Update on Web Cryptography

Jul 21, 2017
by Jiewen Tan
@alanwaketan

Cryptography is the cornerstone of information security, including various aspects such as data confidentiality, data integrity, authentication, and non-repudiation. These provide support for the fundamental technologies of today's Internet like HTTPS, DNSSEC, and VPN. The WebCrypto API was created to bring these important high-level cryptography capabilities to the web. This API provides a set of JavaScript functions for manipulating low-level cryptographic operations, such as hashing, signature generation and verification, encryption and decryption, and shared secret derivation. In addition, it supports generation and management of corresponding key materials. Combining the complete support of various cryptographic operations with a wide range of algorithms, the WebCrypto API is able to assist web authors in tackling diverse security requirements.

This blog post first talks about the advantages of implementing web cryptography through native APIs, and then introduces an overview of the WebCrypto API itself. Next, it presents some differences between the updated SubtleCrypto interface and the older `webkit-` prefixed interface. Some newly-added algorithms are discussed, and finally we demonstrate how to smoothly transition from the `webkit-` prefixed API to the new, standards-compliant API.

Native or Not Native?

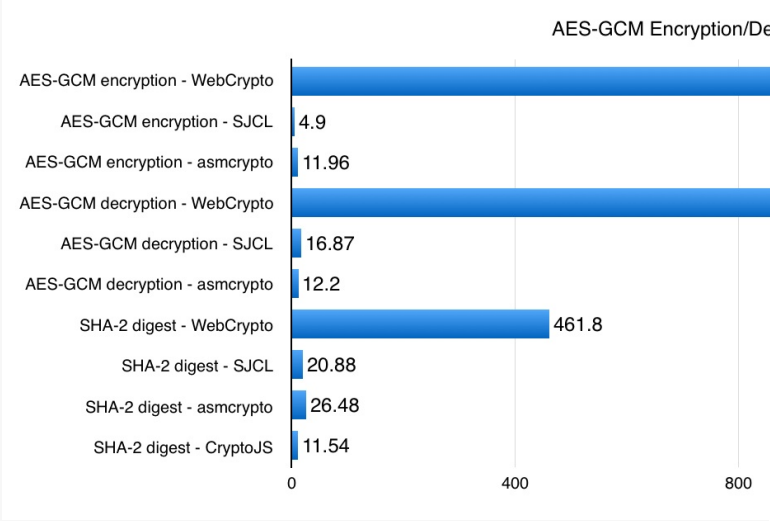
Long before the WebCrypto API was standardized, several JavaScript cryptography libraries were created and have successfully served the open web since. So why bother implementing a web-facing cryptography library built on native APIs? There are several reasons, one of the more important being performance. Numbers tell the truth. We conducted several performance tests to compare our updated WebCrypto API and some famous pure JavaScript implementations.

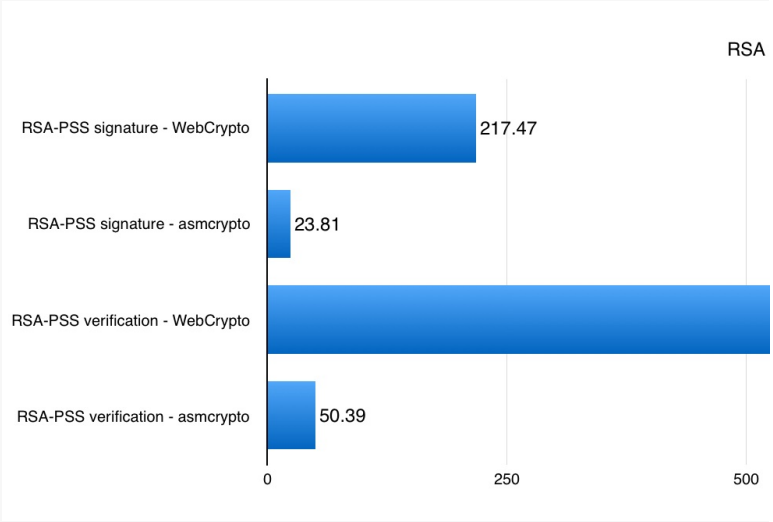
The latest SJCL (1.0.7), asmcrypto.js, and CryptoJS (3.1) were selected for the comparison. The test suite contains:

1. **AES-GCM**: Test encryption/decryption against a 4MB file, repeat certain times and record down the average speed. It uses a 256-bit AES key.
2. **SHA-2**: Hash a 512KB file by SHA-512, repeat certain times and record down the average speed.
3. **RSA**: Test RSA-PSS signature and verification against a 512KB file, repeat certain times and record down the average speed. It uses a 2048-bit key pair and SHA-512 for hashing.

