**Image for the blog background (choose one that fit well):**

* <https://pixabay.com/photos/safety-encryption-ssl-world-2890768/>
* <https://pixabay.com/photos/code-html-computer-internet-it-3477973/>
* <https://pixabay.com/illustrations/hacker-hacking-cyber-security-hack-1944688/>

**Title:**

Overview of what Web Cryptography API is.

**Abstract:**

The Web Cryptography API is now supported by modern browsers, let’s discover which feature it brings as well as pitfalls through an exploration of this new JavaScript API.

**GitHub repository associated that will be moved to public when blog post will be released:**

* <https://github.com/ExcelliumSA/WebCryptographyAPI-Study>

**SEO rules indicated by Mathilde:**

* Paragraphs with fewer than 300 words.
* Keyword used as much as possible:
  + **Cryptography, JavaScript**
* Presence of sections.

**Author(s):**

* Dominique Righetto

# Context

Anyone developing a web application with a front end meet, one day, the need to perform cryptographic operations like hashing, encryption, signatures, etc. from the front-end side (JavaScript code). The habits lead to import and use popular external library like **crypto-js** [1] in order to be portable across all targeted browsers:

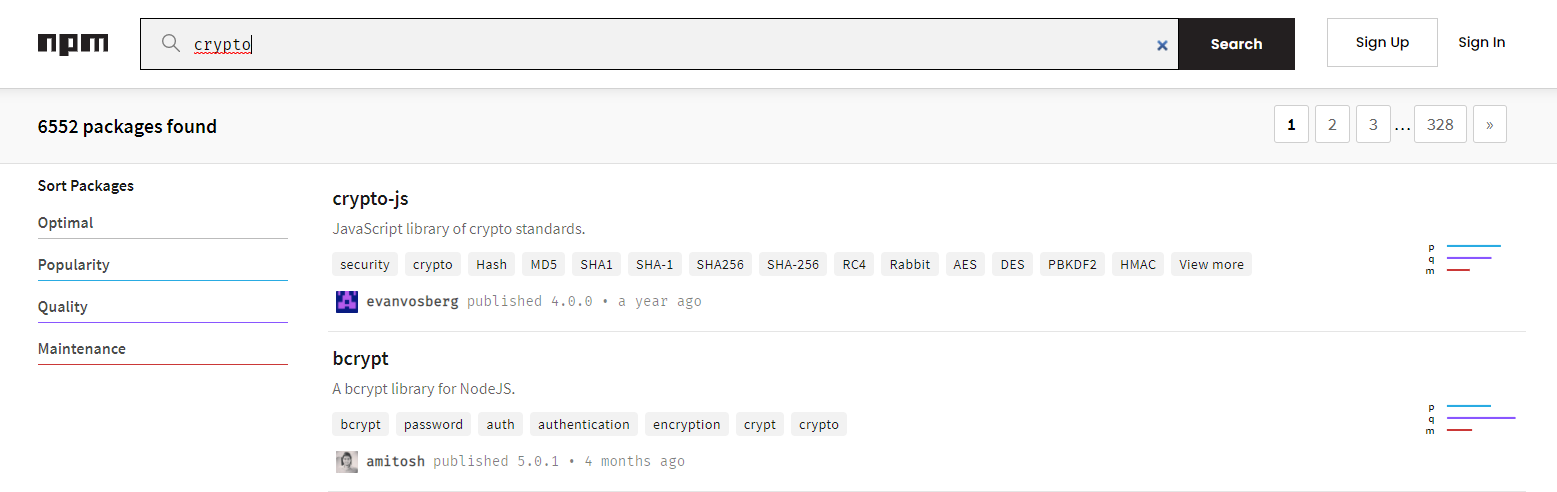


Figure 1: File Figure00a.png

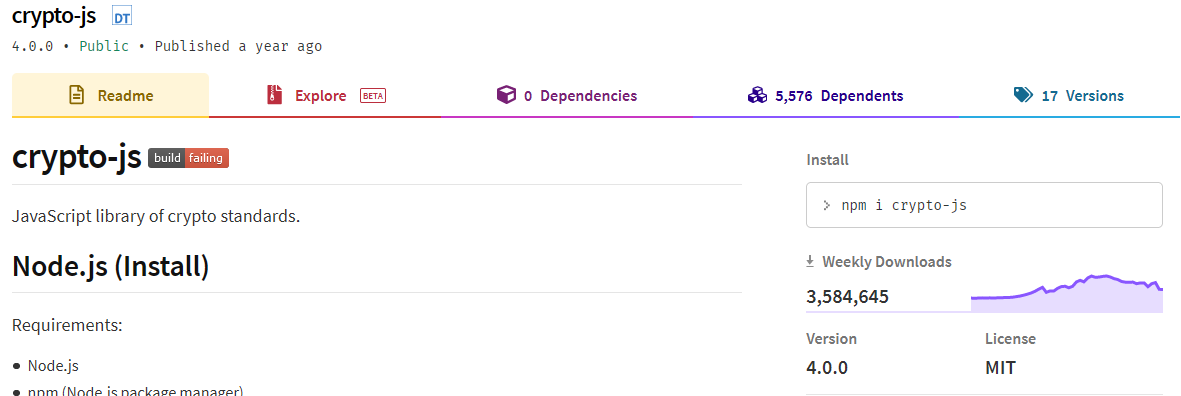


Figure 1: File Figure00b.png

The library work well since years, so, why should I leave it?

