A secure filesender service based on Remote Document Encryption

Eric R. Verheul[⋆]

KeyControls, Radboud University Nijmegen,
P.O. Box 9010, NL-6500 GL Nijmegen, The Netherlands.
eric.verheul@[keycontrols.nl,cs.ru.nl]
Version 0.63, 4 May 2020

Abstract We show how a public key infrastructure (PKI) implemented in current electronic identity documents can form the basis for a secure file sending service. This allows users selecting a recipient PKI certificate based on identity document information (names, but also facial images) signed by the issuing government. These certificates then allow encrypting (large) files for the recipient whereby decryption requires access to the identity document. In other words, RDE-SFS allows an identity document to be used as PKI smartcard. This technique is supported by almost all European passports and can provide for 160 bit security. In addition end-to-end security is supported as encryption and decryption can take in the user internet browser using a JavaScript cryptographic library, e.g. the W3C Web Cryptography API. By using buffered encryption and decryption very large files can be handled with low browser requirements.

Keywords: end-to-end secure file sending, electronic passport, electronic driver license

[⋆] Work done for the Dutch Vehicle Authority (RDW).

Contents

List of Tables

List of Figures

VERSION CONTROL

Version	Date	Description
0.62	2019-06-11	first draft
0.63	2020-03-05	Minor changes, elaborated on registration process in Section 2, added document “mnemonic” during registration and decryption allowing the user recognizing the document required for decryption.

1 Introduction

1.1 Background

A *filesender service* allows a user to send files other users. Such service is particulary relevant when the files are very large, e.g. gigabytes in size, so that file sending by email is not possible. In the context of a filesender service three parties exist, as is depicted in Figure 1 below. For simplicity we only distinguish one receiving user in our descriptions but this can simply extended to multiple users.

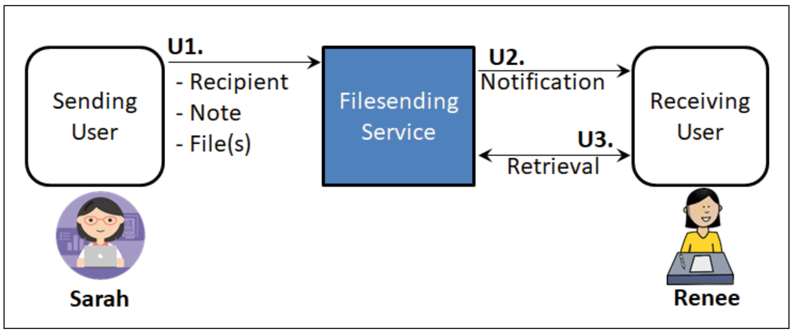

Figure 1. (Secure) filesender service setup

The basic use-case is as follows:

1. The *sending user* Sarah connects to the filesender service using an internet browser and uploads the files and the identity of Renee, the intended *receiving user*, e.g. her email address. Typically Saray will accompany the files with an explanatory note.
2. The receiving user Renee is notified, e.g. through email, that files are available including an explanatory note and a link (URL) to these files.
3. Renee starts up an internet browser and uses the link to retrieve the files. The filesender service removes the files and the explanatory note.

From a security perspective it is prudent to let the sending user authenticate to the filesender service in Step U1. This allows for an sender user identity to be reliably included in the notification to the receiving user in Step U2. In this way the receiving user can assess if she knows/trusts the sending user and if she wants to download the files in Step U3. Basic authentication could be based on user-id/password whereby the sender's email address is used as its user-id. The filesender service checks that the sender user has access to this email address.

In the described setup the files are temporarily stored on a server of the filesender service. This makes these susceptible to compromise there, e.g. by outside hackers or by staff of the filesender service. So when the files contain confidential information, e.g. commercial information or personal data, it is prudent to let the files be stored in encrypted form at the filesender service infrastructure. In some cases the filenames themselves can contain confidential information, e.g. in medical context where the filenames contains the name of the patient. For optimal protection it is best to also encrypt the filenames.

The simplest mechanism achieving such a secure filesender service (SFS) is by letting the sending user Sarah include a cryptographic key (long password) as part of Step U1. The filesender service uses this key to encrypt the files and deletes the key afterwards. Sarah then somehow provides the key to the receiving user Renee. Next Renee includes this key in Step U3 allowing the filesender service to decrypt the files prior to delivering them to Renee. This mechanism is simplest as it does not impose any encryption/decryption capabilities on the users. It is also the mechanism currently used in the "wetransfer" filesender service.[1] Such a mechanism does not provide for *end-to-end security* between sender and user, i.e. protecting both confidentially and the authenticity/integrity of the files sent Indeed, an attacker of the service could get access to the cryptographic key and read or manipulate the files sent. On the other hand, the mechanism allows the filesender service to inspect the files for "illegal content" also avoiding law suites against the filesender service. One can argue that inspection is required if the filesender service does not perform strict (sending) user authentication, e.g. in line with [2].

To provide for end-to-end security, the encryption, respectively the decryption, should take place in the environment of the sending, respectively receiving, user. This could be done by letting the encryption/decryption be done in separate, trusted client software. Such client software could take the form of a signed application or JavaScript code. We note that several JavaScript libraries exist allowing various cryptographic operations to be conducted inside the (sending/receiving) user internet browser, cf. [3]. Particularly interesting is the Web Cryptography API specification of the World Wide Web Consortium (W3C) [18]. This specification defines a low-level interface performing cryptographic operations in internet browsers through JavaScript. These operations are native, i.e. do not any additional JavaScript libraries to be loaded. Apart from security advantages, this also has performance advantages. Most modern internet browsers support Web Cryptography API. Compare `https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API`.

