The W3C Web Cryptography API: Design and Issues

Harry Halpin
World Wide Web Consortium/MIT,
32 Vassar Street
Cambridge, MA 02139 USA
hhalpin@w3.org

ABSTRACT

The W3C Web Cryptography API will be the standard API for accessing cryptographic primitives in Javascript-based environments. We describe the motivations behind the creation of the W3C Web Cryptography API, deal with a number of objections, and then describe the API in detail with motivating use-cases and sample code. Lastly, we outline open issues and how the wider academic and security community can become involved in the process.

Keywords

Web, cryptography, W3C, API, Javascript, standards

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General

General Terms

Security and protection.

1. INTRODUCTION

Recently, the World Wide Web Consortium (W3C) has released as a First Public Working Draft the Web Cryptography API [6], which defines a number of cryptographic primitives to be deployed across browsers and native Javascript environments. This API has been driven mostly by industry as a standard despite the process being open to both invited experts and academics. It has made a number of design choices about how to best expose cryptographic functionality to Web application developers, and these design choices need to be more fully analyzed by the wider security and cryptography communities before implementation in native browser code begins.[1] Currently, all major browser vendors including Apple, Google, Microsoft, Mozilla, and Opera are participating in the Working Group. Thus, one should have every reason to believe this will API will become the de-facto cryptography API for the Web.

First, in Section 1 we will discuss the relationship of the API to common objections against the Javascript runtime environment and describe how it fits into the (still evolving) Web security model. Next, in Section 2, we will review the overall state of the Web Cryptography API and related specifications, such as an API for Key Discovery and a simplified 'high-level' API that abstracts away many of the details from Web application developers and the use-cases that motivate the work. Then in Section 3, we will overview the API itself in detail, providing sample code. Once the overview is complete, we will provide briefly in Section 4 an overview of existing controversial issues where the W3C Working Group has yet to reach consensus. Finally, in Section 6 we will outline future research issues. *Note to readers of first draft, this will change in final version:* This paper is unusual insofar as its final product, the Web Cryptography API, is not yet complete and still open for amendment. Yet it is precisely for these reasons that the W3C would like to ensure wider input from the security and cryptography community, including the academic community, at this stage. The Web Cryptography API will is supposed to have its design finalized and open issues closed in October of this year by reaching 'Last Call' in W3C process, with the call for input ending later in 2014. We expect that this paper will have changes based on Working Group decisions by the time of the final submission. However, we do not expect the core design of the API to change at this time.

2. BACKGROUND AND ASSUMPTIONS

There has been recently a rebirth of interest in cryptography in the browser environment, as exemplified by the release of high-quality Javascript cryptography such as the Stanford Javascript Cryptography Library [7]. As more and more applications transition to the Web, the desire for ordinary users to have more secure Web applications has increased and Web developers are attempting to match those expectations. However, as shown by the controversy over the development of the 'Cryptocat' encrypted chat application, which initially not only recreated their own cryptographic routines in Javascript but also deployed these Javascript libraries insecurely, there is currently no way to successfully within any reasonable threat model deploy cryptography in Javascript securely.[2] The W3C Web Cryptography API has as its mission to give Web application developers the ability to change this state of affairs in order to write secure Web applications that use cryptography.

In October 2012, after a public workshop,[3] the World Wide Web Consortium decided to back the creation of a unified Web Cryptography API due to developer demand. Also, there are a number of related cryptography APIs in the academic literature and various companies such as Apple are currently unifying their cryptography APIs across their various platforms (such as iOS and Mac OS X in the case of Apple). However, there are some fundamental objections to cryptography in Javascript within web browsers as have

[1]However, it should be noted that a pollyfill Javascript version called 'PolyCrypt' has been created by BBN and can be used to explore and test the current Web Cryptography specification, available at *http://polycrypt.net/*.

[2]For details regarding 'cryptocat', see *https://crypto.cat/*.

[3]The workshop was called 'Identity in the Browser.' Details are archived at *http://www.w3.org/2011/identity-ws/*

WS-REST 2014, Seoul, Korea
ACM (to be determined).

been discussed in a number of blog-posts such as "Javascript Cryptography Considered Harmful."[4] These objections can generally be phrased as either objections to the possibility of implementing any cryptosystem securely within Javascript or as objections against the general security model of the Web itself. Lastly, there are open questions about what kind of API makes sense for Web application developers, the vast majority of whom are not cryptographers.