# No troll allowed here

The goal of this blog post is not to say that Web Cryptography API is the best API or that you must replace your existing JS cryptographic API by Web Cryptography API.

The goal is just to present you what Web Cryptography API is and what if offer.

# What is the Web Cryptography API?

From the RFC [2], it is a:

“*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*.”

To summarize, it brings native capabilities to browsers for perform different kind of cryptographic operations.

This API allow performing the following kind of operation:

* Generate cryptographically random values.
* Encrypt and decrypt content using symmetric or asymmetric encryption algorithms.
* Sign and verify the signature using symmetric or asymmetric algorithms.
* Derivate key and bits from base keys or source of bits.
* Generate hash of a content.

In addition, it allows the following operations regarding the key management:

* Generate symmetric or asymmetric key for different kinds of usage.
* Export / import key using secure / insecure methods.

The API do not handle the following aspects [3] (*some points in the list below were directly taken from the RFC*):

* Key (secure) persistence in browser storage systems.
* The provisioning of keys in particular types of key storage, such as secure elements or smart cards.
* Communication and discovery of cryptographic modules.
* The RFC places no normative requirements on how implementations handle key material once all references to it go away.

# Why Web Cryptography API is not the Holy Grail?

The API provide **low-level cryptographic** operations, so, it requires to know “*what you are doing*” when using the different feature. Indeed, the API do not protect you against incorrect usage [5] of the cryptographic algorithms supported.

In addition, like **crypto-js**, the API support deprecated algorithms like MD5, SHA1, ECB mode of operation of AES, CBC mode of operation of AES, etc., thus, no warning will be raised if you decide to use one of them whatever you usage is legit or not.

In the same way, the Web Cryptography API will not warn you or raise an error if you create/use a key with larger key usages that the real needed for the cryptographic operations performed.

To finish, like **crypto-js,** the Web Cryptography APIlet you alone if you need to securely store a key or communicate with cryptographic modules from JavaScript code on the browser.

# So, why should I consider the Web Cryptography API?

Cryptography is a very complex and very error prone domain. Even if you stick to the standard, you can create vulnerable code. Beside incorrect usage of a cryptographic library, one of the pitfalls is the quality of the implementation of the supported algorithms. Indeed, this type of issue is “transparent” for most of the user of a cryptographic library (including for the creator of this blog post) in the way in which it requires specific skills to identify that, the implementation of a cryptographic algorithm is unsafe.

For example, how you ensure, in an accurate way, that the cryptography library that you are using:

* Implement correctly the algorithm that you want to advantage.
* Provide battle tested implementation.
* Patch, in an effective way, any security weakness identified.
* Maintain, over the time, the different implementations of the algorithms supported.
* It is implemented by people having the required skills/experience.

Here come the Web Cryptography API, by using it, you delegate the validations above to the provider of the browser, i.e. Google (Chrome), Mozilla (Firefox), Microsoft (Edge), Apple (Safari) for the main browsers. The section of the RFC named “*Underlying Cryptographic Implementation*” [6] define the following statement:

“*This specification assumes, but does not require, that conforming user agents do not and will not be* ***directly implementing cryptographic operations within the user agent itself****. Historically, many user agents have deferred cryptographic operations, such as those used within TLS,* ***to existing APIs that are available as part of the underlying operating system or to third-party modules that are managed independently of the user agent****.”*

The RFC recommended to browser provider to not implement cryptographic functions themselves but instead leverage industry-recognized modules. It is true that here we are in “*delegation****ception***” because you delegate to the browser that delegate itself to another module the handling of cryptographic operations.

# The end of the white rabbit syndrome [7]

From a patching point of view, if a critical issue is identified in the implementation of an algorithm, you will not need anymore to identify, in urgency, which of your web app is using the vulnerable version of the cryptographic library and then perform X update and redeploy. The patching will be ensured via the auto-update feature of the each browser. Most of them have now such a feature to keep them up-to-date.

To be honest a second, let us agree that entity like Google/Mozilla/Microsoft/Apple are better at cryptography/patch management that your company (mine included). Therefore, even if we are in two-stage delegation as described in the previous section, security level will be better than if you need to patch all your application yourselves.

