EHEALTH SERVICES - WEB ACCESS

INTEGRATING EHEALTH IN A MOBILE WORLD

Contents

Scope ...3
Requirements ...3
Identity & Access Management ...3
Information Security ...4
Architecture ...6
Protocols and Profiles ...6
RESTful API ...6
Identity & Access Management ...6
Information Security ...7
Flows ...8
Identity & Access Management ...8
Information Security ...14
Security Recommendations, Risks & Known Limitations ...25
Trust Model ...25
Federated Identity ...25
Point-to-Point Information Security ...25
CA vs Self-Signed ...25
EHealth Certificates/ETKs ...25
validation ...26
Non-Repudiation ...26
Encryption with public keys ...26
multiple devices ...26
Long term storage ...27
Bearer vs Holder-of-Key ...27
Authorization Code Exchange ...28
SOAP vs REST ...28
Token Exchange ...28
Cryptolib ...30

Web Authentication API 31
References 32

Figures

Figure 1. Authorization Code - Public Client *10*
Figure 2. Authorization Code - Confidential Client *12*
Figure 3. Authorization Client Credentials *13*
Figure 4. Public Key Registration *15*
Figure 5. Symmetric Key Registration *16*
Figure 6. Known Addressee - Synchronous Communication *18*
Figure 7. Known Addressee - Asynchronous Communication *20*
Figure 8. Unknown Addressee *22*
Figure 9. Token Exchange - Public Client *29*
Figure 10. Token Exchange - Confidential Client *30*

SCOPE

This document describes solutions to setup the eHealth services of tomorrow to ease integration of citizens, health professionals and organizations in the context of their work, regardless of environment or device used to connect to those services.

REQUIREMENTS

IDENTITY & ACCESS MANAGEMENT

In order to allow mobile access to eHealth services, we must be able to authenticate ALL users that need to use eHealth services, regardless of device or system used to connect.

We can mainly split the type of users for eHealth services into 2 categories.

1. Persons: Belgian citizen/Foreigner, Professional, Member of Organization, Mandatee

2. Systems

It must be possible to construct a digital identity for any of those.

REGISTRATION

- All users must be registered in an authentic source, available to eHealth (either directly or indirectly)

    - Persons in the National Register with a SSIN (Belgian citizens) or SSIN BIS (Foreigners). Target groups for eHealth include Belgian citizens and foreigners that live here or across the border.

    - Systems must belong to an organization which can be uniquely identified in an authentic source for the particular type of organization.

- Every user must have means to prove his identity online with a digital key. At least one key must be handed to him upon registration.

AUTHENTICATION

- Authentication must be supported for all kind of clients: web (browser), native (mobile app), desktop, server (backend, batch).

    - All clients with user interaction must support web-based authentication. Native or desktop applications need to support the 'AppAuth Pattern' so the web authentication flow can take place in an external user-agent (browser) of the device/pc. This is considered best current practice (RFC8252) for native apps. At the end of the authentication flow, an identityToken must be available for direct calls from the client to the eHealth service(s) in order to prevent the need for re-authentication at every call.

- To authenticate, a user needs to use on of his digital keys to prove he is who he claims to be. EHealth's federated identity model must be reusable for all users.

- All digital keys must meet minimum safety requirements.

    - For persons, the European Eidas regulation defines the levels LOW, SUBSTANTIAL, HIGH. It must be possible for each eHealth Service to decide which level is sufficient.

    - For systems, only asymmetric keys are allowed.
        - All keys need to be registered before use.
        - Use of a CA issued certificate is no requirement.
- A person must be able to use multiple devices to authenticate to eHealth Services.
- A person needs to be able to choose an applicable userprofile (i.e. Citizen, Quality, organization membership, mandate) that will be used for authentication to eHealth Services.
- It must be possible to push the chosen identity to the requested resources or they must be able to pull it.

AUTHORIZATION

- Permissions need to be based on the chosen digital identity for each of the requested resources.
- It must be possible to propagate the permissions to the requested resources or they must be able to pull for them.
- It must be possible to let the user decide whether or not he wants to give permissions to the client application that will use those permissions on his behalf.
- It must be possible for users to revoke granted permissions.

INFORMATION SECURITY

CONFIDENTIALITY

- All communication between client to server must be considered confidential and protected against eaves-dropping, at least when traversing over an unsecure medium, such as internet.
- Medical data should be protected on the message-level to prevent disclosure of data when hopping from point to point over the network. If end-to-end encryption from original sender to final receiver is not required, it should at least be setup point-to-point between those two parties so the medical data is never sent unprotected between two points. Whether or not point-to-point is sufficient should be decided per project.
- Users must to be able to sign and encrypt messages on different devices (laptop, smartphone and tablet) without the need to transfer and expose digital keys between those devices.

INTEGRITY

- When medical data is sent from client to server, it must be signed at the message-level for content integrity.

NON-REPUDIATION

'In law, non-repudiation implies one's intention to fulfill their obligations to a contract. It also implies that one party of a transaction cannot deny having received a transaction, nor can the other party deny having sent a transaction.'

- Non-repudiation with legal binding is out-of-scope for this document. Projects that require this must take apropriate actions: A signature with non-repudiation proof should be placed on the message in clear. The signature and encryption layers mentioned in this document are to protect the message during transport and have no legal non-repudiation proof on their own.

ARCHITECTURE

PROTOCOLS AND PROFILES

RESTFUL API

A RESTful API is a method of allowing communication between a web-based client and server that employs representational state transfer (REST) constraints. It uses HTTP requests to GET, PUT, POST and DELETE data.

EHealth will use a RESTful approach to provide users access to the requested resources. REST is a logical choice for building APIs that allow users to connect and interact with online services using a regular browser or native app. The APIs will be available on an API Gateway.

IDENTITY & ACCESS MANAGEMENT

SAML 2.0

Security Assertion Markup Language (SAML) is a standard for exchanging authentication and authorization data between security domains. SAML 2.0 is an XML-based protocol that uses security tokens containing assertions to pass information about a principal (usually an end user) between a SAML authority, named an Identity Provider, and a SAML consumer, named a Service Provider.

EHealth's central Identity Provider supports most SAML 1.1 and 2.0 profiles for Single Sign On between webapplications. Prefered profile is SAML 2.0 HTTP-POST. For more information, see 'I.AM Overview' in eHealth's technical documentation.

Intended audience of this profile are Service Providers, offering regular webapplications, accessible using a standard browser.

Identity Propagation between AuthorizationServer and Identity Providers (IDP) is also done with this profile.

OAUTH 2.0

The OAuth 2.0 authorization framework, as described in RFC6749, enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.

EHealth supports following grant flows:

- Clients with user-interaction (i.e. Resource Owner is person controlling a mobile device or pc) must use the 'Implicit' or 'Authorization Code' grant flow.
- Clients without user-interaction (i.e. Resource Owner is system) must use the 'Client Credentials' grant flow.

OIDC 1.0

OpenID Connect 1.0 is a simple identity layer on top of the OAuth 2.0 protocol. It enables Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User in an interoperable and REST-like manner.

Ehealth's authorizationserver will play the role of OpenID Provider and implements this profile.

Intended audience is native apps, desktop applications and systems that need to connect to eHealth's APIs.

The id- and accessTokens will contain the identity, chosen by the user, in json claims and the permissions for each requested resource.

JWT

JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties.

Ehealth will use this standard to construct id- and accessTokens to propagate identity and permission information to clients and services.

SIGNED JWT ASSERTION

Specification RFC7523 defines the use of a JSON Web Token (JWT) Bearer Token as a means for requesting an OAuth 2.0 access token as well as for client authentication.

Confidential and system clients must follow this specification to authenticate themselves with a preregistered key.

PKCE

OAuth 2.0 public clients utilizing the Authorization Code Grant are susceptible to the authorization code interception attack. Specification RFC7636 describes the attack as well as a technique to mitigate against the threat through the use of Proof Key for Code Exchange (PKCE, pronounced "pixy").

Public clients must follow this specification.

INFORMATION SECURITY

TLS

The TLS protocol aims primarily to provide privacy and data integrity between two communicating computer applications.

Transport Layer Security can provide confidentiality of traffic between 2 nodes having to talk over an unsecured medium. EHealth will ensure One-way TLS for all its services to provide this confidentiality and to allow the client to be certain whom he is sending the information to.

This type of confidentiality is enough for communication between client and service if it does not contain medical data.

JWS

JSON Web Signature (JWS) represents content secured with digital signatures or Message Authentication Codes (MACs) using JSON-based data structures.

Ehealth will use message-level security using JWS in services that require data integrity on (parts of) messages sent from client to service or back. Only keys that can be linked to the user or client are acceptable. This could be:

- An X.509 Certificate, issued by a trusted CA.
- A public key, added to a white-list of keys, preregistered at eHealth.

EHealth will also use JWS to protect the content of issued id- and accesstokens.

