eHealth Network

Guidelines on

EU DCC Revocation - B2A
Communication between the Backend and the Applications

Version 1.1

2022-03-30

eHealth Network

The eHealth Network is a voluntary network, set up under article 14 of Directive 2011/24/EU. It provides a platform of Member States' competent authorities dealing with eHealth.

Adopted by the eHealth Network on 30 March 2022.

Version history

Version	Date	Changes
1.0	16 February 2022	Initial version
1.1	30 March 2022	Improve technical description for the Partial variable-length SHA encoding.

Contents

1. Introduction

This document is a guideline on how to download and process the revocation list data[1] from the EU DCC Gateway (DCCG) and use it within the associated wallet / verifier apps and validation services. It complements the DCC Revocation B2B concept, which describes the DCCG interfaces / access points for revocation of single DCCs.

The described concept is not binding for the DCC participants: all participants are free to use it, customise it or derive their own solution, so far, it's guaranteed that the revocation list entries are taken into account on behalf of the issuer.

2. Assumptions

This concept is based on the following assumptions:

- All uploaders/downloaders of the revocation list batch data are connected to the EU DCC Gateway
- Maximum amount of DCC revocation entries is 80 Million (worldwide)[2]
- Apps should be able to download incrementally without the need to sync entire lists
- The apps should be able to decrypt and verify the downloaded lists
- Revocation must be processable offline on verifier devices

3. Scope

3.1. In scope

This concept gives a guideline / recommendation on how to implement the local part of the overall revocation system including download revocation list data from DCCG, distribute it to the verification apps and process it within the apps in an efficient and secure way considering the data privacy requirements.

It refers to the following components:

- The local **Revocation Data Service**
  The service which downloads the revocation lists from the DCCG and distributes them to the verifier apps. It can be also queried by the wallet apps in order to determine whether the DCCs they contain have been revoked.

[1] The revocation lists are used for the invalidation of single DCCs. For the mass invalidation of DCC please refer to the Business Rules concept (https://ec.europa.eu/health/ehealth/covid-19_en#:~:text=EU%20DCC%20Validation%20Rules) and the Public Key Governance concept (https://ec.europa.eu/health/ehealth/covid-19_en#:~:text=Volume%205%3A%20Public%20Key%20Certificate%20Governance)

[2] Based on the real amount of data in the revocation databases of selected countries in Europe, we estimate around 4 million valid revocation entries for the EU covering all use cases: fraud (<1%), false issued DCCs (<20%) and suspended DCCs (80%, if all countries support suspension)

- The **Validation App**
  Which downloads the revocation information and processes it during the validation process in order to determine whether the given DCC has been revoked or not.
- The **Wallet App**
  Which is able to query the Revocation Data Service in order to determine whether the DCCs it contains have been revoked or not.

This concept covers following Use Cases:

- **The validation app updates its local revocation list data storage**
  The validation app queries the country-specific instance of the Revocation Data Service on a regular basis, loads all changes of the revocation list (from all participating countries) and updates its local data storage. It optimises the storage for efficient access.
- **The validation app checks whether a scanned DCC has been revoked**
  Within the DCC validation process the validation app queries its own local revocation list data storage in order to determine whether the given (scanned) DCC has been revoked or not.
- **The wallet app checks whether a given loaded DCC has been revoked**
  The wallet app queries the country-specific instance of the Revocation Data Service on a regular basis in order to determine whether any of the DCCs it contains has been revoked.
- **The validation app deletes the expired revocation entries from the local data storage**
  The validation app deletes all expired revocation entries from its own local revocation list data storage on a regular basis. It optimises the storage for efficient access.

3.2. Out of scope

- Possible solutions / concepts on how to identify and revoke certificates in the national systems are not covered in this document.
- This concept also does not consider specific national regulations according to data privacy and security.
- This concept does not describe possible UI/UX solutions and app-UI-workflows.

