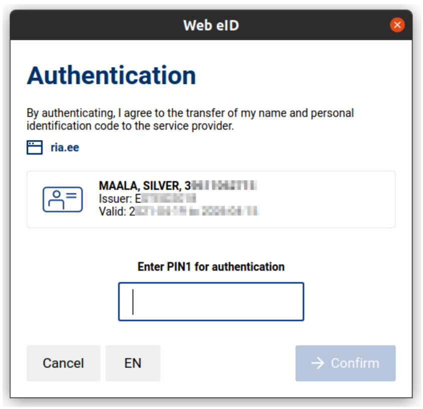


Authenticate

Authentication challenge	
GET URL	http://localhost:3000/auth/challenge
headers	{ "X-Nonce-Length": "44" }
Authentication token	
POST URL	http://localhost:3000/auth/token
headers	{ "Content-Type": "application/json" }
Web-eID library options	
userInteractionTimeout	120000
Language	(not specified)

```
Authentication result will be here...
```

Figure 9. Authenticating option in Web eID mock webapp



Authentication

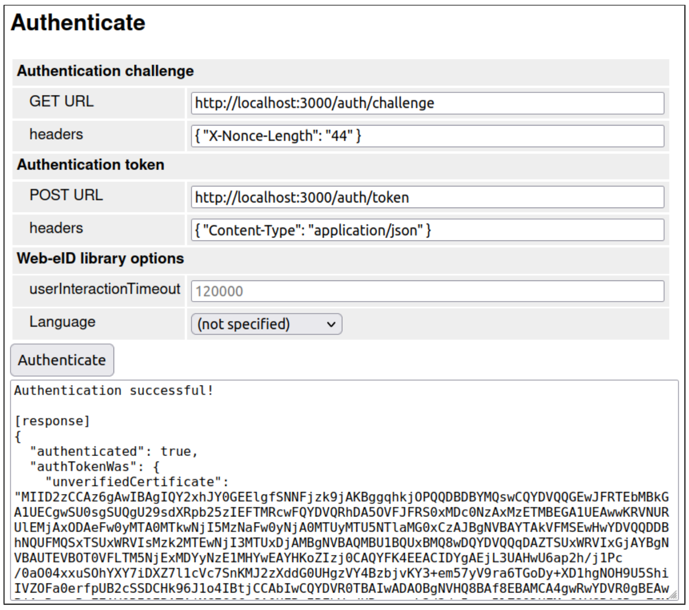
By authenticating, I agree to the transfer of my name and personal identification code to the service provider.

ria.ee

MAALA, SILVER, 3[redacted]
Issuer: E[redacted]
Valid: 2[redacted]

Enter PIN1 for authentication

Figure 10. Authenticating screen in the Web eID project



GET URL: http://localhost:3000/auth/challenge
headers: { "X-Nonce-Length": "44" }
POST URL: http://localhost:3000/auth/token
headers: { "Content-Type": "application/json" }
userInteractionTimeout: 120000
Language: (not specified)

```
Authentication successful!

[response]
{
  "authenticated": true,
  "authTokenWas": {
    "unverifiedCertificate":
"MIID2zCCAz6gAwIBAgIQY2xhJY0GEElgfSNNFjzk9jAKBggqhkjOPQQDBDBYMQswCQYDVQQGEwJFRTEbMBkG
A1UECgwSU0sgSUQgU29sdXRpb25zIEFTMRcwFQYDVQRhDA5OVFJFRS0xMDc0NzAxMzETMBEGA1UEAwwKRVNUR
UlEMjAxODAeFw0yMTA0MTkwNjI5MzNaFw0yNjA0MTUyMTU5NTlaMG0xCzAJBgNVBAYTAkVFMSEwHwYDVQQDDB
hNQUFMQSxTSUxWRVIsMzk2MTEwNjI3MTUxDjAMBgNVBAQMBU1BQUxBMQ8wDQYDVQQqDAZTSUxWRVIxGjAYBgN
VBAUTEVBOT0VFLTM5NjExMDYyNzE1MHYwEAYHKoZIzj0CAQYFK4EEACIDYgAEjL3UAHwU6ap2h/j1Pc
/0aO04xxuSOhYXY7iDXZ7l1cVc7SnKMJ2zXddG0UHgzVY4BzbjvKY3+em57yV9ra6TGoDy+XD1hgNOH9U5Shi
IVZOFa0erfpUB2cSSDCHk96J1o4IBtjCCAbIwCQYDVR0TBAIwADAOBgNVHQ8BAf8EBAMCA4gwRwYDVR0gBEAw
```

Figure 11. Successful authentication in the Web eID mock webapp

3.2 Components

This section provides more details about each of the three components in the Web eID architecture.

3.2.1 Native Application

The Web-eID native application is installed on the user's computer and it performs signing and authentication operations with ID cards. The Web-eID native application is written in C++ and uses the Qt framework to create the UI [20].

The Web eID native applications accept a command and its arguments from the command-line interface (CLI) or from standard input (STDIN). The commands supported are `get-signing-certificate`, `authenticate` and `sign` [20]. Their functions are self explanatory. In our implementation we only have functionality for authenticating the user so in this section we will not look at the `get-signing-certificate` and `sign` commands.

The Web eID native application project's Github defines the arguments for each command [20]:

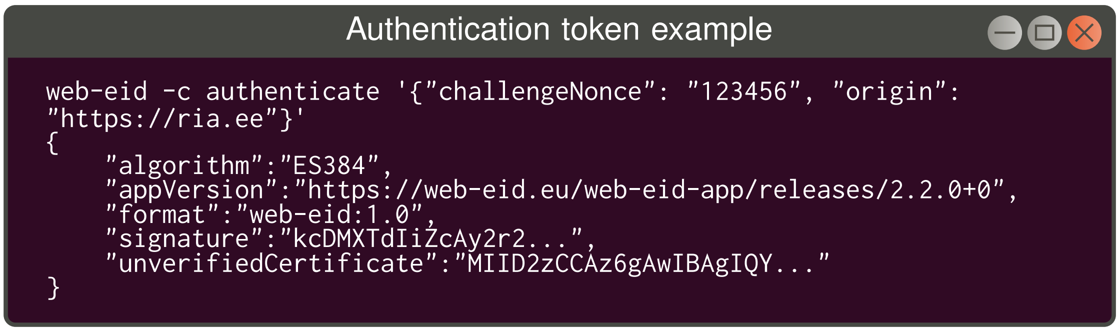
- `origin`: This field is mandatory for all commands. The URL has to be HTTPS and must not end with a slash.
- `challengeNonce`: This is only required for `authenticate` and `sign`. This nonce is signed by the private key on the chip.

When the the native application receives the `authenticate` command and its arguments it opens the UI where the user has to do the steps described in Section 3.1. Once the user enters the required PIN the native application creates the signature from the input with the help of the ID card and then creates the authentication token. When the authentication token is created the native application outputs the token.

