01-Web_Cryptography_API

Introduction

This note gather information obtained during my discovery of the **Web Cryptography API**.

Security attention points

All element marked with ⚠ are points that must taken in account from a security point of view.

Misc

- Table were generated with this tool.

Study roadmap

- [x] Search and explore documentation and tutorials.
- [x] Implement the labs.
- [ ] Create a XLM blog post about all this study.

Sources

- RFC
- MDN DOCS
- MDN SAMPLES
- CHROMIUM

Note for the blog post

The blog post will have the following section:

1. What is the Web Cryptography API?
2. Browser support level?
3. Why it is interesting to use it (pro/con)?
4. Global overview of its capabilities

For the point 4:

Demonstration of usage for the different kind of crypto operations:

- Hash

- HMAC
- Signature
- Key handling
- Security random number generation (PRNG)
- Symetric encryption
- Asymetric encryption

Research

What is the Web Cryptography API?

> Information below are taken from the RFC.

JavaScript API for performing basic cryptographic operations in web applications,
such as hashing, signature generation and verification, and encryption and decryption.

Additionally, it describes an API for applications to generate and/or manage the
keying material necessary to perform these operations.

Cryptographic transformations are exposed via the `SubtleCrypto` interface, which defines a set of methods for performing common cryptographic operations.

This specification does not dictate a mandatory set of algorithms that MUST be implemented (see the reason here) however it provides descriptions for a variety of algorithms that authors may wish to use and that User Agents may choose to implement.

This API does not deal with or address the discovery of cryptographic modules (see the reason here).

This specification does not define any specific mechanisms for the storage of cryptographic keys:

> By default, unless specific effort is taken by the author to persist keys, such as through the use of the Indexed Database API, keys created with this API will only be valid for the duration of the current page (e.g. until a navigation event). Authors that wish to use the same key across different pages or multiple browsing sessions must employ existing web storage technologies

The only requirement for the API is that key material is not exposed to script, except through the use of the `exportKey` and `wrapKey` operations.

⚠ In particular, the API does not guarantee that the underlying cryptographic key material will not be persisted to disk, possibly unencrypted, nor that it will be inaccessible to users or other applications running with the same privileges as the User Agent.

⚠ Any application or user that has access to the device storage may be able to recover the key material, even through scripts may be prohibited.

⚠ This specification places no normative requirements on how implementations handle key material once all references to it go away. That is, conforming user agents are not required to zeroize key

material, and it may still be accessible on device storage or device memory, even after all references to the CryptoKey have gone away.

⚠ Developers making use of the `SubtleCrypto` interface are expected to be aware of the security concerns associated with both the design and implementation of the various algorithms provided.

⚠ This specification includes several algorithms which, in their default usage, can result in cryptographic vulnerabilities. While these concerns may be mitigated, such as through the combination and composition with additional algorithms provided by this specification, authors should proceed with caution and review the relevant cryptographic literature before using a given algorithm: See here.

⚠ Security considerations.

Supported operations, see here and here :

- Get random bytes
- Encrypt / Decrypt using symetric/asymetric algorithm/key
- Sign and verify signature using symetric/asymetric algorithm/key
- Hash
- Generate symetric/asymetric key
- Derive key symetric/asymetric key
- Import/export symetric/asymetric key
- wrap/unwrap symetric/asymetric key

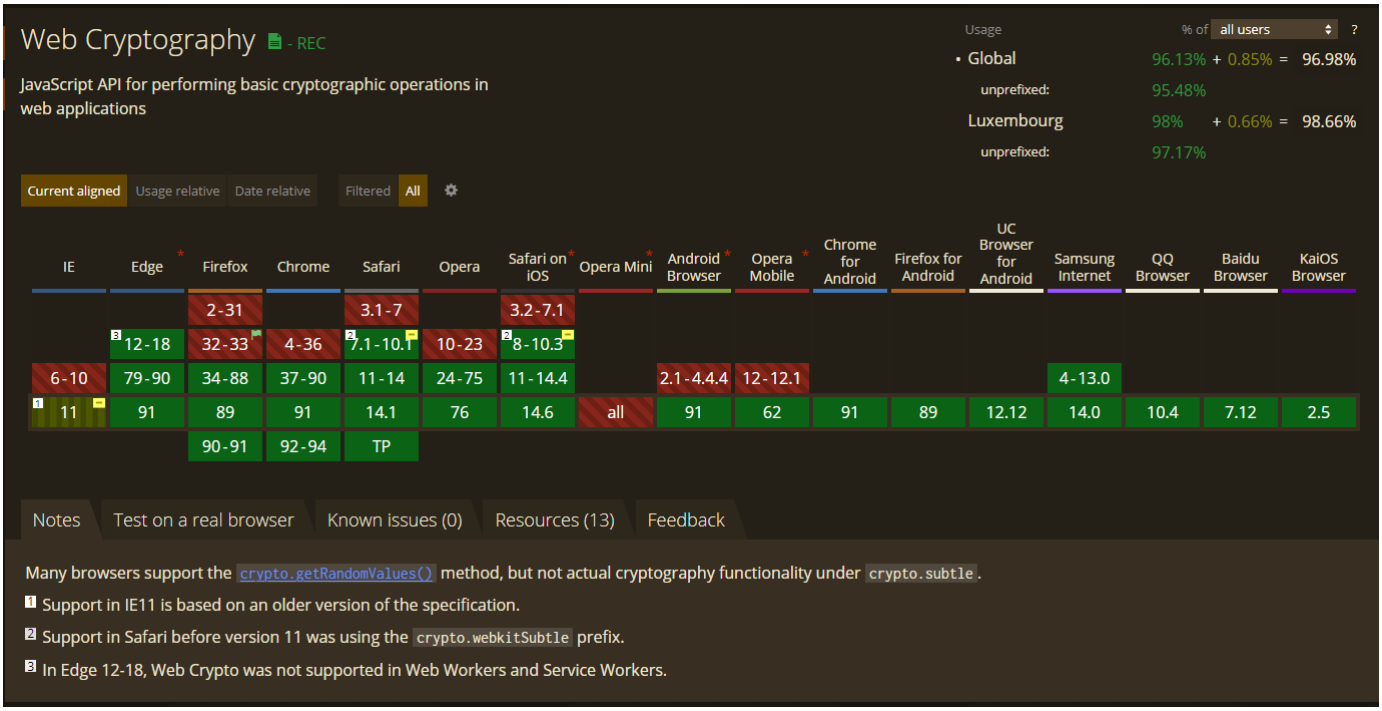
Supported key format, see here :

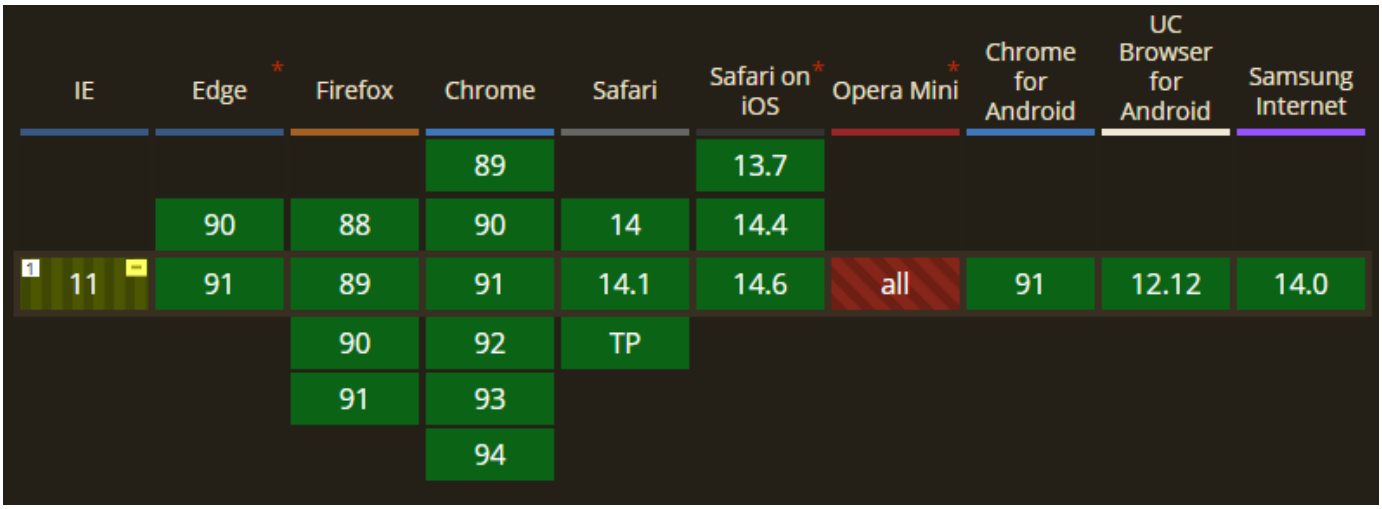
- **raw**: An unformatted sequence of bytes. Intended for secret keys.
- **pkcs8**: The DER encoding of the `PrivateKeyInfo` structure.
- **spki**: The DER encoding of the `SubjectPublicKeyInfo` structure.
- **jwk**: The key is a JsonWebKey dictionary encoded as a JavaScript object.