# It is not always required to make a choice…

Popular cryptographic library, like **crypto-js**, have already started using Web Cryptography API under the hood when it is available:

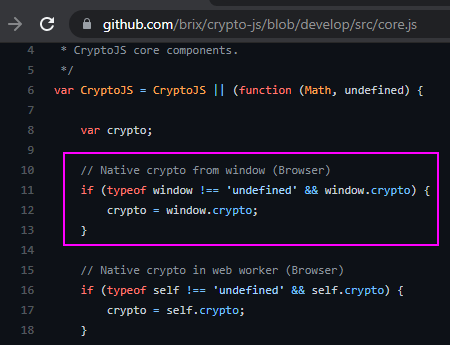


Figure 2: File Figure02.png

Therefore, not need to leave your favorite cryptographic library if it support Web Cryptography API when it is available ☺

TODO:FIND\_OTHER\_LIB\_MIGRATING\_WCA

# Browser support level

On June 2021, it was the following [4]



Figure 3: File Figure01.png

# Web Cryptography API capabilities

This section provides an overview of the capabilities bring by the Web Cryptography API at the time of writing of this post (June 2021).

The source was the Mozilla documentation [8] combined with the RFC [9] as well as validation using the labs presented (see further [10]).

**Algorithms and operations supported:**

|  |  |  |
| --- | --- | --- |
| Cryptographic operation | Supported algorithm | Protocol \* |
| Generate cryptographically random values | - | HTTP  HTTPS |
| Generation of a symmetric key for encryption/decryption | AES-CBC  AES-CTR  AES-GCM | HTTPS |
| Generation of a secret for symmetric signature/validation | HMAC |
| Generation of an asymmetric key pair for encryption/decryption | RSA-OAEP |
| Generation of an asymmetric key pair for signature/validation | RSASSA-PKCS1-v1\_5  RSA-PSS  ECDSA |
| Symmetric encryption/decryption | AES-CBC  AES-CTR  AES-GCM |
| Asymmetric encryption/decryption | RSA-OAEP |
| Symmetric signature and validation | HMAC |
| Asymmetric signature and validation | RSASSA-PKCS1-v1\_5  RSA-PSS  ECDSA |
| Hashing | SHA1  SHA-256  SHA-384  SHA-512 |
| Key derivation / bit derivation | ECDH  HKDF  PBKDF2 |
| Key wrapping - From Mozilla documentation:  “*Exports the key in an external, portable format, and then encrypts the exported key.* | AES-CBC  AES-CTR  AES-GCM  AES-KW  RSA-OAEP |

\* If you are using an insecure protocol for operation requiring HTTPS (named **secure context**), the following error will be raised (the **subtle** attribute member of the **crypto** object is not defined):

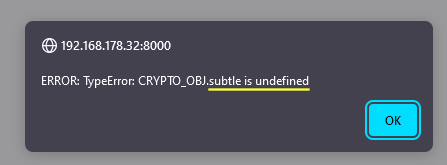


Figure : File Figure03.png

**Key export format:**

* Raw: Array buffer containing the raw bytes for the key.
* PKCS #8. [11]
* SubjectPublicKeyInfo. [12]
* JSON Web Key. [13]

# Avoid hidden trouble with the API

Below is a set of key points to avoid traps of the Web Cryptography API:

1. Do not use deprecated algorithms / Ensure to correctly use an algorithm [14][15]:
   1. If you have any doubt about an algorithm or its correct usage then ask to a friend or colleague with skills in the cryptographic field (the author always does that, it is a wink to you crypto mum).
   2. Doubting and asking questions is a healthy attitude when it comes to choose or correctly use a cryptographic algorithm☺.
2. Do not use the function **getRandomValues()** to generate a key, use the dedicated **generateKey()** function.
3. During the generation of a key, only set key usage consistent for the usage of the key via the parameter named **keyUsages** of the **generateKey()** function [16].
4. If the key you generated is not intended to be exported, for example, because it will be only used for a temporary operation then mark it as non-extractable via the parameter named **extractable** of the **generateKey()** function. In case of doubt set this parameter to false.
5. Never assume that the storage systems (Local Storage, Session Storage, Indexed DB, etc.) provided by the browser are secure from system access point view (they are at least secure from a web origin and JavaScript access point of view).
6. To export a key always use the **wrapKey()** function except if you exactly know why you are doing using the **exportKey()** function.

# Why should I not consider the browser storage systems as secure?

Once generated, a key (a **CryptoKey** object [20] behind the scenes) only expose metadata information to the JavaScript environment like type, algorithms, key usages, etc.