2.1 The Web Security Model

Is Javascript cryptography doomed on the Web? First, there is no a priori reason why cryptographic primitives can not be programmed in Javascript, as exemplified by recent work on the Stanford Javascript Crypto Library.[5] Even with excellent implementations of cryptographic primitives such as the Stanford Javascript Crypto Library, browsers still have to download possibly these Javascript cryptographic code to obtain basic cryptographic functionality not provided natively by Javascript such as key storage. This process of downloading code is one powerful attack vector, as the connection to even the correct cryptographic routines can easily be hi-jacked by man-in-the-middle attacks against the process of determining a TLS connection as has been demonstrated by open source software such as *sslstrip*.[6] There are two paths out of this particular critique. The first involves using larger Web Security model's various access-control mechanisms including HSTS,[7] Content Security Policy,[8] and W3C Cross Origin Resource,[9] to allow websites to automatically switch to TLS connections without the ability of facing a man-in-the-middle attack and then carefully sandbox Javascript code in order to prevent cross-site scripting attacks while still sharing Javascript code across multiple domains. The work of the IETF on key-pinning [10] and IETF Certificate transparency [11] should allay many of the concerns over the CA system itself being compromised when using TLS. In final analysis, the Web Cryptography API provides cryptographic primitives, but the API itself does not provide the security to protect those primitives from known attacks on browsers. To mitigate against those attacks, a developer needs to use other specifications and implementation-specific mitigations.

This suite of W3C and IETF standards would not prevent many attacks, such as lack of proper sanitization in HTML forms leading to SQL injection attacks. However, in principle *any programming environment* is susceptible to poorly designed code. While the Web is exceptionally dangerous insofar as it is a shared and distributed space of code, nonetheless we will posit that it is possible to write Javascript code that is restricted to the same origin that can be secure in all other regards except for dependencies on cryptographic primitives. This assumption is often made in other programming language environments.

In conclusion, much of the critique of Javascript cryptography boils down to a critique of the security model of the Web itself, and as has been shown by the W3C and browser vendors - the Web security model can evolve to *in principle* allows one to write secure Javascript code. The key insight of the Web Cryptography is that a Cryptography API is actually not part of the security model of the Web. In fact, what it provides is *new standardized cryptographic functionality* to existing Web application development environments.

2.2 Javascript Web Cryptography

Due to TLS, almost every web browser and operating system already contains well-verified and reviewed cryptographic algorithms. In the case of Mozilla and in some versions of Chrome, this exists in NSS. Other browsers, such as Safari and Microsoft Internet Explorer, call from the browser to cryptographic routines are passed to the underlying operating system. Given that there is already cryptography functionality hidden in the browser and operating system, the Web Cryptography API simply exposes this already existing and often heavily verified cryptographic functionality to Web application developers through a standardized interface. Due to this constraint, the API itself is somewhat constrained practically by what features already exist cross-browser. Thus, it makes sense to expose ECDSA, but exposing arbitrarily elliptic curves or the mathematical functions needed to 'roll your own' cryptographic primitives such as modular arithmetic and native 'BigNum' integer support in Javascript is in general not feasible. While a case can be made that such functions should be exposed within the API as they are used primarily for implementing cryptographic, these larger changes would impact the entire the entire Javascript environment and thus are more properly standardized as part of Javascript itself in ECMA TC 39.[12] However, we provide a simple *BigNum* type definition in the Web Cryptography API, although we do not provide the underlying operations as native code. Furthermore, we do not deal with secure delete from memory, although most browsers can be run in a FIPs-compliant mode that allows them to have secure memory operations (with considerable cost to performance).

The value of the Web Cryptography API is to enable Web applications that require features such as cryptographically strong random number generation, constant-time cryptographic primitives, and a secure keystore. No one doubts that without these functions, Javascript web cryptography would be doomed. Currently, the situation is indeed dismal, with pseudo-random number generation in Javascript being not cryptographically strong and with no secure key store being available. Also, even well-designed libraries that could in principle be transported safely over the network such as the Stanford Javascript Cryptography can not offer guarantees of constant-time implementation due to the vagaries of cross-browser Javascript optimization. By exposing the primitives already in the browser as constant time functions and providing a suitable abstraction of a key-store, the API can address the current factors that make cryptography in Javascript impossible.