Browser support level?

Support on June 2021:

> Source: https://caniuse.com/cryptography





> Source: https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API#browser_compatibility

Browser compatibility

Crypto

Report problems with this compatibility data on GitHub

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	Node.js
Crypto	11	12	26	11	15	6.1	37	18	26	14	6.1	1.0	15.0.0 ★ ▼
getRandomValues()	11	12	26	11	15	6.1	37	18	26	14	6.1	1.0	15.0.0
subtle	37	12	34	11	24	11 ▼	37	37	34	24	11 ▼	3.0	15.0.0
Secure context required	60	79	75	No	47	No	60	60	No	47	No	8.0	?

Full support

Partial support

No support

Compatibility unknown

Why it is interesting to use it (pro/con)?

The list below present some pros and cons of the using this API.

Pros:

- Ensure that implementation of the algorithms is done by crypto specialists.
- Ensure that security issues on implementation of the algorithms are automatically patched during the browser update process.
- Ensure that the maintenance/update of the API/implementation of the algorithms is done by the browser provider (no abrupt stop of the support).
- Provide, in theory, a "portability" across all browsers supporting the API.

Cons:

- As patching of security issues on implementation of the algorithms is handled by browsers so user must upgrade it browsers itself (warning is possible using JS with browser version info).
- The API support vulnerable algorithms so care must be taken on the algorithms used.
- Security of the key storage is not covered by the API so it must be managed by the app.

- The API RFC places no normative requirements on how implementations handle key material once all references to it go away so exposure of the key is possible in case of memory handling issue.
- As the API provide low level operations, it's up to the developer to ensure a correct usage of the target algorithm according to its specificities (ex: in AES-GCM, requirement of a unique NONCE for each encryption operation).

💡 Even if cons exists, usage of this API is desirable over custom JS libraries if only for the fact it ensure that implementation of the crypto algorithm is performed by specialist in the domain as well the consistent process of patchning provided by the browser. Indeed, most of modern browsers like Firefox, Chrome, Opera and Edge provide auto-update process.

Global overview of its capabilities

📝 Base references used in all code snippets:

```
//See https://developer.mozilla.org/en-US/docs/Web/API/Window/crypto
const CRYPTO_OBJ = window.crypto;
//See https://developer.mozilla.org/en-US/docs/Web/API/TextEncoder
const TEXT_ENCODER = new TextEncoder("utf8");
//See https://developer.mozilla.org/en-US/docs/Web/API/TextDecoder
const TEXT_DECODER = new TextDecoder("utf8");
```

📝 Conversion to HEX function used in some code snippets (credits):

```
//Credits: https://stackoverflow.com/a/40031979/451455
function toHex(buffer) {
    return [...new Uint8Array(buffer)]
        .map(x => x.toString(16).padStart(2, "0"))
        .join("");
}
```

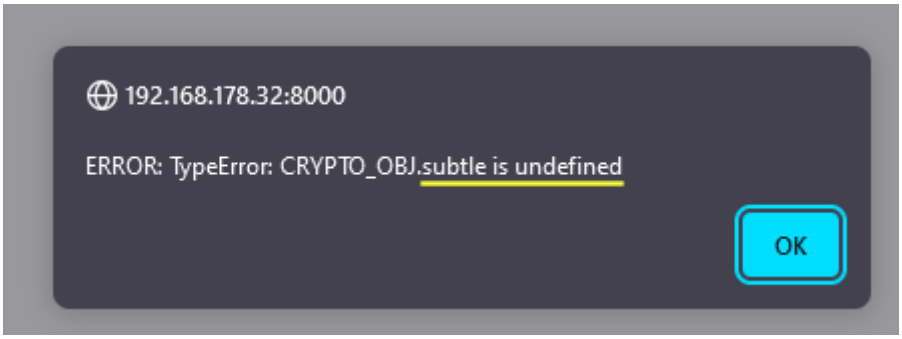
📝 Reference to **Secure Context** means, to simplify, that the protocol used is HTTPS:

SubtleCrypto

Secure context

This feature is available only in secure contexts (HTTPS), in some or all supporting browsers.

The following error is raised when the HTTP protocol (non secure context) is used but it is not supported due to the requirement of a **Secure Context** (`subtle` attribute member of the `crypto` object is not defined):



📝 Everytime it was possible, the strongest algorithms and key lengths were used in order to evaluate the browser supports as well as behavior.

Random values generation

> Source: https://developer.mozilla.org/en-US/docs/Web/API/Crypto/getRandomValues

⚠ Don't use `getRandomValues()` to generate encryption keys. Instead, use the generateKey() method. There are a few reasons for this; for example, `getRandomValues()` **is not guaranteed to be running in a secure context**.

⚠ See value limitation here.

Can be used in the context of the following protocols:

- HTTP: ✔
- HTTPS: ✔

Code snippet:

```
function performRandomValuesGeneration(wantedLength){
    let buffer = new Int32Array(wantedLength);
    cryptoObj.getRandomValues(buffer);
    return toHex(buffer);
}
```

Hashing

> Source: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/digest

Can be used in the context of the following protocols:

- HTTP: ✘
- HTTPS: ✔

Code snippet:

```
async function performSha512Hash(sourceData){
    let dataEncoded = TEXT_ENCODER.encode(sourceData);
    let hashBytes = await CRYPTO_OBJ.subtle.digest("SHA-512",
dataEncoded);
```

```
    return toHex(hashBytes);
}
```

Symmetric key handling

> Source: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/generateKey

> Source: https://developer.mozilla.org/en-US/docs/Web/API/AesKeyGenParams

Can be used in the context of the following protocols:

- HTTP: ✗
- HTTPS: ✓

Code snippet:

```
async function performSymmetricKeyGenerationForEncryptionDecryptionUsageWithAESGCM() {
    //Generate a 256 bits key for AES-GCM symmetric encryption algorithm
    //See https://developer.mozilla.org/en-US/docs/Web/API/AesKeyGenParams
    let aesKeyGenParams = {
        name: "AES-GCM",
        length: 256
    };
    let keyUsages = ["encrypt", "decrypt"];
    //See https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/generateKey
    let cryptoKey = await CRYPTO_OBJ.subtle.generateKey(aesKeyGenParams, true, keyUsages);
    return cryptoKey;
}
```

Symmetric encryption and decryption

> Source: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/encrypt

> Source: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/decrypt

> Source: https://developer.mozilla.org/en-US/docs/Web/API/AesGcmParams

Can be used in the context of the following protocols:

- HTTP: ✗
- HTTPS: ✓

Code snippet:

```
async function performEncryptionDecryptionWithAESGCM(sourceData,
cryptoKey) {
    let nonce = new Int32Array(12); //96 bits
    CRYPTO_OBJ.getRandomValues(nonce);
    //See https://developer.mozilla.org/en-US/docs/Web/API/AesGcmParams
    let additData = new Int32Array(16); //128 bits
    CRYPTO_OBJ.getRandomValues(additData);
    let aesGcmParams = {
        name: "AES-GCM",
        iv: nonce,
        additionalData: additData,
        tagLength: 128 //16 bytes
    };
    let dataEncoded = TEXT_ENCODER.encode(sourceData);
    //See https://developer.mozilla.org/en-
US/docs/Web/API/SubtleCrypto/encrypt
    let encryptedData = await CRYPTO_OBJ.subtle.encrypt(aesGcmParams,
cryptoKey, dataEncoded)
    //See https://developer.mozilla.org/en-
US/docs/Web/API/SubtleCrypto/decrypt
    let decryptedData = await CRYPTO_OBJ.subtle.decrypt(aesGcmParams,
cryptoKey, encryptedData)
    let plainText = TEXT_DECODER.decode(decryptedData);
    let result = {
        encryptedData: toHex(encryptedData),
        cycleSucceed: (sourceData === plainText)
    }
    return result;
}
```

HMAC

> Source: https://developer.mozilla.org/en-US/docs/Web/API/HmacKeyGenParams

> Source: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/generateKey

> Source: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/sign

> Source: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/verify

⚠ For the generation of the secret, specifiy the member **length** of the **HmacKeyGenParams** object, only if **you know what you are doing** from a cryptograpghy point of view.