The content under test was carefully selected to reflect the most frequently used day-to-day cryptography operations and paired with appropriate algorithms. The test platform was a MacBook Pro (MacBookPro11,5) with a 2.8 GHz Intel Core i7 running MacOS 10.13 Beta (17A306f) and Safari Technology Preview 35. Some of the pure JavaScript implementations do not support all of the test content, therefore corresponding results were omitted from those results.

Here are the test results.





As you can see, the difference in performance is staggering. This was a surprising result, since most modern JavaScript engines are very efficient. Working with our JavaScriptCore team, we learned that the causes of these pure JavaScript implementations not performing well is that most of them are not actively maintained. Few of them take full advantage of our fast JavaScriptCore engine or modern JavaScript coding practices. Otherwise, the gaps may not be that huge.

Besides superior performance, WebCrypto API also benefits better security models. For example, when developing with pure JavaScript crypto libraries, secret or private keys are often stored in the global JavaScript execution context. It is extremely vulnerable as keys are exposed to any JavaScript resources being loaded and therefore allows XSS attackers be able to steal the keys. WebCrypto API instead protects the secret or private keys by storing them completely outside of the JavaScript execution context. This limits the risk of the private key being exfiltrated and reduces the window of compromise if an attacker gets to execute JavaScript in the victim's browser. What's more, our WebCrypto implementation on macOS/iOS is based on the CommonCrypto routines, which are highly tuned for our hardware platforms, and are regularly audited and reviewed for security and correctness. WebCrypto API is therefore the best way to ensure users enjoy the highest security protection.

Overview of WebCrypto API

The WebCrypto API starts with `crypto` global object:

```
Crypto
{
    subtle: SubtleCrypto,
    ArrayBufferView getRandomValues(ArrayBufferView array)
}
```

Inside, it owns a `subtle` object that is a singleton of the `SubtleCrypto` interface. The

interface is named `subtle` because it warns developers that many of the crypto algorithms have sophisticated usage requirements that must be strictly followed to get the expected algorithmic security guarantees. The `subtle` object is the main entry point for interacting with underlying crypto primitives. The `crypto` global object also has the function `getRandomValues`, which provides a cryptographically strong random number generator (RNG). The WebKit RNG (macOS/iOS) is based on AES-CTR.

The `subtle` object is composed of multiple methods to serve the needs of low-level cryptographic operations:

```
SubtleCrypto
{
    Promise<ArrayBuffer> encrypt(AlgorithmIdentifier algorithm, CryptoKey key, BufferSource data);
    Promise<ArrayBuffer> decrypt(AlgorithmIdentifier algorithm, CryptoKey key, BufferSource data);
    Promise<ArrayBuffer> sign(AlgorithmIdentifier algorithm, CryptoKey key, BufferSource data);
    Promise<boolean> verify(AlgorithmIdentifier algorithm, CryptoKey key, BufferSource signature, B
    Promise<ArrayBuffer> digest(AlgorithmIdentifier algorithm, BufferSource data);
    Promise<CryptoKey or CryptoKeyPair> generateKey(AlgorithmIdentifier algorithm, boolean extract
    Promise<CryptoKey> deriveKey(AlgorithmIdentifier algorithm, CryptoKey baseKey, AlgorithmIdent
    Promise<ArrayBuffer> deriveBits(AlgorithmIdentifier algorithm, CryptoKey baseKey, unsigned lon
    Promise<CryptoKey> importKey(KeyFormat format, (BufferSource or JsonWebKey) keyData, Algo
    Promise<ArrayBuffer> exportKey(KeyFormat format, CryptoKey key);
    Promise<ArrayBuffer> wrapKey(KeyFormat format, CryptoKey key, CryptoKey wrappingKey, Algo
    Promise<CryptoKey> unwrapKey(KeyFormat format, BufferSource wrappedKey, CryptoKey unwra
}
```