Letting the filesender service use such local encryption and decryption, theoretically allows for end-to-end security but still leaves the users with the classical problem of securely distributing the cryptographic keys. In general this problem can be solved by using *public key certificates*. Such certificates bind the identity of a person to a public key through a signature placed by a trusted party, i.e a certificate service provider. Such providers typically also maintain a directory of all certificates issued accessible by relevant parties, i.e. the sending users in our context. Here the sending user Sarah is provided with the certificate of Renee by

[1] Communication from wetransfer support desk, 3 April 2019.

the filesender service as part of Step U1. This then also allows Sarah to encrypt the files in Step U1 with the public key of Renee. Next Renee decrypts these files in Step U3 using her private key.

Although the use of public key cryptography and digital certificates solve the classical cryptographic key distribution problem it replaces it with the problem of setting up a suitable public key certificate infrastructure (PKI). It is vital that such a PKI provides a reliable binding between the public key and the user identity. Preferably, this binding is based on a face-to-face process whereby the user appears in person to a PKI employees allowing identity proving, e.g. based on an identity document. Compare [1]. However, setting up a suitable PKI is challenging and does not exist in most countries including The Netherlands.

This paper introduces a secure filesender service based an Remote Document Encryption (RDE-SFS) which is based on existing PKIs related to electronic passports, identity cards and driving licenses. This allows selecting certificates of intended receivers based on information (names, but also facial images) from their identity document vouched and signed by the government that issued it. These certificates then allow encrypting information for the receiver in such a way that decryption requires access to the identity document. In other words, RDE-SFS allows identity documents to be used as PKI smartcards.

1.2 Document outline

In Section 2 we functionally describe RDE-SFS. Section 3 discusses the cryptographic prerequisites required for RDE-SFS techniques specified. Section 4 specifies the RDE-SFS techniques themselves. Finally, Section 5 briefly discusses RDE-SFS implementation.

2 Functional description of RDE-SFS

In this section we functionally describe a secure filesender service based an Remote Document Encryption (RDE-SFS). We require that the secure filesender service meets the following high-level requirements:

1. both confidentiality and integrity/authenticity of the file contents and names sent are end-to-end protected,
2. the integrity/authenticity of the note is end-to-end protected,
3. the service supports for part-wise processing of (large) files.

For simplicity we only discuss passports, cf. [6]. However, the same applies to identity cards and driving licenses in many countries including The Netherlands, cf. [5]. Passports of most and all European countries include an electronic chip. This chip allows border control officers retrieving passport information through RFID (contactless communication). The protocols for this are specified by a United Nations agency called International Civil Aviation Organization (ICAO). All personal/identifying information that is physically printed on passports is also accessible electronically from the chip, electronically signed by the country that issued the passport. This information does not only include the first and last names of the citizen but also their date of birth, place of birth and even their facial image (in color). All this personal/identifying information is bound to separate electronic passports PKIs as specified by ICAO in [6]. These PKIs also allow passports proving their authenticity trough specific protocols. The corresponding public key certificates are retrievable from the passport and the private keys securely reside inside the electronic passport usable for the specific protocol only.

One of these PKIs is related to the Chip Authentication (CA) protocol. It is shown in [17] that by using a technique called Remote Document Encryption (RDE), the CA PKI can be used as a regular encryption PKI. That is, it allows any party encrypting data for the passport holder using the CA public key certificate such that this data can only be decrypted using the private key residing in the passport. This functionality is based on a tweak of the CA protocol and probably not intended by ICAO in [6]. RDE is based on *hybrid, authenticated encryption* whereby the actual data is symmetrically encrypted using an (authenticated) encryption algorithm and whereby the secret key itself is encrypted using asymmetric encryption. The asymmetric encryption is essentially based on the Diffie-Hellman key exchange protocol, cf. [16].

We now outline the basic working of RDE-SFS. As indicated in Figure 2 below, we introduce an *RDE-client* for receiving users, next to the internet browser. The RDE-client facilitates that a receiving user is able to interact with its passport through RFID. For simplicity one could think of the RDE-client as an mobile application on a mobile device of the receiving user that supports RFID. In the context of mobile application this is also known as Near Field Communication (NFC). In the RDE-client the complex part of RDE decryption takes places in interaction with the receiving user's passport. We remark that an RDE-client could also take other forms such as a plugin in the user internet browser or as an application next to the internet browser setting up a local server allowing communication with the browser. We further note that only receiving users require an RDE-client.

When receiving user Renee wants to be able to receive encrypted files through RDE-SFS, she first needs to register at the directory service within RDE-SFS. This consists of nine steps:

R1 Renee opens an account in the RDE-SFS directory service (hereafter: service) and accepts the (privacy) conditions thereof. This account will include information allowing Renee to be contacted, typically her email address.

R2 Renee is provided a choice which personal data on the passport is to be provided as part of registration.

R3 Renee consents to a particular choice of personal data to be registered.

R4 Renee lets her RDE-client read the consented data groups from her passport.

R5 The RDE-client suggests Renee a *mnemonic* allowing her to recognize the registered passport later.

R6 The RDE-client sends the consented data groups from Step 4 and the optional mnemonic from Step 5 to the service.

R7 The service validates the data read from the passport of Renee, including the passport issuer electronic signatures.

R8 The service forms and registers an *RDE-certificate* based on the personal data consented in Step 4. If Renee consented to a Mnemonic then the RDE certificate is registered under the Mnemonic.

R9 The service securely wipes all data gained in the registration process other then is required for the RDE-SFS functionality.

For simplicity the steps above only correspond with the *happy flow* and does not deal with the erroneous situations. A real life implementation of these steps should also deal with these situations. An important choice in Step R3 is whether the facial image data is registered or not. The data groups read in Step R4 minimally consist of data group 15 (holding the chip authentication key), but typically also includes data group 1 (holding naming information). If Renee consented to registering her facial image also data group 2 is read. If the passport supports Active Authentication (AA, compare [6,17]) then consent evidence can be generated by placing the consented data in a separate object and having that signed using the AA protocol. As this protocol only allows for signing 8 byte "hash values", cf. [?], it is best to include a RDE-SFS directory service session identifier in the signature. When AA is used, then the certified AA public key (data group 15) should also be retrieved in Step R4. Moreover, the RDE-SFS directory service should validate the AA signature in Step R7.