The authentication token is a JSON structure containing all the data needed to verify the signature. The idea behind the different parameters is explained in the Web eID authentication token format specification [25]:

- `algorithm`: The signature algorithm used to produce the signature.
- `appVersion`: The URL identifying the name and version of the application that issued the token.
- `format`: The type identifier and version of the token.
- `signature`: The Base64 encoded signature.
- `unverifiedCertificate`: The Base64 encoded DER encoded X509 certificate. The public key in the certificate can be used to verify the signature, but the certificate must be verified before.

Figure 12 shows a simplified example of the `authenticate` command.



```
web-eid -c authenticate '{"challengeNonce": "123456", "origin":
"https://ria.ee"}'
{
    "algorithm":"ES384",
    "appVersion":"https://web-eid.eu/web-eid-app/releases/2.2.0+0",
    "format":"web-eid:1.0",
    "signature":"kcDMXTdIiZcAy2r2...",
    "unverifiedCertificate":"MIID2zCCAz6gAwIBAgIQY..."
}
```

Figure 12. The `authenticate` command example in the CLI

The Web eID authentication token format specification describes how the signature in the authentication token is created [25]. The signature is built with the following logic: $signature = sign(hash(origin) + hash(challenge))$. The hash function is the same one used in the "algorithm" field. For example, `SHA256` is used for the `RS256` signature algorithm. The origin and challenge are hashed before concatenation to ensure separation between the two values. Otherwise two different origin/nonce values could have the same hash: $'origin.com' +' abc' \equiv' origin.co' +' mabc'$ [25].

3.2.2 Browser Extension

The Web-eID webextension is a browser extension for Chrome, Firefox and Safari. The webextension allows the web application to communicate with the Web eID native application using Native messaging. Native messaging allows a browser extension to communicate with an application installed on the user's computer [26]. The Web-eID webextension is written in Typescript and JavaScript [21].

3.2.3 Web-eid.js

The web-eid.js is a Javascript library that acts as a thin wrapper on top of the messaging interface provided by the Web eID browser extension. This can be used by the web applications for authentication and digital signing with ID cards. The web-eid.js library is written in Typescript and JavaScript [22].

4 Implementation Details

This section describes how we implemented the eMRTD authentication logic in the Web eID project.

The Web eID project uses an authentication token to allow a web server to authenticate the user. Similarly, our solution also uses an authentication token although the format has been changed. The eMRTD authentication token has a different purpose compared to the current Web eID authentication token. The eMRTD authentication token will only authenticate the chip due to how eMRTD works. After validating the eMRTD authentication token the web application must use the information in the token to do further authentication, like facial recognition over a video feed, to connect the authenticated card to the user. In this thesis we do not look at how a web application can do facial recognition using the photo in the eMRTD authentication token.

In this section we describe the new eMRTD authentication token and the reasoning behind it. We also provide instructions how a web application can verify the new eMRTD authentication token. Finally we outline the modifications made to the Web eID components to enable eMRTD authentication.

4.1 The eMRTD Authentication Token

The Web eID authentication token structure cannot hold the data needed by a web application to authenticate using eMRTD due to the differences between the eID applet and eMRTD applet functionality. A new eMRTD authentication token needs to be designed. This section describes the new eMRTD authentication token and the reasoning behind it. This section also describes to how to validate the eMRTD authentication token.

4.1.1 Design

Looking at the files present on the eMRTD applet file system and looking at what the web application needs for authentication we can model what data needs to be present in the eMRTD authentication token.

The `EF.SOD` file that contains the Document Security Object (SO_D) definitely needs to be in the token. The SO_D is going to be used to verify all the other files in the token and it itself is verified using the Document Signer Certificates in the SO_D. Because we use the SO_D to verify the eMRTD data in the new eMRTD authentication token we can only put full files from the eMRTD applet into the authentication token. If just a subset of some information from a file is put into the token then that data cannot be verified by the web application by checking the hash of the file in the SO_D.

The `signature` field needs to be present. Although the signature will need to be built slightly differently from how it was done in the Web eID project. The eMRTD signature is built as follows: $signature = sign(hash(hash(origin) + hash(challenge))[:8])$.

Both the `origin` and `challenge` need to be hashed separately before concatenation as reasoned in Section 3.2.1. After that the $hash(origin) + hash(challenge)$ needs to be hashed again otherwise even with the hashing algorithm with the shortest output only the origin hash would matter in the signature since we only sign 8 bytes. Or if the concatenation order would be reversed then only the challenge would matter in the signature. Obviously neither of these cases would not work from a security perspective. Only 8 bytes of the hash is signed because eMRTD Active Authentication only supports signing of 8 bytes. The signature is created with the Active Authentication private key (KPr_{AA}). This makes sure that the card is not cloned since the KPr_{AA} is stored in the card's secure memory. This also gives assurance that the token is not being replayed since the nonce is generated by the web server. Unfortunately signing just the 8 bytes means that the hash offers only 64 bit security. This means that an attacker has an easier time of finding a collision for the hash.

The eMRTD applet does not hold a X.509 certificate that contains information about the user (name, ID code, etc.). The only file that contains information about the user is the `EF.DG1` file that contains the MRZ. The MRZ will contain at least the issuing State, document number, date of birth, gender, date of expiry, card holder's nationality and person's name. The MRZ structure can be seen in Figure 7. The MRZ on Estonian ID cards also contains the ID code. Most likely majority of these parameters will not be necessary for a web application to authenticate the user. For example, the document number. Unfortunately there is no way to send just a subset of this information since then it is no longer possible to verify the data due to how file verification with the SO_{D} works.

The `format` and `appVersion` fields need to carry over from the Web eID authentication token and they fulfill the same purpose. Although the values need to be updated to reflect that the token is different.

The Active Authentication public key file (`EF.DG15`) also needs to be present in the token for a web application to be able to verify the signature.

The `algorithm` field also needs to be present as with the original authentication token to describe how the nonce was signed. However here it specifies the hashing algorithm that was used for hashing the values in the `signature` field.

The file that contains the user's facial image (`EF.DG2`) also needs to be in the token. After matching the facial image with the user's face the web application can be sure that the user presenting the card is also the owner of the card.

4.1.2 Structure

Below are the fields in the eMRTD authentication token. The "unverified" prefix in front of some of the fields signifies that the field needs to be verified before using it. The `DG` files and `EF.SOD` file are Base64 encoded DER ASN.1 objects. More details about each of these files can be found in Section 2.2 and "eMRTD specification Part 10: Logical Data Structure (LDS) for Storage of Biometrics and Other Data in the Contactless Integrated

Circuit (IC)" [3]. These are the fields in the new eMRTD authentication token:

- `algorithm`: The hash algorithm that is used to hash the values used to create the signature.
- `appVersion`: The URL identifying the name and version of the application that issued the token. Has the same purpose as in the Web eID authentication token. Although the value here will be different as to reflect that it is the eMRTD version.
- `format`: The type identifier and version of the token. Has the same purpose as in the Web eID authentication token. Although the value here will be different as to reflect that it is the eMRTD version.
- `signature`: The signature created with the `origin`, `challenge` and Active Authentication private key ($\mathrm{KPr}_{\mathrm{AA}}$) as described in the previous section. The signature allows the web application to be sure that the user used a valid ID card, the card is not cloned and the signature is not replayed by an attacker.
- `unverifiedPublicKeyInfo`: The contents of the `EF.DG15` file that have been Base64 encoded. This is the Active Authentication public key ($\mathrm{KPu}_{\mathrm{AA}}$) that corresponds to the Active Authentication private key ($\mathrm{KPr}_{\mathrm{AA}}$) that was used to create the signature. This public key is used for verifying the signature. The content of this is the `SubjectPublicKeyInfo` as defined in RFC-5280 [14].
- `unverifiedDocumentSecurityObject`: The contents of the `EF.SOD` file that have been Base64 encoded. This file is used to verify all other files from the eMRTD chip in the token and it itself can be authenticated using the Document Signer Certificate. This allows for verifying the integrity of the files in the token.
- `unverifiedMrz`: The contents of the `EF.DG1` file that have been Base64 encoded. Contains the MRZ that holds details about the user.
- `unverifiedPhoto`: The contents of the `EF.DG2` file that have been Base64 encoded. Contains the photo of the user.