As the names of these methods imply, the WebCrypto API supports hashing, signature generation and verification, encryption and decryption, shared secret derivation, and corresponding key materials management. Let's look closer at one of those methods:

```
Promise<ArrayBuffer> encrypt(AlgorithmIdentifier algorithm,
                             CryptoKey key,
                             BufferSource data)
```

All of the functions return a `Promise`, and most of them accept an `AlgorithmIdentifier` parameter. `AlgorithmIdentifier` can be either a string that specifies an algorithm, or a dictionary that contains all the inputs to a specific operation. For example, in order to do an AES-CBC encryption, one has to supply the above `encrypt` method with:

```
var aesCbcParams = {name: "aes-cbc", iv: asciiToUint8Array("jnOw99oOZFLIEPMr")}
```

`CryptoKey` is an abstraction of keying materials in WebCrypto API. Here is an illustration:

```
CryptoKey
{
    type: "secret",
    extractable: true,
    algorithm: { name: "AES-CBC", length: 128 },
    usages: ["decrypt", "encrypt"]
}
```

This code tells us that this key is an extractable (to JavaScript execution context) AES-CBC "secret" (symmetric) key with a length of 128 bits that can be used for both encryption and decryption. The `algorithm` object is a dictionary that characterizes different keying materials, while all the other slots are generic. Bear in mind that `CryptoKey` does not expose the underlying key data directly to web pages. This design of WebCrypto keeps the secret and private key data safely within the browser agent, while allowing web authors to still enjoy the flexibility of working with concrete keys.

Changes to WebKitSubtleCrypto

Those of you that have never heard of `WebKitSubtleCrypto` may skip this section and use `SubtleCrypto` exclusively. This section is aimed at providing compelling reasons for current `WebKitSubtleCrypto` users to switch to our new standards-compliant `SubtleCrypto`.

1. Standards-compliant implementation

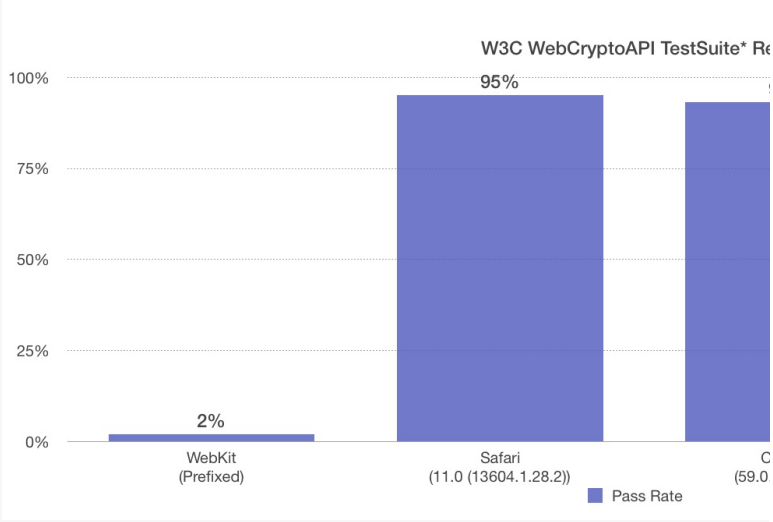
`SubtleCrypto` is a standards-compliant implementation of the current specification, and is completely independent from `WebKitSubtleCrypto`. Here is an example code snippet that demonstrates the differences between the two APIs for importing a JsonWebKey (JWK) format key:

```
var jwkKey = {
    "kty": "oct",
    "alg": "A128CBC",
    "use": "enc",
    "ext": true,
    "k": "YWJjZGVmZ2gxMjM0NTY3OA"
};

// WebKitSubtleCrypto:
// asciiToUint8Array() takes a string and converts it to an Uint8Array object.
var jwkKeyAsArrayBuffer = asciiToUint8Array(JSON.stringify(jwkKey));
crypto.webkitSubtle.importKey("jwk", jwkKeyAsArrayBuffer, null, false, ["encrypt"]).then(function(key)
    console.log("An AES-CBC key is imported via JWK format.");
});

// SubtleCrypto:
crypto.subtle.importKey("jwk", jwkKey, "aes-cbc", false, ["encrypt"]).then(function(key) {
    console.log("An AES-CBC key is imported via JWK format.");
});
```