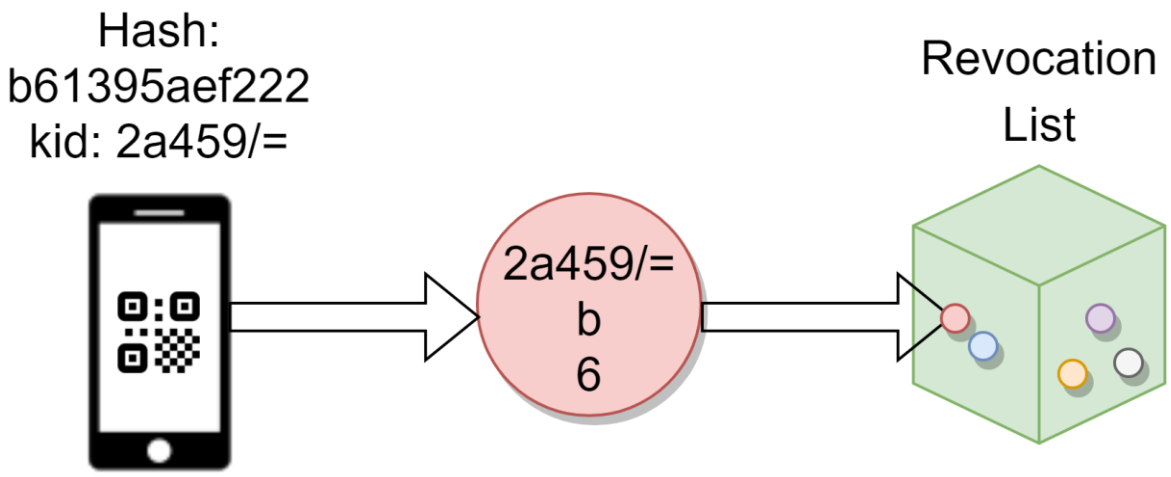
4. Solution

4.1. Considerations

The revocation of single DCCs can result in a very big amount of data. When millions DCC revocation entries are handled at global scale, normal lists with simple hashes can quickly reach hundreds of megabytes. This concept guarantees the offline capabilities for revocation considering both: efficient access to revocation data and the size of the revocation lists data storage. The following chapters describe patterns which help to partition the revocation lists and to speed up the revocation lookup.

4.2. Revocation Entry Lookup

The core question is how to find a revocation entry in an efficient way. For this kind of problem an effective pointing method must be introduced to avoid a delay of the verification process. For instance, during a scan of 1000 people, a delay of 6 seconds means 100 min more delay on the entire process. To save this time, the lookup mechanism must be optimised (striving to achieve O(log(n)) access) and avoid a linear search complexity (O(n)) depending on the number of items. To achieve this goal, the DCC will be used as the basis for the point by using the KID as entry point, and the first and the second byte of the revocation entry (hash)[3] as coordinates to identify where the revocation entry must be located.



4.3. Revocation Partition List

For the support of the concept of the revocation entry pointer, the data must be prepared in small groups to form a cube. The cube can have multiple dimensions e.g. X,Y,Z. Each dimension makes the pointing more precise. Overall, the amount of data remains almost the same (plus overhead for management). Only the organisation of the data is different to support an effective pointing to reach an O(log(n)) access by combining an O(1) access over an static partitioning and an O(log(n)) access within the partitions itself. This reduces the search amount, because most of the data is skipped and needs not to be included in the search.

To split this big amount of data in smaller pieces, it's necessary to find a mechanism which allows for each attendee to calculate and share deltas. For this purpose, the revocation list will be converted into a table where each cell acts as a "revocation partition" with some entries for revocation. Each of those cells is identifiable over X,Y coordinates and can have a separate expiration date, which makes it not necessary to update the entire list over a timeline. To identify a certain revocation entry, the coordinates of the partition must be **derived** from the DCC to have an efficient pointing to the right partition. Then the partition can be loaded and scanned for the existence of an entry. To make this

[3] See Chapter 5.6 “Hash Types” in the B2B Concept for more details.

system even more effective and determine revocations by DSC, the partition lists are additionally grouped by the KID.

So, the revocation partition list is a 3d-cube where each cell contains a list of revocation entries. This concept defines these cells as partitions which can have a maximum amount of 16^2 when a hex encoding is used for the hashes. This mechanism avoids the full scan of lists, because the DCC can point directly on its own revocation partition, which reduces the amount of data to the absolute minimum.

The values of X and Y are defined as the first byte of the sha256 hash, and the second byte of the sha256 hash[4]. Both in combination lead to 65536 partitions per KID when no further encoding is used.

The list itself is split in different parts:

- Partitions
- Chunks
- Slices

A partition contains multiple chunks. One chunk contains multiple slices sorted by date. This means that a DCC can locate the precise chunk by using the first bytes and, if existing, the slice data will hold the precise revocation information which can be easily loaded to memory and removed if the expiring is reached. For instance, when a kid has 4 million hashes revoked, the amount of data will be separated over 256 partitions. Each Partition contains around 15K Hashes, each chunk around 1K, but the slices depending on the settings contain just around 1K/Slice amount hashes. In the end just one 1K is loaded to memory.