The format of the eMRTD authentication token is JSON.

4.1.3 Verification

The web application using eMRTD authentication has to verify all the values in the eMRTD authentication token to be sure the token is valid. To verify the data in the eMRTD authentication token the following steps need to be done by the web application. These are the same steps as with eMRTD Passive Authentication in Section 2.3.1.

1. The web application must parse the `unverifiedDocumentSecurityObject` and extract the Document Signer Certificate (CDS) from the Document Security Object (SO_D).

2. The web application must build and validate the certification path from a Trust Anchor to the CDS. The trust anchor is the Country Signing Certification Authority (C_{CSCA}). Each State has their own C_{CSCA} that issues certificates for the eMRTD PKI [15].

3. The web application must use the Document Signer Public Key from the verified Document Signer Certificate (CDS) to verify the signature of the SO_D.

4. The web application must decode the Base64 encoded `unverifiedPhoto`, `unverifiedMrz`, `unverifiedPublicKeyInfo` fields in the token and hash them. After that the web application must compare these created hashes to the hashes in the Document Security Object (SO_D). The generated hash and the hash in the SO_D must match. If the hashes do not match then the file has been modified and the token is not valid.

5. The web application must verify the signature. The web application first must create $hashToBeSigned = hash(hash(origin) + hash(challenge)[:8])$ using the hash algorithm from the `algorithm` field. Then the web application needs extract the Active Authentication public key (KPu_{AA}) from use the new verified `unverifiedPublicKeyInfo`. The KPu_{AA} and $hashToBeSigned$ must be used to verify the `signature`. If the signature is valid then that means the card has not been cloned.

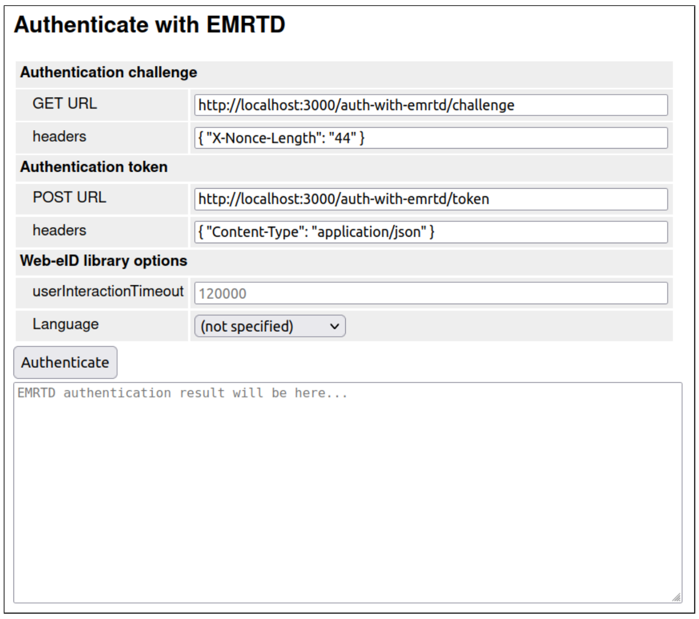
The steps above verify that the card is valid and has not been cloned. Next the web application should parse the MRZ and extract the facial image from the `unverifiedPhoto` and use the photo to do remote facial recognition.

4.2 Usage Example

This section describes how the user can use the new eMRTD authentication feature.

Figure 13 shows the authentication option on the Web eID mock webapp. The user clicks the “Authenticate” button. The web application sends the `authenticate-with-emrtd` command and related arguments to the modified Web eID webextension. The modified webextension in turn sends it to the modified Web eID native application.

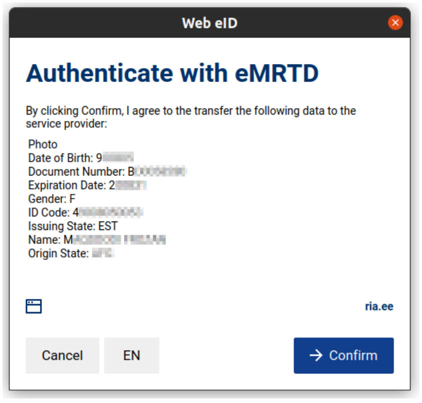
The native application selects the eID applet and reads the necessary personal data files from the applet. Then the native application selects the eMRTD applet and uses the data from the personal data files to negotiate the BAC session keys. Once the native applications reads the MRZ file from the eMRTD applet the native app opens the UI screen shown in Figure 14 where the data from the MRZ is displayed. The user is shown a list of all the data that will be sent to the web application. If the user is fine with sending this data to the web application then they can click the “Confirm” button. This opens the loading screen shown in Figure 16. During the loading screen the native application reads the rest of the required data from the chip, creates the signature and then creates the eMRTD authentication token. Once the native application creates eMRTD authentication token the loading screen closes and the token is returned to the Web eID webextension. The webextension in turn sends it back to the Web eID mock webapp. The resulting authentication token is shown in Figure 11.



Authenticate with EMRTD

Authentication challenge	
GET URL	http://localhost:3000/auth-with-emrtd/challenge
headers	{ "X-Nonce-Length": "44" }
Authentication token	
POST URL	http://localhost:3000/auth-with-emrtd/token
headers	{ "Content-Type": "application/json" }
Web-eID library options	
userInteractionTimeout	120000
Language	(not specified)

Figure 13. New eMRTD authentication option in the Web eID mock webapp

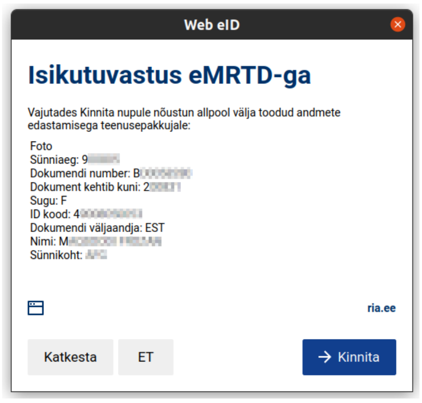

Web eID

Authenticate with eMRTD

By clicking Confirm, I agree to the transfer the following data to the service provider:

Photo
Date of Birth: 9[redacted]
Document Number: B[redacted]
Expiration Date: 2[redacted]
Gender: F
ID Code: 4[redacted]
Issuing State: EST
Name: M[redacted]
Origin State: [redacted]