JWE

JSON Web Encryption (JWE) represents encrypted content using JSON-based data structures.

JSON Web Encryption can provide confidentiality of messages between the encryption point and the decryption point, no matter how many hops are in between, provided that the decryption key is only known at the decryption point. If the sending and receiving point are the first and latest in a communication flow, we can say there is End-To-End-Encryption, if not, we should call it Point-To-Point-Encryption.

This type of confidentiality is required for communication between client and service if it contains medical data. Whether or not encryption needs to be end-to-end or point-to-point can be decided per project.

JWK

A JSON Web Key (JWK), as described in RFC7517, is a JavaScript Object Notation (JSON) data structure that represents a cryptographic key. A JWK Set JSON data structure represents a set of JWKs.

Ehealth uses this specification to publish public keys used in cryptographic operations. Clients validating signatures or encrypting messages for a known addressee can retrieve public keys for this purpose from eHealth's keystore which will return requested keys as a JWK or JWK Set.

WEBAUTHN

The Web Authentication API (W3C WebAuthn) is an extension of the Credential Management API that enables strong authentication with public key cryptography

EHealth will use this recommendation to register public keys for signing and encryption and link those to users and the clients/devices they are using.

FLOWS

IDENTITY & ACCESS MANAGEMENT

REGISTRATION

Registration of persons is managed by the Belgian Government:

- Unique identifier in the National Registry (SSIN or SSIN BIS).
- At least one digital key (eID or TOTP at this time of writing).
- With the European eIDAS regulation, European citizens will be able to prove their identity online if their country provides them with means to do so.

Registration of clients and the organizations owning them is managed by authorized users in eHealth's API Portal.

AUTHENTICATION/AUTHORIZATION

For persons requesting access to webapplications and REST services, eHealth uses a federated Identity Model.

SAML clients will connect directly to eHealth's Identity Provider.

OIDC clients will connect to eHealth's Authorizationserver which will delegate authentication to eHealth's Identity Provider.

EHealth's central Identity Provider will itself delegate authentication to the CSAM Federal Authentication Service (FAS), available as a Gcloud Service. FAS itself is an Identity Provider that offers different authenticationmethods (either directly or indirectly through partner services) so the user can use one of his digital keys.

After initial authentication, FAS will return the unique identifier of the user (SSIN or SSIN BIS) to eHealth's Identity Provider which will offer the user a choice of acceptable profiles for the requested resource(s).

In case the client is using SAML, permissions are retrieved by eHealth's Identity Provider itself and added to the SAMLToken as an attribute.

In case the client is using OIDC, permissions are retrieved by the Authorizationserver for each resource that the requesting client will need to access and added to the JWT accessToken as a claim.

See flows below for different OIDC clients.

For SAML Clients, See 'I.AM IDP' on eHealth's documentation site.

PUBLIC CLIENT

A client is considered public if it can't keep a secret to itself.

This flow is typically used by native apps without a serverside backend or clients that run entirely embedded into a browser.

The schema below shows the flow for a native app on a smartphone.

Figure 1. Authorization Code - Public Client

1. The client, on behalf of the resource owner, initiates the authorization process by opening and targeting the platform browser to the authorization URL of the Authorization Server, with parameters reflecting the client identity and a call back URL that he is sure to be able to intercept when the whole process is finished in step 9. (With apps this is done by a private or claimed URI scheme registered and recognized by the operating system as a way to redirect to an application in the manifest). The Authorization Server will process the given parameters if the client is registered.

   If the user's identity can be established by the authorizationserver (i.e. he has an active, authenticated session on the server), steps 2-8 are skipped.

2. If the user's identity cannot be found (he has no authenticated session on the server), the user is redirected to eHealth's identity provider (IDP) who will ensure identity.

   If the user's identity can be established by the IDP (i.e. he has an active, authenticated session on the server), steps 3-5 are skipped.

3. If the user's identity cannot be found (he has no authenticated session on the server), the user is redirected to CSAM FAS (IDP) who will ensure identity. FAS will only show authenticationmethods with a Level of Assurance (LoA) as defined by eHealth in the SAML Request.

4. Because the user browses those pages in his trusted browser, the URL he reads on top of the page is Fedict's own, the credentials the user will use to prove his identity will thus never be read by a third party or the authorization server. The user could for example choose "Itsme" as his digital key as a means of authentication.

5. EHealth IDP gets back an assertion from FAS that the user successfully completed his authentication process. With this assertion comes a technical level indicating the strength of the authentication method that was used. Strength of authentication can be used as one of many parameters to decide to grant a particular principal access or not. The assertion will also contain the SSIN or SSIN BIS of the user.

6. EHealth IDP can now query authentic sources for applicable profiles of the user based on his unique identifier (SSIN or SSIN BIS) and offer the user a choice between those.

7. The Authorization Server gets back an assertion from eHealth IDP with the profile, chosen by the user, and can now initiate a session for that user.

8. Based on the received profile, the AuthorizationServer will request permissions from eHeatlh's Policy Decision Point (PDP) for the given context (user, resource and environment). The assembled data is stored in the Authorization Server's userstore so the next time the delegation steps to the IDP and PDP can be skipped as long as the user's authenticated session remains valid.

   Not shown in the flow for the sake of clarity is the one-time presentation of consent to the user. A page will appear and ask the user if he agrees that the client may act on his behalf and receive permissions to do so. This consent is stored in the authorizationserver and not asked again. A user can review and revoke his consents at all times using the account service of the authorizationserver.

9. The delegation process of authorization is materialized by an authorization code, being a simple handle to an access token that will have to be fetched in a separate step by the client itself (while providing proof of its identity). This code is sent to the callback URL that the client registered. The client is supposed to be able to extract it from that context.

   The client obtains the code from the callback. A web client will take it from the URL, a JavaScript will read it from the fragment and an app will register a specific protocol or URL handler so that the call is translated into an intent to the client application and gather the parameters from it. This part is not covered by SSL so eHealth requires clients to protect themselves against attacks on this by implementing PKCE. See further.

10. The client will open a backchannel to the authorization server token endpoint to exchange his authorization code handle for an access token. As the request (and delegation) was made to a specific client, the authorization server must make sure here that he is talking to the right client, as to not issue an access token to a rogue client. As public clients cannot keep key material secret, we cannot rely only on this. Public clients must therefore use PKCE (RFC7636) as protocol when requesting tokens.

11. An access token is returned directly to the client on the same confidential and authenticated connection. None of the user agent or any other apps has seen that access token. The client should treat it as extremely confidential, it must be short lived and never stored.

12. When the client needs to access a resource on behalf of its user, he can address its request to the right resource server. Provided he presents the access token with the request, and that access token has the correct scopes and audience for the requested operation, the resource server should honour the request as if it was from the user itself. Privacy logs can show that user's identity.

CONFIDENTIAL CLIENT

A client is confidential if it can keep a secret which offers the opportunity to let it identify itself using a private key only known by that client. This opens other possibilities as well. If it can keep a key secret, it can also keep other things secret (such as session information or an offline token).

This is typically used by the backend of a native app or a regular webapplication, hosted by a trusted partner. The authorization code will be exchanged using the platform browser. The tokens will be exchanged in a direct communication between authorization and backend server.

Figure 2. Authorization Code - Confidential Client

1. The client, on behalf of the resource owner, initiates the authorization process by redirecting the platform browser to the authorization URL of the Authorization Server, with parameters reflecting the client identity and a call back URL that he is sure to be able to intercept when the whole process is finished in step 9. (On backends this is done with a URL that leads back to the server on which the backend runs). The Authorization Server will process the given parameters if the client is registered.

Steps 2-9, see 'public client'

The client obtains the code from the callback as the URI points to a URL which ends up at the backend server. The backend server can retrieve the code from the URL.

10. The client will open a backchannel to the authorization server token endpoint to exchange his authorization code handle for an access token. As the request (and delegation) was made to a specific

client, the authorization server must make sure here that he is talking to the right client, as to not issue an access token to a rogue client. For a client to be considered confidential, eHealth requires the use of a signed JWT Assertion as described in RFC7523. Using a simple id and password is considered weak.

Before the client can authenticate itself, its public key should be registered in eHealth's Keystore. See Key Registration Flow. This flow should be executed by an authorized user of the organization that is the owner of the client or an eHealth administrator.

Steps 11-12, see 'public client'

SYSTEM CLIENT

A client is considered to be a system if there is no human interaction and it is acting on behalf of itself (as resource owner) and not on behalf of a person controlling the client.

System clients do not use the authorization_code flow but the Client Credentials flow.

Typical use of this client is a batch application or a webapp backend that requires access to backend resources, not on behalf of an end-user but on behalf of the platform it belongs to.

