Security Analysis of the W3C Web Cryptography API

Kelsey Cairs[1] Harry Halpin[2] Graham Steel[3]

[1]Washington State University, Seattle, USA
[2]W3C/Inria, Paris, France
[3]Cryptosense, Paris, France

Security Standardization Research Conference, NIST, Dec 5th 2016

Outline

Javascript Cryptography

Considered Harmful?

- Javascript lacked a cryptographic PNRG (*Math.random*)
- No *BigInt* support
- People creating their own insecure Javascript APIs (*OpenPGP.js*)
- Or secure ones like Stanford Javascript Crypto Library

World Wide Web Consortium (W3C)

- Standards body for Web standards like HTML5
- XML-DSIG, Content Security Policy (XSS attack prevention), Web Authentication ...
- Identity in the Browser Workshop (*http://www.w3.org/2011/identity-ws/*)
- Consensus from browser vendors to fix browser crypto

Role of Formal Verification

Security API

- Provide as much functionality as possible
- Yet prevent attacks and errors (high vs. low-level API)
- A *security API* consists of a set of functions that are offered to some other program that uphold some security properties, regardless of the program making the function calls and what functions are called (Bond, 2001)
- No clear threat model, but clear security properties
- Can we prove security properties for standard APIs in browser **before** standardization?

Formal Verification of APIs

Set-up

- Using model checking and theorem proving to verify security properties
- Dolev-Yao (DY) model: Crypto-primitives are functions on bitstrings

Tools

- **Alloy**: SAT solving over infinite models (*Trusted Platform Module 1.2*)
- **Scyther**: Unbounded sessions, no control flow (*Signal*)
- **Tamarin**: Unbounded sessions, mutable global state (*TLS*)
- **Proverif**: Unbounded sessions, Horn clauses (*Signal*)
- **AVISPA**: Unbounded sessions, mutable global state, based on rewrite rules with SAT solver (*Web Crypto API*)

W3C Web Cryptography API

Web Cryptography API

W3C Candidate Recommendation *11 December 2014*

This version:
http://www.w3.org/TR/2014/CR-WebCryptoAPI-20141211/
Latest Editor's Draft:
https://dvcs.w3.org/hg/webcrypto-api/raw-file/tip/spec/Overview.html
Latest Published Version:
http://www.w3.org/TR/WebCryptoAPI/
Previous Version(s):
http://www.w3.org/TR/2014/WD-WebCryptoAPI-20140325/
Editors:
Ryan Sleevi, Google, Inc. <sleevi@google.com>
Mark Watson, Netflix <watsonm@netflix.com>
Participate:
Send feedback to public-webcrypto@w3.org (archives), or file a bug (see existing bugs).

Abstract

This specification describes a JavaScript API for performing basic cryptographic operations in web applications, such as ha
API for applications to generate and/or manage the keying material necessary to perform these operations. Uses for this A
integrity of communications.

W3C Web Cryptography API

Overview

- *RandomSource*: Pseudorandom number generation.
- *CryptoKey*: JSON object for key material.
- *CryptoOperation*: Functions such as encryption and wrapping, along with error codes.

Key Types

- **Type**: Public, private or secret (symmetric)
- **Extractable**: A boolean specifying whether the key material may be exported to Javascript
- **Algorithm**: The algorithm used to create the key
- **Usages**: Attributes which specify the key's allowed operations

Applications of WebCrypto API

Examples

- Netflix
- uProxy (Google)
- Signal
- Crypto.cat
- Digital Signatures for eGovernment

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

var algorithmSign = {
  name: "RSA-PSS",
  saltLength: 32,
  hash: {
    name: "SHA-256"
  }
};
```