One critique that we purposefully do not address is the malleability of the Javascript run-time, where one function can be overridden by another function given the same name in a particular runtime (a technique called 'polyfill'). While this would seem to render all Javascript cryptography untrustworthy, this is not the case. First, in the hands of responsible and trusted developers operating within the same origin, polyfilling functions is useful. For example, it allows new experimental functions to be used in Web applications, such as a new hashing function, without touching the underlying code. While it is tempting to believe that the Web Cryptography API should restrict polyfilling over its functions, this would hender trusted developers that need polyfills and fracture the Javascript environment from the Web environment. Thus, the Web Cryptography API is released to developers as a native Web API in the same fashion as any other API in related W3C standards such as HTML5.

[4] At *http://www.matasano.com/articles/javascript-cryptography/*.

[5] See *https://crypto.stanford.edu/sjcl/*

[6] See details of Moxie Marlinspike's well-known attack at *http://www.thoughtcrime.org/software/sslstrip/*.

[7] *https://tools.ietf.org/html/rfc6797*

[8] *http://www.w3.org/TR/CSP/*

[9] *http://www.w3.org/TR/cors/*

[10] Such as the Tack proposal at *https://tools.ietf.org/html/draft-perrin-tls-tack-02*

[11] See *https://tools.ietf.org/html/draft-laurie-pki-sunlight-12* for details.

[12] *http://www.ecma-international.org/memento/TC39.htm*.

We assume that the Web application is well-designed and thus the malleability is not a threat. Lastly, we are producing the Web Cryptography API as a native cryptographic library rather than encouraging the use of cryptographic browser plug-ins, despite plug-ins being currently 'best practice' on the Web in applications such as the latest version of Cryptocat. The reason for this is straightforward. While browser code and underlying cryptographic code has been heavily invested in and in the case of open-source code subject to wide review, browser plug-ins are generally not: There are a large number of 0-day attacks on browser-plug ins, and thus while they may be resistant to cross-site scripting and other common attacks, they often expose new threats to the browser environment. So, most browser vendors will be for security reasons encouraging developers to write Web applications directly in Javascript and phasing out plug-ins over their next release cycles. Equally objectionable is that the use of plug-ins to provide cryptographic materials binds the application to a single browser environment. For example, the use of a government-mandated ActiveX plug-in binds users to Internet Explorer for e-commerce in Korea, an issue that so irritated users that it has become an election issue. Ironically, this plug-in was later revealed to store its key material insecurely.

2.3 Host-based security

The Web Cryptography API also makes a number of strong assumptions that some in the security community find objectionable. In particular, Patrick Ball states that "CryptoCat is one of a whole class of applications that rely on what's called *host-based security*. The most famous tool in this group is Hushmail, an encrypted e-mail service that takes the same approach. Unfortunately, these tools are subject to a well-known attack. I'll detail it below, but the short version is if you use one of these applications, your security depends entirely the security of the host. This means that in practice, CryptoCat is no more secure than Yahoo chat, and Hushmail is no more secure than Gmail. More generally, your security in a host-based encryption system is no better than having no crypto at all."[13] The Web Cryptography API is, like other Web APIs, built around a host-based security model that enforces (with access-control via CSP and CORS) the same-origin policy. Even in perfectly designed Web App, the behavior as regards the DOM (Document Object Model) is *completely controlled* by the host.[14]

We would argue that host-based security is a feature, not a bug. Contrary to some, an argument can be made that client security can easily be less than host-based security. Average users do not, in general, have any motivation to update their systems more often or with better safeguards than the security professionals that are employed by major corporations that provide hosts Web application hosting. The question is whether or not one should trust the updating mechanism of *any* software. Ball and Schneir could argue that at least the user is in control of the updates on the client, while on the server-side the host can invisibly update the software. Again, the assumption is that the host may be compromised and invisibly update malicious software or break its assurances. However, this can be done easily on clients as well (for example, via a rootkit), so we should assume at least a parity in security between host and clients. There may even be reasons to believe that applications on a remote host are superior: most 0-day attacks can be performed before proper updates to client devices can correct the attack. Host-based security may scale better: It is simply easier and more secure to have hosts quickly update Javascript once rather than deal with the inevitable lags of client-based updates. Indeed, for this reason browser plug-ins are slowly being eliminated in general.