Can be used in the context of the following protocols:

- HTTP: ✕

- HTTPS: ✔

Code snippet:

ℹ This include the **secret** generation for signature operation.

```
async function performSecretGenerationForSignVerifyUsageWithHMAC() {
    //Generate a secret (cryptoKey) for HMAC operation with SHA-512
    //See https://developer.mozilla.org/en-US/docs/Web/API/HmacKeyGenParams
    let hmacKeyGenParams = {
        name: "HMAC",
        hash: "SHA-512"
    };
    let keyUsages = ["sign", "verify"];
    //See https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/generateKey
    let cryptoKey = await CRYPTO_OBJ.subtle.generateKey(hmacKeyGenParams, true, keyUsages);
    return cryptoKey;
}

async function performSignVerifyWithHMAC(sourceData, cryptoKey) {
    let algorithm = "HMAC";
    let dataEncoded = TEXT_ENCODER.encode(sourceData);
    //See https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/sign
    let signature = await CRYPTO_OBJ.subtle.sign(algorithm, cryptoKey, dataEncoded);
    //See https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/verify
    let isValid = await CRYPTO_OBJ.subtle.verify(algorithm, cryptoKey, signature, dataEncoded);
    let result = {
        signature: toHex(signature),
        cycleSucceed: isValid
    }
    return result;
}
```

Asymmetric key handling

> Source: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/generateKey

> Source: https://developer.mozilla.org/en-US/docs/Web/API/RsaHashedKeyGenParams

> Source: https://www.keylength.com/en/3/

⚠ For the generation of the key pair, specifiy for the member **publicExponent** of the **RsaHashedKeyGenParams** object another value than 65537 (`new Uint8Array([1, 0, 1])`), only if **you know what you are doing** from a cryptograpghy point of view.

Can be used in the context of the following protocols:

- HTTP: ✗
- HTTPS: ✔

Code snippet:

```
async function performAsymmetricKeyGenerationForEncryptionDecryptionUsageWithRSAOAEP() {
    //RSA was chosen because EC was not supported by the "algorithm" parameter at the time of the POC (June 2021):
    //See https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/encrypt#parameters
    //Generate a RSA-OAEP key pair with a size of 4096 bits
    //See https://developer.mozilla.org/en-US/docs/Web/API/RsaHashedKeyGenParams
    //See https://www.keylength.com/en/3/
    //See https://developer.mozilla.org/en-US/docs/Web/API/RsaHashedKeyGenParams#properties
    let rsaHashedKeyGenParams  = {
        name: "RSA-OAEP",
        modulusLength: 4096,
        publicExponent: new Uint8Array([1, 0, 1]),
        hash: "SHA-512"
    };
    let keyUsages = ["encrypt", "decrypt"];
    //See https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/generateKey
    let cryptoKeyPair = await CRYPTO_OBJ.subtle.generateKey(rsaHashedKeyGenParams, true, keyUsages);
    return cryptoKeyPair;
}
```

Asymmetric encryption and decryption

> Source: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/encrypt

> Source: https://developer.mozilla.org/en-US/docs/Web/API/RsaOaepParams

Can be used in the context of the following protocols:

- HTTP: ✗
- HTTPS: ✔

Code snippet:

```
async function performEncryptionDecryptionWithRSAOAEP(sourceData,
cryptoKeyPairPublicKey, cryptoKeyPairPrivateKey) {
    let labelData = new Int32Array(32); //256 bits
    CRYPTO_OBJ.getRandomValues(labelData);
    //See https://developer.mozilla.org/en-US/docs/Web/API/RsaOaepParams
    let rsaOaepParams = {
        name: "RSA-OAEP",
        label: labelData
    };
    let dataEncoded = TEXT_ENCODER.encode(sourceData);
    //See https://developer.mozilla.org/en-
US/docs/Web/API/SubtleCrypto/encrypt
    let encryptedData = await CRYPTO_OBJ.subtle.encrypt(rsaOaepParams,
cryptoKeyPairPublicKey, dataEncoded)
    //See https://developer.mozilla.org/en-
US/docs/Web/API/SubtleCrypto/decrypt
    let decryptedData = await CRYPTO_OBJ.subtle.decrypt(rsaOaepParams,
cryptoKeyPairPrivateKey, encryptedData)
    let plainText = TEXT_DECODER.decode(decryptedData);
    let result = {
        encryptedData: toHex(encryptedData),
        cycleSucceed: (sourceData === plainText)
    }
    return result;
}
```

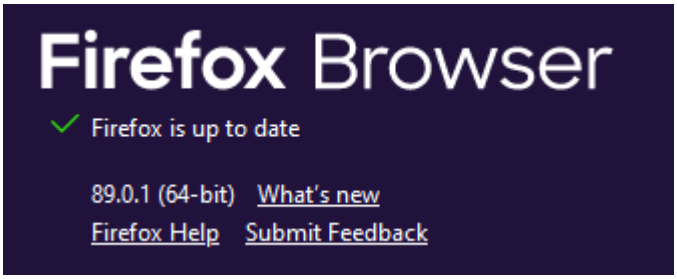
💡 Observations:

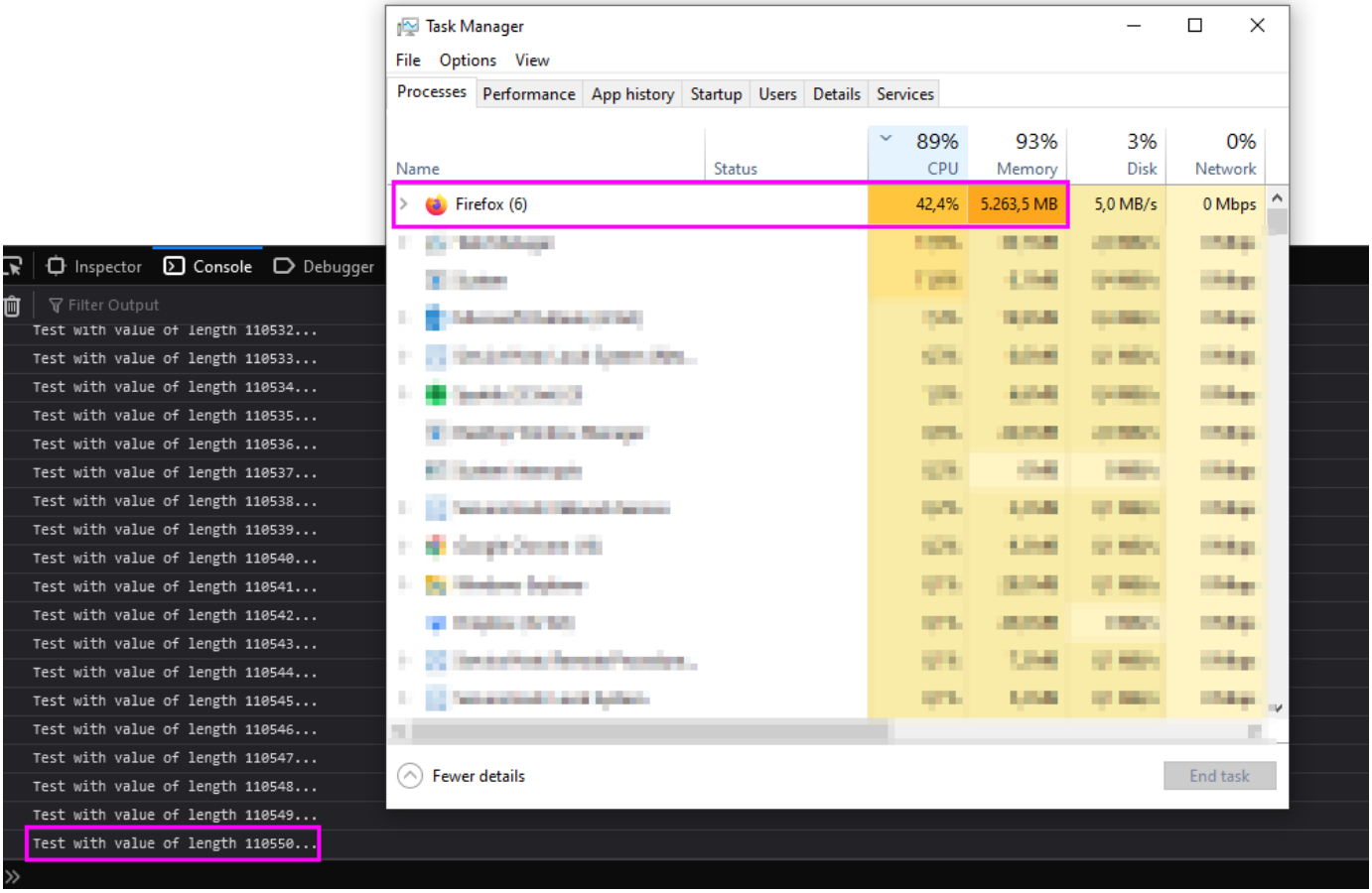
- For the test cases above "100", the browser raise an operation specific error. This error is consitent because the amount of data is big for a asymmetric encryption operation. This behavior prevent the browser to hang or crash due to the launching of a huge processing.
- Asymmetric encryption is targeted for a small data like the protection of a symmetric key during the exchange for later symmetric encryption operation.
- If the following code is used, on Firefox (last version on June 2021) it continue to encrypt the data provided whatever the size and made the browser unstable:

```
function
performIdentificationOfContentLengthLimitForEncryptionWithRSAOAEP(cryptoKe
```