Figure 14. eMRTD authentication confirmation screen in the Web eID native application


Web eID

Isikutuvastus eMRTD-ga

Vajutades Kinnita nupule nõustun allpool välja toodud andmete edastamisega teenusepakkujale:

Foto
Sünniaeg: 9[redacted]
Dokumendi number: B[redacted]
Dokument kehtib kuni: 2[redacted]
Sugu: F
ID kood: 4[redacted]
Dokumendi väljaandja: EST
Nimi: M[redacted]
Sünnikoht: [redacted]

Figure 15. eMRTD authentication confirmation screen in the Web eID native application in Estonian

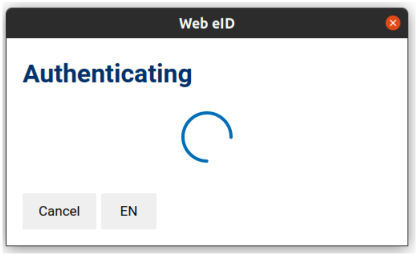
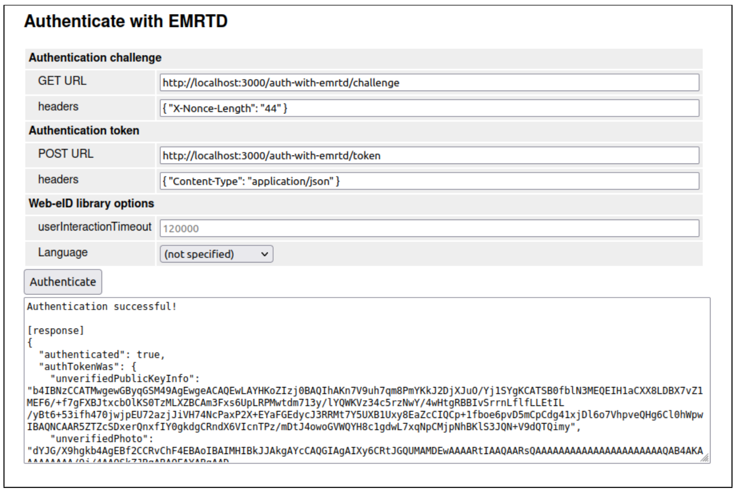


Figure 16. Loading screen after pressing the “Confirm” button in the eMRTD confirmation screen (Figure 14)


Authenticate with EMRTD

Authentication challenge	
GET URL	http://localhost:3000/auth-with-emrtd/challenge
headers	{ "X-Nonce-Length": "44" }
Authentication token	
POST URL	http://localhost:3000/auth-with-emrtd/token
headers	{ "Content-Type": "application/json" }
Web-eID library options	
userInteractionTimeout	120000
Language	(not specified)

```
Authentication successful!

[response]
{
  "authenticated": true,
  "authTokenWas": {
    "unverifiedPublicKeyInfo":
"b4IBNzCCATMwgewGByqGSM49AgEwgeACAQEwLAYHKoZIzj0BAQIhAKn7V9uh7qm8PmYKkJ2DjXJuO/Yj1SYgKCATSB0fblN3MEQEIH1aCXX8LDBX7vZ1
MEF6/+f7gFXBJtxcb0lKS0TzMLXZBCAm3Fxs6UpLRPMwtdm713y/lYQWKVz34c5rzNwY/4wHtgRBBIvSrrnLflfLLEtIL
/yBt6+53ifh470jwjpEU72azjJiVH74NcPaxP2X+EYaFGEdycJ3RRMt7Y5UXB1Uxy8EaZcCIQCp+1fboe6pvD5mCpCdg41xjDl6o7VhpveQHg6Cl0hWpw
IBAQNCAAR5ZTZcSDxerQnxfIY0gkdgCRndX6VIcnTPz/mDtJ4owoGVWQYH8c1gdwL7xqNpCMjpNhBKlS3JQN+V9dQTQimy",
    "unverifiedPhoto":
"dYJG/X9hgkb4AgEBf2CCRvChF4EBAoIBAIMHIBkJJAkgAYcCAQGIAgAIXy6CRtJGQUMAMDEwAAAARtIAAQAARsQAAAAAAAAAAAAAAAAAAAAAQAB4AKA
```

Figure 17. Mock webapp showing the new eMRTD authentication token after successful authentication

Should be noted that the Web eID mock webapp does not do any validation on the token. In an actual web application the server also needs to verify the eMRTD token as specified in the previous section. The facial image from the authentication token can also be used to do further authentication on the user.

Figure 18 shows the high-level overview of how our solution works.