[13] *https://www.schneier.com/blog/archives/2012/08/cryptocat.html*

[14] The DOM is the primary abstract syntax tree of HTML that is manipulated for presentation and interaction within the browser. See the HTML5 specificatiom *http://www.w3.org/TR/html5/* for more information.

Lastly, implicit in this argument is that client devices such as the FreedomBox[15] are more secure than storing data on a host such as Google Drive (or some other cloud-serving service in the jurisdiction of the user's choosing). It can be argued this is not to be true in many scenarios. For example, client-device seizures are likely at least as common if not moreso than server seizures and legal compunction to release data, especially in high risk zones such as Syria. Furthermore, even the evidence of running cryptographically-enabled secure services on a client can be used as evidence against someone. In these cases, storing data securely 'on the cloud' makes far more practical sense. Mistaking *proximity for security* is a simple conceptual error. However, this does not necessarily mean that the host should have control over private key material. The key management and store of the Web Cryptography API has been designed in such a way that it should be possible for secret key material to be stored in such a way that the host does not, if the application is built correctly, control the keys of the user. This would allow encrypted data to be stored in the host that the host cannot decrypt.

Is releasing this cryptography in Javascript to developers responsible? There are lots of use-cases, and while some of them are controversial, the potential benefits of enabling richer web-apps that can use cryptography outweighs some of the controversial points that have been raised about the availability of cryptographic APIs in the browser. Given the current dangerously insecure state of Javascript cryptography and the fact that developers are already re-implementing cryptographic functions in Javascript insecurely, we will have to trust the Web developer community to get better at building more secure Web applications.

3. OVERVIEW OF WEB CRYPTOGRAPHY

The W3C Web Cryptography Working Group has prepared a group of specifications rather than a single API: one use-case document [5], one normative specification that can fulfill the use-cases [6], as well as two (likely) non-normative documents describing a high-level API [2] and key discovery [8]. We will go over the use-cases and only briefly touch upon the high-level API and key discovery documents before exploring in detail the Web Cryptography API itself.[16]

3.1 Use-cases and Scope

The amount of possible cryptographic functionality that Web application developers could need is huge, ranging from simple primitives to advanced certificate functionality. In this regard, the Web Cryptography Working Group applied a two-step process. As motivated in the previous section, we restricted our assumptions to a host-based security model around the same origin policy without major modifications to the underlying Javascript runtime. The open question is what precise primitives are to be released.

A core number of primitives were deemed to be necessary for any application, and these were termed *primary features*. Primary features in scope are: key generation, encryption, decryption, deletion, digital signature generation and verification, hash/message authentication codes, key transport/agreement, strong random num-

[15] *https://www.freedomboxfoundation.org/*

[16] This work presents the work of the larger W3C Web Cryptography Working Group, not just individual author who is Team Contact for the Working Group. For example, many of the key design decisions have been taken by the editor of documents. In the case of the API, this is Ryan Sleevi, and would be Mark Watson for the Key Discovery work.

ber generation, key derivation functions, and key storage and control beyond the lifetime of a single session. In addition, the API should be asynchronous and must prevent or control access to secret key material and other sensitive cryptographic values and settings. Encryption and decryption include both symmetric and asymmetric cryptography.

A number of features were demanded by the community and would be up for inclusion if suitable use-cases could be found that could not be covered by primary use-cases. These include: control of TLS session login/logout, derivation of keys from TLS sessions, a simplified data protection function, multiple key containers, key import/export, a common method for accessing and defining properties of keys, and the lifecycle control of credentials such enrollment, selection, and revocation of credentials with a focus enabling the selection of certificates for signing and encryption. A number of features are explicitly out of scope, in particlar those dealing with device-specific features and larger trust models, such as special handling directly for non-opaque key identification schemes, access-control mechanisms beyond the enforcement of the same-origin policy, and functions in the API that require smartcards. For more detail, please see the W3C Web Cryptography Working Group Charter.[17] API features that are not implemented due to time constraints can be put in a non-normative 'roadmap' document for future work of the next version of the API. For the existing sample code that implements this use-cases, see the Web Cryptography Use-Cases and Requirements document [5]. Note that new use-cases can still be accepted at this time until the document reaches Candidate Recommendations status.