4.4. Revocation List Optimizations

Depending on the size of the lists and the number of hashes, it makes sense to optimise the entire cube for downloading. Otherwise, the app would need to download everything again. This can be done using the coordinates. If the number is small, it makes no sense to split in the full-blown partition size. For this purpose, this concept introduces the Mode “POINT” and “VECTOR”. Point uses no coordinates (one static partition per KID), Vector just X as coordinate (up to 16 partitions per KID) and coordinate uses 256 partitions per kid within hexadecimal encoding.

The lists itself can be represented in different formats (see chunk format below). The format itself needs to be chosen according to local regulations.

4.5. Revocation Batch Download

In order to set up the partitions, the revocation data must be downloaded from the revocation batch endpoints from the gateway. Each downloaded batch should be resolved to a flat structure of country, kid, hash, hashType and expiringDate. To prepare the partitioning, each row should contain the X,Y,Z[5] value as well (for sorting and grouping).

[4] We assume byte-/binary- encoding.

[5] Where X is the first byte, Y is the second byte and Z is the third byte of the value.

NOTE: If hashes are delivered twice, the second entry is ignored.

4.6. Revocation List Partition Content

4.6.1. Overview

The partitions within the revocation lists can contain the revoked hashes in different formats represented as chunks to optimise the lookup behaviour for a device. This allows it to provide the revocation lists optimised for each device (e.g. selected by User Agent) or for memory footprint reduction etc. Each of the chunks can have another format which makes it easy to migrate in the future to other formats within the own app. To coordinate it a meta data structure is introduced per partition.

4.6.2. Metadata

4.6.2.1. Concept

The partition metadata contains a prefix tree in binary form to make a lookup with the calculated hash to find the right chunk. This shall avoid:

- loading a “hot spot” partition in one shot into the memory
- control over type and version of the used data distribution

To support the lookup each partition metadata has the following data structure:

```
{
     “id”:string,
     “kid”:string,
     “X”:char, // if missing (null) the section name is considered to be X
     “Y”:char, // (missing if X is missing) if missing section name = Y
     “expires”:{Date}, // (optional) Maximum of the section expiration dates
      “chunks”:[{
             // If “X” is missing in the partition, section = X
             // If “X” exists and “Y” is missing, section = Y
             // If both “X” and “Y” are both available, section = Z
             “section”:”b”,

             “expires”:{date},
             “chunk”:{“type”:”Bloom”,”hash”:’333ffff’,“version”:”1.0” }},{
             “section:“9”,
             “expires”:{date},
             “chunk”:{“type”:”Hash”,”hash”:’dddd’, “version”:”3.0”  }}]
}[6]
```

[6] This example uses hex encoding for X,Y and Z.

4.6.2.2. Calculation

The items above can be calculated by the following algorithm in the backend:

At first it must be found out which mode has to be used:

a) Group received batches by kid and count of data
b) Use following optimization schema:
   i) POINT for less than 2^{13} revocation entries
   ii) VECTOR for less than 2^{17} revocation entries
   iii) COORDINATE for more revocation entries

After the mode selection the data must be sorted ascending by kid by expiring date and for X or for X and Y. Depending on the mode, the hashes can be grouped by kid and first byte (POINT), by kid first+second byte (VECTOR) or by kid first, second and third byte (COORDINATE). With this configuration and in hex encoding, the coordinate configuration can store around 4M per kid (16^3 * 1000 = 4096000 Entries, 256 Partitions with 16 Chunks per Partition)

Example:

List of hashes for partition (X=e,Y=d):

(1) ed**a**41aab68a44e6e71b912abc8f2fd7844a599346a9303886c5abe96f15375f4
(2) ed**d**41aab68a44e6e71b912abc8f2fd7844a599346a9303886c5abe96f15375f4
(3) 5d**c**41aab68a44e6e71b912abc8f2fd7844a599346a9303886c5abe96f15375f4
(4) 1d**e5**1aab68a44e6e71b912abc8f2fd7844a599346a9303886c5abe96f15375f4
(5) 6d**e6**1aab68a44e6e71b912abc8f2fd7844a599346a9303886c5abe96f15375f4

The assumption is now to have an amount less than defined for POINT. The result is then:

```
{
   “e”:{... 2 Hashes},
   “5”:{... One Hash},
   “1”:{... One Hash},
   “6”:{... One Hash}
}
```

Is the Mode Changed to Vector, the partitions are split by the first byte:

X=”e”

```
{
   “d”:{...},//contains 2
}
```

X=”5”

```
{
```

```
   “d”:{...},//contains 1
}
```

With the change to Coordinate:

```
X=”e”, Y=”d”

{
   “a”:{...},//contains 1,
   “d”:{...},//contains 1
}
```

Calculation Considerations

The tree can be built hierarchically, but it should be considered to build it in a flat layer. Within a hexadecimal range, there are 256 partitions possible. Each partition can have 16 chunks. This results in an amount of 4096 chunks. Assuming 1000 Items per chunk, there are more than 4M Hashes possible per kid, which seems to be enough for the most use cases. If the size is not fixed, there are more than 4M possible. Which calculation is used, depends on the app needs and design.

