SKM Simple Key Management API

Release 1.0.0

Axiomatic Systems

May 08, 2015

Contents

SKM - Simple Key Management

The SKM API is a simple REST API designed to interface to local or remote key servers. It is an **Open Specification** that may be implemented by different software packages, like the SkeyMa Key Server , or online services, such as the ExpressPlay service. Different implementations of this API may differ slightly in the way they deal with access control, logging, and other functions, but the core SKM REST API documented here is common to all the implementations.

The purpose of the SKM API is to provide a very simple interface for software and services that need a simple and convenient way to store and/or retrieve cryptographic keys. This includes content packagers, head-end scramblers, DRM license servers, etc.

Contents:

CHAPTER 1

SKM REST API

The SKM API is an interface to a remote or local storage and management of cryptographic keys (media encryption keys for example). Each Key object has a unique key ID (KID) and a value (typically a 16-byte random number, like an AES-128 cipher key). Keys are stored encrypted with a KEK (Key Encryption Key) using a key wrapping algorithm (AES Key Wrap, **RFC 3394**). Each KEK has a corresponding KEKID (Key Encryption Key ID) which can be caller-assigned, or automatically computed by the server by deriving from the KEK value using a one way function. The server does not store the KEKs. Keys can be stored and retrieved in their encrypted form (in this case it is up to the caller of the API to perform the key wrapping and/or unwrapping), or they can be stored and retrieved in clear-text form by passing the KEK as an API parameter whenever needed (this this case, it is the server that performs the key wrapping and/or unwrapping).

1.1 Data Model

A Key object has the following properties:

- KID (Unique)
- Key Value (byte array, encrypted with KEK). This value is not stored on the server
- EK (encrypted representation of the Value). This is stored on the server
- KEK_ID (String that identifies a KEK. May be derived from KEK through a one-way function)
- Info (arbitrary string)
- Content ID (arbitrary string used to remember the association of the key with a specific content/file)
- Expiration (if not null, the date/time at which the Key object expired; it may be removed automatically by the server)
- Last Update (date/time at which the Key object was last updated)

JSON representation

```
{
    "kid":        <hex>    // (KID, 32 hex chars
    "k":          <hex>    // Key Value (not stored in DB, dynamically computed from "ek")
    "ek":         <hex>    // Key Value encrypted with KEK using AES Key Wrap
    "kekId":      <string> // KEK identifier
    "info":       <string> // (optional)
    "contentId":  <string> // (optional)
    "expiration": <date>   // (optional) expiration date of the object (ISO 8601 Extended Format)
    "lastUpdate": <date>   // (server-assigned) (ISO 8601 Extended Format)
}
```

1.2 Encrypted vs Clear-text Key Values

Keys are always stored on the server in their encrypted form. When Key values are exchanged through the API (either sent or received), they may be expressed either in their encrypted form (AES Key Wrap with a KEK), or in their raw clear-text form. If Key values are exchanged in their encrypted form, it is up to the caller to wrap or unwrap the key values with the appropriate KEK. If a `kek` URL query parameter is present, the server will automatically perform the Key value wrapping or unwrapping.

1.3 REST API

1.3.1 Common API parameters

kek (optional). When specified, this parameter contains the 16-byte KEK used to unwrap the key value, in hexadecimal (32 hex characters)

1.3.2 Optional API parameters and extensions

Note: Some implementations of the SKM API require API callers to authenticate themselves. This may be implemented in a number of different ways, including using a URL query parameter (like for example an API key passed in a query parameter, named `apiKey`, `customerAuthenticator`, or something similar). URL query parameters must be separated with an `&` character.

1.3.3 Special KID syntax

In order to facilitate the work of API clients, a 16-byte KID can be replaced by a `^` character followed by a string. In that case, the KID is simply computed as the 16-byte truncated SHA1 hash of the string.

For example, if the KID is specified as the string `^kid1`, the actual KID value is `80ea8bc8a58f990ad1f76bc665b30bfa`.