The use-cases the group has currently accepted are, as given by the use-case document [5]:

Multi-factor Authentication: A web application may wish to extend or replace existing username/password based authentication schemes with authentication methods based on proving that the user has access to some secret keying material.

Protected Document Exchange: When exchanging documents that may contain sensitive or personal information, a web application may wish to ensure that only certain users can view the documents, even after they have been securely received, such as over TLS. One way that a web application can do so is by encrypting the documents with a secret key, and then wrapping that key with the public keys associated with authorized users.

Cloud Storage: When storing data with remote service providers, users may wish to protect the confidentiality of their documents and data prior to uploading them.

Document Signing: A web application may wish to accept electronic signatures on documents. The web application must be able to locate any appropriate keys for signatures, then direct the user to perform a signing operation over some data, as proof that they accept the document. European eID legislation and work from ABC4Trust [18] provides more details.

Data Integrity Protection: When caching data locally, an application may wish to ensure that this data cannot be modified in an offline attack. In such a case, the server may sign the data that it intends the client to cache, with a private key held by the server. The web application that subsequently uses this cached data may contain a public key that enables it to validate that the cache contents have not been modified by anyone else.

Secure Messaging: While TLS/DTLS may be used to protect messages to web applications, users may wish to directly secure messages using schemes such as off-the-record (OTR) messaging. The Web Cryptography API enables OTR by allowing key agreement to be performed so that the two parties can negotiate shared encryption keys and message authentication code (MAC) keys, to allow encryption and decryption of messages, and to prevent tampering of messages through the MACs.

In general, Web applications Web are developing towards *multi-channel* communication across multiple origins where data and even entity authentication of the client is required. This loosely-coupled design architecture should allow both data integrity and even entity authentication across networks of communicating web applications by the use of the Web Cryptography API combined with CSP and CORS. For example, digital signatures are very useful for authentication across multiple hosts (like signing OAuth2 tokens to verify the client rather than using symmetric shared secrets). Not only does this defeat an attack vector of cookie-snatching attacks, it allows possibilities of possibly powerful technologies such as federated identity without HTTP-redirection phishing attacks. Applications of this type such as Mozilla Persona are already planning on using the Web Cryptography API to enable this kind of functionality.[19]

3.2 The Web Cryptography Family of Specifications

The use-case document, has already been described and will be a non-normative deliverable [5]. The main deliverable is the Web Cryptography API itself [6]. However, there are two more deliverables that may or may not become normative, namely a document to describe key discovery (that has privacy concerns) [8] and a high-level API to enable Web Application developers to use the API more easily (built as a shim on top of the Web Cryptography API) [2].

Web Cryptography Use-Cases and Requirements: For each suggested new feature for the Web Cryptography API outside of the primary API features given in the scope (including currently listed secondary API features), a concrete use-case must be described and produce clearly defined requirements that are agreed upon by the Working Group. These are kept track of in a use-case document with sample code for each use-case [5]

Web Cryptography API: Commonly-used cryptographic primitives made available to Web application developers via a standardized API to facilitate their operation. This API is a set of bindings that can be thought of as equivalent in functionality to OpenSSL bindings, but provided natively as constant time functions while relying on platform-specific key storage functionality via a suitable abstraction layer. The API is described in more detail than provided in this paper in the API document itself [6], although some of the design rationale is more detailed in this paper.

Web Cryptography Key Discovery: This specification describes an API for discovering named, origin-specific pre-provisioned cryptographic keys for use with the Web Cryptography API. Pre-provisioned keys are keys which have been made available to the UA by means other than the generation, derivation, imporation functions of the Web Cryptography API. These keys are currently limited to origin-specific keys [8].

High-level Web Cryptography API: The Web Cryptography API is not a simple API geared towards the average web developer, rather its use requires near-expert knowledge of cryptography, so a 'high-level' API has been designed around fewer use cases and is not concerned with backward-compatibility with existing crypto

[17] Available at *http://www.w3.org/2011/11/webcryptography-charter.html*

[18] See *https://abc4trust.eu/*

[19] Formerly called BrowserID, see *https://login.persona.org/* for more information.

systems and protocols, leveraging the IETF JOSE JWA, JWK and JWE JSON formats for algorithms, public keys and cipher data.[20] This design has been heavily influenced by the Stanford Javascript Cryptography Library and KeyCzar[21], with its main aim advantage being the use of a standardized JOSE JSON data-format for data and key material rather than custom formats.