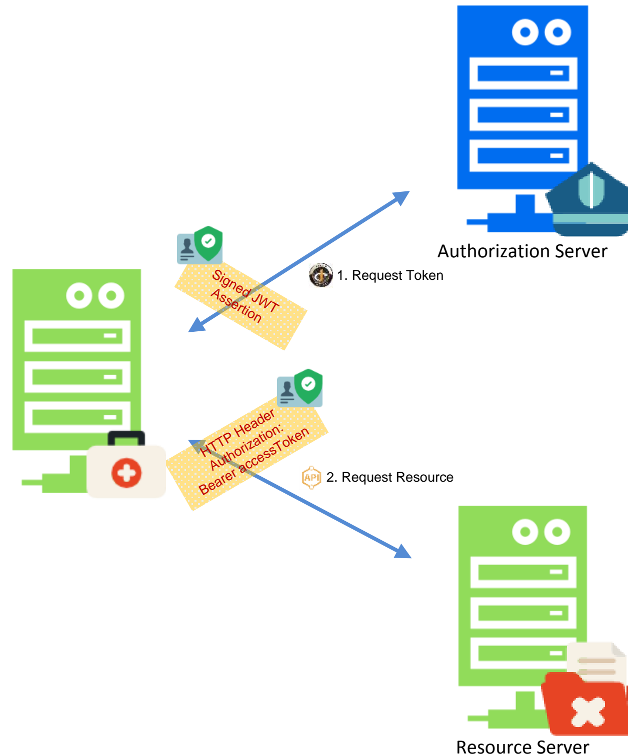

Figure 3. Authorization Client Credentials

1. The client connects directly to the token endpoint of the authorization server to request a token on behalf of itself. The authorization server will process the given parameters. As it is the token endpoint, the client needs to use the same authentication as for confidential clients in the authorization_code flow to make sure it is not a rogue client requesting a token on behalf of the registered client. System clients therefore need to use a signed JWT Assertion, as described in RFC7523.

There is no notion of authenticated browser sessions in this flow as there is no end-user. There is no redirection to eHealth IDP and therefore no FAS authentication, no choice of profile. There is no request to eHealth's PDP for permissions. The claims to be returned to the client are static and always the same. They are configured when the client is registered as system client.

Registration of the client should include registration of its public key in eHealth's Keystore. See Key Registration Flow. This flow should be executed by an authorized user of the organization that is the owner of the client or an eHealth administrator.

2. When the client needs to access a resource, he can address its request to the right resource server. Provided he presents the access token with the request, and that access token has the correct scopes and audience for the requested operation, the resource server should honour the request.

INFORMATION SECURITY

To transfer medical data from sender to receiver, security at the transport level is not enough. Signing and encrypting the data at the message level, is required.

To allow users to sign and encrypt messages in apps on multiple devices, they should be able to use different keys. Keys must be registered at eHealth before they can be used.

Scope of these keys is providing authentication, content integrity and confidentiality for data in transit. Projects requiring long-term storage or non-repudiation cannot rely solely on this setup.

KEY REGISTRATION

EHealth provides different flows to register a public or symmetric key as both are meant for different purposes.

PUBLIC KEY

A Public Key, in contrast to a Private Key, is considered to be known or knowable by anyone. It is always paired with a Private Key, which is considered to be known by noone else but the owner of it. To reach this goal, it is considered best practice to keep private keys on a secure storage or smartcard from which they cannot be extracted. However, on the mobile devices of today, there is not much support for this. It is nevertheless advised to keep the key as safe as possible and apps generating keypairs on behalf of the user should prevent exposure of the private keys.

Public keys must be registered with a unique credential-id and linked to a combination of the user's identity and the client's identifier which means a client can register one or more keys for the same client/app, one for each device on which the client/app gets installed by the user (for example: app installed on both smartphone and tablet).

Public keys can serve two purposes:

- Sign a message for content integrity and authentication
- Encrypt a message for confidentiality to a receiver

Clients should have the possibility to register a different key for both usages or one and the same.

Keys can not only be registered, they can also be unregistered to revoke them for whatever reason. Ehealth will not delete those keys but keep a trace to make it possible to request validation of keys for a given date.There should be however a limit per client+user combo to prevent misuse or explosion of the keyStore.

Scope of eHealth's public keys is data in transit which means they are not meant for long-term validation or encryption of messages. Projects that require this must use other means to reach that goal.

Keys will only be queryable within reasonable time limits (to be decided).

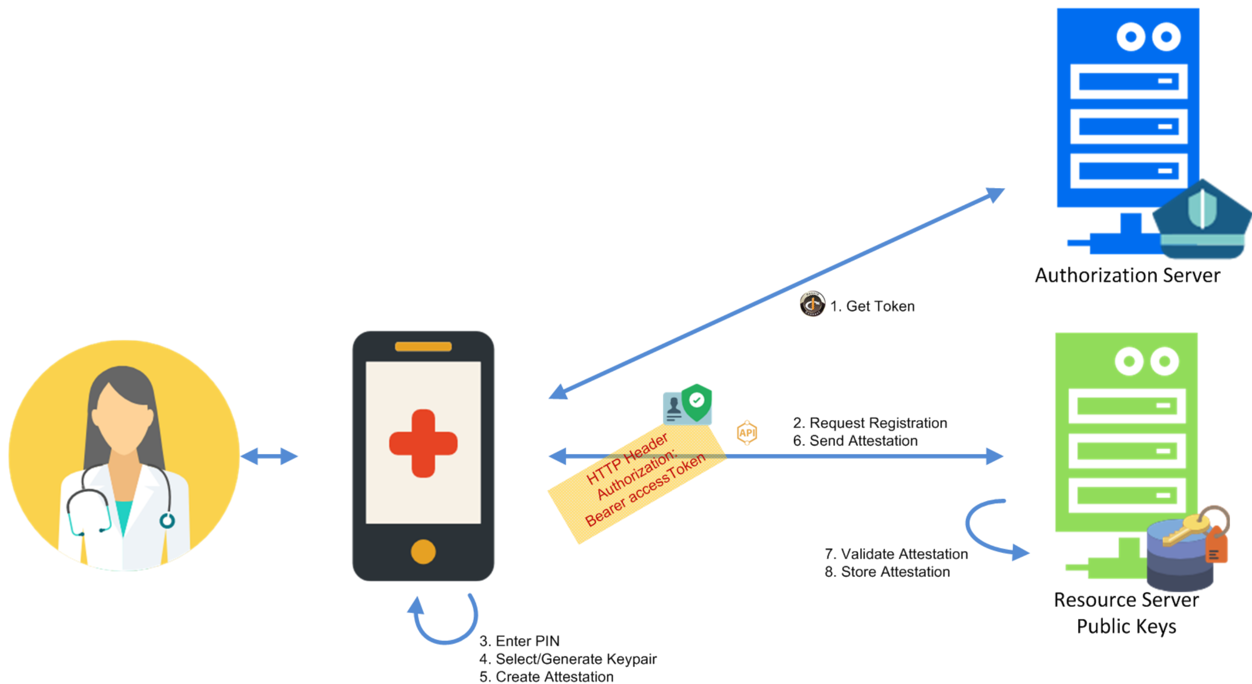

Figure 4. Public Key Registration

1. Client/app requests authorization at eHealth. See authentication/authorization flows for a full description.

2. If the client has received permissions to register public keys on behalf of the user, this will be mentioned in the accessToken and the client can perform the initial registration request.

EHealth follows the W3C Web Authentication recommendation (W3C WebAuthn) to register keys. The initial request for registration is outside of the scope of WebAuthn but the following steps are described. This means eHealth will return a challenge, user info and relying party info to the client. See W3C WebAuthn Recommendation for full details.

3. Before doing anything, the client or device that will generate the keypair will typically ask for some form of user verification, like a PIN or thumbprint, to prove that the user is present and consenting the registration.

4. After the user verification, the client or device will create a new asymmetric key pair and safely store the private key for future reference.

5. The client or device will generate and sign an attestation, including the newly generated public key.

The attestation can be signed by the generated key itself (self-attestation) or the device that generated the key in case it supports this (such as a WebAuthn compliant Authenticator). The latter can be used to prove that the keypair was generated and protected by a device, recognized to be secure.

The client should also offer the user to choose a meaningful name for his key for later consultation and revocation. This name will be transmittable to the eHealth key store.

6. The client sends the attestation to eHealth.

7. EHealth will validate the attestation to ensure that the registration was complete and not tampered with.

   Validation includes the challenge, origin and signature. If the attestation was signed by a recognized authenticator device, that chain will be validated as well. A complete list of validation steps can be found in the WebAuthn recommendation.

8. Assuming that the checks pan out, eHealth will store the new public key associated with the user's account for future use.

   The key will be linked to the user and the requesting client, both mentioned in the accessToken: Identifier (Id, Type), applicationIdentifier. EHealth's Keystore API will have search options for those fields, including the unique credentialId of the key, generated by the client or authenticator device, as described in the WebAuthn recommendation. An expiration policy is added to the key.