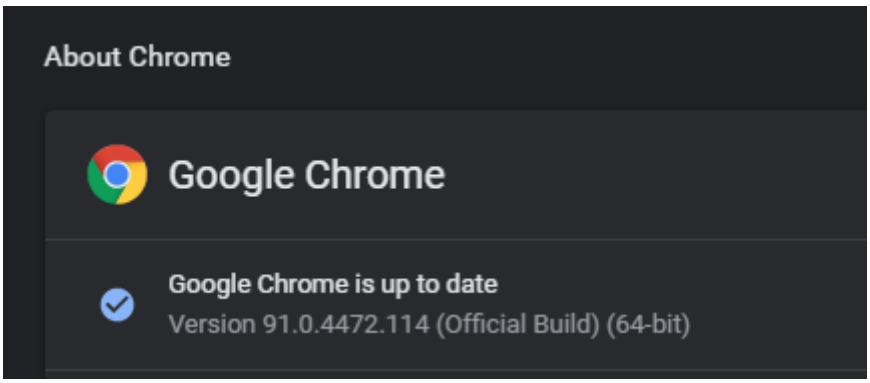
```
yPairPublicKey) {
    let labelData = new Int32Array(32);
    CRYPTO_OBJ.getRandomValues(labelData);
    let rsaOaepParams = {
        name: "RSA-OAEP",
        label: labelData
    };
    let dataEncoded = null;
    for (let i = 100; i < 1000000; i++) {
        console.debug("Test with value of length " + i + "...");
        dataEncoded = TEXT_ENCODER.encode("T".repeat(i));
        CRYPTO_OBJ.subtle.encrypt(rsaOaepParams, cryptoKeyPairPublicKey,
dataEncoded).catch(err => {
            console.warn(err);
            return i;
        });
    }
}
```

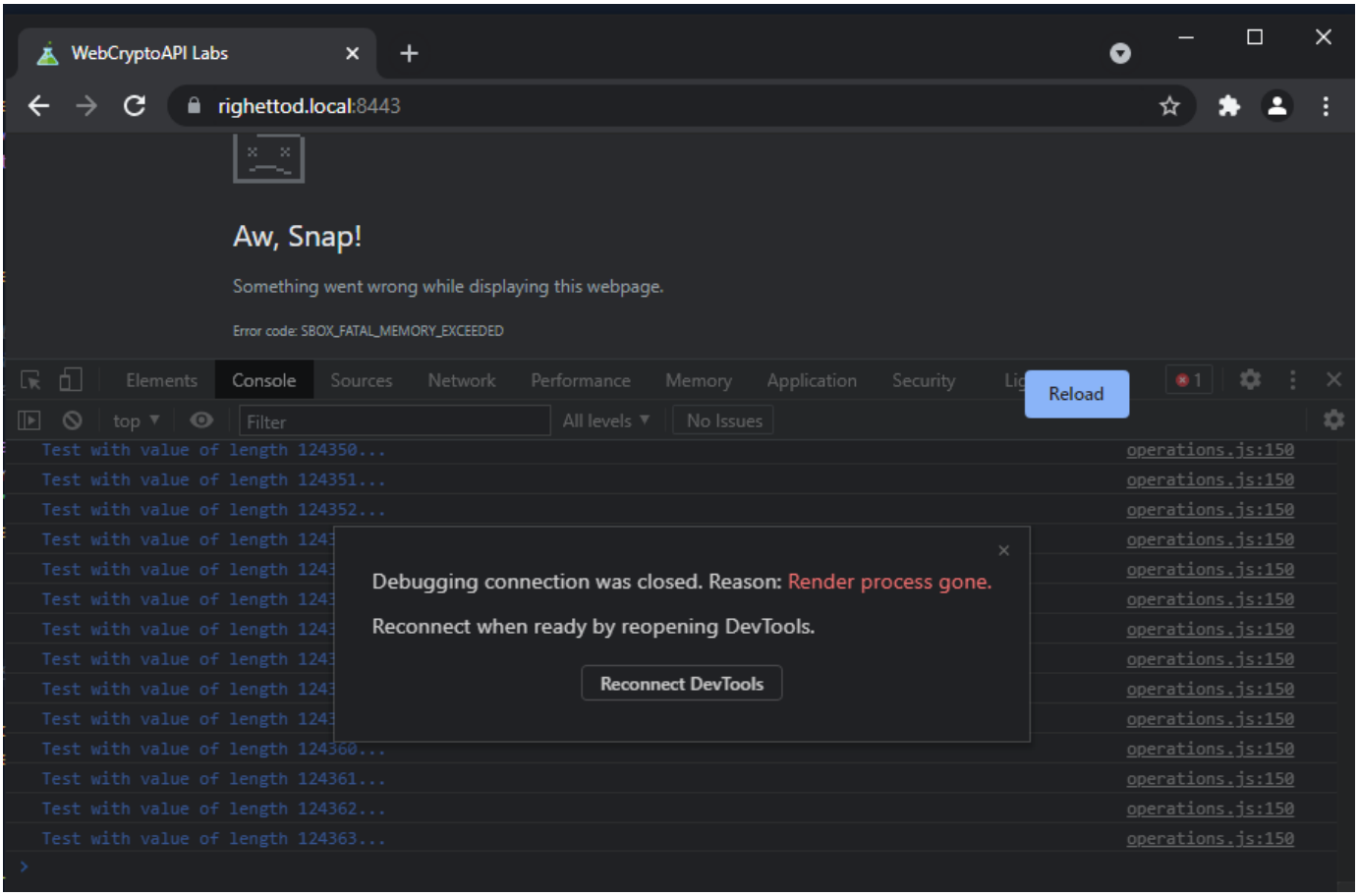
Run on Firefox my laptop, catch clause was not invoked and browser became unstable (kill of the tab required):



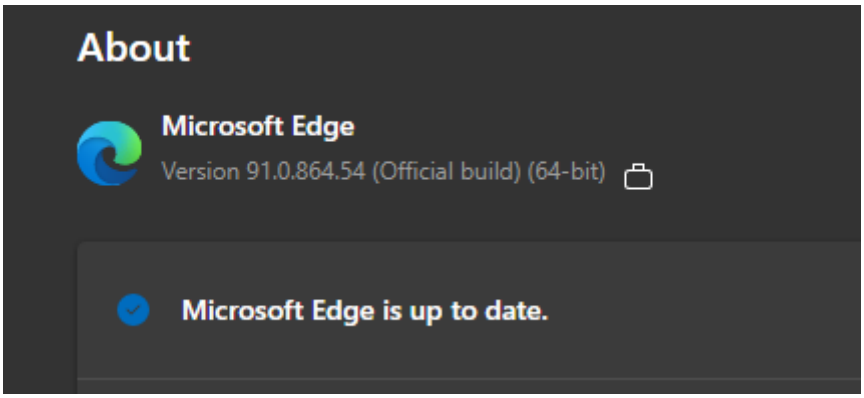


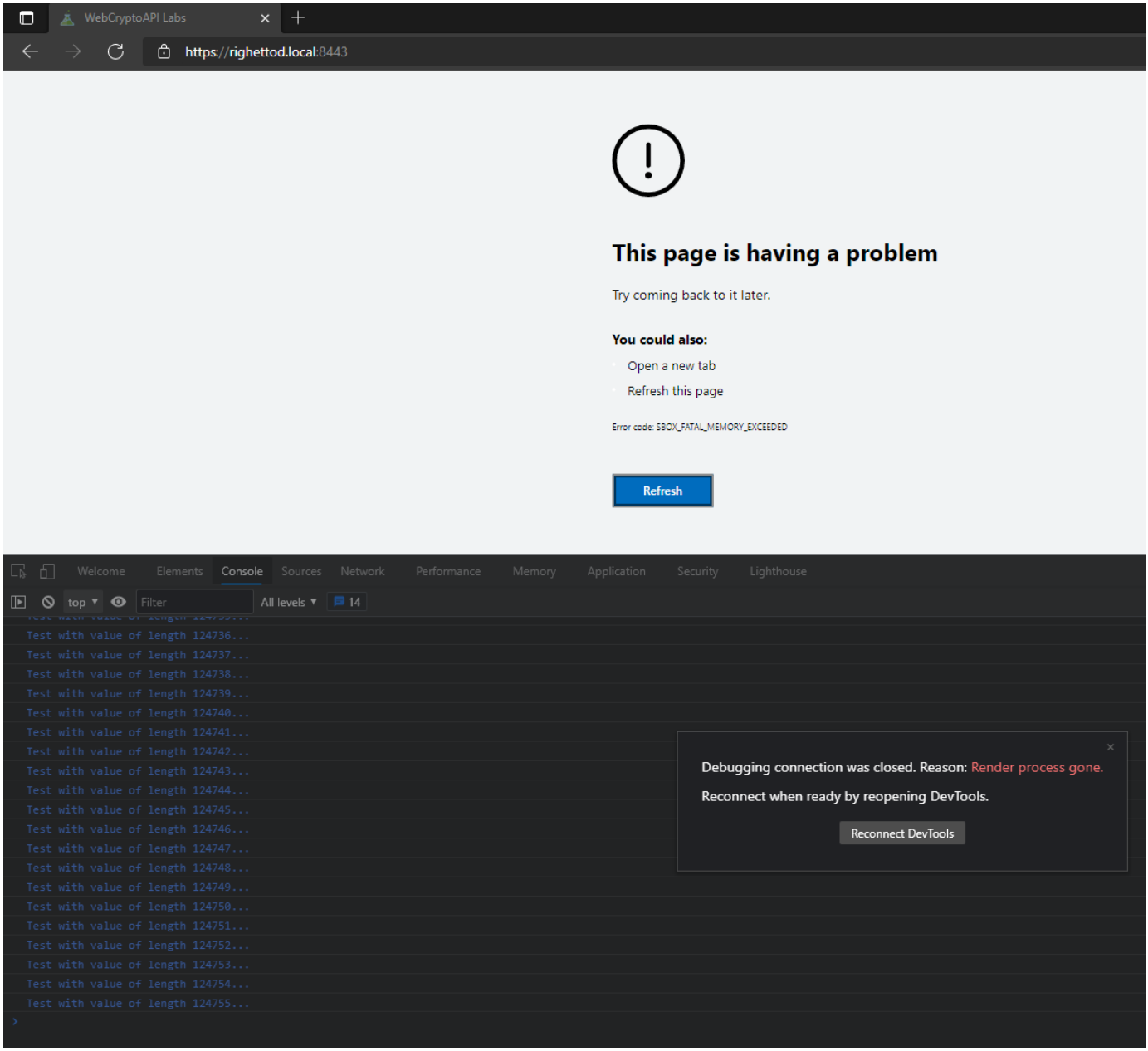
Same test on Chrome (last version on June 2021), browser stay stable and raise the following error without causing any instability:





Same behavior than Chrome on Edge (last version on June 2021):





Signature

> Source: https://developer.mozilla.org/en-US/docs/Web/API/EcKeyGenParams

> Source: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/generateKey

> Source: https://developer.mozilla.org/en-US/docs/Web/API/EcdsaParams

> Source: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/sign

> Source: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/verify

Can be used in the context of the following protocols:

- HTTP: ✕
- HTTPS: ✔

Code snippet:

ℹ This include the **key pair** generation for signature operation.

```
async function performAsymmetricKeyGenerationForSignVerifyUsageWithECDSA()
{
    //Generate a ECDSA key pair with the P-521 elliptic curve
    //See https://developer.mozilla.org/en-US/docs/Web/API/EcKeyGenParams
    let ecKeyGenParams = {
        name: "ECDSA",
        namedCurve: "P-521"
    };
    let keyUsages = ["sign", "verify"];
    //See https://developer.mozilla.org/en-
US/docs/Web/API/SubtleCrypto/generateKey
    let cryptoKeyPair = await
CRYPTO_OBJ.subtle.generateKey(ecKeyGenParams, true, keyUsages);
    return cryptoKeyPair;
}

async function performSignVerifyWithECDSA(sourceData,
cryptoKeyPairPublicKey, cryptoKeyPairPrivateKey) {
    //Generate a ECDSA with SHA-512 signature and verify it
    //See https://developer.mozilla.org/en-US/docs/Web/API/EcdsaParams
    let ecdsaParams = {
        name: "ECDSA",
        hash: "SHA-512"
    };
    let dataEncoded = TEXT_ENCODER.encode(sourceData);
    //See https://developer.mozilla.org/en-
US/docs/Web/API/SubtleCrypto/sign
    let signature = await CRYPTO_OBJ.subtle.sign(ecdsaParams,
cryptoKeyPairPrivateKey, dataEncoded);
    //See https://developer.mozilla.org/en-
US/docs/Web/API/SubtleCrypto/verify
    let isValid = await CRYPTO_OBJ.subtle.verify(ecdsaParams,
cryptoKeyPairPublicKey, signature, dataEncoded);
    let result = {
        signature: toHex(signature),
        cycleSucceed: isValid
    }
    return result;
}
```

Key derivation

> Source: https://developer.mozilla.org/en-US/docs/Web/API/Pbkdf2Params

> Source: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/importKey

> Source: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/deriveKey

> Source: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/wrapKey#aes-kw

> Source: https://developer.mozilla.org/en-US/docs/Web/API/CryptoKey

> Source: https://cryptosense.com/blog/parameter-choice-for-pbkdf2

⚠ Security note about usage of the different derivation algorithms based on the type of input data provided.

Can be used in the context of the following protocols:

- HTTP: ✕
- HTTPS: ✔

Code snippet:

ℹ Here a common case was chosen, i.e. derivate a key from a provided password by the user.

```
async function performKeyDerivationFromPassword(iterationCount,
basePassword) {
    let saltData = new Int32Array(16);
    CRYPTO_OBJ.getRandomValues(saltData);
    //PBKDF2 algorithm will be used
    //See https://developer.mozilla.org/en-
US/docs/Web/API/SubtleCrypto/deriveKey#supported_algorithms
    //See https://developer.mozilla.org/en-US/docs/Web/API/Pbkdf2Params
    //See https://cryptosense.com/blog/parameter-choice-for-pbkdf2
    let pbkdf2Params = {
        name: "PBKDF2",
        hash: "SHA-512",
        salt: saltData,
        iterations: iterationCount
    }
    //Use the importKey() function to import the initial password as
CryptoKey
    //Flag it as not exportable and for Key/Bits derivation usages in
order it can only be used
    //to obtain a derivated CryptoKey
    let dataEncoded = TEXT_ENCODER.encode(basePassword);
    let baseCryptoKeyUsages = ["deriveBits", "deriveKey"];
    //See https://developer.mozilla.org/en-
US/docs/Web/API/SubtleCrypto/importKey
    let baseCryptoKey = await CRYPTO_OBJ.subtle.importKey("raw",
```

```
dataEncoded, "PBKDF2", false, baseCryptoKeyUsages);
    //Obtain a derivated key
    //See https://developer.mozilla.org/en-
US/docs/Web/API/SubtleCrypto/wrapKey#aes-kw
    let aesKeyGenParams = {
        name: "AES-KW",
        length: 256
    }
    let derivatedCryptoKeyUsages = ["wrapKey", "unwrapKey"];
    //See https://developer.mozilla.org/en-
US/docs/Web/API/SubtleCrypto/deriveKey
    let derivatedCryptoKey = await
CRYPTO_OBJ.subtle.deriveKey(pbkdf2Params, baseCryptoKey, aesKeyGenParams,
true, derivatedCryptoKeyUsages);
    //See https://developer.mozilla.org/en-US/docs/Web/API/CryptoKey
    return derivatedCryptoKey;
}
```

Key secure import/export

> Source: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/importKey

> Source: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/exportKey

> Source: https://developer.mozilla.org/en-US/docs/Web/API/CryptoKey

The `CryptoKey` / `CryptoKeyPair` object do not expose the key content to the Javascript. Only not sensitive properties of the key are exposed like type, algorithm, usages, etc.

The functions `importKey()` / `exportKey()` allow to import/export key from/to an unprotected format. By "unprotected" it means that the content of the key is directly accessible in the initial (import case) / target (export case) format. In this way the content is **unprotected** during the transfert.

The functions `unwrapKey()` / `wrapKey()` allow to perform the same objective than functions `importKey()` / `exportKey()`. However, the content of the key is encrypted with a key (`CryptoKey` / `CryptoKeyPair` object) provided during the call. In this way the content is **protected** during the transfert.

⚠ It is important to no that, the be exported, a `CryptoKey` / `CryptoKeyPair` object must have the attribute **extractable** been defined to **true** during the generation/import of the key. Otherwise export will be refused.

⚠ Usage of the unprotected or protected way are both suitable depending on the usage but it's up to the project team to choose the one fitting the use case.

⚠ Same remark for the **usages** attribute. It must be defined according to target situation of usage of the key. Never define it to all possibilities:

`CryptoKey.usages`

An `Array` of strings, indicating what can be done with the key. Possible values for array elements are:

- `"encrypt"`: The key may be used to `encrypt` messages.
- `"decrypt"`: The key may be used to `decrypt` messages.
- `"sign"`: The key may be used to `sign` messages.
- `"verify"`: The key may be used to `verify` signatures.
- `"deriveKey"`: The key may be used in `deriving a new key`.
- `"deriveBits"`: The key may be used in `deriving bits`.
- `"wrapKey"`: The key may be used to `wrap a key`.
- `"unwrapKey"`: The key may be used to `unwrap a key`.