The issue of using keys as 'super-cookies' caused the separation of Web Cryptography Key Discovery from the Web Cryptography API, and the inability of the import and export of arbitrary key material outside of the same-origin policy. With the exception of ephemeral keys, its often desirable for applications to strongly associate users with keys. These associations may be used to enhance the security of authenticating to the application, such as using a key stored in a secure element as a second factor, or may be used by users to assert some identity, such as an e-mail signing identity. As such, these keys often live longer than their counterparts such as usernames and passwords, and it may be undesirable or prohibitive for users to revoke these keys. Because of this, keys may exist longer than the lifetime of the browsing context and beyond the lifetime of items such as cookies, thus presenting a risk that a user may be tracked even after clearing such data. This is especially true for keys that were pre-provisioned for particular origins and for which no user interaction was provided. One design goal of the core API is then to match the life-time guarantees of keys to that of the rest of the browser environment, which key discovery of any pre-provisioned key violates. The details of Key Discovery are beyond scope, but consist of adding a *findKey* function that discovers some named key. See the Key Discovery Document for details [8].

It is well understood that developers may have difficulty in using the core Web Cryptography API, as such as an API forces a developer to explicitly provide all the default values that they wish to use. While helper functions can be provided for initialization vectors, it was felt that using defaults in the API might lead to possibly very dangerous default usage of operations. The Web Cryptography API assumes the user is implementing a cryptosystem where many of the defaults have already been specified in the specification of said system. In general, simpler *jQuery*-like libraries will likely be evolved by the market to address the needs of most Web developers who wish to do simple tasks on a per-application basis, for whom the Web Cryptography API is too hard to use. However, the Working Group decided that a sample high-level API can be implemented as a library on top of the Web Cryptography API as a good example. For example, in the high-level API reasonable defaults for key lengths are chosen and encryption always features signing. This API is further detailed in its own document and so will not be explored further here [2].

Note that the details of any user experience (such as prompts) will not be normatively specified, although they may be informatively specified for certain function calls. While this may be desirable for operations involving key import, export, and private material key material access, browser vendors have historically been unable to standardize elements of the graphical user interface, including the infamous 'TLS' icon case.

4. EXAMPLES

A number of examples may clarify the usage of the API. The first is to generate a signing key pair and sign some data, and this is given in Figure 1. More examples, including that of symmetric key encryption, are given in the specification [6] and the use-case document [5].

[20]See *https://tools.ietf.org/wg/jose/* for details. Note that this JSON format corrects some of the parsing ambiguities found in ASN.1 [4].

[21]*http://www.keyczar.org/*

5. OPEN ISSUES

A number of open issues are still being tackled by the Working Group. These are kept in a publically available repository for comment.[22] One issue is key derivation. Further distinction is needed to clarify the differences between key generation and key derivation. Should they be distinguished by their inputs (key generation takes parameters, while key derivation takes parameters and keys), by their outputs (key generation generates keys, key derivation generates opaque bytes as secret material), or is there some other construct to distinguish the two?

There is an open question as to how the API should support key wrap and unwrap operations. Should they be distinct operations, independent from key import/export, or should they be part of the parameters supplied during import/export?

There are no per-algorithm security considerations as these considerations may change over time and no algorithm registry. Yet there is no central place for developers to go to that lets them determine which algorithms are known to be broken in certain ways.

Finally, currently the algorithm provides no defaults and only a few helper functions. Further feedback is needed from developers t detemrine if this is indeed the best way forward, or if certain constructs from a 'high-level' API can simplify usage of the low-level Web Cryptography API.

6. CONCLUSIONS AND NEXT STEPS

The creation of the Web Cryptography API has been heavily influenced by prior academic work such as the Stanford Javascript Library, but also presented a number of hard practical questions where empirical research is needed [7]. One if the ability of developers to use cryptography APIs in general. For example, while it seems that users will generally use the highest-level of abstraction available to them, it is still an open empirical debate if one should present a 'dangerous' lower-level API in conjunction with a more safe 'higher-level' API or rather a single API with appropriate defaults. A large-scale study of the APIs usage amongst web developers would be appreciated. Larger and more thorough studies need to be done, both of deployed Web application code [7] and testing new designs.