In a naive implementation an RDE-certificate would simply be formed as *all* information read from the user passport signed by the country that issued the passport including the CA public key. However, this would imply that users need to reveal *all* information printed on their information such as full first and last names, date and place of birth and their facial image. The legal basis for data processing of the directory user is consent of the user and users would typically not consent to all this information being revealed to other users. Moreover, in some countries, including the Netherlands, the passport also contains the user social security number which is legally not allowed to be revealed to other people. However, if one does not reveal all information, then the signature of the country that issued the passport cannot be validated by the sending user. To remedy this and to allow for flexibility we let the directory service validate the relevant information read from the passport, i.e. the signatures of the country that issued the passport, but then forms its *own* RDE certificates based on that. This happens in Step R6 of the registration process. These RDE certificates bind the CA public key read from the passport together with personal information read from it that the user is willing to share with others. These RDE certificates also binds to other identifiers of the user, most notably to its email address. Depending of the contractual arrangements, the original data read form the passport could either be deleted or (securely) archived for dispute handling. The directory service is not required to use its own certificates and could simply store the relevant user data, including the CA public key. However, this would make the directory vulnerable to compromise.

The mnemonic in Steps R7 and R8 allows Renee to recognize the passport registered in the decryption phase. This is particulary helpful when Renee has multiple passports or when more identity documents could be used in the service. In the Netherlands both the national identity card and driving licence also support RDE. The directory service could suggest a mnemonic, e.g. the last three characters of the document number. Alternatively, one could let the user choose the mnemonic.

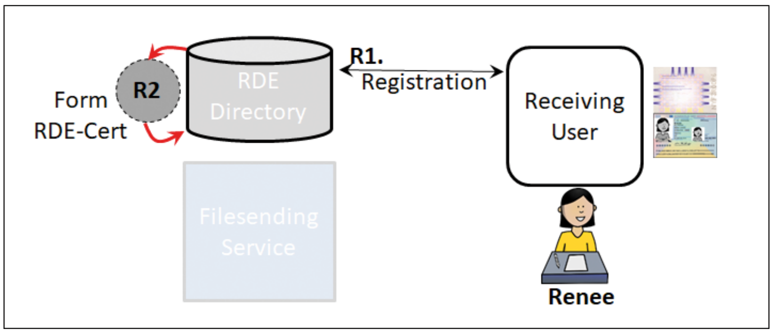

Figure 2. RDE-SFS user registration

After registration is completed for a user, the registered RDE-certificate can now be used as indicated in Figure 3. The RDE-SFS encryption process consists of the following steps in the happy flow, i.e. when no issues or errors occur:

E1 The sender connects her browser to the RDE-SFS server. She optionally needs to authenticate before she can proceed. The sender enters the email address or other identifying information on the intended recipient.

E2 The RDE-SFS server queries the RDE directory service and presents the relevant RDE-certificate(s) if that exists. The sender inspects the offered certificates, e.g. by comparing the name information or the facial image.

E3 The sender selects the RDE-certificate of the intended recipient. The sender writes an optional explanatory note and selects the files she wants to send to the recipient.

E4 The browser generates an RDE session key S and an RDE encryption of S. The browser uses S to supplement the note with an authentication tag. The sender browser generates a list holding the names of the files selected. The browser then encrypts the filename list using S. Next the browser RDE encrypts the contents of each the selected files using S.

E5 The browser sends the results from the previous step in an RDE session key file, a note file, a metafile and encrypted files to the RDE-SFS server.

It might be prudent requiring users to authenticate before access to the RDE-SFS server as indicated in the Step E1. Indeed, senders get access to personal data of other participants in Step 2 which is best to be protected. Moreover, such authentication ensures that the files sent in Step E5 are authenticated. In Step

E5 it seems simplest to replace the filenames with temporary sequential ones, e.g. File1, File2 et cetera protecting the original names in the encryption process. In the happy flow the RDE-SFS decryption process consists of the following steps:

D1 The receiver is notified of the files available on the RDE-SFS server. The explanatory note is also part of the notification but not supplemented with an authentication tag. Part of the notification is a URL on the RDE-SFS server.

D2 The receiver clicks the URL and connects her browser to the RDE-SFS server. The explanatory note is shown (again) and the number of available (encrypted) files. The receiver accepts the sent files.

D3 The browser collects the RDE encrypted session file and generates on the fly a public-private key pair. Next the browser sends the contents of the RDE encrypted session file and the public key to the RDE-Client, e.g. through a QR code.

D4 The RDE-client indicates the passport required (mnemonic) to the receiver and interacts with the receiver's passport to extract the RDE session key.

D5 The session key is encrypted by the RDE-client with the browser public key from step D3.

D6 The encrypted session key is sent to the browser through the RDE-SFS server.

D7 The browser retrieves the explanatory note including authentication tag and the encrypted filename list L. The browser obtains the session key by using the private key from Step D3 and uses this to verify the authentication tag on the explanatory note and to decrypt the filename list L. Next the browser RDE downloads each of the encrypted files, decrypts them writing them under the name in the filename list. The decryption process ensures that that the names of the files sent match those received.