SYMMETRIC KEY

Ehealth allows the use of symmetric keys only for encryption.

For authentication, messages need to be signed with asymmetric key material.

Symmetric keys are typically used when the addressee of the message is unknown as we have been using for several years allready (SOAP Service KGSS).

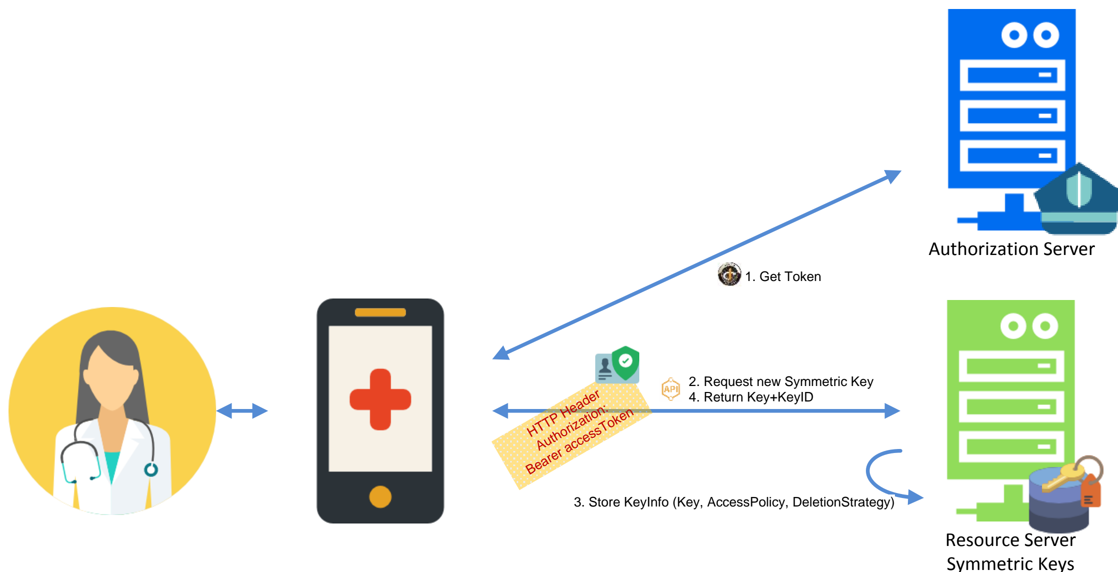

Figure 5. Symmetric Key Registration

1. Client/app requests authorization at eHealth. See authentication/authorization flows for a full description.

2. If the client has received permissions to register symmetric keys on behalf of the user, this will be mentioned in the accessToken and the client can continue by requesting a new key at eHealth. The client needs to add an accessPolicy to define who can request access to the key.

3. EHealth will generate a new key. It will be stored together with the accessPolicy and a deletion strategy to make sure its validity is limited in time.

4. Ehealth will return the key and a unique identifier to the client so he can use it when referencing the key in encrypted messages.

SIGN+ENCRYPT FOR KNOWN ADDRESSEE

If the Addressee of the message is known, his registered public key(s) can be used to encrypt the content of the message.

As the receiver may have multiple devices to read his message, the sender should encrypt the message so it can be read on each device.

Each device can have its own key(s) - per app or per device in case secure storage is available – so a sender must encrypt the message so it can be decrypted by any of the active keys of the receiver.

Before encrypting the message, the sender must sign the message for content integrity and authentication of the author.

All keys used in a signing or encryption step must first be registered at eHealth.

SYNCHRONOUS

In this flow, the sender communicates directly to the receiver, like an online service.

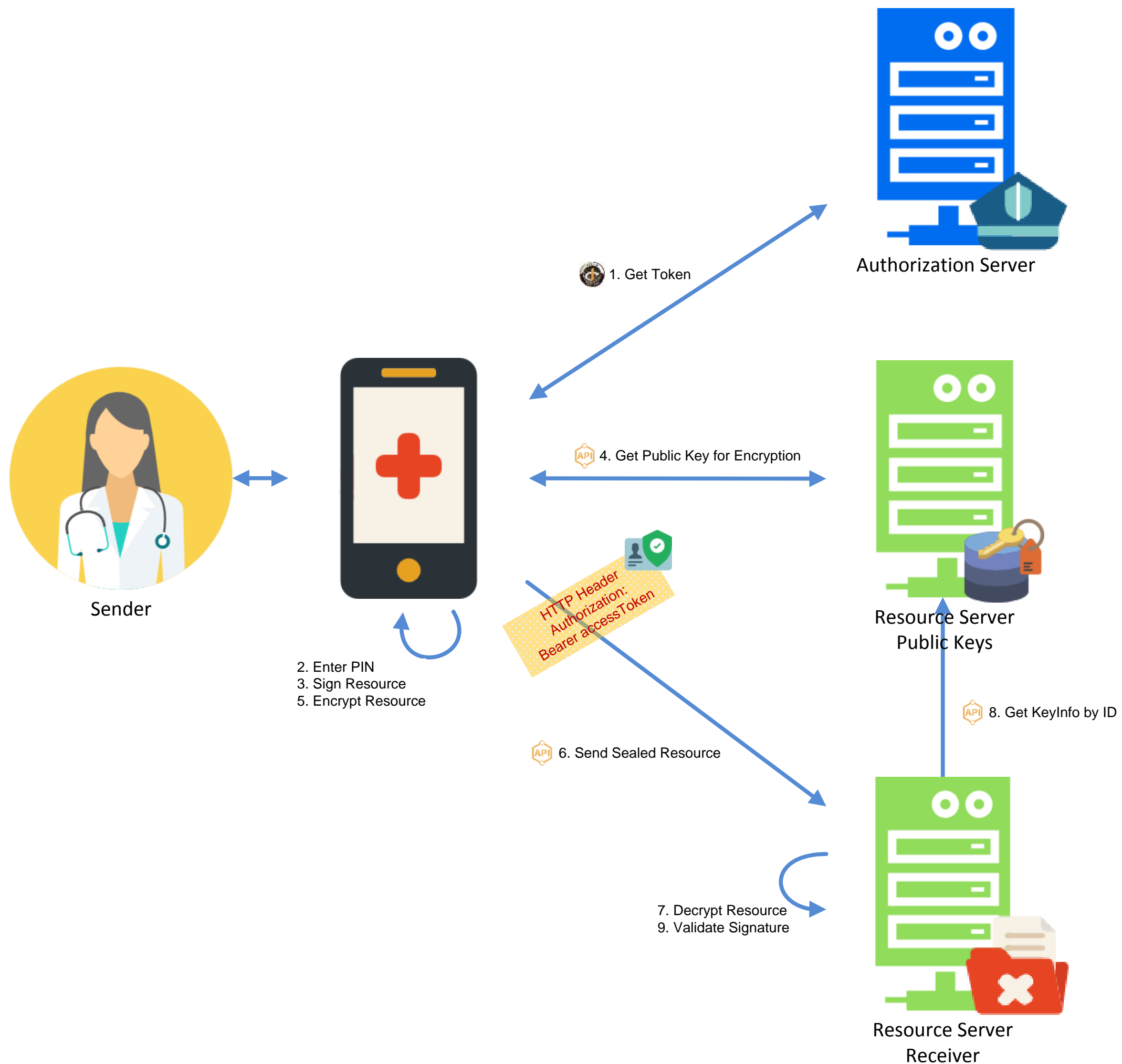

Figure 6. Known Addressee - Synchronous Communication

1. The client requests authorization at eHealth. See authentication/authorization flows for a full description.

   The accessToken returned to the client will contain a reference to the registered, active key(s) linked to the combination of the given client-user, for future reference. See section 'Security Recommendations, Risks & Known Limitations'.

2. The client requests the user to unlock his private key.

3. The client signs the message for content integrity and message authentication.

   The signature must be placed with a key of the user, registered at eHealth and active for the given client. Those are referenced in the accessToken, received in step 1. This ties the pieces together: key, client and end-user.

4. The client gets the public key of the receiver that can be used for encrypting messages to the particular resource.

   How the public key is selected from the resource server depends on how the project/app/client is setup to send messages from sender to receiver. The Public Keys Resource Server will support sufficient filtering options, such as: unique identifier of the receiver (SSIN), application identifier (clientID), usage (enc). The user should only be asked to select a key if there are multiple receivers to choose from.

5. The client encrypts the message with the key received in step 4

6. The client sends the sealed message to the receiver. The accessToken, received at the end of step 1, is added to the request.

7. If the accessToken contains sufficient privileges, the receiver decrypts the message.

8. To validate the signature on the message and to authenticate the owner of the key used for it, the receiver can use the reference of the key to get it from eHealth's keystore, verify the signature with it and verify if the reference is claimed in the accessToken, which proves it is a valid one, linked to the claimed user. This verifies that the sender of the message is also the author and that the receiver was the intended audience of the original author.