On the other hand, more formal research is needed on the larger framework of the Web Cryptography API and the Web security model, and recent academic work in formal analysis of APIs such as PKCS#11 may be very useful [3]. The Web security model (as based on standards from the W3C and IETF) is woefully underanlayzed, despite its rising popularity as the preferred method of delivering even high-value applications with security implications. For example, even very basic theoretical divisions such as distinguishing between a *Web Attacker* with control of the Javascript runtime environment in a browser and a *Network Attacker* with control over the HTTP upgrade to a TLS session have only recently been made and formally studied [1]. Currently, there is a larger problem: that the entire Web Security Model needs to be formalized and modeled, and it only makes sense formalizing the security analysis of the Web Cryptography as part of this larger analysis.

The core design of the W3C Web Cryptography API has finalized [6], however, it is still very much a work-in-progress with a number of open issues and possible changes where the wider input of the community is necessary before the W3C sets the API 'in

[22]Available at *http://www.w3.org/2012/webcrypto/track/*

```
var algorithmKeyGen = { name: "RSASSA-PKCS1-v1_5", modulusLength: 2048,
  publicExponent: new Uint8Array([0x01, 0x00, 0x01])};

var algorithmSign = {
  name: "RSASSA-PKCS1-v1_5",
  hash: {name: "SHA-256",} };

var keyGen = window.crypto.subtle.generateKey(algorithmKeyGen, false,["sign"]);

keyGen.oncomplete = function(event) {
  var signer = window.crypt.sign(algorithmSign, event.target.result.privateKey);
  signer.oncomplete = function(event) {
    console.log("The signature is: " + event.target.result);
  }
  signer.onerror = function(event) {
    console.error("Unable to sign");}

  var dataPart1 = convertPlainTextToArrayBufferView("hello,");
  var dataPart2 = convertPlainTextToArrayBufferView(" world!");

  signer.process(dataPart1);
  signer.process(dataPart2);
  signer.finish();
};

keyGen.onerror = function(event) {
  console.error("Unable to generate a key.");
};
```

Figure 1: Public Key Signature Example

stone' by implementing it and establishing a uniform test-suite in order to determine compliance. Thus, we would appreciate from the community an overview of the open issues of the Working Group before the W3C Web Cryptography standardizes the API as a Candidate Recommendation. Lastly, there will likely be a next version of the API, and this version may either expose 'lower-level' primitives such as the functionality needed to support zero-knowledge proofs or it may aim to support more complex demands, such as certificates and smartcards.

7. REFERENCES

[1] Akhawe, D., Barth, A., Lam, P. E., Mitchell, J., and Song, D. Towards a formal foundation of web security. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, IEEE Computer Society (Washington, DC, USA, 2010), 290–304.

[2] Dahl, D. High-level Web cryptography API. Working draft, W3C, 2013. https://dvcs.w3.org/hg/webcrypto-highlevel/raw-file/tip/Overview.html.

[3] Delaune, S., Kremer, S., and Steel, G. Formal security analysis of PKCS#11 and proprietary extensions. *J. Comput. Secur. 18*, 6 (Sept. 2010), 1211–1245.

[4] Kaminsky, D., Patterson, M. L., and Sassaman, L. PKI layer cake: new collision attacks against the global x.509 infrastructure. In *Proceedings of the 14th international conference on Financial Cryptography and Data Security*, FC'10, Springer-Verlag (Berlin, Heidelberg, 2010), 289–303.

[5] Rangathan, A. Web Cryptography Use-cases. Working draft, W3C, 2013. http://dvcs.w3.org/hg/webcrypto-usecases/raw-file/tip/Overview.html.

[6] Sleevi, R. Web Cryptography API. Working draft, W3C, 2013. http://www.w3.org/TR/WebCryptoAPI/.

[7] Stark, E., Hamburg, M., and Boneh, D. Symmetric cryptography in Javascript. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, IEEE Computer Society (Washington, DC, USA, 2009), 373–381.

[8] Watson, M. Web Cryptography Key Discovery. Working draft, W3C, 2013. https://dvcs.w3.org/hg/webcrypto-keydiscovery/raw-file/tip/Overview.html.