It is a requirement from the RFC that the content of the key was not accessible to the JavaScript environment [21] – See below, in yellow, the exposed content of a CryptoKey are runtime:

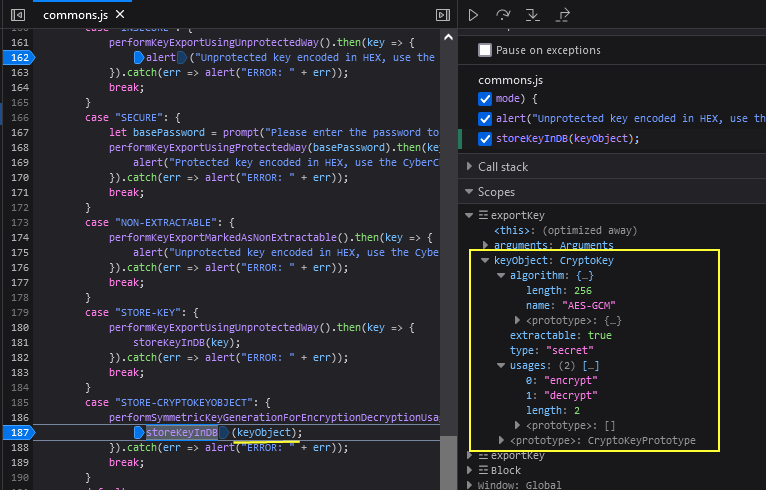


Figure 5: File Figure05.png

The only way to access to the key content from the JavaScript environment is to export it via the **exportKey()** or **wrapKey()** function.

A common and quick reflex can be to use the **exportKey()** function and store the exported content of the key in the IndexedDB browser storage systems like suggested by the RFC in section named “*Key Storage*” [21].

A question came in mind; does the browser protect the content stored in the IndexedDB storage when data are at rest?

Let us verify…

1) Store an exported key [22] in an IndexedDB using Firefox:

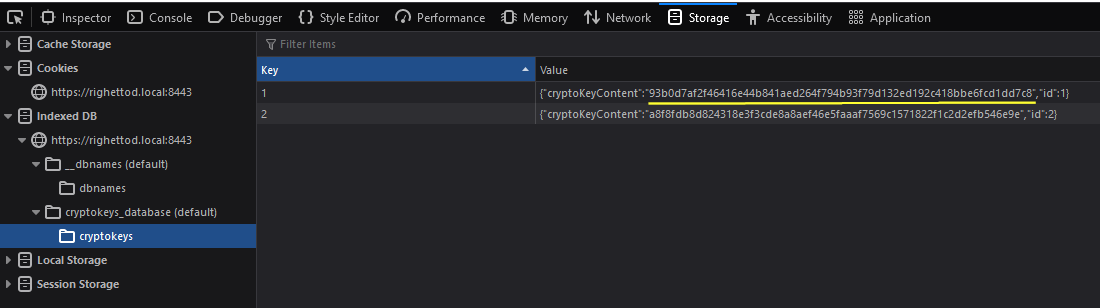


Figure 6: File Figure06a.png

2) Check at the file system level, if it is possible to access to the content of the key by reading the SQLite database used to store IndexedDB data:

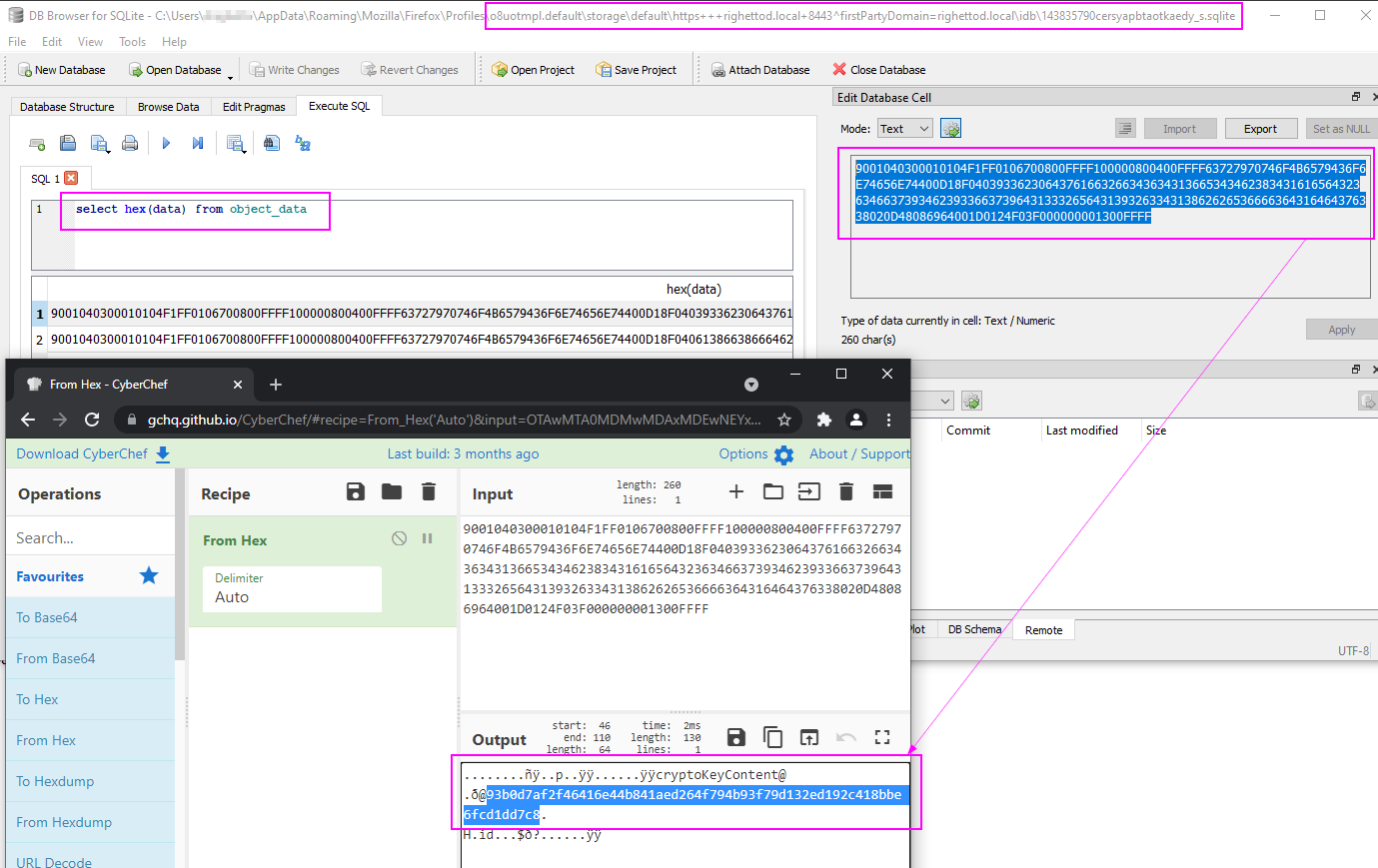


Figure 7: File Figure06b.png

We can access the stored content ☹

Same operation using Chrome, notes that it use LevelDB [23] format instead of SQLite for the storage:

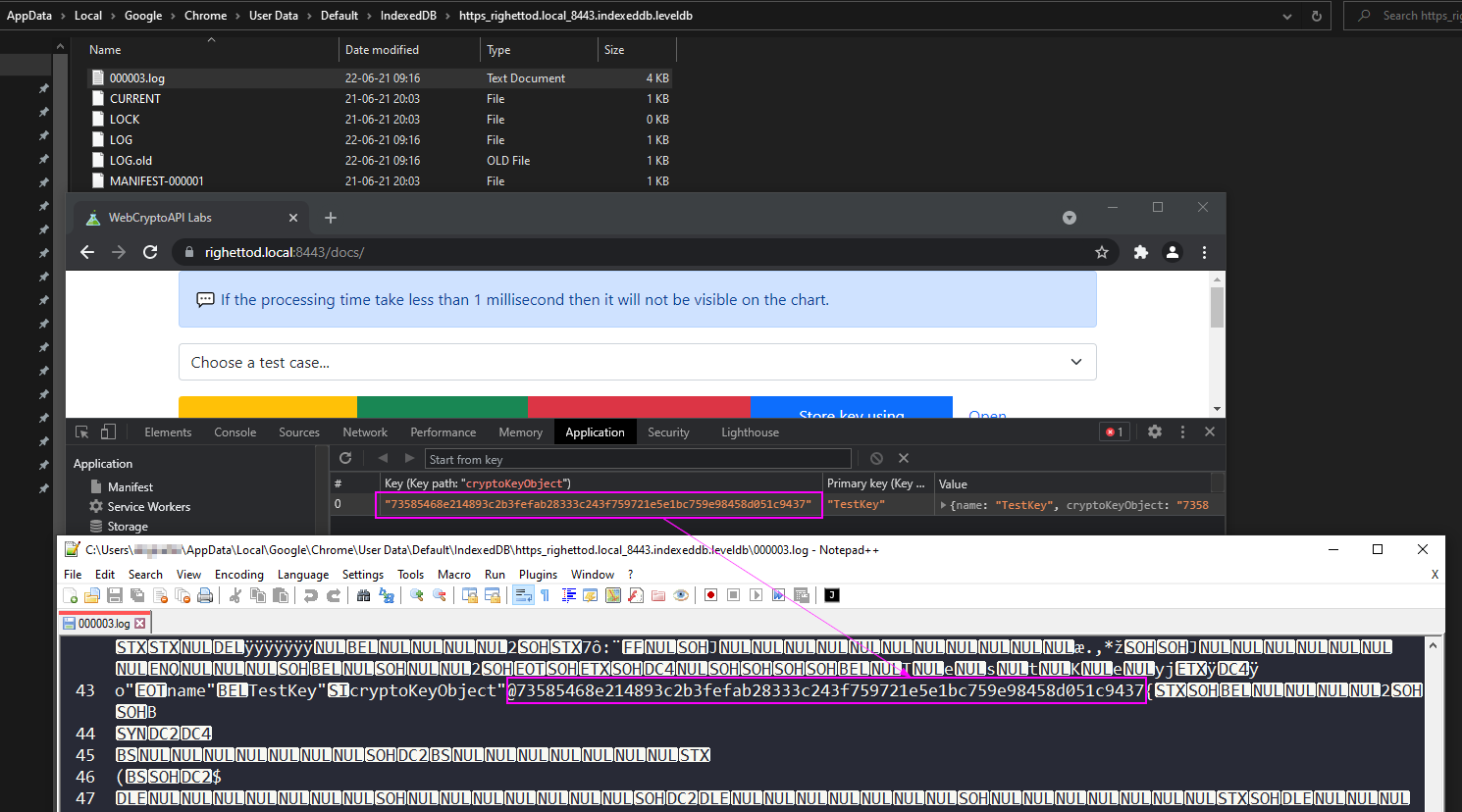


Figure 8: File Figure07.png

We can access the stored content too ☹

Same test for Edge (Edge also uses the LevelDB format):

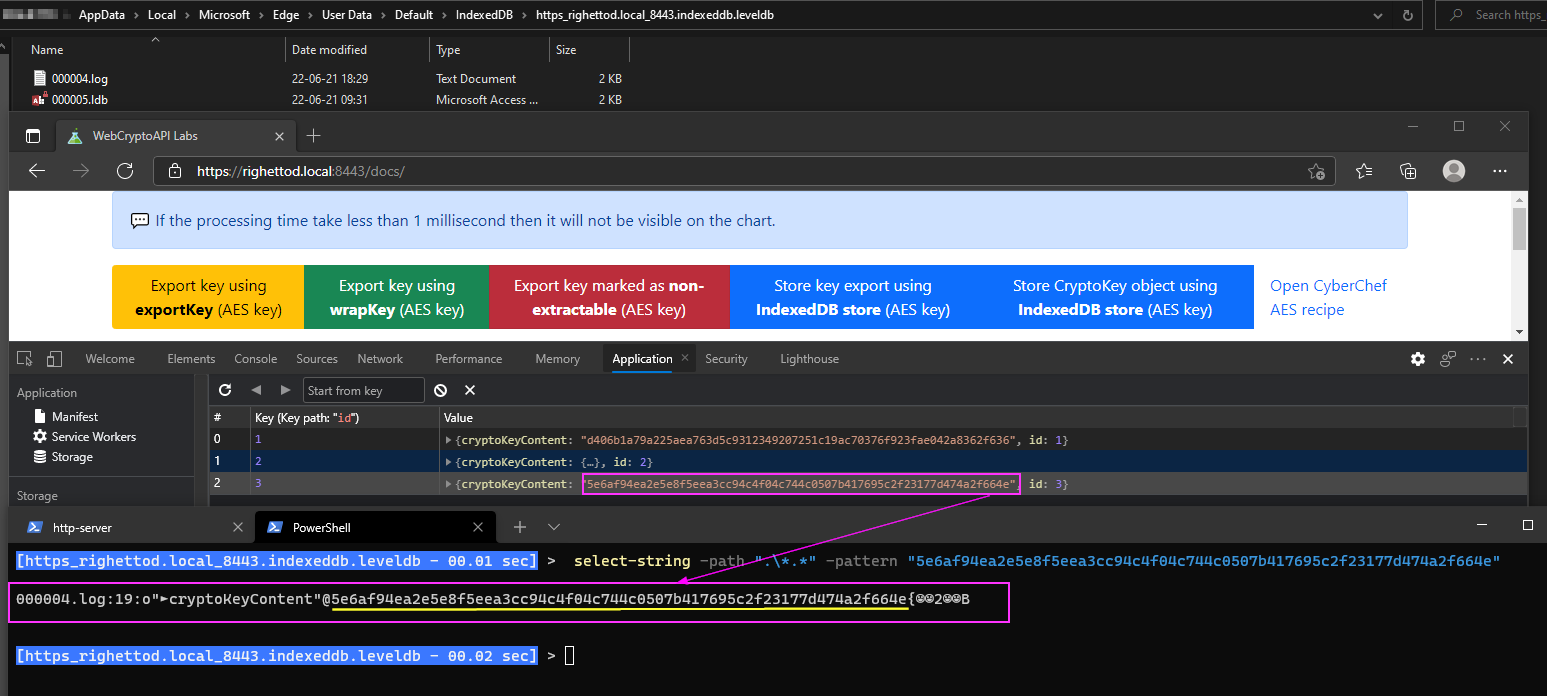


Figure 9: File Figure08.png

Same result ☹

In conclusion, consider the browser storage systems as unsafe from a file system point of view.