1.3.4 Root URL

Note: The URLs below are relative to a root URL for the Key Management Service on a server. In these examples, the root URL path is `/`. But a server may use a different root URL path. For example, if the Key Management Service on a server named `api.service.expressplay.com` is at a root URL path `/keystore`, then the URL relative path `keys/{kid}` would result in a final URL `https://api.service.expressplay.com/keystore/keys/{kid}` if accessed with HTTPS.

1.3.5 API URLs

`POST /keys`

Create a new Key object

The `POST` body is either empty, in which case a brand new object, with a random KID, is created and returned (a KEK **must** be supplied through the `kek` URL query parameter), or contain a JSON object with a partial or complete Key object.

A partial Key object is one where not all fields are set. Fields that are not set are automatically generated by the server as follows:

kid If the KID is not specified, a new random KID is chosen by the server. If the KID is specified, a new object is created with the specified KID, unless a Key object with the same KID already exists, in which case the rest of the POST body is ignored and the existing Key object is returned with an `HTTP 200` status code.

k If the clear-text Key Value is not specified, a new random key value will be chosen, using a cryptographically-strong random number generator.

ek If the encrypted Key Value is not specified, a new random key value will be chosen, using a cryptographically-strong random number generator.

kekId If a KEK ID is not specified, the server generates a KEKID value uniquely identifying the KEK

info If the Key Info is not specified, this field remains empty

contentId If the Content ID is not specified, this field remains empty

expiration If no Expiration is specified, the expiration field is not set and the object does not expire.

The response contains a JSON object representing the Key object that was created or found.

Note: It is important to set the `Content-Type` HTTP header to `application/json` when issuing a `POST` request with a JSON body

---

Example Request: create a new random Key object

```
POST /keys?kek=000102030405060708090a0b0c0d0e0f HTTP/1.1
```

Response

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /keys/4e2df6b45e8257e187b2802b22ae7418

{
  "kid":    "4e2df6b45e8257e187b2802b22ae7418",
  "k":      "a9b9033df0b9ca5447839e3d074817a0",
  "ek":     "5dbd06c0056b42fe0b8cf406679620c31bd619732730433d",
  "kekId": "#1.afe008a381bdac03b412a92d54b92ddf"
}
```

Example Request: create a new Key object with all fields already set

```
POST /keys?kek=000102030405060708090a0b0c0d0e0f HTTP/1.1

{
  "kid":        "4e2df6b45e8257e187b2802b22ae7418",
  "k":          "a9b9033df0b9ca5447839e3d074817a0",
  "kekId":      "my-kek-id-1",
  "contentId": "urn:mynamespace:my-content-id-1234",
  "info":       "some comment"
}
```

Response

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /keys/4e2df6b45e8257e187b2802b22ae7418

{
```

```
    "kid":       "4e2df6b45e8257e187b2802b22ae7418",
    "k":         "a9b9033df0b9ca5447839e3d074817a0",
    "ek":        "5dbd06c0056b42fe0b8cf406679620c31bd619732730433d",
    "kekId":     "my-kek-id-1",
    "contentId": "urn:mynamespace:my-content-id-1234",
    "info":      "some comment"
}
```

Query Parameters

- **kek** – (optional) KEK used to unwrap the key value

Request JSON Object

- **kid** (*string*) – KID (32 hex characters)
- **k** (*string*) – Clear-text Key value (hex) [requires that the 'kek' query parameter be passed]
- **ek** (*string*) – Encrypted Key value (hex) [mutually exclusive with the presence of a 'k' field]
- **kekId** (*string*) – (optional) KEK Id
- **info** (*string*) – (optional) Key info
- **contentId** (*string*) – (optional) contentId
- **expiration** (*string*) – (optional) Object expiration data/time

Status Codes

- 200 OK – an existing Key object was found and returned
- 201 Created – a new Key object successfully created

PUT /keys/{kid}

Update a Key object

The `PUT` body must contain a JSON object for a partial or complete Key object. Fields that are not specified in the body will not be updated. The `kid` field, if present in the body, is ignored.