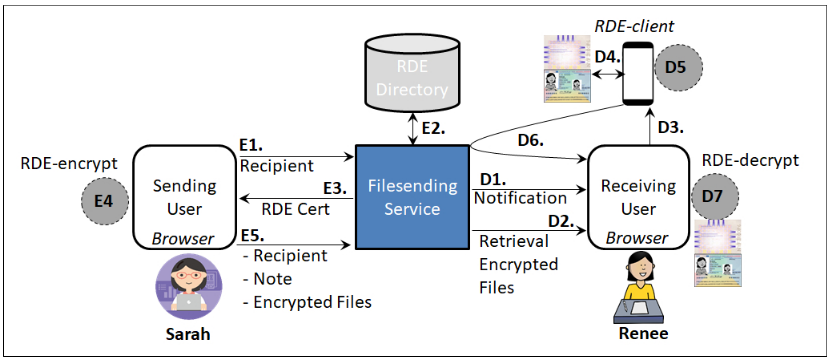

Figure 3. RDE-SFS usage

3 Cryptographic prerequisites

3.1 AES encryption

In version 1 of the RDE-SFE specification, the encryption of data is based on AES deployed in the so-called Galois/Counter Mode (GCM) mode [13]. This is an algorithm for authenticated encryption with associated data, i.e. protecting both confidentiality and authenticity/integrity of data. The rationale for this choice is also that AES-GCM is part of the Web Cryptography API and natively supported in regular internet browsers. Apart from encryption AES-GCM allows additional data as input that is not encrypted but only authenticated. AES-GCM is based on AES-CTR [12] protecting confidentiality supplemented with an AES based message authentication code (GMAC).

The AES-GCM algorithms allows for various configurations. The input of the AES-GCM algorithm consists of a key K, an initialisation vector IV, plaintext data D and additional authenticated data A. The output the AES-GCM algorithm consists of encrypted data C and an authentication tag T. In essence, the AES-GCM algorithm first encrypts the data D deploying AES in so-called Counter mode (AES-CTR, cf. [12]) using the initialisation vector. Next, AES-GCM runs the encrypted data and the additional data A through a AES based Message Authentication Code function called GMAC which forms the authentication tag T. The AES-GCM decryption algorithm is very similar to the AES-GCM encryption algorithm. As part of the decryption the authentication tag is verified and if this is not successful the decryption process fails and delivers no plaintext.

In calling the AES-GCM algorithm several times with the same key K it is important that initialisation vector IV is different as otherwise the encrypted data leaks secret information. Analysing the GCM specifications, the simplest way to achieve this is to use a 96 bit (12 byte) initialisation vector that incremented in each call. That is, the first call uses an initialisation vector that is equal to 0 (considered as a 96 bit number), then 1 et cetera. In the AES-GCM application in RDE-SFS the key K is chosen randomly in each session, allowing to choose predicable initialisation vectors.

We let $(C, T) = \mathrm{ENC}(IV, K, D, A)$ denote the AES-GCM encryption of plaintext data D and additional authenticated data A under key (256 bit) K using a (96 bit) initialisation vector IV. In version 1 of the RDE-SFE specification we choose the maximal authentication tag length (128 bit, i.e. 16 byte) and key size (256 bit, i.e. 32 byte). We also let $\mathrm{DEC}(IV, K, T, C, A)$ denote the corresponding AES-GCM authenticated decryption of ciphertext C and additional authenticated data A based on initialisation vector IV, key K and authentication tag T. This decryption can fail if the authentication tag is incorrect in which no plaintext is returned.

If the plaintext data D is empty then so is the encrypted data C and the authentication tag is (by definition) the GMAC authentication tag on the additional authenticated data A. That is, an AES-GCM implementation also conveniently caters for a MAC algorithm. We note that the GMAC value is independent of the initialisation vector. This will also be used in the version 1 of the RDE-SFE specification as indicated in the next section.

3.2 Hashing and Message Authentication Codes

In the RDE-SFS specification we use a secure hash function $\mathcal{H}(.)$. As part of the version 1 RDE-SFS specification we stipulate the use of the SHA-256 hash function [9]. We also require the generation Message Authentication Codes to protect the authenticity and integrity of data. As part of the version 1 RDE-SFS specification we specify the use of the GMAC algorithm as this is conveniently available from the used AES-GCM implementation. We will use the maximal authentication tag length (128 bit, i.e. 16 byte) and denote the computation of the GMAC value under key K by $\mathcal{M}(K, .)$. The GMAC keys we use are 256 bits in size. We assume that the data input of $\mathcal{M}(K, .)$ is a byte array but we loosely allow using string input also. In the latter case, the string is assumed to consist of printable ASCII characters considered as a byte array of the same length consisting of the string ASCII byte values, i.e. without the closing zero byte.

3.3 Elliptic Curve based Diffie-Hellman

In the RDE-SFS specification we use the so-called Diffie-Hellman key exchange protocol to allow secure communication between the receiver browser and its RDE-client. See [16]. In these protocols, an additive group $\mathbb{G} = (\langle G \rangle, +)$ of order q generated by a generator element G plays an important role. We use additive notation as this is customary in the context of elliptic curve groups we deploy in practice. We assume that q is prime. For any natural scalar n and element $H \in \langle G \rangle$ we define the *(point) multiplication* nH as adding H n-times, e.g. $2H = H + H$. As $nH = mH$ if and only if $n = m \bmod q$ we can represent scalars as elements of $\mathbb{F}_q$. A randomly, or cryptographically secure pseudo randomly, chosen element from a set is denoted by $\in_R$. We assume that the group $(\langle G \rangle, +)$ satisfies the usual cryptographic security properties, i.e., that the so-called discrete logarithm, the Diffie-Hellman and the Decision Diffie-Hellman problems are intractable. See [4]. In the Diffie-Hellman key exchange protocol two parties Alice and Bob each generate a public/private key pair. For instance, Alice generates public key $A = aG$ with private key $a \in_R \mathbb{F}_q^*$ and Bob generates public key $B = bG$ with private key $b \in_R \mathbb{F}_q^*$ Next they sent each other their public keys from which both can compute the shared key $S = abG = aB = bA$. This shared key can then be used for secure communication between Alice and Bob. From S an AES-GCM key is derived. In the basic Diffie-Hellman setup Alice and Bob choose fresh public keys A, B as part of the secure communication setup. In a more advanced setup, called half-certified Diffie-Hellman cf. [8, Protocol 12.5], the public key A of Alice is fixed and wrapped in a certificate together with information identifying Alice. Only Bob chooses a fresh public key B and private key b which are also known as *ephemeral* keys.