9. The receiver validates the signature.

ASYNCHRONOUS

In this flow, the sender does not send the message directly to the receiver. Instead it sends it to a central location, a depot, which will stock the message for some time so the receiver is able to retrieve it at some time in the future.

As the scope of this architecture is data in transit, the receiver should get, decrypt and validate the message within a reasonable timeframe.

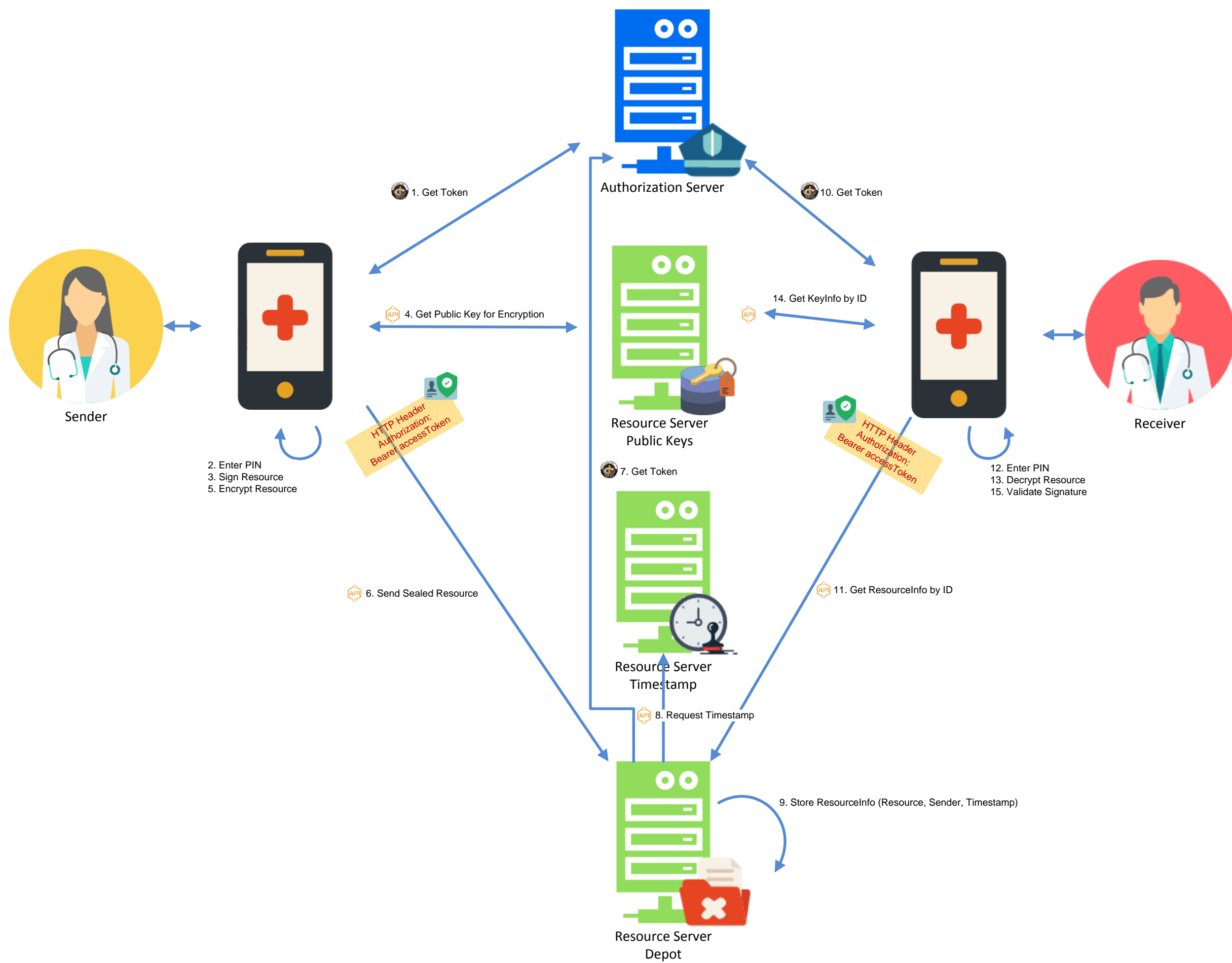

Figure 7. Known Addressee - Asynchronous Communication

1. The client requests authorization at eHealth. See authentication/authorization flows for a full description.

   The accessToken returned to the client will contain a reference to the registered, active key(s) linked to the combination of the given client-user, for future reference.

2. The client requests the user to unlock his private key.

3. The client signs the message for content integrity and message authentication.

   The signature must be placed with a key of the user, registered at eHealth and active for the given client. Those are referenced in the accessToken, received in step 1. This ties the pieces together: key, client and end-user.

4. The client gets the public key(s) of the receiver that can be used for encrypting messages to the given receiver for a given application (e.g. ehbox).

   How the public key is selected from the resource server depends on how the project/app/client is setup to send messages from sender to receiver. The Public Keys Resource Server will support sufficient filtering options, such as: unique identifier of the receiver (SSIN), application identifier (clientID), usage (enc). The user should only be asked to select a key if there are multiple receivers to choose from.

5. The client encrypts the message with the key(s) received in step 4.

6. The client sends the sealed message to the depot. The accessToken, received at the end of step 1, is added to the request.

7. If the accessToken contains sufficient privileges, the depot will accept the message and add a timestamp on it for proof of reception. For this, the depot requests a token on behalf of itself (client credentials grant, see authentication/authorization flows for a full description) to get permission to request a timestamp.

8. The depot requests a timestamp on the received data.

   The timestamp should at least cover the sealed message. It could also include the accessToken to harden the validity of the claims inside it.

9. The depot will store the message, the claims identifying the sender of the message (including the references to the registered, active keys of the sender at that time) and the timestamp as proof of reception.

   How the receiver is notified that a message is available for him at the depot is out of scope of this document and can be decided per project that uses a depot between sender and receiver.

10. The client of the receiver requests authorization at eHealth on his behalf. See authentication/authorization flows for a full description.

11. The client gets the message by its ID from the depot. The accessToken, received in previous step, is added for authentication/authorization.

    The depot will also send the identity of the sender and the timestamp, linked to the message. The format may be project-dependent and is outside of the scope of this document.

12. The client requests the user to unlock the decryption key of the app/device.

    If the message was encrypted BEFORE the decryption key was registered at eHealth, the client will not be able to decrypt the message unless another and previously registered decryption key was installed on the device for the given client.

13. The client decrypts the resource.

14. To validate the signature on the message and to authenticate the owner of the key used for it, the receiver can get the key from eHealth using the reference mentioned in the signature. The time of the timestamp returned by the depot can be added to be able to retrieve a key that is no longer active now but was at the time the message was sent. To verify that the sender was also the author of the message, the key must also be referenced in the identity of the sender. This info was present in the accesstoken sent to the depot and should be transmitted to the receiver. This verifies that the entity that sent the message to the depot is also the author, that the key was active at the time the message was sent and that the receiver was the intended audience of the original author.

    The timestamp, placed on the message in step 8 and stored in step 9, gives the receiver sufficient proof that the message was received in that state at that exact time. If such proof is not available, the receiver should use the current time when validating the key and signature on the message.

15. The receiver validates the signature.

SIGN+ENCRYPT FOR UNKNOWN ADDRESSEE

In case the sender doesn't know the addressee, he can't use a Public Key as he doesn't know which one to select.

For this flow, we'll use a symmetric key, registered at eHealth.

Below is the same asynchronous setup as for known addressees but with symmetric keys as this is the most likely scenario for unknown addressees.

Before encrypting the message, the sender must sign the message for content integrity and authentication of the author.

All keys used in a signing or encryption step must first be registered at eHealth.