Note: It is important to set the `Content-Type` HTTP header to `application/json` when issuing a `PUT` request with a JSON body

---

Example Request: change the contentId of a Key object

```
PUT /keys/11a48707853ed5f13485f161523ffdc4 HTTP/1.1

{
  "contentId": "urn:namespace:x1234yyu"
}
```

Response

```
HTTP/1.1 200 OK
```

Query Parameters

- **kek** – (optional) KEK used to unwrap the key value

Request JSON Object

- **k** (*string*) – Clear-text Key value (hex) [requires that the 'kek' query parameter be passed]

- **ek** (*string*) – Encrypted Key value (hex) [mutually exclusive with the presence of a 'k' field]
- **kekId** (*string*) – (optional) KEK Id
- **info** (*string*) – (optional) Key info
- **contentId** (*string*) – (optional) contentId
- **expiration** (*string*) – (optional) Object expiration data/time

Status Codes

- 200 OK – no error
- 404 Not Found – key not found

GET /keys/{kid}

Returns one Key object

Example Request (without specifying the KEK)

```
GET /keys/11a48707853ed5f13485f161523ffdc4 HTTP/1.1
```

Example Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "kid":   "11a48707853ed5f13485f161523ffdc4",
  "ek":    "b6862c586af0d70fdc594deb7b254bb38937113dbc6411ea",
  "kekId": "#1.afe008a381bdac03b412a92d54b92ddf"
}
```

Example Request (with KEK)

```
GET /keys/11a48707853ed5f13485f161523ffdc4?kek=000102030405060708090a0b0c0d0e0f HTTP/1.1
```

Example Response

```
HTTP/1.1 200 OK
Content-Type: application/json


{
  "kid":   "11a48707853ed5f13485f161523ffdc4",
  "k":     "d4783a651c96a872daa145ce1a378153",
  "kekId": "#1.afe008a381bdac03b412a92d54b92ddf"
}
```

Query Parameters

- **kek** – (optional) KEK used to unwrap the key value

Status Codes

- 200 OK – no error
- 400 Bad Request – bad request (ex: wrong KEK passed)
- 404 Not Found – key not found

GET /keys/{kid}/value
Returns one Key value

Instead of returning a complete JSON Key object, this request returns only the Key Value, as a hex string. If a KEK is passed in the *kek* query parameter, the response body contains the raw clear-text value of the Key object. If no KEK is passed, the response body contains the encrypted Key Value, prefixed with a # character

Example Request (with KEK)

```
GET /keys/00112233445566778899aabbccddeefc/value?kek=00112233445566778899aabbccddeeff HTTP/1.1
```

Example Response

```
HTTP/1.1 200 OK
Content-Type: text/plain

1234123412341234123412341234123412341234
```

Example Request (without KEK)

```
GET /keys/00112233445566778899aabbccddeefc/value HTTP/1.1
```

Example Response

```
HTTP/1.1 200 OK
Content-Type: text/plain

#ffaf1dae9201d1adf62770dca5ddb77ad773a79369e39986
```

Query Parameters

- **kek** – (optional) KEK used to unwrap the key value

Status Codes

- 200 OK – no error
- 400 Bad Request – bad request (ex: wrong KEK passed)
- 404 Not Found – key not found

GET /keys/{kid1},{kid2},...
Returns mutliple Key objects

When multiple KIDs are specified, separated by ',' characters, multiple Key objects can be retrieved with a single request. Just like for other requests, each KID may be expressed as a 32-character hex string, or a '^' followed by an arbitrary string. The response body contains a JSON array of Key objects

Example Request

```
GET /keys/00112233445566778899aabbccddeefb,00112233445566778899aabbccddeefa,0011223344556677889
```

Example Response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "kid":  "00112233445566778899aabbccddeeff",
    "k":    "ea85a33da18d55ffead60509a5666ad1"
  },
  {
```

```
        "kid":  "00112233445566778899aabbccddeefa",
        "k":    "0ae81ee0bc16917f3758324c151f7010"
      },
      {
        "kid":  "00112233445566778899aabbccddeefb",
        "k":    "a0a1a2a3a4a5a6a7a8a9aaabacadaeaf"
      }
    ]
```