With the new interface, one no longer has to convert the JSON key to `UInt8Array`. The SubtleCrypto interface is indeed significantly more standards-compliant than our old WebKitSubtleCrypto implementation. Here are the results of running W3C WebCrypto API tests:



This test suite is an improved one based on the most updated web-platform-tests GitHub repository. Pull requests are made for all improvements: #6100, #6101, and #6102.

The new implementation's coverage is around 95% which is 48X higher than our `webkit-` prefixed one! The concrete numbers for all selected parties are: 999 for prefixed WebKit, 46653 for Safari 11, 45709 for Chrome 59, and 18636 for FireFox 54.

2. DER encoding support for importing and exporting asymmetric keys

The WebCrypto API specification supports DER encoding of public keys as SPKI, and of private key as PKCS8. Prior to this, `WebKitSubtleCrypto` only supported the JSON-based JWK format for RSA keys. This is convenient when keys are used on the web because of its structure and human readability. However, when public keys are often exchanged between servers and web browsers, they are usually embedded in certificates in a binary format. Even though some JavaScript frameworks have been written to read the binary format of a certificate and to extract its public key, few of them convert a binary public key into its JWK equivalent. This is why support for SPKI and PKCS8 is useful. Here are code snippets that demonstrate what can be done with the `SubtleCrypto` API:

```
// Import:
// Generated from OpenSSL
// Base64URL.parse() takes a Base64 encoded string and converts it to an Uint8Array object.
var spkiKey = Base64URL.parse("MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAwCjRCtFwv
crypto.subtle.importKey("spki", spkiKey, {name: "RSA-OAEP", hash: "sha-256"}, true, ["encrypt"]).the
    console.log("A RSA-OAEP key is imported via SPKI format.");
});

// Export:
var rsaKeyGenParams = {
    name: "RSA-OAEP",
    modulusLength: 2048,
    publicExponent: new Uint8Array([0x01, 0x00, 0x01]),  // Equivalent to 65537
    hash: "sha-256"
};
crypto.subtle.generateKey(rsaKeyGenParams, true, ["decrypt", "encrypt"]).then(function(keyPair) {
    crypto.subtle.exportKey("spki", keyPair.publicKey).then(function(binary) {
        console.log("A RSA-OAEP key is exported via SPKI format.");
    });
});
```

A live example from a third party to generate public key certificates can be found here.

3. Asynchronously execute time-consuming `SubtleCrypto` methods

In the previous `WebKitSubtleCrypto` implementation, only `generateKey` for RSA executes asynchronously, while all the other operations are synchronous. Even though synchronous operation works well for methods that finish quickly, most crypto methods are time-consuming. Consequently, all time-consuming methods in the new `SubtleCrypto` implementation execute asynchronously:

Method	encrypt	decrypt	sign	verify	digest	generateKey*	deriveKey	deriveBits	in
Asynchronous	✓	✓	✓	✓	✓	✓	✓	✓	

Note that only RSA key pair generation is asynchronous while EC key pair and symmetric key generation are synchronous. Also notice that AES-KW is the only exception where synchronous operations are still done for `wrapKey/unwrapKey`. Normally key size is a few hundred bytes, and therefore it is less time-consuming to encrypt/decrypt such small amount of data. AES-KW is the only algorithm that directly supports `wrapKey/unwrapKey` operations while others are bridged to `encrypt/decrypt` operations. Hence, it becomes the only algorithm that executes `wrapKey/unwrapKey` synchronously. Web developers may treat every `SubtleCrypto` function the same as any other function that returns a promise.