As indicated in the next section RDE is based on Elliptic Curve based Diffie-Hellman (ECDH) where the group is stipulated by the passport of the receiver user. Modern passports use elliptic curve groups based on one of the NIST curves [11] or Brainpool curves [7]. Current European passports, including the Dutch ones, are based the Brainpool320r1 curve. As we are required to perform ECDH operations with the curve used in the passport in the sender browser, we also use the same curve for the Diffie-Hellman key exchange protocol in the receiver browser. That is, in the RDE-SFS specification we only use one elliptic curve group $(\langle G\rangle, +)$. As also indicated in Section 5 the Web Cryptography API only supports the NIST curves so we need to partially use a separate JavaScript library for ECDH in RDE-SFS implementations.

3.4 RDE encryption session keys and key derivation

We refer to [17] for a general background on Remote Document Encryption (RDE). RDE is a tweak of the earlier mentioned Chip Authentication (CA) protocol from part 11 of [6]. In essence the CA protocol consists of setting up a secure messaging channel between passport and a reader application based on the Diffie-Hellman key exchange protocol.

As follows from [17], Remote Document Encryption (RDE) is a tweak of the Chip Authentication (CA) protocol which is based the half-certified Diffie-Hellman key exchange protocol. In the CA protocol the passport holds the certified public key and the application reading the passport chooses an ephemeral key. As in Section 3.3 we let $(\langle G\rangle, +)$ denote the group used in the passport as part of the CA protocol. Current European passports, including the Dutch ones, are based on the elliptic curve group based on the Brainpool320r1 curve, cf. [7].

RDE is based on Protocols 4 and 5 from [17] according to RDE input parameters P. These parameters consist of the triple $(n, F_{\text{Id}}, F_{\text{Cont}})$ introduced in [17]. Here F_{Id} identifies a data group and n is the number of bytes to read from it as part of RDE. F_{Cont} represents the full contents of this data group. It is indicated in [17] that n needs not be too large ensuring the bytes read fit a standard response of 255 bytes. Taking into account secure messaging overhead, [17] indicates n should be less than 223 for AES based secure messaging and less than 231 for DES based secure messaging. The choice of n is further upper bounded by the number of bytes available in the data group to be read.

Protocols 4 and 5 from [17] result into an output triple $(Z, \overline{\text{RB}_\text{s}}, \bar{M})$. Here Z is an ephemeral public key, i.e. a point on the elliptic curve the Chip Authentication (CA) protocol is based on. $\overline{\text{RB}_\text{s}}$ is a protected READ BINARY command $\overline{\text{RB}_\text{s}}$ and $\bar{M}$ is the protected response including the status words. This includes both the protected and unprotected ones, i.e. 4 bytes in total. We let that these three parts be represented as byte arrays.

The protected response $\bar{M}$ is the basis for the RDE encryption. Essentially this consists of an encryption with a block cipher $E_K(.)$ of the data read and an 8 byte Message Authentication Code (MAC) over the result. Compare Part 11 of [6]. The block cipher is either Triple DES (3DES) or AES and the MAC algorithm is either CMAC or based on CBC. Modern passports are based on

AES and CMAC. For this the CA protocol uses the Diffie-Hellman key exchange protocol to derive two suitable keys $K = K_{\text{ENC}}, K_{\text{MAC}}$.

As part of this RDE-SFS specification (version 1) we let F_{Id} be the data group holding the CA public key, i.e. data group 14. Moreover we let $n = 1$, i.e we only read one byte from data group 14, which is always equal to 0x6E. This byte corresponds to the data group "Tag" which is equal to the hexadecimal nibble 6 followed by the data group number hexadecimal notation, i.e. E. One can easily derive that the encrypted content in the case the blockcipher is AES equals:

$$\text{AES}_K(\text{AES}_K(\texttt{00000000000000000000000000000001}) \oplus \texttt{6E800000000000000000000000000000}).$$

In case the blockcipher is 3DES the encrypted content equals:

$$\text{3DES}_K(\text{3DES}_K(\texttt{0000000000000001}) \oplus \texttt{6E80000000000000}).$$

Here the inside encryptions correspond with the initialisation vector based on the (first) Send Sequence Counter. The byte arrays on the right corresponds with the first byte (0x6E) of the data group followed with padding i.e. 0x80 followed with zero bytes filling the blocksize.

Using a message authentication function as part of a key derivation function is well accepted, cf. [14]. Hence the 8-byte MAC value used can be considered suitable cryptographically random. It is well accepted that 3DES and AES are pseudorandom permutations (PRP) implying that for a random K (appropriate for DES/AES), the functions $AES_K(.)$ $DES_K(.)$ cannot be distinguished from a random permutation on the set of byte arrays of length 16. This implies that the inside encryptions are suitably cryptographically random. In total the AES encryption and the CMAC constitutes to 16+8=24 bytes, i.e. 192 bits. The DES encryption and CBC-MAC constitutes to 8+8=16 bytes, i.e. in a 128 bit RDE session key. Both are suitable large. As we remarked earlier, modern passports are based on AES and CMAC where RDE-SFE resulting in a 192 bit RDE session key. Modern passports are based on the 320 bit Brainpool320r1 curve implying that the resulting RDE security corresponds with 160 bit.

For future reference we specify in Algorithm 1 the generation of a RDE Session Key and its encrypted form. In Algorithm 2 we specify how the RDE Session Key can be decrypted from its encrypted form. Algorithm 3 specifies how we use the RDE Session Key to derive several other encryption and authentication keys from.