4.6.2.3. Data Distribution Format

The partition metadata can be optionally and packed as Sign1 Message (Tag 18) when the sharing for the app requires a signature. This can have the following structure:

- Protected Header
  - Signature Algorithm (alg, label 1)
  - Key Identifier (kid, label 4)
- Payload
- Signature (of associated RSC)

The signature is following the definitions of the hCert Spec[7]. All signatures must map to the RSC kid and the kid MUST be in the protected header and is defined as the first 8 bytes of the SHA256 fingerprint of the used DGC.

An example payload CBOR format can be found in the Appendix[8].

4.6.3. Chunks

4.6.3.1. Considerations

All chunks should be generated according to the applicable personal data protection legislation (GDPR and/or other applicable laws). Anonymous data structures as bloom filters should be prefered, if possible, in combination with revocation crosscheck features within the wallet app. Overall, the chunks should be generated so that each chunk does not have a mix of items with very

[7] https://github.com/ehn-dcc-development/hcert-spec/blob/main/hcert_spec.md#332-signature-algorithm.
[8] METADATA CBOR DEFINITION

different expiration dates. All chunks/slices should be generated in a way that the difference in expiration is not bigger than 24 hours.

4.6.3.2. Hash Lists

All revocation entries can be distributed in hash lists without any compression or similar. These lists can be loaded in standard data structures to find a hash. An example Object for CBOR can be found in the appendix (Hash List CBOR Format).

4.6.3.3. Bloom Filters

To reduce the data privacy/memory footprint of provided UCIs, a bloom filter can be used to reduce the size of provided revocation lists. The hash function used is SHA256. All other parameters can be selected by the participants. The generated Bloom Filter is inserted in the partition as payload together with the associated parameters. An example Object for CBOR can be found in the appendix (Bloom Filter Format). For bloom filters the associated parameters are the important point: it's recommended to pack not more than 1000 Entries within one filter, with an amount of around 20 hash functions, to reach an probability of false positives of around 1E-10[9]. The exact values and settings should be provided to the app to adjust within the verification all the time the calculation.

4.6.3.4. Partial variable-length SHA encoding

To reduce the data footprint while offering the same privacy guarantees as Bloom filters, ordered tables of d-subset bits of SHA hashes can be used.. Lookup of a revoked certificate is performed via binary search in the ordered table. In more detail:

- We take the first d bits of their hash values, following the bits used for the partition lookup.
- These are stored ordered in an array (or any data structure with random access in O(1)) of length n.
- To lookup a particular certificate we take the same d bits of its hash value and use them as key in a binary search in the ordered array.

The probability of a false positive is $2^{-(d+c-\lg n)}$, where d Is the number of bits we take from the hash (the length of the encoding), $c \in \{0, 8, 16\}$ is the number of bits used for the partition lookup, and n is the total number of revocation keys. That means that if we want to achieve 2^{-p} precision, we must use $d = \lg n - c + p$ bits from each hash, compared to the $1.44\,p$ bits cost incurred by Bloom filters

This structure allows countries to make a trade-off between the privacy provided and the rate of false positives they find acceptable. The more bits they use the less chance of false positives, less privacy and more data transfer costs.

We recommend choosing a value of d which is byte aligned - so divisible by 8 - as this because that makes sense given modern computers.

Some indicative numbers:

[9] https://hur.st/bloomfilter/?n=1000&p=1.0E-10&m=&k=20

- For 100,000,000 revocations, stored with two bytes for partition lookup, using 40 bits from the hash will provide a precision of 1.67×10^{15} (scale of one in a trillion false positive rate).
- For 100,000,000 revocations, stored with two bytes for partition, using 32 bits from the hash will provide a precision of 6.55×10^{12} (scale of one in a billion false positive rate).