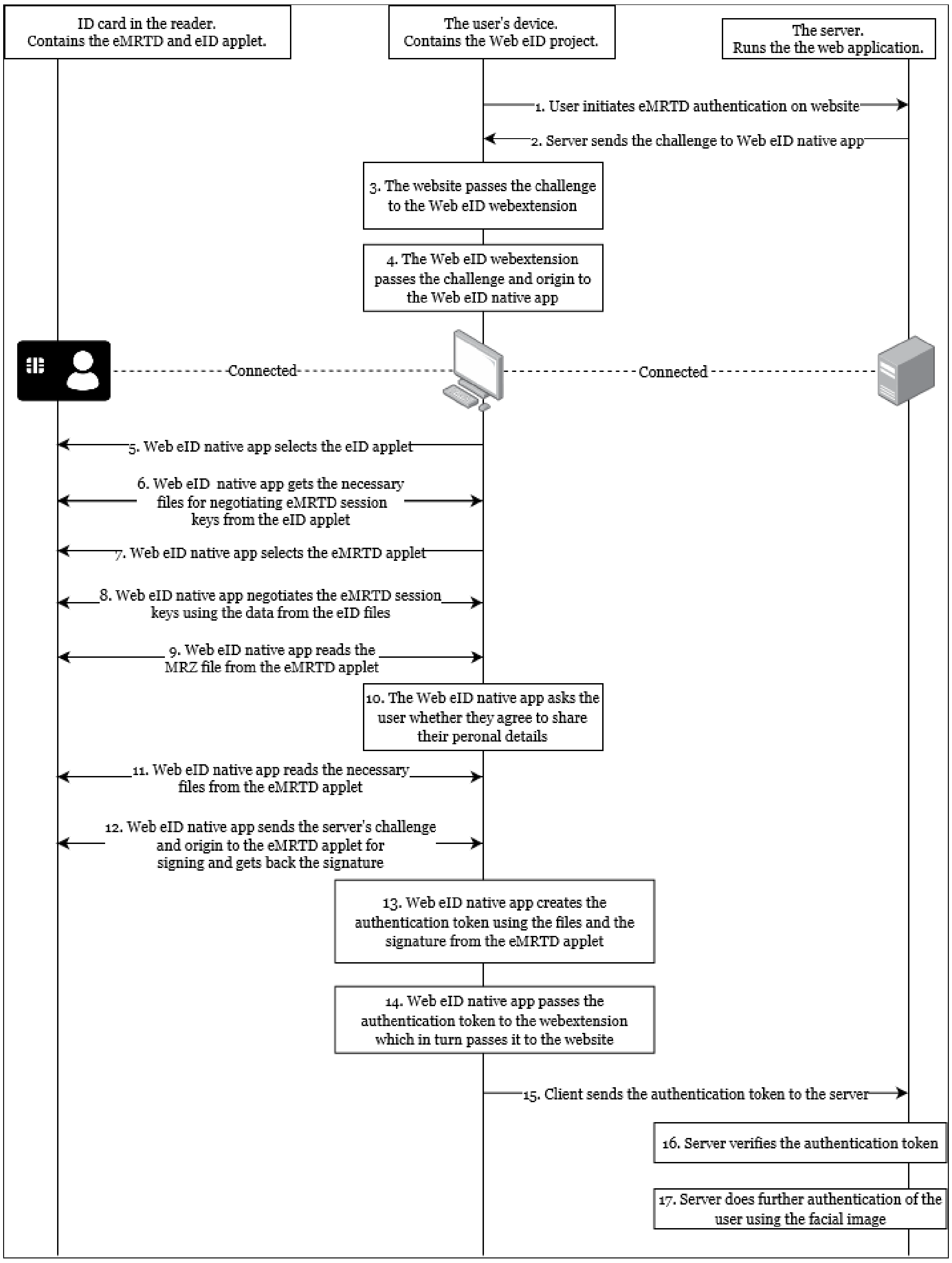

ID card in the reader.
Contains the eMRTD and eID applet.
The user's device.
Contains the Web eID project.
The server.
Runs the the web application.
1. User initiates eMRTD authentication on website
2. Server sends the challenge to Web eID native app
3. The website passes the challenge to the Web eID webextension
4. The Web eID webextension passes the challenge and origin to the Web eID native app
Connected
Connected
5. Web eID native app selects the eID applet
6. Web eID native app gets the necessary files for negotiating eMRTD session keys from the eID applet
7. Web eID native app selects the eMRTD applet
8. Web eID native app negotiates the eMRTD session keys using the data from the eID files
9. Web eID native app reads the MRZ file from the eMRTD applet
10. The Web eID native app asks the user whether they agree to share their peronal details
11. Web eID native app reads the necessary files from the eMRTD applet
12. Web eID native app sends the server's challenge and origin to the eMRTD applet for signing and gets back the signature
13. Web eID native app creates the authentication token using the files and the signature from the eMRTD applet
14. Web eID native app passes the authentication token to the webextension which in turn passes it to the website
15. Client sends the authentication token to the server
16. Server verifies the authentication token
17. Server does further authentication of the user using the facial image

Figure 18. High-level overview of our solution

4.3 Forked Projects

To add eMRTD authentication support to the Web eID project the following components were modified. The Appendix contains instructions on how to run these forked Web eID components.

- Web eID native application [20]
- Web eID webextension [21]
- The web-eid.js library [22]
- Web eID mock webapp [24]

The following sections describe the changes to each of the Web eID component in more detail.

4.3.1 Native application

We forked the Web eID native application [20] and added eMRTD support to it. The forked project is on Github [27].

We tried to keep the new eMRTD code and original Web eID code as separate as possible due to the differences between the two. Also should this fork ever be merged into the Web eID project having the new eMRTD code separate makes checking the code easier and we can be sure we did not introduce bugs to the original code.

From a high-level view the main changes and additions to the Web eID native application projects were:

- We added a new `authenticate-with-emrtd` command that allows the user to use the new eMRTD functionality.
- We added functionality that reads files off the eID applet. These files contain data needed to establish the session keys with BAC.
- We added functionality that allows the Web eID native application to communicate with the eMRTD applet. This involved implementing the secure communication logic: creating the session keys using BAC, encrypting and decrypting the messages between the native application and the chip.
- We added functionality that reads files from the eMRTD applet. These files were needed to create the eMRTD authentication token.
- We added functionality that parses the DER encoded files and extracts the required information from the files. This extracted data was used in the UI and for creating the eMRTD authentication token.

- We added new UI views. We created a new authentication confirmation screen that displays data from the MRZ. The new authentication confirmation screen can be seen in Figure 14. We also created a loading screen that is shown after the user clicks “Confirm” in the confirmation screen. This is to improve the user experience since reading the larger files from the chip takes awhile. The new loading screen screen can be seen in Figure 16.

- We added translations for the eMRTD UI. The eMRTD UI supports both Estonian (ET) and English (EN). Should be noted that currently the language changing functionality does not work correctly. The list of user details is not updated until the application restarts. The confirmation screen in Estonian can be seen in Figure 15.

First to create the session keys using BAC we need the document number, date of birth and expiration date. Usually this this information is extracted from the MRZ, but without session keys we cannot read the `EF.DG1` file that contains the MRZ from the eMRTD applet. Fortunately the eID applet holds all the data we need albeit in separate files. The ID card applet (EstEID) specification lists all the files present on the eID applet [28]. We first select the eID applet and read the the document number, date of birth and expiration date files. A very important point here is that different versions of ID cards and ID cards from other countries might use different file names. The cards might not even have the necessary files. In such a case our implementation might break. One possible solution for this would be to store details of various ID cards in the native application. If the required personal data files are not present in the eID applet then the native application can use PACE instead. The user would only need to enter the CAN number on the card which is easier than entering the document number, date of birth and expiration date.

We then switch to the eMRTD applet and negotiate the session keys using BAC. BAC was selected because it is supported by all cards [4] and it is the easiest to implement. After that we read the `EF.DG1` file that contains the MRZ and display the authentication confirmation screen screen to the user. The confirmation screen is shown in Figure 14. The confirmation screen displays the parsed MRZ contents showing what data will be sent to the web application. It should be noted that when the issuer field in the MRZ is not “EST” then we log a debugging message that warns that non Estonian ID card MRZ fields might be different and in such a case the values in the UI might be displayed incorrectly. For example, the MRZ on German ID cards is different, among other differences the MRZ lacks the ID code [29].

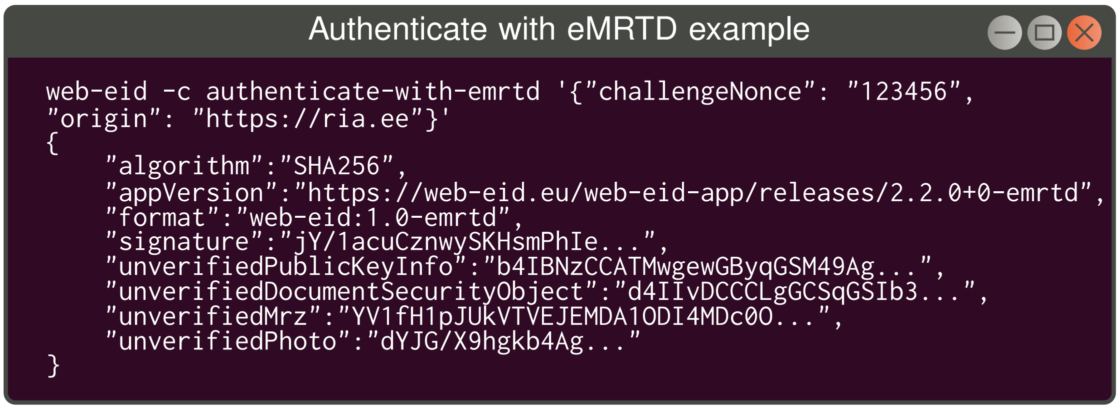
When the user clicks on the “Confirm” button the loading screen is shown and the rest of the necessary files are read. The loading screen can be seen in Figure 16. The loading screen was added to show the user that the native application is doing work. Reading large files off the chip takes time. Overall the time from the moment the user clicks “Confirm” to the moment the token is created seems to be around 17 seconds. This is due to having to read the `EF.DG2` which contains the photo of the person. Without reading the photo the same process takes 3 seconds. The `EF.DG2` size seems to be around 18 KB. If there were no loading screen then the “Confirm” screen would be shown the entire time and would make it seem that the program crashed even though native app is just reading the files.