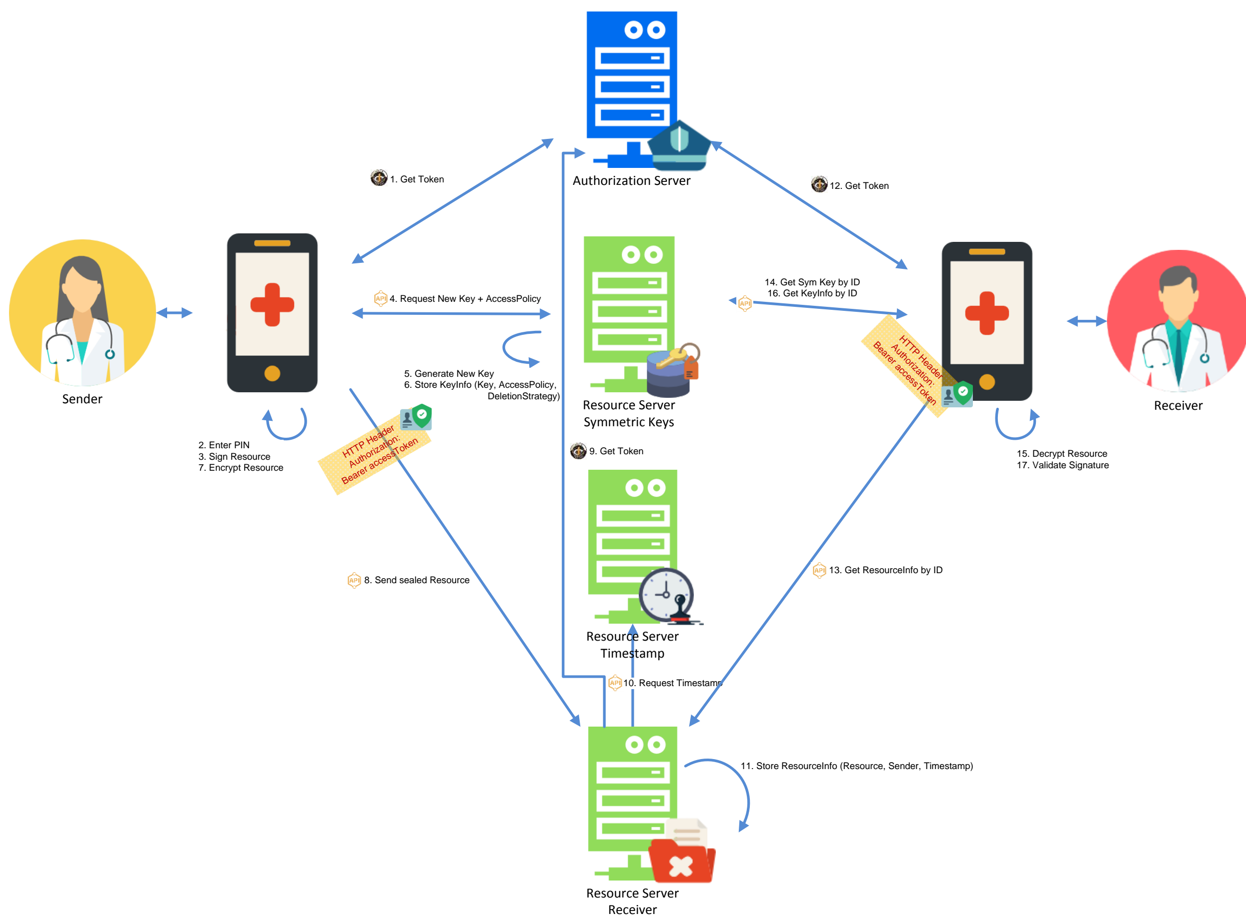

Figure 8. Unknown Addressee

1. The client requests authorization at eHealth. See authentication/authorization flows for a full description.

   The accessToken returned to the client will contain a reference to the registered, active key(s) linked to the combination of the given client-user, for future reference.

2. The client requests the user to unlock his private key.

3. The client signs the message for content integrity and message authentication.

The signature must be placed with a key of the user, registered at eHealth and active for the given client. Those are referenced in the accessToken, received in step 1. This ties the pieces together: key, client and end-user.

4. The client requests a symmetric key and defines who should get access to it.

5. EHealth generates a new key.

6. EHealth stores the new key, its identifier, the accessPolicy and a deletion strategy and returns the key with its identifier.

7. The client encrypts the message with the key(s) and adds the reference to the key in clear.

8. The client sends the sealed message to the depot. The accessToken, received at the end of step 1, is added to the request.

9. If the accessToken contains sufficient privileges, the depot will accept the message and add a timestamp on it for proof of reception. For this, the depot requests a token on behalf of itself (client credentials grant, see authentication/authorization flows for a full description) to get permission to request a timestamp.

10. The depot requests a timestamp on the received message.

The timestamp should at least cover the sealed message. It could also include the accessToken to harden the validity of the claims inside it.

11. The depot will store the message, the identity of the sender of the message and the timestamp as proof of reception.

How the receiver is notified that a message is available for him at the depot is out of scope of this document and can be decided per project that uses a depot between sender and receiver.

12. The client of the receiver requests authorization at eHealth on his behalf. See authentication/authorization flows for a full description.

13. The client gets the message by its ID from the depot. The accessToken, received in previous step, is added for authentication/authorization.

The depot will also send the identity of the sender and the timestamp, linked to the message. The format may be project-dependent and is outside of the scope of this document.

14. The client gets the symmetric key from eHealth. The accesstoken is added so the accessPolicy can be applied based on that.

15. The client decrypts the resource.

16. To validate the signature on the message and to authenticate the owner of the key used for it, the receiver can get the key from eHealth using the reference mentioned in the signature. The time of the timestamp returned by the depot can be added to be able to retrieve a key that is no longer active now but was at the time the message was sent. To verify that the sender was also the author of the message, the key must also be referenced in the identity of the sender. This info was present in the accesstoken sent to the depot and should be transmitted to the receiver. This verifies that the entity that sent the message to the depot is also the author, that the key was active at the time the message was sent and that the receiver was the intended audience of the original author.

The timestamp, placed on the message in step 8 and stored in step 9, gives the receiver sufficient proof that the message was received in that state at that exact time. If such proof is not available, the receiver should use the current time when validating the key and signature on the message.

17. The receiver validates the signature.

SECURITY RECOMMENDATIONS, RISKS & KNOWN LIMITATIONS

TRUST MODEL

To tie all the pieces together, different stakeholders need to take their responsibility and they need to trust every party for doing its job securely. This means that no party can solely verify end to end if all security precautions were taken. They need to work together to accomplish this.

This might seem a risk but this trust model is inherent to how the internet works today and is how we allready set up Single-Sign-On (SSO) years ago for regular webapplications, accessible using a browser. All stakeholders should agree on how a client's request can be traced back from end to end. Each stakeholder should take apropriate actions to fulfull his duty.

FEDERATED IDENTITY

As eHealth relies on FAS for authentication of the end-user, FAS relies on partners to provide strong authenticationmethods and identity is propagated from service to service, you need to trust different parties to take the necessary precautions to prevent identity theft.

POINT-TO-POINT INFORMATION SECURITY

MESSAGE AUTHENTICATION

If signed messages between sender and receiver are stored temporarely in a depot, the receiver can not verify itself directly if the sender of the message was indeed the author. It must trust the depot that it correctly authenticated the sender of the message and the propagated identity indeed is that sender.

The receiver can than, after decryption of the message, verify if the sender was indeed the author, if it wishes to do so.

MESSAGE ENCRYPTION

If an encrypted message is decrypted at some point in the network before the device of the intended receiver, we don't have end-to-end encryption but only point-to-point encryption. If medical data is sent from sender to receiver, the decryption point must encrypt the message again to send it to the intended receiver so the medical data remains confidential over the network until it reaches the device of the receiver.

We need to trust decryption points that medical data is never exposed to unauthorized parties.

CA VS SELF-SIGNED

EHEALTH CERTIFICATES/ETKS

At this moment, eHealth allready offers a Public Key Infrastructure (PKI) with keypairs for signing and encryption and a certificate for the signing keypair, issued by a trusted CA. This is heavily used in the SOAP services of our SOA architecture. There is no intention to change that for the time being. For our REST services, however, we need something else/more. Users can only request one eHealth certificate. This will not be sufficient for a lot of users, especially in a mobile environment. To use the eHealth certificate in a browser, the keypair needs to be installed in the browser, to use it on more then 1 device (e.g. smartphone, tablet and pc), the keypair needs to be copied from one device to another or stored in a cloud with risk of exposure. This would go against our best practice advice: leave private keys in one place, where they are generated in the first place.

We will however support them in our REST services as in fact, the keypairs themselves are no different of the keypairs that we will accept in our Key Registration flow, as described in this document.

The public keys of the keypairs that are registered when an eHealth certificate or ETK is requested will be automatically added to eHealth's keystore so users in possession of an eHealth certificate can use their eHealth certificate to sign messages as described in this document on those devices where it is available. For encryption, the public key of the ETK of the receiver can be used if one is available.

VALIDATION

By allowing registration of public keys that are not linked to certificates, issued by trustworthy Certificate Authorities (CA), we can not rely on those to prove that a key is still valid and linked to a given user.

The link between keys and identities is maintained by eHealth in its keystore.

This means that the keystore in which eHealth stores its keys must be protected against modifications by unauthorized parties. This is out of scope of this document.

To confirm that a given key is registered and active for a given client-user combination, a reference to the key will be added as claim to accessTokens requested by a given client to assert that the key is active at the time the token is requested and registered for the given user and client. Servers accepting signed messages can use the info in the signature and accessToken to verify validity of the key and identity.

RFC7800 (see references) provides such a claim if there is only 1 key to reference. If there are more, similar claims should be used.

EHealth's Keystore REST API will also provide a JWK endpoint to publish the keys with search options, such as the unique credentialId.