4.6.3.5. Evaluation of the different chunk formats

General evaluation:

Chunk Format	**Pros**	**Cons**
Hash Lists	+ No false positives (the probability for false positives is astronomically low)	- Very big amount of data to be distributed to the mobile devices - Very big amount of data on the mobile devices - Possible data privacy issues - Suitable for up to 10M revocation entries
Bloom Filters	+ Low data privacy issues (Bloom filters are considered as anonymous data by many EU countries) + Fast calculation on device + Low amount of data on the mobile devices + Already tested according performance	- The probability for a false positive is $1/10^{10}$
Partial variable-length SHA-encoding	+ Very low amount of data on the mobile devices + Very low amount of data to be distributed to the mobile devices + Fast calculation on device	- Possible data privacy issues - The probability for a false positive is $1/10^{12}$

Evaluation according given key attributes:

Attribute	**Hash Lists**	**Bloom Filters**	**Partial variable-length SHA-encoding**
Data distribution to the mobile Devices	worst	good	best
Speed of calculation on the mobile device	worst	best	best
Amount of data stored on the mobile devices (for verifier apps)	worst	good	best

Probability for false positives	best	worst	middle
Possible data privacy issues in different MS	worst	best	middle

4.7. Revocation List Signing

The revocation list data should be signed by the creating authority to prove the trustworthiness of the revocation data within the verifier app. For this purpose a new "Revocation Signer Certificate" must be used (DSC should not cross used). The RSC needs to be generated by each country according to certificate governance security requirements. With this certificate all revocation items should be signed before distributing it to the apps. The signing may be optional if the data is directly distributed via an API where the source can be verified over a TLS certificate.

4.8. List Distribution

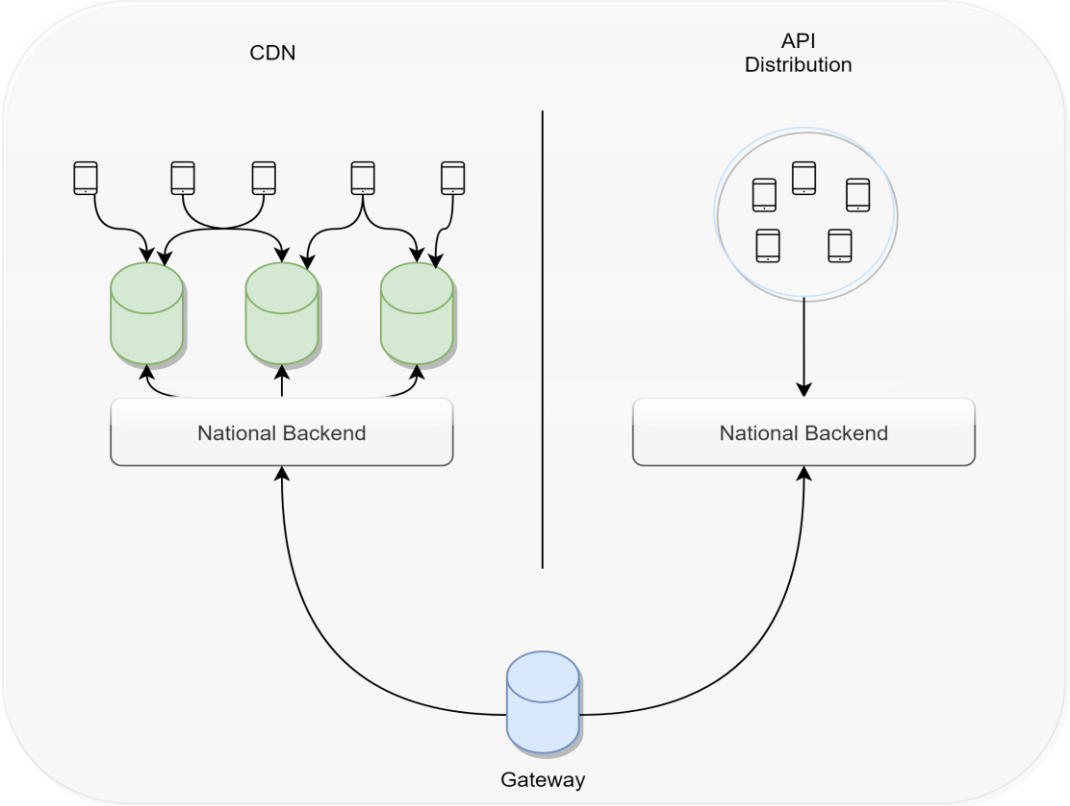
4.8.1. Concept

The download for the app must be as minimal as possible, because the download amount of millions of devices can be a very high traffic for the network. For this purpose all APIs should provide endpoints which deliver only the last modified partitions and/or compressed data. This can be realised by distributing just the latest links to CDN content or by using an API which works with eTags and If-None-Match/If-Match/If-Modified-Since headers.