During the loading screen we read the `EF.SOD` which is the Document Security Object (SO_D), `EF.DG2` which contains the photo of the person, `EF.DG14` which contains the eMRTD Active Authentication related data and `EF.DG15` which contains the eMRTD Active Authentication public key info. We then extract the hash algorithm from `EF.DG14` and use the hash algorithm to create the signature as described in the Section 4.1.1 and Section 4.1.2. We then use the read files and created signature to create the eMRTD authentication token.

In the code we also commented out the HTTPS protocol check for the “origin” URL. Otherwise to use the Web eID mock webapp a TLS certificate would need to be created and set up in the application and browser. Allowing HTTP is only to make it easier to set up the forked Web eID components and test it.

Additionally we modified the libpcsc-cpp library [30] that is used by the Web eID native application. The libpcsc-cpp library was created by the Web eID team and it is for accessing smart cards. In the libpcsc-cpp library we disabled the transaction guard check logic. This is because we cannot start a transaction outside of the library and the library does not allow us to send bytes to the card without a transaction running. We are not sure what the transaction guard actually does, its purpose is not documented. That being said the Web eID native app seems to work without it, but there could be cases where something might break because this transaction functionality is not used.

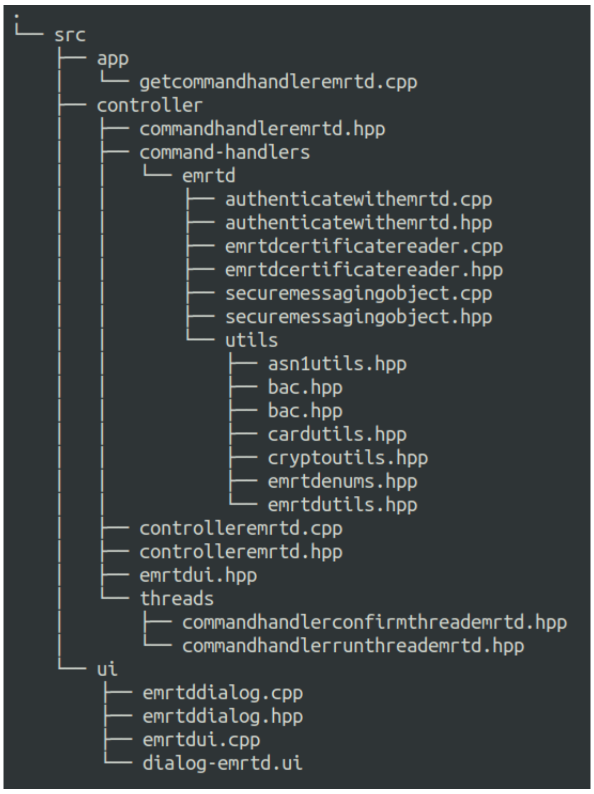
The simplified pseudo-code for the eMRTD authentication token creation logic that was implemented is shown in Algorithm 1. Figure 19 shows an example how to use the new `authenticate-with-emrtd` command using the CLI. In Figure 19 the first line is the CLI entry and below that is the returned eMRTD authentication token.



```
web-eid -c authenticate-with-emrtd '{"challengeNonce": "123456",
"origin": "https://ria.ee"}'
{
    "algorithm":"SHA256",
    "appVersion":"https://web-eid.eu/web-eid-app/releases/2.2.0+0-emrtd",
    "format":"web-eid:1.0-emrtd",
    "signature":"jY/1acuCznwySKHsmPhIe...",
    "unverifiedPublicKeyInfo":"b4IBNzCCATMwgewGByqGSM49Ag...",
    "unverifiedDocumentSecurityObject":"d4IIvDCCCLgGCSqGSIb3...",
    "unverifiedMrz":"YV1fH1pJUkVTVEJEMDA1ODI4MDc0O...",
    "unverifiedPhoto":"dYJG/X9hgkb4Ag..."
}
```

Figure 19. The `authenticate-with-emrtd` command example in the CLI

Figure 20 shows the files that we added to the Web eID project. In addition the the files shown in Figure 20 many existing files also needed to be modified. The files in the `utils` directory hold logic that is used by the rest of the files in the application. The `command-handlers/emrtd` directory files handle the creation of the eMRTD authentication token. The `ui` directory holds logic that displays the UI views to the user. The rest of the files are there due to the architecture of the application.



```
.
└── src
    ├── app
    │   └── getcommandhandleremrtd.cpp
    ├── controller
    │   ├── commandhandleremrtd.hpp
    │   ├── command-handlers
    │   │   └── emrtd
    │   │       ├── authenticatewithemrtd.cpp
    │   │       ├── authenticatewithemrtd.hpp
    │   │       ├── emrtdcertificatereader.cpp
    │   │       ├── emrtdcertificatereader.hpp
    │   │       ├── securemessagingobject.cpp
    │   │       ├── securemessagingobject.hpp
    │   │       └── utils
    │   │           ├── asn1utils.hpp
    │   │           ├── bac.hpp
    │   │           ├── bac.hpp
    │   │           ├── cardutils.hpp
    │   │           ├── cryptoutils.hpp
    │   │           ├── emrtdenums.hpp
    │   │           └── emrtdutils.hpp
    │   ├── controlleremrtd.cpp
    │   ├── controlleremrtd.hpp
    │   ├── emrtdui.hpp
    │   └── threads
    │       ├── commandhandlerconfirmthreademrtd.hpp
    │       └── commandhandlerrunthreademrtd.hpp
    └── ui
        ├── emrtddialog.cpp
        ├── emrtddialog.hpp
        ├── emrtdui.cpp
        └── dialog-emrtd.ui
```

Figure 20. The files added to the Web eID native application

4.3.2 web-eid.js

We forked the Web eID web-eid.js [22] library and added eMRTD support to it. The forked project is on Github [31].

Here no new files were added. We just added new Typescript request and response objects for the `authenticate-with-emrtd` command. We also added a function that allows a web application to use the new `authenticate-with-emrtd` command.

4.3.3 Webextension

We forked the Web eID webextension [21] and added eMRTD support to it. The forked project is on Github [32].

First we made this project use our own forked version of the web-eid.js library. This allows our forked Web eID webextension to use our new eMRTD request and response objects we added to the web-eid.js library fork. We then added the "authenticate-with-emrtd.ts" file that handles the `authenticate-with-emrtd` command. We also modified some existing files and added code that calls our new logic. This project required little modifications because the webextension in essence is just a proxy between the web application and the native application.