Algorithm 1 *GEN_RS*(): Generation of random RDE Session Key S and its RDE-encryption $(P, Z, \overline{\text{RB}_\text{s}})$

```
1: Apply protocol 4 from [17] and implied RDE-parameters $P$ resulting
   in byte arrays: an ephemeral public key $Z$, a protected READ BINARY
   command $\overline{\text{RB}_\text{s}}$ and a protected response $\bar{M}$ (including all 4 status words)
2: Return $\bar{M}$ as the RDE Session Key $S$ and $(P, Z, \overline{\text{RB}_\text{s}})$ as its
   RDE-encryption
```

Algorithm 2 $DEC\_RS(P, Z, \overline{\mathrm{RB_s}})$: Decryption of RDE-encrypted session key S from $(P, Z, \overline{\mathrm{RB_s}})$

```
1: Apply protocol 5 from [17] according to RDE-parameters $P$ using the
   passport of the user, the ephemeral public key $Z$, and the protected
   READ BINARY command $\overline{\mathtt{RB_s}}$ resulting in a protected response, a byte
   array $\bar{M}$ (including all 4 status words).
2: If Step 2 is not successful return Error                    // Bad input
3: Return $\bar{M}$ as the RDE Session Key $S$
```

Algorithm 3 $K_{\mathrm{AE}}(S, i)$: Generation of the i-th authenticated encryption key from session key S (byte array)

```
1: If $i > 2^{32}$ return error
2: Represent $i$ as a unsigned integer in a 4 byte array $I$
3: Return $K_{\mathrm{AE}}(S, i) = H(S \,||\, I)$
```

4 Specification of RDE-SFS encryption and decryption

In this section we specify the cryptographic techniques deployed within RDE-SFS. We make some specific implementation choices to simplify the presentation. Various variants of the specification outlined are possible. The first choice we make is that we assume that the files selected by the sender are non-empty. Empty files support can be easily arranged but requires various exception handling, complicating the presentation.

The result of any successful RDE-SFS encryption process of the sender is an RDE-SFS session identifier *RID* and a number of non-empty associated files on the RDE-SFS server. The resulting filenames take the format "R_*i*_*j*" where the "R" stands for Result and the i, j are positive integers denoted in their ASCII representation, e.g. "R_1_1". The first integer i denotes the *file sequence number* whereas the j denotes the *part number*. The part numbers are consecutive, e.g. a file R_2 could be decomposed into three partial files R_2_1, R_2_2, R_2_3. Each RDE-SFS session contains a parameter MaxB holding a natural number. All parts except the last part consist of MaxB 16 byte blocks, whereas the last part contains less or equals MaxB 16 byte blocks. This block size of 16 bytes corresponds with the AES block size allowing convenient buffer-wise encryption and decryption.

As indicated in Table 1 some result files are not be divided in parts and MaxB should be large enough to support for this. A minimal choice of MaxB of 640 representing 10 megabytes of data in a part seems an appropriate choice. As the files selected by the user are assumed non-empty, so are the result files. Using parts allows decomposing a large file in smaller units. We will simply talk about (result) file R_*i* effectively meaning the consecutive concatenation of all its parts. Each result file R_*i*_*j* has an associated (and undivided) Authentication Tag which will be contained in an additional file with name AT_*i*_*j*. The contents of AT_*i*_*j* is an AES-GCM authentication tag of the data (bytes) of the result file prepended with the name of the file, i.e the string "R_*i*_*j*" considered as a byte array. In this setup each part of the result file has it own authentication tag. One can argue that is more efficient to have only one authentication tag for the whole file R_*i*, i.e. the concatenation of all parts. We have this not chosen for the one HMAC approach for two reasons. Firstly, the chosen approach allows the receiver browser to verify the authentication tag of each part. This allows early breaking off the decryption and does require the browser to process all parts. This is an advantage for large files in particular. Secondly, the one tag approach assumes that the user browser is able to compute an AES-GCM tag incrementally. That is, that the browser keeps internal state that is updated with each file part being processed and finalized with the last part. Although this is commonly supported in most cryptographic libraries, it is not customary in JavaScript cryptographic libraries most notably the Web Cryptography API [18].

Sequence Number	**Meaning**	**Encrypted**	**Result file**	**AT file**
1	Version File	No	R_1_1	AT_1
2	RDE session key file	Yes (implicit)	R_2_1	AT_2
3	Note File	No	R_3_1	AT_3
4	Metadata File	Yes	R_4_1	AT_4
$i \geq 5$	Actual files sent	Yes	R_i_*	AT_i_$*$

Table 1. Semantics of RDE result files

The first file resulting from a successful RDE-SFS encryption process will be the *version file* R_1_1. This file consists of one part only, i.e. only consists of file R_1_1, and holds the version number of the RDE-SFS implementation. In this document we specify the first RDE-SFS version in which the version file only holds the ASCII character 1. The version file is not encrypted but has an associated authentication file. In the second RDE-SFS version this will hold the ASCII character 2 and so on. The version file allows the receiver's browser to decide the specific implementations choices made. Everything we specify from this point forward relates to RDE-SFS version 1.

The second file R_2_1 is the *RDE session key file* holding the RDE encrypted session key allowing the receiver to decrypt the sent files. The third file F_4 is the *note* file holding the explanatory note in plaintext. The fourth file R_3_1 is the *metadata file*, holding in encrypted form the metadata associated with the RDE-SFS encryption process. It holds the number n of files in the RDE-SFS encryption instance, the original names and for each file the number of parts it consists of. The metadata file also holds the number MaxB discussed earlier. From the metadata file the receiver's browser is able to determine all (partial) files existing in the RDE-SFS instance at the RDE-SFS server.

The fifth file F_5 and further contains the actual (large) files sent by the sender in the order they where processed by the sender's browser. That is, the name of the i-th processed file ($i = 1, 2, \ldots, n-1$) will occur as the i-th name in the list in the metadata file and will be stored in file parts F_4+i_j. Each such file has an authentication tag placed in AT_4+i_j.

In the current description, the receiver only gets information on the files sent as part of RDE decryption. With reselect to file names this seems an appropriate choice. However, one might argue that the receiver should at least should get information on the number of files sent and their size. If this is desired, we suggest placing that information in a mandatory part of the note file.