4. Web worker support

Besides making most of the APIs asynchronous, we also support web workers to allow another model of asynchronous execution. Developers can choose which one best suit their needs. Combining these two models, developers now could integrate cryptographic primitives inside their websites without blocking any UI activities. The `SubtleCrypto` object in web workers uses the same semantics as the one in the `Window` object. Here is some example code that uses a web worker to encrypt text:

```
// In Window.
var rawKey = asciiToUint8Array("16 bytes of key!");
crypto.subtle.importKey("raw", rawKey, {name: "aes-cbc", length: 128}, true, ["encrypt", "decrypt"]).
    var worker = new Worker("crypto-worker.js");
    worker.onmessage = function(evt) {
        console.log("Received encrypted data.");
    }
    worker.postMessage(localKey);
});

// In crypto-worker.js.
var plainText = asciiToUint8Array("Hello, World!");
var aesCbcParams = {
```

```
    name: "aes-cbc",
    iv: asciiToUint8Array("jnOw99oOZFLIEPMr"),
}
onmessage = function(key)
{
    crypto.subtle.encrypt(aesCbcParams, key, plainText).then(function(cipherText) {
        postMessage(cipherText);
    });
}
```

A live example is here to demonstrate how asynchronous execution could help to make a more responsive website.

In addition to the four major areas of improvement above, some minor changes that are worth mentioning include:

- `CryptoKey` interface enhancement includes renaming from `Key` to `CryptoKey`, making algorithm and usages slots cacheable, and exposing it to web workers.
- `HmacKeyParams.length` is now bits instead of bytes.
- RSA-OAEP can now import and export keys with SHA-256.
- `CryptoKeyPair` is now a dictionary type.

Newly added cryptographic algorithms

Together with the new `SubtleCrypto` interface, this update also adds support for a number of cryptographic algorithms:

1. AES-CFB: CFB stands for cipher feedback. Unlike CBC, CFB does not require the plain text be padded to the block size of the cipher.
2. AES-CTR: CTR stands for counter mode. CTR is best known for its parallelizability on both encryption and decryption.
3. AES-GCM: GCM stands for Galois/Counter Mode. GCM is an authenticated encryption algorithm designed to provide both data authenticity (integrity) and confidentiality.
4. ECDH: ECDH stands for Elliptic Curve Diffie–Hellman. Elliptic curve cryptography (ECC) is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields. ECC requires smaller keys compared to RSA to provide equivalent security. ECDH is one among many ECC schemes. It allows two parties each of whom owns an ECC key pair to establish a shared secret over an insecure channel.
5. ECDSA: ECDSA stands for Elliptic Curve Digital Signature Algorithm. It is another ECC scheme.
6. HKDF: HKDF stands for HMAC-based Key Derivation Function. It transforms secrets into key, allowing to combine additional non-secret inputs when needed.
7. PBKDF2：PBKDF2 stands for Password-Based Key Derivation Function 2. It takes a password or a passphrase along with a salt value to derive a cryptographic symmetric key.
8. RSA-PSS: PSS stands for Probabilistic Signature Scheme. It is an improved digital signature algorithm for RSA.

This set of new algorithms not only adds new functionality, e.g. key derivation functions, but also benefits developers from higher efficiency and better security by replacing existing ones having the same functionalities. To demonstrate the benefits, sample code snippets written with selected new algorithms are presented in the following. Implementations under these examples are not written with the best practices and therefore are for demonstration only.