```
window.crypto.subtle.generateKey(algorithmKeyGen,
false, ["sign","verify"]).then(
function(key) {
  var dataPart1 = convertPlainTextToArrayBufferView("hello,");
  var dataPart2 = convertPlainTextToArrayBufferView(" world!")
  return window.crypto.subtle.sign(algorithmSign,
  key.privateKey)
  .process(dataPart1)
  .process(dataPart2)
  .finish();
 },
  console.error.bind(console, "Unable to generate a key")
).then(
  console.log.bind(console, "The signature is: "),
  console.error.bind(console, "Unable to sign")
);
```

Security Goals

Security Assumption

The origin is trusted when the WebCrypto API is initialized and secrets are successfully encrypted and stored on the client.

Threat Model

A temporary compromise of the Javascript environment after secrets have been encrypted by WebCrypto and stored on the client (XSS attack). Attacker goal is to decrypt secrets.

Security Property

Access to the raw key material that is private, secret, or explicitly typed as non-extractable should not be accessible to Javascript.

AVISPA Model

Keys

$$keystore(K) : key \rightarrow fact$$

Attacker Goal

$$\begin{aligned} &step\ i\_encrypt(M, K) := \\ &\quad iknows(M) \wedge iknows(K) \\ &\quad \Rightarrow iknows(scrypt(K, M)) \\ &step\ i\_decrypt(M, K) := \\ &\quad iknows(scrypt(K, M)) \wedge iknows(K) \\ &\quad \Rightarrow iknows(M) \end{aligned}$$

Attacks on WebCrypto API

Goal

Systematically modeling different use cases using AVISPA and assessing the resulting attacks on the Web Crypto API

WebCrypto API Attack Overview

- **Export Attack:** Exporting extractable key data and changing usages.
- **API Attack:** Using API calls to recover clear text of encrypted communication via building on the attack on key wrapping.

Export Attack

Attack Overview

Usages can be added and changed simply by wrapping and unwrapping the extractable key:
$wrap(skey, ikey), unwrap(skey, ikey)$

AVISPA Model

Instance Variables: $key, ikey : key$
$st : type$
Initial State: $sym(skey) \wedge sym(ikey)$
$\wedge keystore(skey, st) \wedge keystore(ikey, st)$
$\wedge extract(skey) \wedge usages(ikey)$
Goal: $addUsage() : encryptUsage(skey)$

API Attack

Extending to Key Exchange

As key wrapping is a composition of export and encrypt, if an attack existed on a wrapped key, then the same attack would apply to an encrypted message that uses this wrapped key.

- *Symmetric encryption* The sender wraps the key using a symmetric key shared with the receiver who unwraps the key
- *Asymmetric encryption* The sender wraps the key using public key for the receiver who unwraps with the corresponding private key
- *Symmetric encryption with asymmetric signing* The symmetric encryption case augmented by signing with the sender's private key
- *Asymmetric encryption with asymmetric signing* The asymmetric encryption case augmented by signing with the sender's private key

Fixing attack

Using distinct keys for each direction of communication and using distinct *usages* attributes prevents this type of attack.

CFRG draft: Security Guidelines for Cryptographic Algorithms in the W3C Web Cryptography API

Algorithm/Mode	legacy	future	Note
RSAES-PKCS1-v1_5	×	×	
RSA-OAEP	✓	✓	
RSASSA-PKCS1-v1_5	✓	×	No security proof
RSA-PSS	✓	✓	
ECDSA	✓	×	Weak provable security results
ECDH	✓	✓	
AES-CBC	✓	✓	NB not CCA secure
AES-CFB	✓	✓	NB not CCA secure
AES-CTR	✓	✓	NB not CCA secure
AES-GCM	✓	✓	
AES-CMAC	✓	✓	
AES-KW	✓	×	No public security proof
HMAC	✓	✓	
DH	✓	✓	
SHA-1	×	×	See text
SHA-256	✓	✓	

Fixing the WebCrypto API

Recommendations for Errors

- All errors caused by improper padding or incorrect key length/formatting are indistinguishable. (Padding errors will be returned from a different subroutine than the other errors and be discovered first, so any information about the *source* of the error is potentially a distinguishing factor.)
- Lengths of unwrapped keys are verified to match one of the predefined key lengths (not accepted)
- All bytes of padding are checked for conformance (not accepted).

High-level API

Defaults?

- Randomize the IVs
- *AES-GCM* mode for symmetric crypto
- *RSA-PSS* should be used for digital signatures
- emphRSA-OAEP should be used for encryption.
- *ECDH* for Diffie-Hellman Key Exchange (Curve 25519 when added)
- *SHA-256* for hash functions
- *HMAC* for MACs
- *Key size* 2048 for RSA, 256 for symmetric and EC crypto.

Take-home message

For any future API

- Key-wrapping must use special operating environment to keep private ke material secure
- Enforce *usages* on keys by default
- Keep any information out of *error codes*
- Beware of “backwards-compatible” arguments for algorithms
- Larger issues re isolation and key storage (keys are super-cookies for tracking!) on the Web

Next Steps for Standards Research

API issues

- APIs seem simple, but more tricky to test than protocols.
- Real-world applications use multiple APIs with user permissions and (possibly conflicting) security and privacy goals.
- Can we integrate provable security properties into specs? (*WebIDL*)
- Get independent security expertise involved early
- Don't assume major vendors know what they are doing

Start modeling in design stage

- See work on *TLS 1.3* for good example.
- Make formal verification part of conformance testing.
- Automatic Generation of test-suite?

Any Questions?