Query Parameters

- **kek** – (optional) KEK used to unwrap the key value

Status Codes

- 200 OK – no error
- 400 Bad Request – bad request (ex: wrong KEK passed)
- 404 Not Found – key not found

GET /keys/{kid1},{kid2},.../value
Returns multiple Key object values

This variant of the multiple-KID request returns the key values only instead of an array of JSON Key objects. As with the single-KID Key value request, the response body contains Key values either in raw clear-text form (when a KEK is passed), or in wrapped form (prefixed with '#'). The Key values in the response body are separated by ',' characters

Example Request (with KEK)

```
GET /keys/00112233445566778899aabbccddeefb,00112233445566778899aabbccddeefa,0011223344556677889
```

Example Response

```
HTTP/1.1 200 OK
Content-Type: text/plain

a0a1a2a3a4a5a6a7a8a9aaabacadaeaf,0ae81ee0bc16917f3758324c151f7010,ea85a33da18d55ffead60509a5666a
```

Example Request (without KEK)

```
GET /keys/00112233445566778899aabbccddeefb,00112233445566778899aabbccddeefa,0011223344556677889
```

Example Response

```
HTTP/1.1 200 OK
Content-Type: text/plain

#7c98f3e4d60636d4aef4977d12dbfe75611dbd03e54dffef,#83017d13dc5067c1cff0ecab23184fd721832ad61f79e
```

Query Parameters

- **kek** – (optional) KEK used to unwrap the key value

Status Codes

- 200 OK – no error
- 400 Bad Request – bad request (ex: wrong KEK passed)
- 404 Not Found – key not found

DELETE /keys/{kid}
Delete a Key object

Example Request

```
DELETE /keys/00112233445566778899aabbccddeefb HTTP/1.1
```

Example Response

```
HTTP/1.1 200 OK
```

Status Codes

- 200 OK – no error
- 404 Not Found – key not found

GET /keys
Returns all the Key objects stored on the server

Example Request

```
GET /keys HTTP/1.1
```

Example Response

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "kid":   "11a48707853ed5f13485f161523ffdc4",
    "ek":    "b6862c586af0d70fdc594deb7b254bb38937113dbc6411ea",
    "kekId": "#1.afe008a381bdac03b412a92d54b92ddf"
  },
  {
    "kid":   "f0bacfca77d36361179b36a4cbee8abf",
    "ek":    "a23ce0ab465f36a56f6e2863b16778cb7c7064662c1cbfa0",
    "kekId": "#1.afe008a381bdac03b412a92d54b92ddf",
    "info":  "foobar"
    }
]
```

Query Parameters

- **kek** – (optional) KEK used to unwrap the key value

Status Codes

- 200 OK – no error

GET /keycount
Returns the number of Key objects stored on the server, as a JSON object with a "keyCount" integer field

Example Request

```
Get /keycount HTTP/1.1
```

Example Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "keyCount": 1567
}
```

Status Codes

- 200 OK – no error

CHAPTER 2

SKM API Examples and Cookbook

2.1 Create a new Key, with a server-assigned value and KID

POST an empty body, or a body with an empty `{}` JSON Object. The server will create a new random key and assign it a new random KID.