At first, the app should retrieve information about the available lists associated with a kid, to know what kind of “operation mode” should be applied. With this information, the app can differentiate what kind of algorithm for the selection shall be applied. This can be the case when the revocation entries per kid growing up rapidly (e.g. for suspensions, fakes etc.) With this mechanism the app can be triggered to change the mode without losing verification performance. This information should be provided per app version. The following chapters describe how such a Distribution Structure could look like. **Derivations may be done at any time by the participants.**

4.8.2. Distribution Structure

The distribution of the content can be done via CDN or direct API calls to the national backend. All calls should be optimised with eTags or conditional get calls (“If-Match/If-None match) to avoid unnecessary network traffic. Each delivered content should be signed if a CDN is used.



4.8.3. Download Optimizations

Before the distribution of the partitions and chunks, there can be some assumptions made to optimize the download for the app. Especially for lower end devices the handling of a complete revocation list can be difficult. The following assumptions should be considered:

a) For domestic use, it's not necessary to hold all revocation lists for each kid. To download by default the lists to national KIDs and the list for unknown KIDs may be fit for the most of domestic use cases
b) For the Traveler Check use cases may it be better to hold the lists in the order of travel rates (e.g. Mallorca with a lot of germans/britain's visitors) It would make perhaps more sense to download revocation lists according to incoming flights and download some lists dynamically from a local server
c) KIDs which are removed from the trustlist, can be removed for the download as well
d) Wallet Apps need not the entire list because they own just a few certificates of the holder. This can be realized over requesting the single partition or information if the certificates are valid or not.

4.8.4. Delete Optimizations

To avoid expensive re-downloads, the data should be grouped in chunks which have the same expiration date. The app can decide then which chunks can be deleted from the storage without any re-download of metadata information.

4.8.5. Wallet App Lookup

The usage of bloom filters or other probabilistic data structures results in the acceptance of a higher false positive rate than with SHA256 hashes, which must be mitigated by a comparement functionality over the wallet app/website. For instance, if a verifier app highlights a digital covid certificate as revoked, each holder must be able to verify this statement by checking their own digital covid certificates against the revocation list. To avoid the loading of the entire revocation lists, the wallet apps should be able to verify the state of their own certificates. This request can be made over:

a) Download of associated partitions
b) Authorised Online Lookup for revocation

All wallet apps must consider the GDPR regulations for the revocation list handling.

Authorization

For an online lookup an authorization from the personal data protection legislation perspective (GDPR and/or other applicable laws) is necessary. A public endpoint for checking the revocation state should therefore have an endpoint with an authorization concept. This can be:

a) An authorization over the keypair which was used for claiming a certificate
b) An authorization by ZKP (Zero Knowledge Proof)
c) An authorization by a Challenge Response
d) An authorization by other systems (e.g. National ID, OIDC etc.)
e) An authorization by DCC itself + Cross Check against Bloomfilter AND Hashlist

It doesn't matter which system is used in the national implementation, but it must be guaranteed that access is only limited to the current holder of the certificate. An identity checkup of the holder before the lookup is optional and left to the national regulations.

4.8.6. Lookup Proof

Normally the lookup of certificate revocation status is normally made by the holder in situations where any kind of check fails (e.g. at the Border). In these situations, the verifier needs to be sure that the challenge of the holder was justified. This can be realised over generating lookup proofs which are signed by the revocation list provider. The best lookup proof would be in this case a new issued DCC or a token provided by a validation service to fulfil the claim of compensation.

4.8.7. Example API

4.8.7.1. List Information

This endpoint delivers a list of existing revocation lists by kid with information regarding the partition list. The entire list should be hashed to identify changes easier by an eTag and If-None-Match Headers.

```
Route: /lists
Verb: GET
If-None-Match: 33feea222579184aefd
Content-Type: application/json
Response: Array with KID Objects

[{
   “Kid”:string,
   “Mode”:Coordinate|Vector|Point,
   “hashType”:”UCI|SIGNATURE|COUNTRYUCI”,
   “expired”:{DATE},
   “lastUpdated”:{DATE}
}]
```

Request Headers

Field	Type	Value
If-None-Match	String	Should contain eTag

Response Headers

Field	Type	Value
eTag	String	Should contain an sha256 hash to identify the last list.