4.3.4 Mock Webapp

We forked the Web eID webextension mock webapp [24] and added eMRTD support to it. The forked project is on Github [33]. The forked Web eID mock webapp is used for demonstration purposes and it simulates a web application that uses eMRTD to authenticate the user. It is important to note that this mock webapp does not validate the token, the token is just displayed.

This project used the web-eid.js library from the NPM registry. We replaced the NPM web-eid.js library reference with our own local forked version of the web-eid.js. The Web eID webextension mock webapp uses Handlebars to create the UI and Node.js/express to serve files. We added a new section to the Handlebars “webeid.hbs” file that creates a new eMRTD authentication section in the UI along with the “authenticate-with-emrtd.js” file that adds functionality to the new eMRTD UI section. This new UI section can be seen in Figure 13. We also added a new “AuthWithEmrtdController.js” that handles the back end logic for the eMRTD authentication. We also modified some existing files and added code that invokes our new logic.

Algorithm 1 The high-level view of the eMRTD authentication token creation algorithm in the Web eID native application

```
Require: challengeNonce, origin
 1: # Native application starts.
 2:
 3: # Select eID applet.
 4: selectApplet('A000000077010800070000FE00000100')
 5:
 6: # The calculateCheckDigit calculates the check digit used in the MRZ.
 7: documentNumber ← readFile('EF.5007')
 8: documentNumberCheckDigit ← calculateCheckDigit(documentNumber)
 9: dateOfBirth ← readFile('EF.5005')
10: dateOfBirthCheckDigit ← calculateCheckDigit(dateOfBirth)
11: expirationDate ← readFile('EF.5008')
12: expirationDateCheckDigit ← calculateCheckDigit(expirationDate)
13:
14: # Select eMRTD applet and then select LDS applet.
15: selectApplet('A00000024710FF')
16: selectApplet('A0000002471001')
17:
18: secret ← documentNumber + documentNumberCheckDigit + dateOfBirth
              + dateOfBirthCheckDigit + expirationDate + expirationDateCheckDigit
19: secureMessaging ← establishBacSessionKeys(secret)
20:
21: mrzEmrtd ← secureMessaging.readFile(keys, 'EF.DG1')
22:
23: # Confirmation UI view is shown to the user. When user clicks on the "Confirm"
    button the loading screen UI view is shown.
24:
25: photo ← secureMessaging.readFile(keys, 'EF.DG2')
26: publicKeyInfo ← secureMessaging.readFile(keys, 'EF.DG15')
27: documentSecurityObject ← secureMessaging.readFile(keys, 'EF.SOD')
28:
29: dg14 ← secureMessaging.readFile(keys, 'EF.DG14')
30: hashAlgorithm ← getHashAlgorithmName(dg14)
31:
32: # Hashing is done using the hashAlgorithm.
33: dataToBeSigned ← hash(hash(challengeNonce) + hash(origin))[:8]
34: signature ← secureMessaging.sign(dataToBeSigned)
35:
36: return createAuthenticationToken(signature, mrzEmrtd, hashAlgorithm,
                                     publicKeyInfo, documentSecurityObject)
```

5 Future Work

In this section we present further improvements that should be added to our implementation.

Some additional features should be added to the Web eID native application. The Web eID native application should support PACE as a fallback. It might be possible that the eID applet of a country does not have the necessary document number, date of birth and expiration date files that are needed to negotiate the BAC session keys. Manually entering the CAN for PACE is a lot easier than manually entering the document number, date of birth and expiration date. Also as a sanity check perhaps the native app can do the signature validation and the file validation using the SO_D before returning the eMRTD authentication token to the caller. The Web eID native app supports English, Estonian, Finnish, Hungarian, Russian. Our eMRTD UI views only support English and Estonian. Support for Finnish, Hungarian, Russian should be added as well. Our implementation has only been tested with the Estonian ID card. The Web eID native application supports many ID card versions from different countries. Support for other ID cards should be added as well. Finally, we marked some spots in the code using the `TODO` keyword. These are mainly marking features that could not be completed due to time constraints. For example, the user's facial image should be displayed in the "Confirm" screen. Some `TODO`s also mark bugs. For example, the bug where the language is changed but the change only takes effect the next time the native application runs.

Tests should be added for the new eMRTD logic in the Web eID project. This is to ensure that future changes don't break the eMRTD authentication logic.

There should also be code examples for different programming languages and perhaps frameworks that implements the eMRTD authentication token validation logic correctly. This can be used by actual service providers in their web applications to support eMRTD authentication.

We only tested our solution with Ubuntu. The Web eID native application should work with Windows and macOS as well.

Finally, after these fixes and improvements perhaps the eMRTD fork can be merged into the Web eID project.

6 Conclusion

In this thesis we added eMRTD authentication support to the Web eID project enabling the authentication of users over the web using the eMRTD applet on new ID cards. We also gave an overview of key points of the eMRTD specification and Web eID project.

Our solution works by having the Web eID project create a new eMRTD authentication token. This eMRTD authentication token contains files from the eMRTD applet and a signature created by the eMRTD applet. The signature is a signed hash of origin and the challenge sent by the web application. Thanks to the eMRTD security features the eMRTD authentication token can be used by the web application to verify that the ID card is valid. When the web application uses a video feed to match the facial image of the user with the facial image in the eMRTD authentication token then the web application can be sure of the user's identity.

This approach is more convenient from the user's perspective because the user does not have to enter any PIN codes. They just have to have their card in the card reader. Of course this authentication method is applicable in fewer scenarios than the regular ID card authentication.

At the end we also gave an overview of work still needing to be done for the eMRTD authentication code to be fully ready. Hopefully when those changes are implemented this eMRTD authentication logic can be merged into the Web eID project.

References

[1] Republic of Estonia. Information System Authority. New ID card version will have changes for developers. https://www.id.ee/artikkel/id-kaardi-uue-versiooniga-kaasnevad-muudatused-arendajate-jaoks/.

[2] Web eID. https://www.id.ee/en/article/web-eid/.

[3] ICAO. Machine Readable Travel Documents Part 10: Logical Data Structure (LDS) for Storage of Biometrics and Other Data in the Contactless Integrated Circuit (IC). *Doc 9303*, Eighth Edition, 2021. https://www.icao.int/publications/documents/9303_p10_cons_en.pdf.

[4] ICAO. Machine Readable Travel Documents Part 11: Security Mechanisms for MRTDs. *Doc 9303*, Eighth Edition, 2021. https://www.icao.int/publications/documents/9303_p11_cons_en.pdf.

[5] Burak Can Kus. *Use of Electronic Identity Documents for Multi-Factor Authentication.* MSc thesis, University of Tartu, 2021. https://comserv.cs.ut.ee/ati_thesis/datasheet.php?id=72515&year=2021.

[6] Burak Can Kus. eMRTD Face Access. https://github.com/Fethbita/eMRTD_face_access, 2021.

[7] ICAO. Machine Readable Travel Documents Part 1: Introduction. *Doc 9303*, Eighth Edition, 2021. https://www.icao.int/publications/documents/9303_p1_cons_en.pdf.