Example 1: AES-GCM

Prior, AES-CBC is the only available block cipher for encryption/decryption. Even though it does a great job for protecting data confidentiality, yet it doesn't protect the authenticity (integrity) of the produced ciphers. Hence, it often bundles with HMAC-SHA256 to prevent silent corruptions of the ciphers. Here is the corresponding code snippet:

```
// Assume aesKey and hmacKey are imported before with the same raw key.
var plainText = asciiToUint8Array("Hello, World!");
var aesCbcParams = {
    name: "aes-cbc",
    iv: asciiToUint8Array("jnOw99oOZFLIEPMr"),
}

// Encryption:
// First encrypt the plain text with AES-CBC.
crypto.subtle.encrypt(aesCbcParams, aesKey, plainText).then(function(result) {
    console.log("Plain text is encrypted.");
```

```
    cipherText = result;

    // Then sign the cipher text with HMAC.
    return crypto.subtle.sign("hmac", hmacKey, cipherText);
}).then(function(result) {
    console.log("Cipher text is signed.");
    signature = result;

    // Finally produce the final result by concatenating cipher text and signature.
    finalResult = new Uint8Array(cipherText.byteLength + signature.byteLength);
    finalResult.set(new Uint8Array(cipherText));
    finalResult.set(new Uint8Array(signature), cipherText.byteLength);
    console.log("Final result is produced.");
});

// Decryption:
// First decode the final result from the encryption step.
var position = finalResult.length - 32; // SHA-256 length
signature = finalResult.slice(position);
cipherText = finalResult.slice(0, position);

// Then verify the cipher text.
crypto.subtle.verify("hmac", hmacKey, signature, cipherText).then(function(result) {
    if (result) {
        console.log("Cipher text is verified.");

        // Finally decrypt the cipher text.
        return crypto.subtle.decrypt(aesCbcParams, aesKey, cipherText);
    } else
        return Promise.reject();
}).then(function(result) {
    console.log("Cipher text is decrypted.");
    decryptedText = result;
}, function() {
    // Error handling codes ...
});
```

So far, the codes are a bit complex with AES-CBC because the extra overhead of HMAC. However, it is much simpler to achieve the same authenticated encryption effect by using AES-GCM as it bundles authentication and encryption together within one single step. Here is the corresponding code snippet:

```
// Assume aesKey are imported/generated before, and the same plain text is used.
var aesGcmParams = {
    name: "aes-gcm",
    iv: asciiToUint8Array("jnOw99oOZFLIEPMr"),
}

// Encryption:
crypto.subtle.encrypt(aesGcmParams, key, plainText).then(function(result) {
    console.log("Plain text is encrypted.");
    cipherText = result; // It contains both the cipherText and the authentication data.
});

// Decryption:
crypto.subtle.decrypt(aesGcmParams, key, cipherText).then(function(result) {
    console.log("Cipher text is decrypted.");
    decryptedText = result;
}, function(error) {
    // If any violation of the cipher text is detected, the operation will be rejected.
    // Error handling codes ...
});
```

It is just that simple to use AES-GCM. This simplicity will definitely improve developers' efficiency. A live example can also be found here to demonstrate how AES-GCM can prevent silent corruption during decrypting corrupted ciphers.

Example 2: ECDH(E)

Block ciphers alone are not sufficient to protect data confidentiality because secret (symmetric) keys need to be shared securely as well. Before this change, only RSA encryption was available for tackling this task. That is to encrypt the shared secret keys and then exchange the ciphers to prevent MITM attacks. This method is not entirely secure as

perfect forward secrecy (PFS) is difficult to guarantee. PFS requires session keys, the RSA key pair in this case, to be destroyed once a session is completed, i.e. after a secret key is successfully shared. So the shared secret key can never be recovered even if the MITM attackers are able to record down the exchanged cipher and access the recipient in the future. RSA key pairs are very hard to generate, and therefore maintaining PFS is really a challenge for RSA secret key exchange.