# Built-in protection against local DOS

Browsers implements protection preventing usage of cryptographic functions in a way that can cause instability or browser crashes. Below are some example of limitation implemented.

Here an example of limitation implemented in the function **getRandomValues()** – The request length cannot be superior to 65536 bytes:

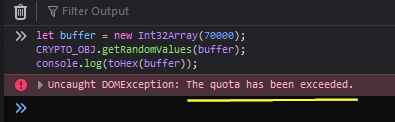


Figure 10: File Figure09.png

Another example of limitation implemented, here in the **encrypt()** function when used with an asymmetric key pair – In this example, the input data to encrypt is too large:

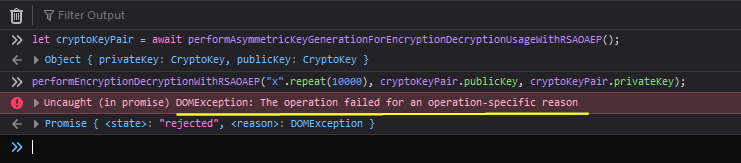


Figure 11: File Figure10.png

However, for the example above, depending on the way in which the cryptographic function is called, it may bypass the protection above and trigger another one or not...

When this test [24] is called, then, it trigger the sandbox protection on Chrome and Edge, at no moment the browser becomes unstable:

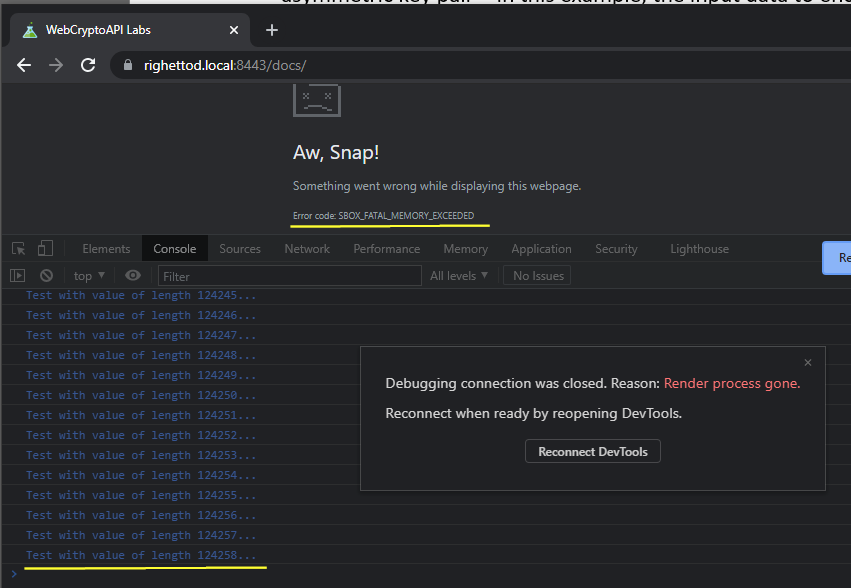


Figure 12: File Figure11.png

On Firefox, the browser becomes unstable and must be killed using the task manager. See this video [25] for a complete demonstration of the behaviour.

# The lab

In order to experiment all elements mentioned in this post, a lab was created [17].

The lab has the following user interface and provides a playground to discover/explore/test the different feature provided by the Web Cryptography API:

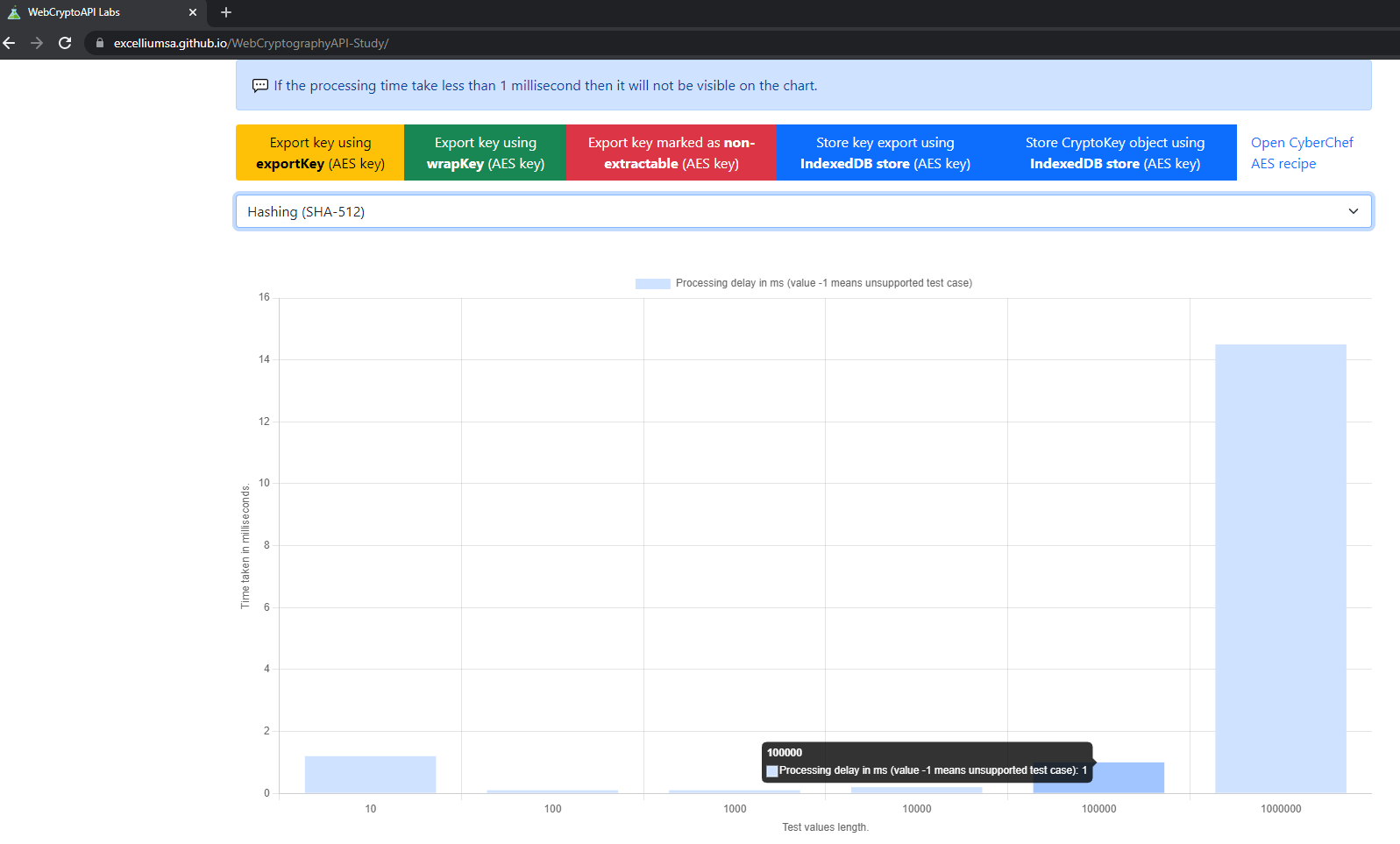


Figure 13: File Figure04.png

The lab features were tested on the last version (on June 2021) of Chrome, Firefox and Edge. All features were fully document to allow an easy access.

All cryptographic operations were centralized in a JS script named “**operations.js**” to facilitate the reading/understanding of the code as well as the modification to explore custom test cases [18].

All study note gathered were provided [19] in order to provide all information/hypothesis/assumptions that were used/made to create this blog post.

# Conclusion

TODO

# Credits

Dominique Righetto

# References

1. <https://www.npmjs.com/package/crypto-js>
2. <https://w3c.github.io/webcrypto/>
3. <https://w3c.github.io/webcrypto/#scope-out-of-scope>
4. <https://caniuse.com/cryptography>
5. <https://cwe.mitre.org/data/definitions/329.html>
6. <https://w3c.github.io/webcrypto/#concepts-underlying-implementation>
7. <https://en.wikipedia.org/wiki/White_Rabbit>
8. <https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API>
9. <https://w3c.github.io/webcrypto/#algorithm-overview>
10. <https://github.com/ExcelliumSA/WebCryptographyAPI-Study>
11. <https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/importKey#pkcs_8>
12. <https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/importKey#subjectpublickeyinfo>
13. <https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/importKey#json_web_key>
14. <https://www.keylength.com/en/3/>
15. <https://bettercrypto.org/#theory>
16. <https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/generateKey#parameters>
17. <https://excelliumsa.github.io/WebCryptographyAPI-Study/>
18. <https://github.com/ExcelliumSA/WebCryptographyAPI-Study/blob/main/docs/js/operations.js>
19. <https://github.com/ExcelliumSA/WebCryptographyAPI-Study/blob/main/study-note.pdf>
20. <https://developer.mozilla.org/en-US/docs/Web/API/CryptoKey>
21. <https://w3c.github.io/webcrypto/#concepts-key-storage>
22. <https://github.com/ExcelliumSA/WebCryptographyAPI-Study/blob/main/docs/js/commons.js#L179>
23. <https://en.wikipedia.org/wiki/LevelDB>
24. <https://github.com/ExcelliumSA/WebCryptographyAPI-Study/blob/main/docs/js/operations.js#L140>
25. <https://github.com/ExcelliumSA/WebCryptographyAPI-Study/blob/main/demo-instability.mp4>