Registered keys should be inactivated if they aren't used for a given time-period as we cannot rely on the responsibility of the enduser to invalidate keys that are no longer used by him. However, endusers that wish to do so, should have the option to consult their active keys and invalidate them immediately if needed. The registration endpoint should provide a GUI and API for this.

NON-REPUDIATION

The scope of signatures in the proposed architecture is authentication, not non-repudiation in a legal context. For this, Qualified Certificates would be required, issued by a trustworthy CA.

Projects that need this as a requirement should perform additional actions, such as adding a qualified signature and timestamp on the message in the clear before the layers are added for transport as outlined in this document.

ENCRYPTION WITH PUBLIC KEYS

MULTIPLE DEVICES

In a mobile context, it is very likely that users have more then one device to read their messages (e.g. a smartphone, a tablet, a laptop). As we should not force users to copy their private key from device to device or store it somewhere in a public cloud with the risk of exposure, they should be able to decrypt messages with different keys, one for each device they are using. This means, the sender needs to encrypt the message for all the registered keys of the receiver for the given application. The JWE Specification (RFC7516) supports encrypting for multiple recipients so that shouldn't be an issue. Another option would be to use symmetric keys but then the

keystore cannot be maintained by any party that plays a role in sending the message from sender to receiver. Otherwise, that party would be able to decrypt the message.

When public keys are used, clientapplications can register multiple keys on behalf of their users, one for each of the user's devices they are installed on.

EHealth should set a maximum on keys that are active for a given user and application (e.g. 3 keys would allow the application to be setup simultaneously on 3 devices). By doing that, we can keep control over the size of the keystore, load of queries and required encryption process (An encrypted message will contain an encrypted part for every key).

LONG TERM STORAGE

Using public keys, registered for a limited timeframe, means that decryption is only possible for messages that are generated after publication of the key used by the clientapplication and before it expires in the registry. Outside that timeframe the sender won't know about that key and won't use it for encryption. To decrypt older messages on a new device, a previously registered key, valid at the time the message was encrypted, should be installed on that device (This should be discouraged as it goes against the recommendation to expose keys from one device to another).

As the scope of our architecture is data in transit, projects that require long term validation and encryption should perform additional actions. Once messages are retrieved, decrypted and validated, other means should be applied to reach that goal.

BEARER VS HOLDER-OF-KEY

The standard way to achieve authentication/authorization in RESTful services is OAuth 2.0.

The Oauth protocol only defines the use of bearer tokens as means for authentication/authorization. This means that the client using the token does not need to provide any proof to the server that it is the legitimate owner of the token.

It would be more secure if we could use proof-of-possession (pop) tokens where each request is signed with a key, therefore proving ownership over a token (in which the key is mentioned). WS-Security defines such a standard for SAML Tokens: SAML Holder-Of-Key. However, WS-Security was set up for SOAP. Oauth, used for RESTful APIs, does not (yet) define such a specification. So far, there only exists a standard to represent a key in a JWT Token (RFC7800) but when it comes to requesting such a token or using it, there are only drafts available (such as 'draft-ietf-oauth-pop-key-distribution-03' to request it or 'draft-sakimura-oauth-jpop-04' to use it). This means we cannot set this up unless we build something for which no agreed standard exists. This would force all clients and servers to build custom code. As there are many libraries around that do support bearer tokens, this decision would be expensive for all parties.

This might seem as a risk but in fact using bearer accessTokens is not less secure then how all major browsers communicate with webapplications around the world: After the first request to a server, hosting a webapplication, a browser receives a secure random sessionID, which it needs to use in every subsequent request. The server will link that session to the user. The browser does not need to prove ownership over the ID. The sessionID can therefore also be considered as a bearer token.

To prevent identity theft, it is very important that the bearer token (sessionID, accessToken) is only sent when the transport layer is secured (TLS).

It is also very important to limit the validity of accessTokens in time. They should be shortlived (a matter of minutes, not hours) as they don't need any proof when being used.

AUTHORIZATION CODE EXCHANGE

OAuth 2.0 public clients utilizing the Authorization Code Grant are susceptible to the authorization code interception attack. There exists however a specification describing the attack as well as a technique to mitigate against the threat through the use of Proof Key for Code Exchange (PKCE, pronounced "pixy", RFC-7636).

Public clients are advised to use that technique when communicating with eHealth's authorization server.

SOAP VS REST

The solutions in this document are meant to be setup for eHealth REST services, available on eHealth's API Gateway.

There is no intention to start a make-over of eHealth's SOA architecture or the SOAP services that are published on it. That would have too much impact on existing flows and would be very expensive. Therefore, eHealth SOAP services are deliberately out-of-scope and the way how Identity & Access Management and Information Security is set up for them will remain as is.

We do however provide a bridge between both worlds, from REST to SOAP, when it comes to authentication, as this helps providing SSO and the possibility to use SOAP services either directly or indirectly from mobile devices.

TOKEN EXCHANGE

Token Exchange at eHealth provides the possibility to exchange an OAuth accesstoken for a SAML HOK Token that can be used as authentication for SOAP services.

It is based on the internet draft 'OAuth 2.0 Token Exchange' (draft-ietf-oauth-token-exchange-12).

The user controlling the client will be requested to grant permission to the client to act on his behalf and exchange his accessToken for a SAML Token. At any time, he will be able to revoke that permission as he can do for all permissions using the account service of eHealth Authorization Server.

PUBLIC CLIENT

A client is considered public if it can't keep a secret to itself.

This flow is typically used by native apps without a serverside backend or clients that run entirely embedded into a browser.

The schema below shows the flow for a native app on a smartphone.

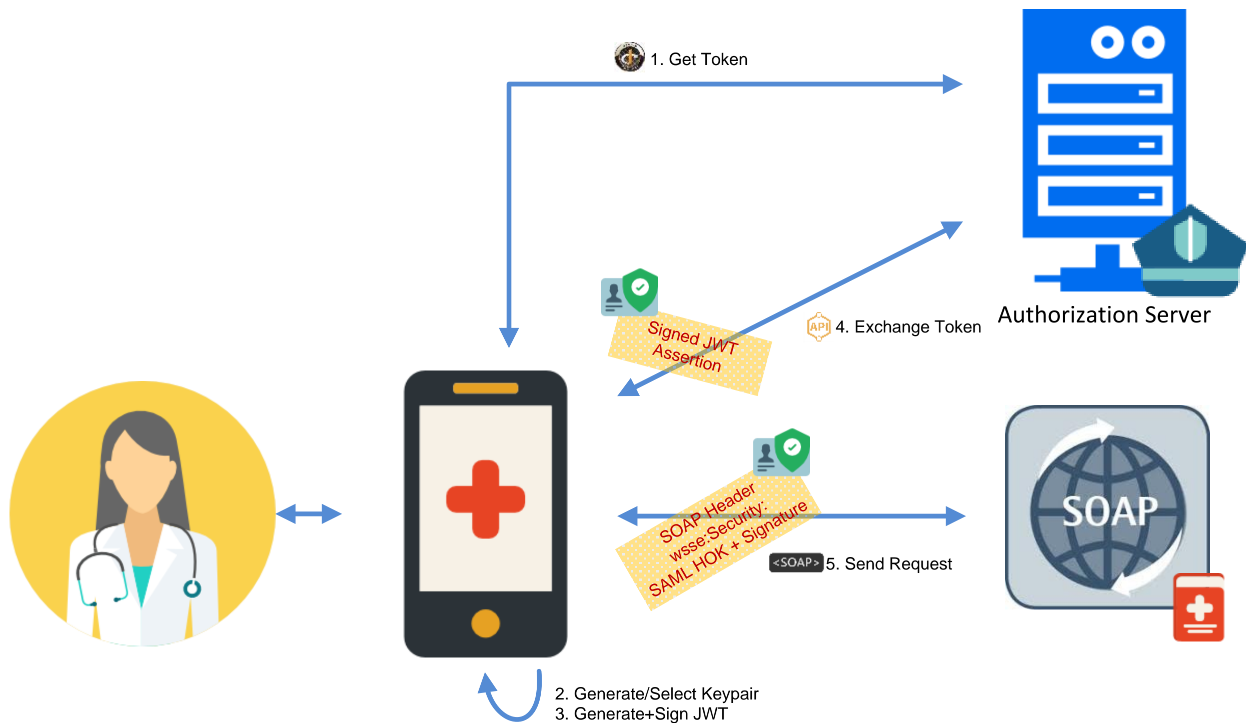

Figure 9. Token Exchange - Public Client

1. The client requests authorization at eHealth. See authentication/authorization flows for a full description.
2. The client requests the user to generate a new keypair or select an existing.

   In case a new keypair is generated, the client should register it at eHealth. See key registration flows for a full description.

3. The client generates a JWT assertion and signs the message with the selected keypair.
4. A request is sent to the authorizationserver including the accessToken received in step 1 as subjectToken, the signed JWT assertion as actorToken and a request to get a SAML HOK Token.