Response Codes

Code	Description
200	OK
304	Not modified, when the eTag matches and the definition has not been modified.

Kid Object

Field	Type	Value

kid	string	Kid of the DSC which are associated with the entries
settings	Array of Setting Objects	Array with available list modes

Setting Object

Field	Type	Value
Mode	Enum (coordinate|vector|point)	Defines what kind of modus should be applied. Coordinate (X,Y), Vector (X, Y=null), Point (X=null,Y=null)
hashType	Enum	Uses Hash Types
Expired	Date	Date of Expiration
lastUpdated	Date	Last Updated

4.8.7.2. List Partitions

This call should deliver all existing partitions for a certain kid. If the call returns 412, the /lists must be loaded again, because the partitions have been changed.

/lists/{kid}/partitions

Verb: GET
If-Match 2344ffffff
Content-Type: application/gzip | Accept-Encoding: gzip
Response: Zip or COSE SIGN1 Message

[Metadata] //metadata Array

Request Headers

Field	Type	Value
If-Match	String	Value of the given eTag by the /lists call
If-Modified-Since	Date	Date of the last changed partitions

Response Codes

Code	Description
200	OK
412	Precondition Failed.

4.8.7.3. List Chunks

Delivers an zip or array of the existing chunks within a partition.

/lists/{kid}/partitions/{id}/chunks

Verb: GET
If-Match 3333dd3444
Content-Type: application/gzip | Accept-Encoding: gzip
Response: Zip or COSE SIGN1 Messages

```
[
  chunk1,
  chunk2
]
```

Note: The partition ID should be a combined value of x and y.

Request Headers

Field	Type	Value
If-Match	String	Value of the given eTag by the /lists call
If-Modified-Since	Date	Date of the last changed partitions

Response Codes

Code	Description
200	OK
412	Precondition Failed.

4.8.7.4. Retrieve Chunk

Delivers a compressed chunk.

/lists/{kid}/partitions/{id}/chunks/{chunkId}

Verb: GET
If-Match: 5822115ba444aaa43b5e33f (eTag)
Content-Type: application/gzip | Accept-Encoding: gzip
Response: Zip or COSE SIGN1 Message

Request Headers

Field	Type	Value
If-Match	String	Value of the given eTag by the /lists call

Response Codes

Code	Description
200	OK
412	Precondition Failed.

4.10. Data Storage

The data should be stored in a local database or file system which is indexed by kid, x,y, and chunkID. A special chunk id is “meta”. In the case of a database the necessary revocation data can be accessed by querying the metadata, finding the right chunk, and loading the chunk in the memory for comparison.

If the “expires” date is reached, the chunk can be deleted.

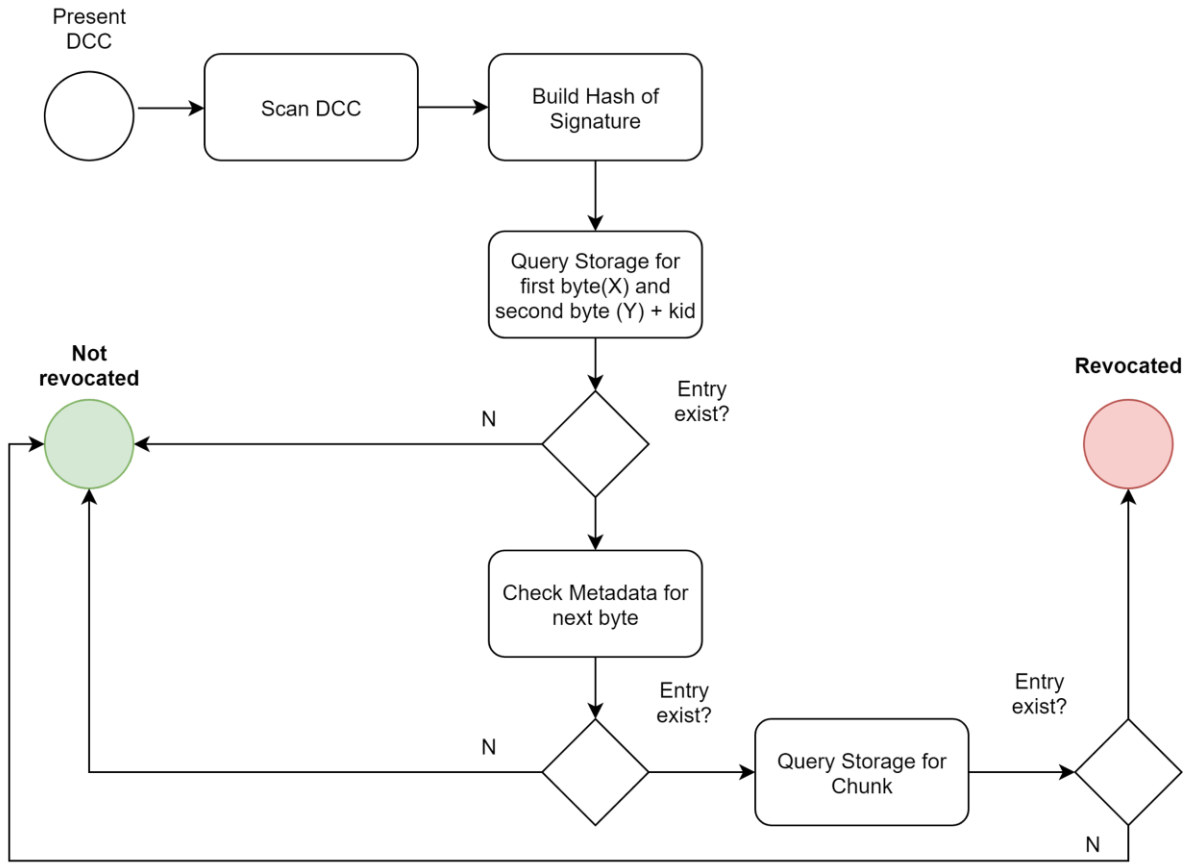
Field	**Type**
kid	byte
x	byte
y	byte
z	byte
chunkId	varchar
expires	DateTime