For simple presentation of the RDE-SFS encryption protocol 3 and decryption protocol 4 it is convenient to specify two building blocks dealing with the RDE encryption and decryption of files, in the browser of the sender and receiver. In these protocols $\mathrm{BN}_{96}(i,j)$ denotes the 16 byte array representation of the integer

$$i{\cdot}2^{32} + j$$

for $0 \leq i \leq 2^{96}-1$ and $0 \leq j \leq 2^{32}-1$. This can alternatively be described as the byte representation of integer i in 12 byte (96 bit) concatenated with the byte representation of integer j in 4 byte (32 bit).

Protocol 1 A-ENC-FILE($F, i, \mathrm{MaxB}, K_{\mathrm{AE}}$): Authenticated encryption of file F with sequence number $1 \leq i < 2^{70}$ using key K_{AE} storing in maximal MaxB 16 byte blocks at RDE-SFS server.

```
 1: If $F$ is empty, then return error
 2: Set $j = 1$
 3: Try reading MaxB 16 byte blocks bytes from file $F$ in PlainBuf
 4: If PlainBuf is empty go to Line 11
 5: Form $(C, T) = \mathtt{ENC}(\mathtt{BN}_{96}(i, j), K_{\mathrm{AE}}, \mathtt{PlainBuf}, \texttt{"R\_}i\texttt{\_}j\texttt{"})$
 6: Store EncBuf in partial file R_$i$_$j$ at RDE-SFS server
 7: Store $T$ in file AT_$i$_$j$ at RDE-SFS server
 8: $j = j + 1$
 9: If $j = 2^{32}$ return error                          // File too big (not likely)
10: Go to Line 3
11: Return information on stored partial files
```

In Protocol 2 we specify RDE-SFS encryption of one file and storing it in the browser local filesystem.

Protocol 2 A-DEC-FILE($\{\mathrm{R\_}i\mathrm{\_}j\}_{j=1}^{l}, \{\mathrm{AT\_}i\mathrm{\_}j\}_{j=1}^{l}, \mathrm{MaxB}, \mathrm{Name}, K_{\mathrm{AE}}$): Authenticated decryption using keys K_{AE}, partial files $\{\mathrm{F\_}i\mathrm{\_}j\}_{j=1}^{l}$ and $\{\mathrm{AT\_}i\mathrm{\_}k\}_{j=1}^{l}$ in maximal MaxB 16 byte blocks storing in file in browser under filename Name.

```
1: If any of the $\{\mathtt{R\_}i\mathtt{\_}j\}_{j=1}^{l}, \{\mathtt{AT\_}i\mathtt{\_}j\}_{j=1}^{l}$ is empty, then return error
2: If $l > 2^{32}$ , then return error     // too much result files (not likely)
3: Create empty file with name Name in browser local filesystem
4: For $j = 1$ to $j = l$ do
5:    Try reading Max 16 byte blocks bytes from file R_$i$_$j$ in EncBuf
6:    Read contents $T'$ from file AT_$i$_$j$
7:    Compute PlainBuf $= \mathtt{DEC}(\mathtt{BN}_{96}(j), K_{\mathrm{AE}}, T', \mathtt{EncBuf}, \texttt{"R\_}i\texttt{\_}j\texttt{"})$ and return error
      on authentication failure
8:    Concatenate PlainBuf to file with name N in browser local filesystem
9: Return information on stored partial files
```

In Protocols 3 and 4 we specify the RDE-SFS encryption and decryption protocol. As indicated in Section 3.2 we use GMAC as message authentication code algorithm. A GMAC value can be computed by only feeding the AES-GCM algorithm with additional authenticated data and no data to encrypt. This means that the routines $\mathcal{M}()$ and A-ENC() mentioned below are closely related. For clarity we have chosen different notation.

Protocol 3 RDE-SFS encryption

```
1: Sender connects her browser to the RDE-SFS server and authenticates
2: Browser and RDE-SFS server negotiate MaxB, the maximal number of 16 byte
   blocks processable in the Browser  // could be fixed in implementation
3: Sender enters identifying data of receiver and selects RDE certificate Cert
4: RDE-SFS server creates TransactionId                 // link to results
5: Sender writes explanatory note N and selects files with names $F_1, F_2, \ldots F_n$
6: Browser calls Algorithm 1 to get session key $S$ and its RDE-encryption $(P, Z, \overline{\texttt{RB}_\texttt{S}})$
   // Calls to Algorithm 3 for derived keys refered to below
7: Browser forms byte contents $V$ of version file, computes $\mathcal{M}(K_{\text{AE}}(S, i), \texttt{"R\_1\_1"}||V)$
   and stores these in files R_1_1 and AT_1 respectively
8: Browser forms byte contents $R$ of RDE session key file, computes
   $\mathcal{M}(K_{\text{AE}}(S, 2), \texttt{"R\_2\_1"}||R)$ and stores these in files R_2_1 and AT_2 respectively
9: Browser forms byte contents $N$ of the Note file, computes $\mathcal{M}(K_{\text{AE}}(S, 3), \texttt{"R\_3\_1"}||N)$
   and stores these in files R_3_1 and AT_3 respectively
10: For $i = 1$ to $n$ do
11:   Browser runs A-ENC($F_i, i$, MaxB, $K_{\text{AE}}(S, 4{+}i), K_{\text{AE}}(S, 4{+}i)$) of Protocol 1 encrypting
      and storing the results in parts
12: Browser forms byte contents $M$ of metadata file including MaxB, computes
    $(M', T)$=ENC(BN$_{96}$(1), $K_{\text{AE}}(S, 4), M, \texttt{"R\_4\_1"}||M$) storing $M', T$ in files R_4_1, AT_4
13: Browser returns information on processed information
```

In Step 5 of Protocol 4 below we let the browser generate a Diffie-Hellman public/private key pair. For implementational simplicity we let the this key pair be based on the same elliptic curve group the core RDE based encryption is based on. The rationale for this is that the sender browser software is required to be able to deal with this elliptic curve group anyway. For European passports this means that the Diffie-Hellman public/private key pair is based on one of the Brainpool groups. As these groups are not supported by the W3C Web Cryptography API we require separate (JavaScript) code for this, e.g. a part of the Stanford Javascript Crypto Library, cf. [15]