Note: An exmple of usage of the unprotected import was shown in the function `performKeyDerivationFromPassword()`

Can be used in the context of the following protocols:

- HTTP: ✗
- HTTPS: ✓

Sample 1: Export a key using the unprotected way

Code snippet:

```
async function performKeyExportUsingUnprotectedWay() {
    let cryptoKey = await
performSymmetricKeyGenerationForEncryptionDecryptionUsageWithAESGCM();
    //See https://developer.mozilla.org/en-
US/docs/Web/API/SubtleCrypto/exportKey
    let exportedKeyContent = await CRYPTO_OBJ.subtle.exportKey("raw",
cryptoKey);
    return toHex(exportedKeyContent);
}
```

Sample 2: Export a key using the protected way

Code snippet:

```
async function performKeyExportUsingProtectedWay(basePassword) {
    let cryptoKeyToExport = await
performSymmetricKeyGenerationForEncryptionDecryptionUsageWithAESGCM();
    let cryptoKeyForProtection = await
performKeyDerivationFromPassword(10000, basePassword);
    //See https://developer.mozilla.org/en-
```

```
US/docs/Web/API/SubtleCrypto/wrapKey#supported_algorithms
    let wrapAlgo = {
        name: "AES-KW"
    }
    //See https://developer.mozilla.org/en-
US/docs/Web/API/SubtleCrypto/wrapKey
    let exportedKeyContent = await CRYPTO_OBJ.subtle.wrapKey("raw",
cryptoKeyToExport, cryptoKeyForProtection, wrapAlgo);
    return toHex(exportedKeyContent);
}
```

Sample 3: Export a key marked as non extractable

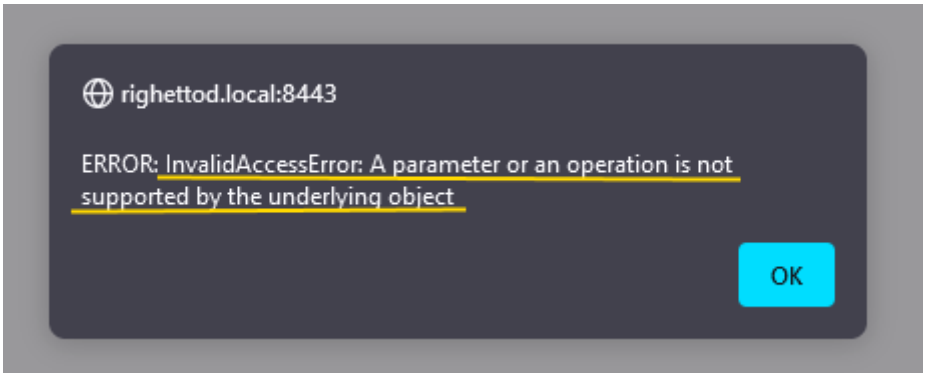
- `extractable` is a `Boolean` indicating whether it will be possible to export the key using `SubtleCrypto.exportKey()` or `SubtleCrypto.wrapKey()`.

Code snippet:

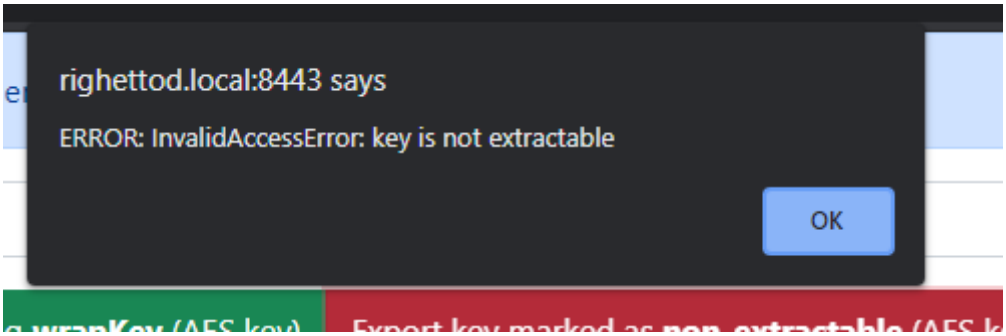
```
async function performKeyExportMarkedAsNonExtractable() {
    //Generate a non extractable key
    let extractable = false;
    let aesKeyGenParams = {
        name: "AES-GCM",
        length: 256
    };
    let keyUsages = ["encrypt", "decrypt"];
    //See https://developer.mozilla.org/en-
US/docs/Web/API/SubtleCrypto/generateKey#parameters
    let cryptoKey = await CRYPTO_OBJ.subtle.generateKey(aesKeyGenParams,
extractable, keyUsages);
    //Try to export it
    let exportedKeyContent = await CRYPTO_OBJ.subtle.exportKey("raw",
cryptoKey);
    return toHex(exportedKeyContent);
}
```

An error is correctly raised when the export is tried.

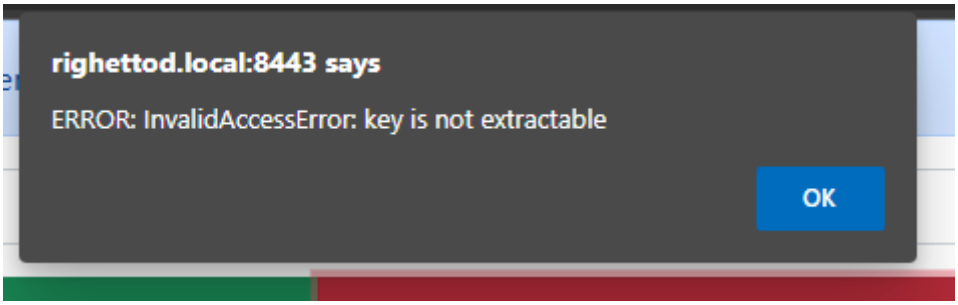
On Firefox:



On Chrome:



On Edge:



Extra

Try to access to the content of a CryptoKey directly via the browser files

> Source: https://stackoverflow.com/a/49479890/451455

> Source: https://security.stackexchange.com/a/244506

> Source: https://w3c.github.io/webcrypto/#concepts-key-storage

> Source: https://caniuse.com/indexeddb

> Source: https://github.com/w3c/IndexedDB/issues/191

> Source: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Using_IndexedDB#libraries

From this extract from the RFC:

```
In practice, it is expected that most authors will make use of the Indexed
Database API
[INDEXEDDB], which allows associative storage of key/value pairs, where
the key is some string
identifier meaningful to the application, and the value is a CryptoKey
object.

This allows the storage and retrieval of key material, without ever
exposing that key material to
the application or the JavaScript environment. Additionally, this allows
authors the full
flexibility to store any additional metadata with the CryptoKey itself.
```

The IndexedDB wrapper Dexie (proposed by the MDN page about IndexedDB) was used to simplify the storage code.

⚠ It was observed that, as the key content is not accessible to JS code, then, the key must be exported prior to be stored.

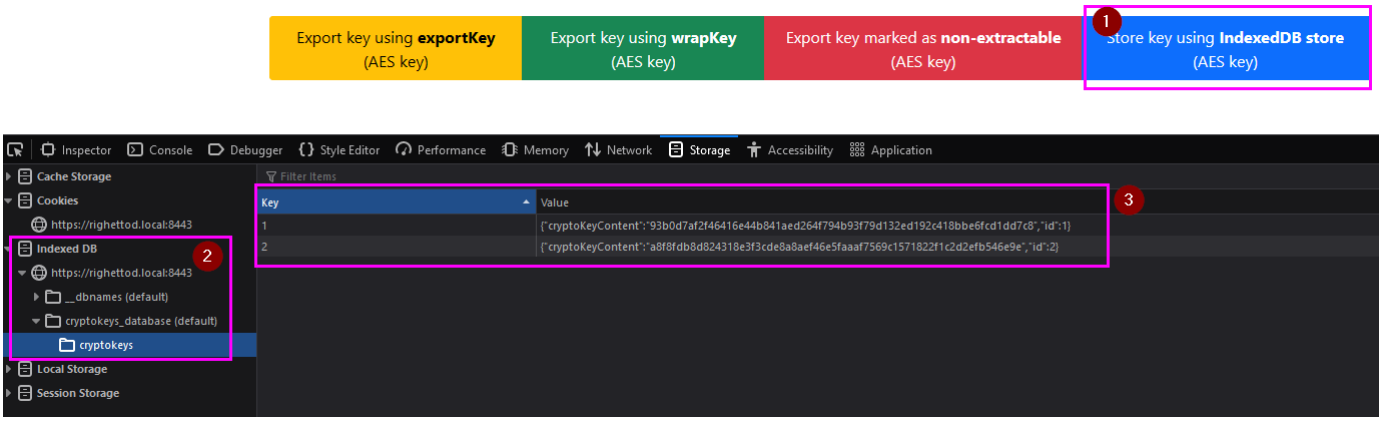
In the demo the insecure way via `export()` was used in order to see if the **IndexDB** was offering a protection layer

Code snippet for the export and storage of the key:

```
//See https://developer.mozilla.org/en-
US/docs/Web/API/IndexedDB_API/Using_IndexedDB#libraries
//See https://github.com/dfahlander/Dexie.js
let INDEXEDDB_KEYS = new Dexie("cryptokeys_database");

function renderInit() {
    //...
    INDEXEDDB_KEYS.version(1).stores({
        cryptokeys: "++id,cryptoKeyContent"
    });
}

function storeKeyInDB(cryptoKey) {
    INDEXEDDB_KEYS.cryptokeys.put({
        cryptoKeyContent: cryptoKey
    }).then(function () {
        alert("Key content inserted into the IndexedDB instance.");
    }).catch(function (error) {
        alert("ERROR: " + error)
    });
}
```