[8] ICAO. Machine Readable Travel Documents Part 9: Deployment of Biometric Identification and Electronic Storage of Data in MRTDs. *Doc 9303*, Eighth Edition, 2021. https://www.icao.int/publications/documents/9303_p9_cons_en.pdf.

[9] Estonian Police and Border Guard Board. ID card samples (in Estonian), August 28, 2020. https://www.politsei.ee/et/juhend/id-kaardi-naeidised-1.

[10] Russ Housley. Cryptographic Message Syntax (CMS). RFC 3369, September 2002.

[11] ICAO. Machine Readable Travel Documents Part 5: Specifications for TD1 Size Machine Readable Official Travel Documents (MROTDs). *Doc 9303*, Eighth Edition, 2021. https://www.icao.int/publications/Documents/9303_p5_cons_en.pdf.

[12] ICAO. Portrait Quality. (Reference Facial Images for MRTD). 2018. https://www.icao.int/Security/FAL/TRIP/Documents/TR%20-%20Portrait%20Quality%20v1.0.pdf.

[13] ICAO. Technical Advisory Group On Machine Readable Travel Documents (Tag/MRTD), Revision of Doc 9303 – Machine Readable Travel Documents. 2014. https://www.icao.int/Meetings/TAG-MRTD/TagMrtd22/TAG-MRTD-22_WP03-rev.pdf.

[14] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and David Cooper. Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile. RFC 5280, May 2008.

[15] ICAO. Machine Readable Travel Documents Part 12: Public Key Infrastructure for MRTDs. *Doc 9303*, Eighth Edition, 2021. https://www.icao.int/publications/Documents/9303_p12_cons_en.pdf.

[16] ICAO. Access to epassport chip. https://www.icao.int/Security/FAL/PKD/BVRT/Pages/Document-readers.aspx.

[17] Yifei Liu, Timo Kasper, Kerstin Lemke-Rust, and Christof Paar. E-passport: Cracking basic access control keys with copacobana. 2007. https://informatik.rub.de/wp-content/uploads/2022/01/epasscrack_sharcs_2007.pdf.

[18] ICAO. Machine Readable Travel Documents Part 4: Specifications for Machine Readable Passports (MRPs) and other TD3 Size MRTDs. *Doc 9303*, Eighth Edition, 2021. https://www.icao.int/publications/documents/9303_p4_cons_en.pdf.

[19] IDEMIA. LDS Applet V10 EAC with AA Configuration – Public Security Target. https://www.commoncriteriaportal.org/files/epfiles/st-2018_04.pdf.

[20] Web eID. Web-eID Native Application. https://github.com/web-eid/web-eid-app, 2023.

[21] Web eID. Web-eID Webextension. https://github.com/web-eid/web-eid-webextension, 2023.

[22] Web eID. Web-eid.js. https://github.com/web-eid/web-eid.js, 2023.

[23] Web eID. Web eID: electronic identity cards on the Web. https://github.com/web-eid/web-eid-system-architecture-doc, 2022.

[24] Web eID. Web-eID Mock Webapp. https://github.com/web-eid/web-eid-webextension-mock-webapp, 2023.

[25] Estonian Information System Authority Mart Sõmermaa. Web eID Authentication Token Format Specification. 2022. https://web-eid.github.io/web-eid-system-architecture-doc/web-eid-auth-token-v2-format-spec.pdf.

[26] MDN web docs. Native messaging. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Native_messaging.

[27] Silver Maala. Web-eID Native Application with eMRTD support. https://github.com/silvergithub999/emrtd-web-eid-app, 2023.

[28] Republic of Estonia. Information System Authority. Estonia id1 chip/app 2018 technical description. Document Release Version: V1.0. https://www.id.ee/wp-content/uploads/2021/08/td-id1-chip-app-4.pdf.

[29] Federal Republic of Germany. The Federal Ministry of the Interior and Community (BMI). Data on the ID Card. https://www.personalausweisportal.de/Webs/PA/EN/citizens/german-id-card/data-on-the-id-card/data-on-the-id-card-node.html.

[30] Silver Maala. Fork of the libpcsc-cpp library. https://github.com/silvergithub999/emrtd-libpcsc-cpp, 2023.

[31] Silver Maala. web-eid.js library with eMRTD support. https://github.com/silvergithub999/emrtd-web-eid.js, 2023.

[32] Silver Maala. Web-eID Webextension with eMRTD support. https://github.com/silvergithub999/emrtd-web-eid-webextension, 2023.

[33] Silver Maala. Web-eID Mock Webapp with eMRTD support. https://github.com/silvergithub999/emrtd-web-eid-webextension-mock-webapp, 2023.

[34] Mart Sõmermaa (mrts). Dockerfile docker-qt-cmake-gtest-valgrind-ubuntu. https://github.com/mrts/docker-qt-cmake-gtest-valgrind-ubuntu/blob/master/Dockerfile.

Appendix

Installation Instructions

In this section we give instructions on how to run the forked Web eID components.

When cloning the repositories make sure the directories are in the same parent directory. This is because the Web eID mock webapp fork component expects the web-eid.js library to be in the `./emrtd-web-eid-webextension/lib/web-eid.js` path.

Here are the instructions on how to run the forked Web eID components:

1. The following packages need to be installed to be able to compile and run the forked Web eID components [34]:

Terminal

```
$ apt-get update
$ apt-get install -y --no-install-recommends \
      git \
      build-essential \
      clang clang-format \
      pkg-config \
      cmake \
      libgtest-dev \
      libssl-dev \
      dh-make devscripts cdbs fakeroot \
      valgrind \
      qtbase5-dev qt5-default libqt5svg5-dev \
      qttools5-dev \
      pcscd libpcsclite-dev \
      nodejs
```

2. Here are the commands to install the Web eID native application eMRTD fork [27]:

Terminal

```
$ git clone --recurse-submodules
git@github.com:silvergithub999/emrtd-web-eid-app.git
$ cd emrtd-web-eid-app
$ cmake . && cmake --build . --target install && cp
src/app/web-eid /usr/local/bin/web-eid && cp src/app/web-eid
/usr/bin/web-eid
```

3. Here are the commands to install the Web eID webextension eMRTD fork [32]:

Terminal

```
$ git clone --recurse-submodules
git@github.com:silvergithub999/emrtd-web-eid-webextension.git
$ cd emrtd-web-eid-webextension/lib/web-eid.js
$ npm install && npm run build
$ cd ../..
$ npm install && npm run clean build package
```

To add the browser extension to Firefox go to the URL `about:debugging#/runtime/this-firefox` and click on the “Load Temporary Add-on...” button. In the window that opens select the manifest file from ’./emrtd-web-eid-webextension/dist/firefox/manifest.json’.

4. Here are the commands to install the Web eID mock webapp eMRTD fork [33]:

Terminal

```
$ git clone git@github.com:silvergithub999/emrtd-
web-eid-webextension-mock-webapp.git
$ cd emrtd-web-eid-webextension-mock-webapp
$ npm install && npm run start
```

The final command runs the server on port 3000. To access the web page go to `http://localhost:3000/`. There click on the “Web-eID” button and you will reach the authentication page.

II. Licence