Protocol 4 RDE-SFS decryption

```
1: Receiver is notified through URL on the note and files available
2: Receiver follows URL connecting browser to stored files under TransactionId
3: Browser reads version file R_1_1 and gives error if version is unsupported
4: Browser reads RDE session file R_2_1 and AT_2
5: Browser generates ECDH public key $U = uG$ and private key $u \in_R \mathbb{F}_q^*$
6: Browser interacts with RDE-client providing it TransactionId,$U$ and R_2_1
   contents
   // Could be interaction based on QR-code
7: RDE-client indicates the passport required (mnemonic) and interacts with
   receiver passport obtaining RDE session key $S$
8: RDE-client generates ephemeral key $E$ and Diffie-Hellman shared secret $D$
9: RDE-client computes $(S', T)$=ENC(BN$_{96}$(1), $K_{\text{AE}}(D, 1), S$, "") and sends $E, S', T$ to
   RDE-SFS server linking to TransactionId
10: Browser is provided $E, S', T$ by RDE-SFS server  // through TransactionId
11: Browser computes $D$ using $E$ and private key $u$ from Step 5
12: Browser computes $S$ = DEC(BN$_{96}$(1), $K_{\text{AE}}(D, 1), T, S'$, "") returning error on
    authentication failure
13: Browser computes $(IV', S)$ = DEC(1, $K_{\text{AE}}(D, 1), S'$)
```

```
14: Browser forms $K_{\text{AE}}(S, 1)$ and validates version file using AT_1
15: Browser forms $K_{\text{AE}}(S, 2)$ and validates RDE file using AT_2
16: Browser forms $K_{\text{AE}}(S, 3)$ and validates Note file using AT_3
17: Browser forms $K_{\text{AE}}(S, 4)$ and validates encrypted full metafile using AT_4
18: Browser forms $K_{\text{AE}}(S, 4)$, decrypts full metafile and obtains list of stored
    partial files F_$i$_$*$, AT_$i$_$*$ and list of original file names $F_1, F_2, \ldots F_n$
19: For $i = 1$ to $n$ do
20:     Browser runs A-DEC($\{$F_$i$_$j\}_{j=1}^{l}$, AT$_i$_$j\}_{j=1}^{l}$, MaxB, $F_i$, $K_{\text{AE}}(S, 4+i)$, $K_{\text{AE}}(S, 4+i)$
        of Protocol 2 decrypting and resassembling orginal files
21: Browser returns information on processed information
```

We remark that the setup of RDE-SFS can also easily be adapted to situation without RDE, i.e. when sender and receiver share a key (password) P. In this situation, the sender browser generates a random nonce N at least 16 byte, place this in the result file R_2_1 and uses a session key $S = \mathcal{H}(P||N)$ instead of the RDE based session key. Steps 5-11 in Protocol 4 are then not necessary. By this approach an RDE-SFS can support passwords too.

5 RDE-SFS implementation suggestion

We suggest the RDE certificates to be based on ECDSA using one of NIST curves, e.g. P-384. We suggest implementing all RDE-SFS cryptography, other than ECDH/core RDE, with the Web Cryptography API in browers, cf. [18]. As the Web Cryptography API also supports ECDSA based on the NIST curves, verification of the RDE certificates in the sending browser is also conveniently possible. For ECDH and core RDE we suggest using the ECDH routines from the Stanford Javascript Crypto Library, cf. [15]. These routines allow using various elliptic curves through their simplified Weierstrass equation which will allow support of the Brainpool curves as commonly used in European passports.

6 References

1. ETSI, EN 319 411-1, Electronic Signatures and Infrastructures (ESI); Policy and security requirements for Trust Service Providers issuing certificates; Part 1: General requirements, V1.2.2, April 2018.
2. European Commission (2015), eIDAS implementing regulation 2015/1502 on setting out minimum technical specifications and procedures for assurance levels for electronic identification, September 2015.
3. Github, See https://gist.github.com/jo/8619441; retrieved 03/04/2019.
4. Darrel Hankerson, Alfred Menezes, Scott Vanstone, Guide to Elliptic Curve Cryptography, Springer-Verlag Berlin, Heidelberg, 2003.
5. International Organization for Standardization (ISO), ISO/IEC 18013-3, Information technology — Personal identification — ISO-compliant driving license — Part 3: Access control, authentication and integrity validation, second edition 2017-04.
6. International Civil Aviation Organization (ICAO), Doc 9303 (multiple parts), Machine Readable Travel Document, Seventh Edition, 2015.
7. Internet Engineering Task Force (IETF), ECC Brainpool standard curves and curve generation, RFC 5639, March 2010.
8. Alfred J. Menezes, Scott A. Vanstone, Paul C. Van Oorschot, Handbook of Applied Cryptography, CRC Press, 1996.
9. NIST, Secure Hash Standard (SHS), FIPS PUB 180-4, August 2015.
10. NIST, Advanced Encryption Standard (AES), FIPS 197, November 26, 2001.
11. NIST, Digital Signature Standard (DSS), FIPS 186-4, July 2013.
12. NIST, Recommendation for Block Cipher Modes of Operation, Special Publication 800-38A, December 2001.
13. NIST, Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, Special Publication 800-38D, November, 2007.
14. NIST, Recommendation for Key Derivation Using Pseudorandom Functions, SP 800-108, October 2009.
15. Stanford Javascript Crypto Library, cf. http://bitwiseshiftleft.github.io/sjcl/.
16. D.G. Stinson, Cryptography: theory and practice, CRC press, 1995.
17. Eric Verheul, Remote Document Encryption - encrypting data for e-passport holders, 9 Jun 2017. Available from: https://arxiv.org/abs/1704.05647.
18. Web Cryptography API, World Wide Web Consortium (W3C), https://www.w3.org/TR/WebCryptoAPI/.