   The authorization server will validate both tokens. If the accessToken contains permissions to do a tokenExchange, the authorization server will accept the request and generate a SAML HOK Token with the key from the validated actorToken as keyInfo so the client will be able to prove ownership over the token.

5. When sending requests to SOAP Services, the client will add the SALM HOK Token in every SOAP Header and sign every SOAP Body with his private key to tie the token to the request.

CONFIDENTIAL CLIENT

A client is confidential if it can keep a secret which offers the opportunity to let it identify itself using a private key only known by that client. This opens other possibilities as well. If it can keep a key secret, it can also keep other things secret (such as session information or an oauth refresh token).

This is typically used by the backend of a native app or a regular webapplication, hosted by a trusted partner. The authorization code will be exchanged using the platform browser. The tokens will be exchanged in a direct communication between authorization and backend server.

In this flow the end-user gives permission to the backend server to send SOAP calls on his behalf with a SAML HOK Token.

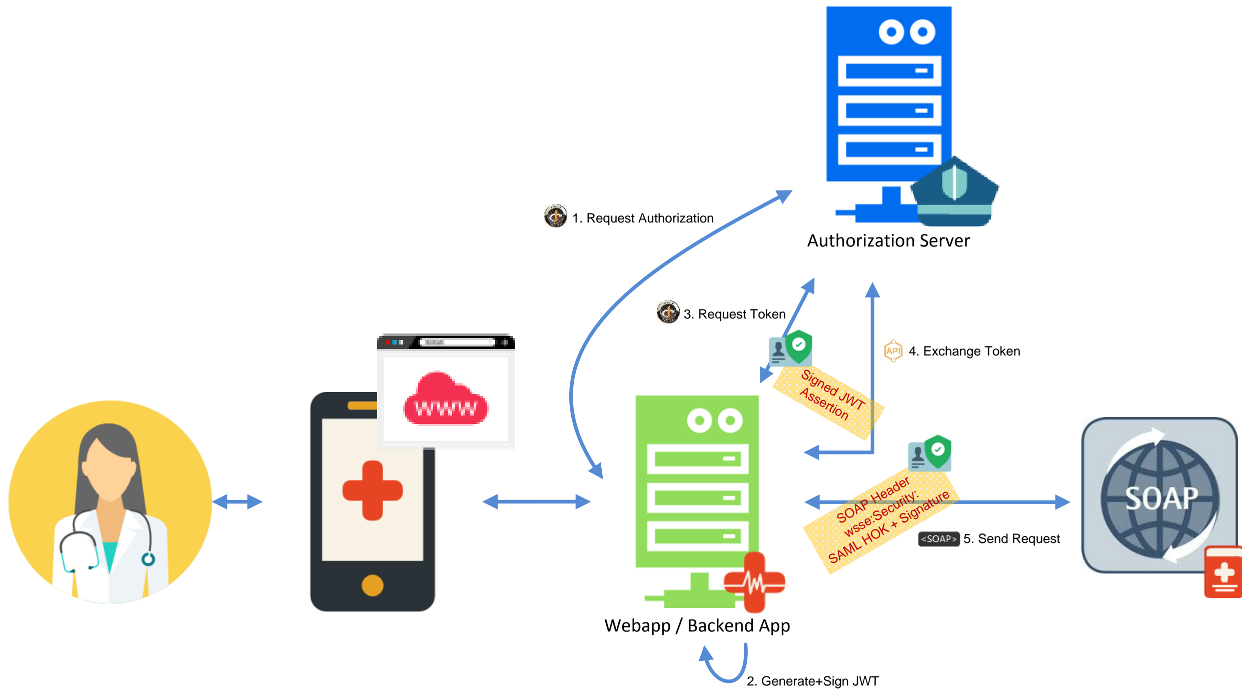

Figure 10. Token Exchange - Confidential Client

1. The client requests authorization at eHealth. See authentication/authorization flows for a full description.
2. The client generates a JWT assertion and signs it with his private key
3. The client requests an accessToken with the authorization code received in step 1 and the signed JWT assertion to authenticate itself.
4. The client sends a token exchange request to the authorizationserver including the accessToken received in step 3 as subjectToken, his signed JWT assertion from step 2 as actorToken and a request to get a SAML HOK Token.

   The authorization server will validate both tokens. If the accessToken contains permissions to do a tokenExchange, the authorization server will accept the request and generate a SAML HOK Token with the key from the validated actorToken as keyInfo so the client will be able to prove ownership of the token.

5. When sending requests to SOAP Services, the client will add the SALM HOK Token in every SOAP Header and sign every SOAP Body with his private key to tie the token to the request.

CRYPTOLIB

EHealth Cryptolib is a software development kit designed to protect medical data using the Cryptographic Message Syntax (CMS), signed with an eHealth certificate and encrypted with an eHealth ETK (See section above on eHealth Certificates/ETKs).

The architecture of CMS is built around certificate-based key management.

As this document proposes a solution for information security by using JSON Web Signatures (JWS) and JSON Web Encryption (JWE) without certificates, CMS and the eHealth Cryptolib are of no use in our REST services.

There are good reasons to choose another setup for those REST services, such as:

- JSON (and extensions like JWS, JWE) are widely adopted standards in the mobile world. Consequently, there are many opensource libraries available to use those standards. Therefore, there's no need to provide something like cryptolib.
- No limitation of one key per user. Number of eHealth certificates is limited to one per user as they are issued by an external CA. To support multiple devices, we need at least one per device.
- No dependency on external CAs.
- Auto-generation and registration of new keys on the fly.

As we are just starting with REST services, there's no cost for partners to rebuilt existing services. For our SOAP services, we will keep using CMS messages with cryptolib, for that reason.

This doesn't mean that CMS or cryptolib can't be used at all on mobile devices or in browsers but it won't work out of the box for the reasons allready mentioned above. Projects that wish to support sealing or unsealing CMS messages on mobile devices or in browsers should provide apropriate means to do so, such as:

- A detection mechanism to switch between CMS or JSON (if they choose to support both) and use the correct sealing or unsealing process.

At least a keystore with the user's ehealth certificate and/or ETK and a truststore with the trusted CAs must be installed on the mobile device(s) or browser(s) of the user. This goes against the recommendation not to distribute the private key(s) from the location where they were originally generated and augments the risk of key exposure.

WEB AUTHENTICATION API

This API specifies how to register public keys and use them for authentication to a server using a browser and an authenticator device.

We can align ourselves with this API to register keys as the process, intention and result is the same: register a public key from a user-agent and link it to an authenticated user.

At the time of writing this API is a W3C Candidate Recommendation, not yet final.

Once the specification becomes final, hopefully, clients and devices will start supporting it. This will make it easier to integrate with eHealth to register keys.

REFERENCES

EHEALTH DOCUMENTATION

I.AM Overview - https://www.ehealth.fgov.be/ehealthplatform/nl/search?q=i.am%20overview

I.AM IDP - https://www.ehealth.fgov.be/ehealthplatform/nl/search?q=i.am%20idp

SPECIFICATIONS

OpenID Connect 1.0 - http://openid.net/specs/openid-connect-core-1_0.html

SAML 2.0 Web Browser SSO Profile - http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf

Web Services Security SAML Token Profile Version 1.1.1 - http://docs.oasis-open.org/wss-m/wss/v1.1.1/wss-SAMLTokenProfile-v1.1.1.html

[RFC 6749] The OAuth 2.0 Authorization Framework - https://tools.ietf.org/html/rfc6749

[RFC 7515] JSON Web Signature (JWS) - https://tools.ietf.org/html/rfc7515

[RFC 7516] JSON Web Encryption (JWE) - https://tools.ietf.org/html/rfc7516

[RFC 7517] JSON Web Key (JWK) - https://tools.ietf.org/html/rfc7517

[RFC 7519] JSON Web Token (JWT) - https://tools.ietf.org/html/rfc7519

[RFC 7523] JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants - https://tools.ietf.org/html/rfc7523

[RFC 7636] Proof Key for Code Exchange by OAuth Public Clients - https://tools.ietf.org/html/rfc7636

[RFC 7800] Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs) - https://tools.ietf.org/html/rfc7800

[RFC 8252] OAuth 2.0 for Native Apps - https://tools.ietf.org/html/rfc8252

DRAFTS AND RECOMMENDATIONS

OAuth 2.0 Token Exchange - https://tools.ietf.org/html/draft-ietf-oauth-token-exchange-12

OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution - https://tools.ietf.org/html/draft-ietf-oauth-pop-key-distribution-03

The OAuth 2.0 Authorization Framework: JWT Pop Token Usage - https://tools.ietf.org/html/draft-sakimura-oauth-jpop-04

[W3C WebAuthn] Web Authentication: An API for accessing Public Key Credentials - https://www.w3.org/TR/webauthn/