


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

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

Data source

- [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
- Security random number generation (PRNG)
- Symetric encryption
- Asymmetric encryption

If possible for each crypto operation compare it [CryptoJS](#) with the same context: The risk it t cause a debate/drama...so let's see during the lab/research...

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 symmetric/asymmetric algorithm/key
- Sign and verify signature using symmetric/asymmetric algorithm/key
- Hash
- Generate symmetric/asymmetric key
- Derive key symmetric/asymmetric key
- Import/export symmetric/asymmetric key
- wrap/unwrap symmetric/asymmetric key (???)

Supported key format, see [here](#) :

- **raw**: An unformatted sequence of bytes. Intended for secret keys.
- **pcks8**: 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>.

Web Cryptography REC

JavaScript API for performing basic cryptographic operations in web applications

Usage
% of all users
Global
96.13% + 0.85% = 96.98%
unprefixed: 95.48%
Luxembourg
98% + 0.66% = 98.66%
unprefixed: 97.17%

Current aligned
Usage relative
Date relative
Filtered
All

IE	Edge *	Firefox	Chrome	Safari	Opera	Safari on iOS *	Opera Mini *	Android Browser *	Opera Mobile *	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
		2-31		3.1-7		3.2-7.1										
	12-18	32-33	4-36	7.1-10.1	10-23	8-10.3										
6-10	79-90	34-88	37-90	11-14	24-75	11-14.4		2.1-4.4.4	12-12.1				4-13.0			
11	91	89	91	14.1	76	14.6	all	91	62	91	89	12.12	14.0	10.4	7.12	2.5
		90-91	92-94	TP												

Notes
Test on a real browser
Known issues (0)
Resources (13)
Feedback

Many browsers support the `crypto.getRandomValues()` method, but not actual cryptography functionality under `crypto.subtle`.

- Support in IE11 is based on an older version of the specification.
- Support in Safari before version 11 was using the `crypto.webkitSubtle` prefix.
- In Edge 12-18, Web Crypto was not supported in Web Workers and Service Workers.




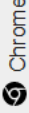

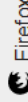
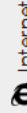

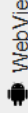
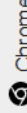
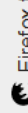


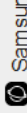

IE	Edge *	Firefox	Chrome	Safari	Safari on iOS *	Opera Mini *	Chrome for Android	UC Browser for Android	Samsung Internet
			89		13.7				
	90	88	90	14	14.4				
11	91	89	91	14.1	14.6	all	91	12.12	14.0
		90	92	TP					
		91	93						
			94						


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.

💡 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 reference used in all code snippets:

```
const cryptoObj = window.crypto;
```

📄 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("");
}
```

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**.

Code snippet:

```
//Return value like "1568bc6d71bc627d488b"
//for the call performRandomValuesGeneration(10)
function performRandomValuesGeneration(wantedLength) {
  let buffer = new Int32Array(wantedLength);
  cryptoObj.getRandomValues(buffer);
  return toHex(buffer);
}
```

Can used in the context of a HTTP (insecure) or a HTTPS (secure) connection.

Hashing

TODO

Key handling

TODO

Signature

TODO

Encryption

TODO