Because the server only stores encrypted keys, the `kek` parameter is required

Request

```
POST /keys?kek=000102030405060708090a0b0c0d0e0f HTTP/1.1
```

Response

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /keys/4e2df6b45e8257e187b2802b22ae7418

{
  "kid":   "4e2df6b45e8257e187b2802b22ae7418",
  "k":     "a9b9033df0b9ca5447839e3d074817a0",
  "ek":    "5dbd06c0056b42fe0b8cf406679620c31bd619732730433d",
  "kekId": "#1.afe008a381bdac03b412a92d54b92ddf"
}
```

2.2 Get a key by KID, with server auto-creation of the key if it does not exist

POST a body with a partial JSON Key Object, including a `kid` field. If a Key Object with that KID already exists on the server, that object is returned, with an `HTTP 200` response code. If no such Key Object already exists, a new one is created, with a new random key value.

Because the server only stores encrypted keys, the `kek` parameter is required

Request

```
POST /keys?kek=000102030405060708090a0b0c0d0e0f HTTP/1.1
Content-Type: application/json

{
```

```
  "kid":   "4e2df6b45e8257e187b2802b22ae7418",
}
```

Response if no key with that KID exists on the server

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /keys/4e2df6b45e8257e187b2802b22ae7418

{
  "kid":   "4e2df6b45e8257e187b2802b22ae7418",
  "k":     "a9b9033df0b9ca5447839e3d074817a0",
  "ek":    "5dbd06c0056b42fe0b8cf406679620c31bd619732730433d",
  "kekId": "#1.afe008a381bdac03b412a92d54b92ddf"
}
```

Response if a key with that KID already exists on the server

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "kid":   "4e2df6b45e8257e187b2802b22ae7418",
  "k":     "a9b9033df0b9ca5447839e3d074817a0",
  "ek":    "5dbd06c0056b42fe0b8cf406679620c31bd619732730433d",
  "kekId": "#1.afe008a381bdac03b412a92d54b92ddf"
}
```

2.3 Using KIDs generated from strings

In some cases, it may be useful to have KIDs that correspond to a well-defined scheme, so that they can follow a pattern instead of being randomly generated.

For example, let's say we are deploying a system with live TV channels for streaming. Each channel must be encrypted with a different key, and the keys must change every day. Instead of picking a random KID for each channel for each day, a simpler approach is to use a pattern where we assign a key name to each channel/day. We can represent the channel by its name and the day by the string YYYY.MM.DD. For example, the key name for channel CNN on December 18 2014 would be: CNN.2014.12.18

We can now use the `^string` KID syntax instead of using hex KID representations. To obtain the key for Channel CNN for December 18 2014, we would get:

`GET /keys/^CNN.2014.12.18`

or, if we want the server to auto-create the key if it doesn't already exist:

`POST /keys`

```
Content-Type: application/json

{
  "kid": "^CNN.2014.12.18"
}
```

This is much more convenient than having to remember a different random KID for each day for each channel.

2.4 Wrapping keys client-side

Sometimes it may be desirable to perform key wrapping/unwrapping on the client side, instead of passing a KEK (Key Encryption Key) and ask the server to do it. For instance, the client may want to use a specific cryptographic random number generator, or may not want to pass a KEK to the server. This, of course, requires the client to be able to perform the proper AES Key Wrap cryptographic operations. To keep the wrapping/unwrapping entirely client-side, simply ommit the `kek` query parameter in requests and supply the `ek` value when creating the key.

Request

```
POST /keys HTTP/1.1
Content-Type: application/json

{
  "kid":   "4e2df6b45e8257e187b2802b22ae7418",
  "ek":    "5dbd06c0056b42fe0b8cf406679620c31bd619732730433d",
  "kekId": "my-kek-id-1234"
}
```

Response

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /keys/4e2df6b45e8257e187b2802b22ae7418

{
  "kid":   "4e2df6b45e8257e187b2802b22ae7418",
  "ek":    "5dbd06c0056b42fe0b8cf406679620c31bd619732730433d",
  "kekId": "my-kek-id-1234"
}
```

/keycount

`GET /keycount`, 10

/keys

`GET /keys`, 10
`GET /keys/^CNN.2014.12.18`, 14
`GET /keys/{kid1},{kid2},...`, 8
`GET /keys/{kid1},{kid2},.../value`, 9
`GET /keys/{kid}`, 7
`GET /keys/{kid}/value`, 7
`POST /keys`, 4
`PUT /keys/{kid}`, 6
`DELETE /keys/{kid}`, 9

R