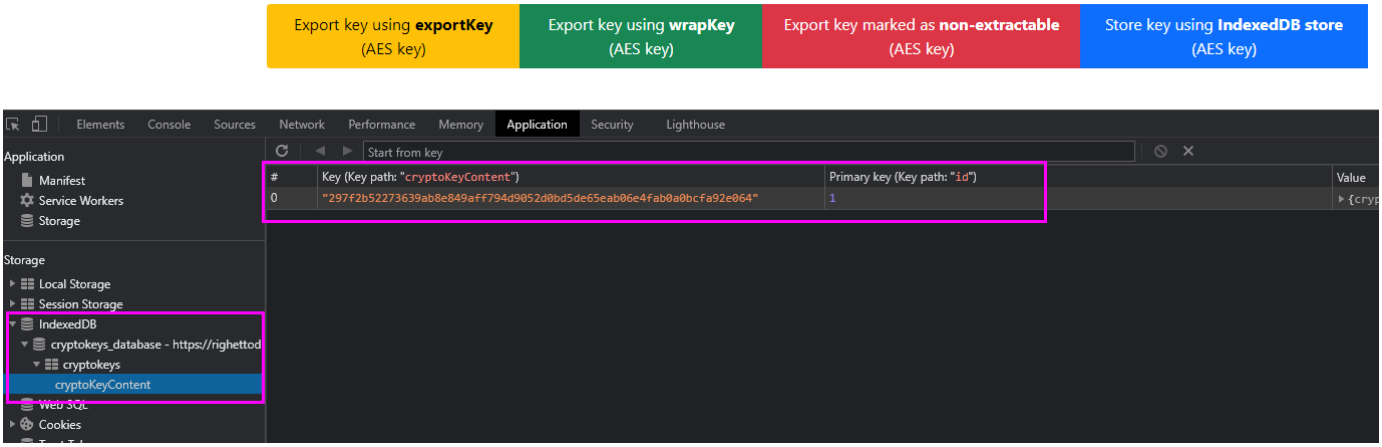
```
function exportKey(mode) {
    switch (mode) {
        //...
        case "STORE-KEY": {
            performKeyExportUsingUnprotectedWay().then(key => {
                storeKeyInDB(key);
            }).catch(err => alert("ERROR: " + err));
            break;
        }
        default:
            alert("Unsupported mode!");
    }
}
```

Execution of the code on Firefox

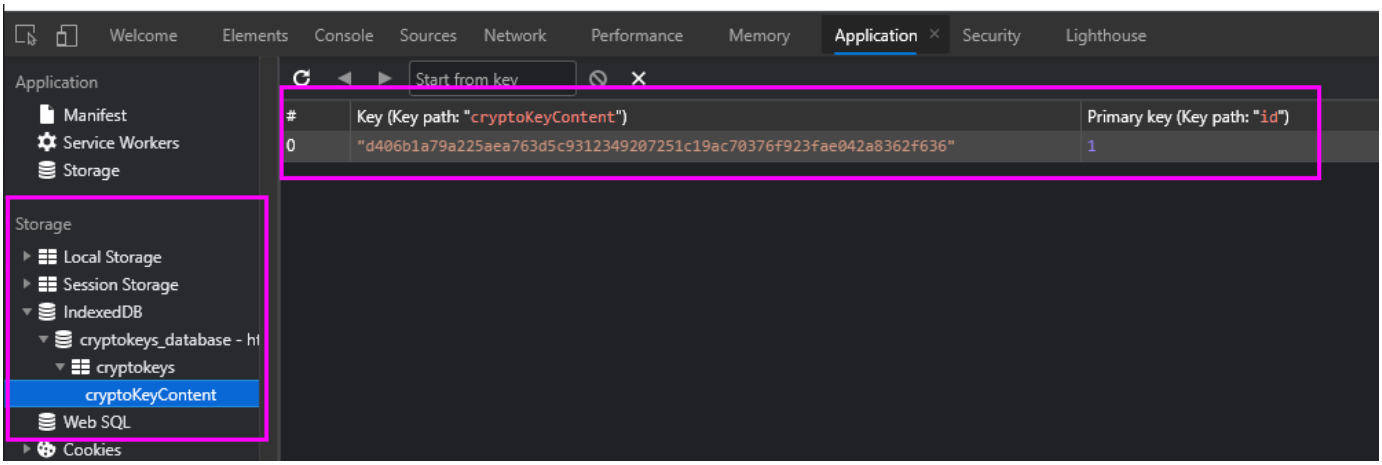


:

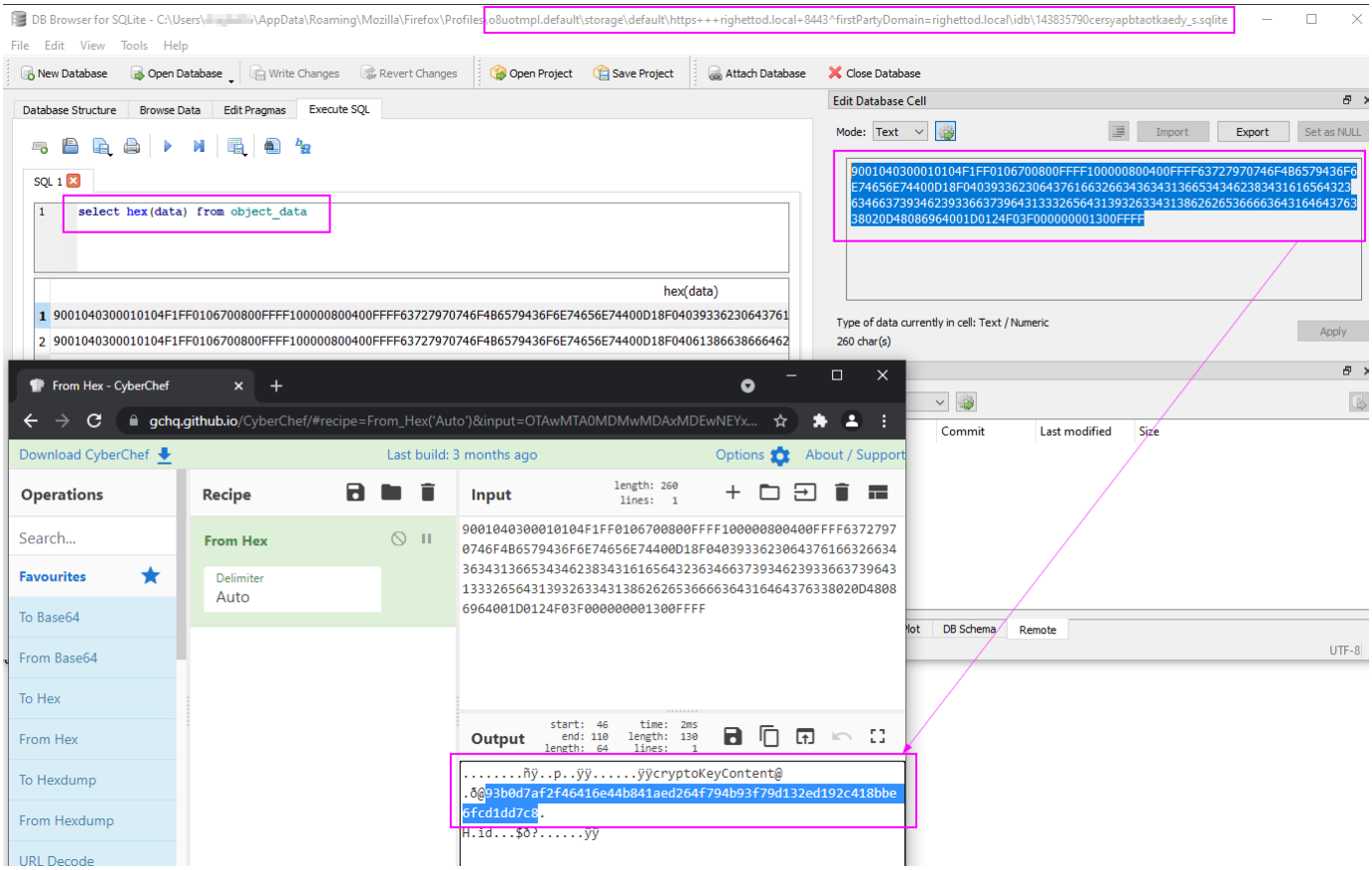
Execution of the code on Chrome:

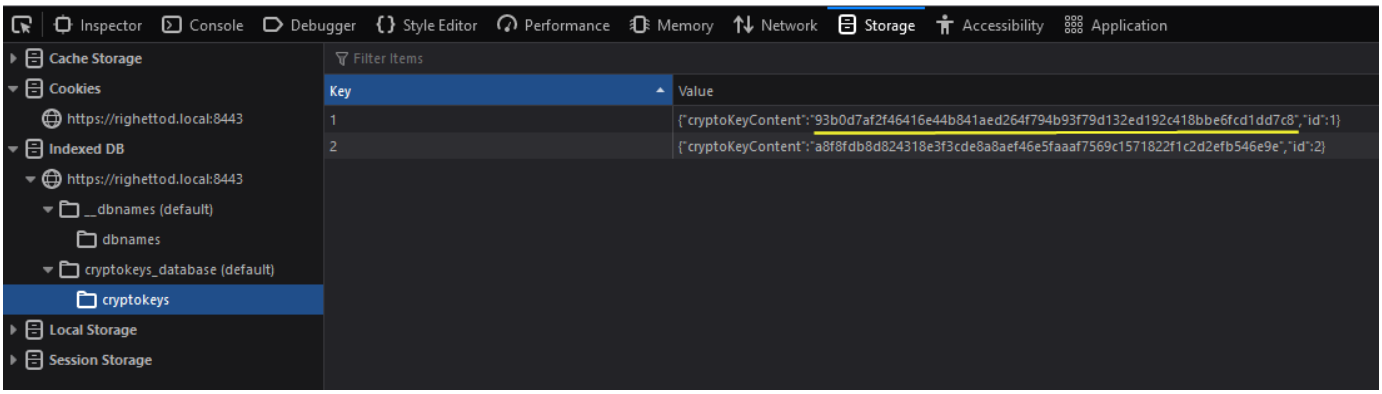


Execution of the code on Edge:



The content stored in not protected when at rest, example for Firefox:

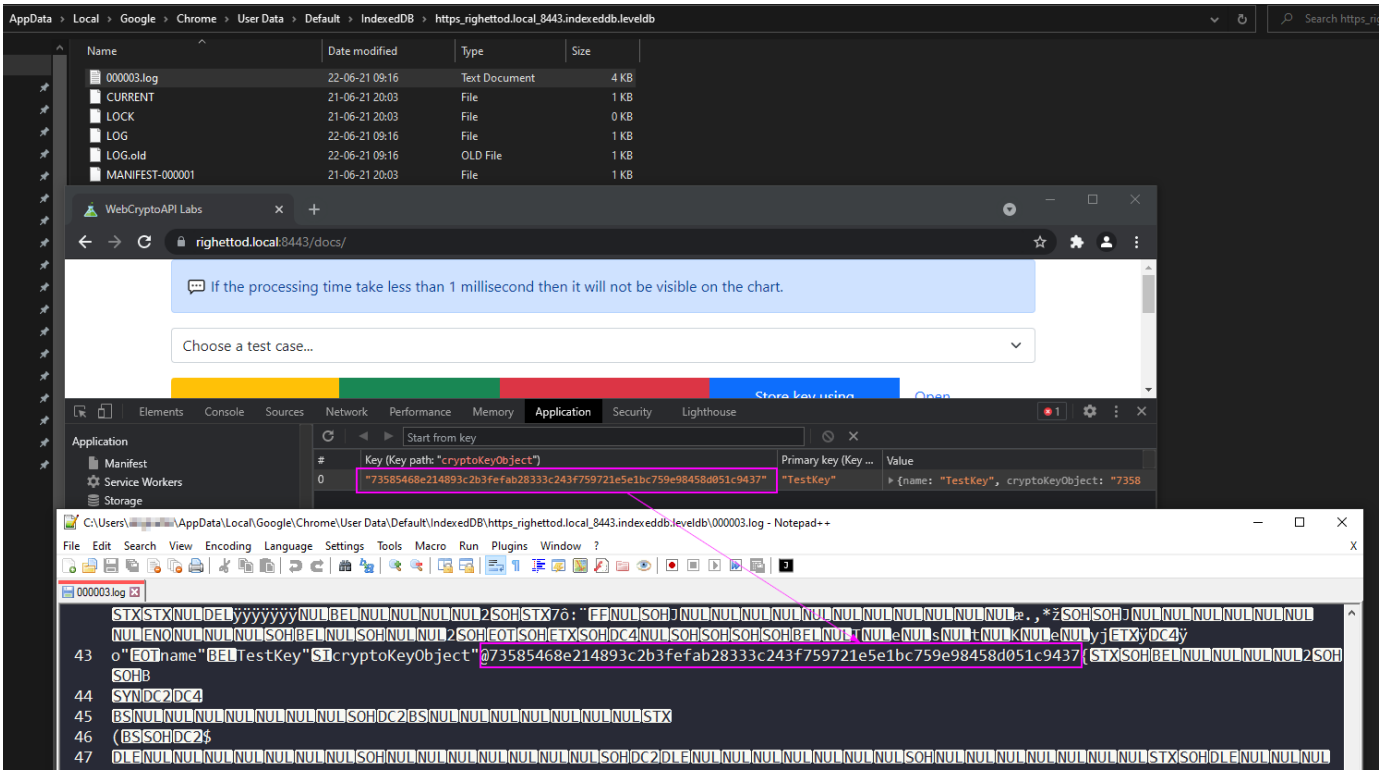




Chrome use a DB using the **LevelDB** format instead of SQLite:

- https://en.wikipedia.org/wiki/LevelDB
- https://www.cclsolutionsgroup.com/post/hang-on-thats-not-sqlite-chrome-electron-and-leveldb

But the content is not protected too when at rest:



Based on this part of the extract from the RFC:

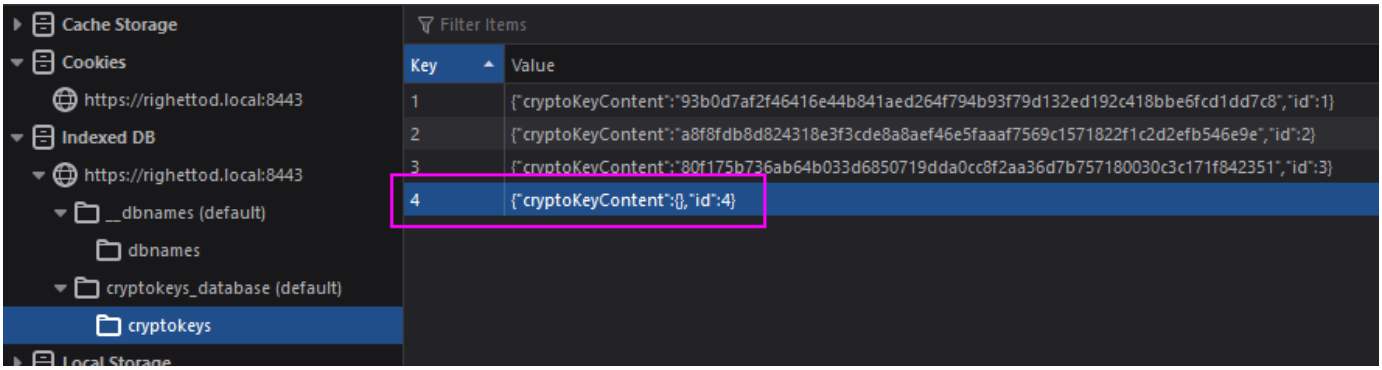
```
This allows the storage and retrieval of key material, without ever
exposing that key material to
the application or the JavaScript environment.
Additionally, this allows authors the full flexibility to store any
additional metadata with the
CryptoKey itself.
```

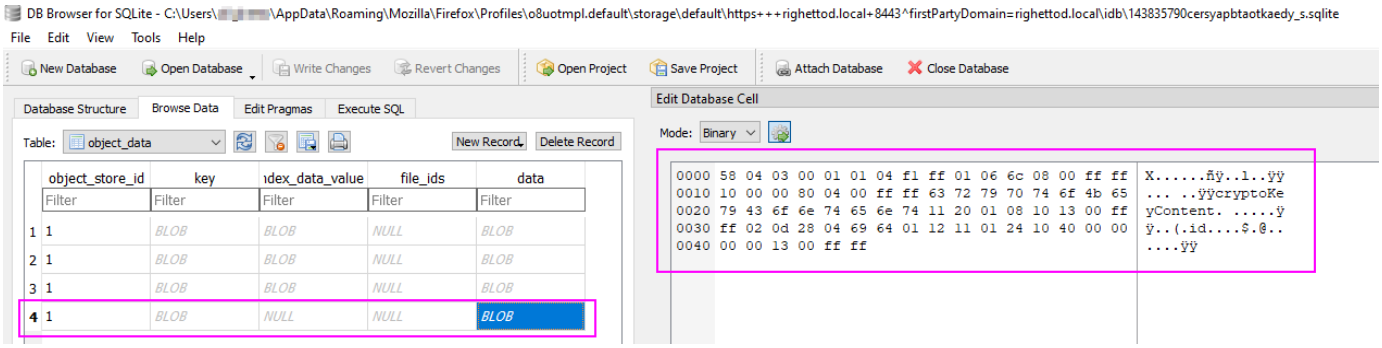
A tentative was performed to store the CryptoKey object directly into the IndexedDB instance to follow the RFC:

```
function storeKeyInDB(cryptoKey) {
    INDEXEDDB_KEYS.cryptokeys.put({
        cryptoKeyContent: cryptoKey
    }).then(function () {
        alert("Key content inserted into the IndexedDB instance.");
    }).catch(function (error) {
        alert("ERROR: " + error)
    });
}

function exportKey(mode) {
    switch (mode) {
        //...
```

```
        case "STORE-CRYPTOKEYOBJECT": {

performSymmetricKeyGenerationForEncryptionDecryptionUsageWithAESGCM().then
(keyObject => {
                storeKeyInDB(keyObject);
            }).catch(err => alert("ERROR: " + err));
            break;
        }
        default:
            alert("Unsupported mode!");
    }
```

The object stored was empty (check performed on Firefox):





But as the IndexedDB content is stored unprotected when at rest, storing the CryptoKey object or an insecure export of the key do not bring any added value from a security point view.