4.11. Entry Matching

4.11.1. Coordinate Matching

The entry matching works in the following way:

1) Calculate the Hash(es) of the DCC
2) Query the storage for the current bytes (X/Y) and kid to find the metadata of the Partition
3) If not, no revocation exists. If found, query for the chunk by using the next byte (and optional dates)
4) If no chunk is found, no revocation exists. Is a chunk found, query it for entries.
5) Contains the chunk the hash, the DCC was revoked.



4.11.2. Point Matching

Is the kid restricted to a point mode, X,Y is set to null. All other steps are the same as in the coordinate mode.

4.11.3. Vector Matching

Is the kid restricted to a point mode, Y is set to null and X is calculated by the first byte of the hash. All other steps are the same as in the coordinate mode.

5. APPENDIX

5.1. ZIP File Example

A zip file can have the following entry structure:

- KID
    - Paritition ID
        - ChunkID
            - Slice Expireing Date
                - Hash.dat (Payload e.g. Bloom Filter)

5.2. Metadata CBOR Format Example

5.2.1. Payload

Field	Major Type	Sub Type	Description
Kid	2	Byte	Bytes of kid
X	2	Byte	Bytes of x
Y	2	Byte	Bytes of y
Version	2	Byte string	Version of metadata
Meta	4	Array of Maps	Array of Map Elements of Type Content Map

5.2.2. Content Map

A Content Map is tagged by value in the range of 65536-1330664269.

Field	Major Type	Sub Type	Description
Tag	2	Byte string	Any arbitrary tag which can be defined by the backend.

Chunks	5	Array	Map of a chunk list object.

5.2.3. Chunk List Object

Field	Major Type	Sub Type	Description
chunkId	2	Byte string	chunkId
Section	2	Byte string	Section
Expire	5	DateTime	Posix Value (Tag 1)
Chunk	5	Map	Map of a chunk Object

5.2.4. Chunk Object

Field	Major Type	Sub Type	Description
type	2	Byte string	BLOOM|HASHLIST
version	2	Byte string	Semver Version
cid	2	Byte string	Chunk id

5.3. Hash List CBOR Format Example

Field	Major Type	Sub Type	Description
kid	2	Byte	Bytes of kid

x	2	Byte	Bytes of x
y	2	Byte	Bytes of y
z	2	Byte	Bytes of z
chunkid	2	Byte string	Unique Id.
hash	4	Byte String Array	Blank Bytes of Hashes in array
expired	0	DateTime	Posix Value (Tag 1)

5.4. Bloom Filter CBOR Format Example

Field	Major Type	Sub Type	Description
kid	2	Byte	Bytes of kid
x	2	Byte	Bytes of x
y	2	Byte	Bytes of y
z	2	Byte	Bytes of z
chunkid	0	Unsigned Integer	Unique Id.
filter	2	Byte	Blank Bytes of Bloom Filter.
k	0	Unsigned Integer	Amount of Hash Calculations.
expired	0	DateTime	Posix Value (Tag 1)