However, maintaining PFS is a piece of cake for ECDH simply because EC key pairs are easy to generate. In average, it takes about 170 ms to generate a RSA-2048 key pair on the same test environment shown in the first section. On the contrary, it only takes about 2 ms to generate a P-256 EC key pair which can provide comparable security to a RSA-3072 alternative. ECDH works in the way that the involved two parties exchange their public keys first and then compute a point multiplication by using the acquired public keys and their own private keys, of which the result is the shared secret. ECDH with PFS is referred as Ephemeral ECDH (ECDHE). Ephemeral merely means that session keys are transient in this protocol. Since the EC key pairs involved with ECDH are transient, they cannot be used to confirm the identities of the involved two parties. Hence, other permanent asymmetric key pairs are needed for authentication. In general, RSA is used as it is widely supported by common public key infrastructures (PKI). To demonstrate how ECDHE works, the following code snippet is shared:

```
// Assuming Bob and Alice are the two parties. Here we only show codes for Bob's.
// Alice's should be similar.
// Also assumes that permanent RSA keys are obtained before, i.e. bobRsaPrivateKey and aliceRsaPu
// Prepare to send the hello message which includes Bob's public EC key and its signature to Alice:
// Step 1: Generate a transient EC key pair.
crypto.subtle.generateKey({ name: "ECDH", namedCurve: "P-256" }, extractable, ["deriveKey"]).then
    console.log("EC key pair is generated.");
    bobEcKeyPair = result;

    // Step 2: Sign the EC public key for authentication.
    return crypto.subtle.exportKey("raw", bobEcKeyPair.publicKey);
}).then(function(result) {
    console.log("EC public key is exported.");
    rawEcPublicKey = result;

    return crypto.subtle.sign({ name: "RSA-PSS", saltLength: 16 }, bobRsaPrivateKey, rawEcPublicKey
}).then(function(result) {
    console.log("Raw EC public key is signed.");
    signature = result;

    // Step 3: Exchange the EC public key together with the signature. We simplify the final result as
    // a concatenation of the raw format EC public key and its signature.
    finalResult = new Uint8Array(rawEcPublicKey.byteLength + signature.byteLength);
    finalResult.set(new Uint8Array(rawEcPublicKey));
    finalResult.set(new Uint8Array(signature), rawEcPublicKey.byteLength);
    console.log("Final result is produced.");

    // Send the message to Alice.
    // ...
});

// After receiving Alice's hello message:
// Step 1: Decode the counterpart from Alice.
var position = finalResult.length - 256; // RSA-2048
signature = finalResult.slice(position);
rawEcPublicKey = finalResult.slice(0, position);

// Step 2: Verify Alice's signature and her EC public key.
crypto.subtle.verify({ name: "RSA-PSS", saltLength: 16 }, aliceRsaPublicKey, signature, rawEcPublicK
    if (result) {
        console.log("Alice's public key is verified.");

        return crypto.subtle.importKey("raw", rawEcPublicKey, { name: "ECDH", namedCurve: "P-256" }
    } else
        return Promise.reject();
}).then(function(result) {
    console.log("Alice's public key is imported.");
    aliceEcPublicKey = result;

    // Step 3: Compute the shared AES-GCM secret key.
    return crypto.subtle.deriveKey({ name: "ECDH", public: aliceEcPublicKey }, bobEcKeyPair.privateKe
}).then(function(result) {
    console.log("Shared AES secret key is computed.");
```

```
    aesKey = result;

    console.log(aesKey);

    // Step 4: Delete the transient EC key pair.
    bobEcKeyPair = null;
    console.log("EC key pair is deleted.");
});
```

In the above example, we omit the way how information, i.e. public keys and their corresponding parameters, is exchanged to focus on parts that WebCrypto API is involved. The ease to implement ECDHE will definitely improve the security level of secret key exchanges. Also, a live example to tell the differences between RSA secret key exchange and ECDH is included here.

Example 3: PBKDF2

The ability to derive a cryptographically secret key from existing secrets such as password is new. PBKDF2 is one of the newly added algorithms that can serve this purpose. The derived secret key from PBKDF2 not only can be used in the subsequent cryptographically operations, but also itself is a strong password hash given it is salted. The following code snippet demonstrates how to derive a strong password hash from a simple password:

```
var password = asciiToUint8Array("123456789");
var salt = asciiToUint8Array("jnOw99oOZFLIEPMr");

crypto.subtle.importKey("raw", password, "PBKDF2", false, ["deriveBits"]).then(function(baseKey) {
    return crypto.subtle.deriveBits({name: "PBKDF2", salt: salt, iterations: 100000, hash: "sha-256"},
}).then(function(result) {
    console.log("Hash is derived!")
    derivedHash = result;
});
```

A live example can be found here.

The above examples are just a tip of capabilities of WebCrypto API. Here is a table listing all algorithms that WebKit currently supports, and corresponding permitted operations of each algorithm.

Algorithm name	encrypt	decrypt	sign	verify	digest	generateKey	deriveKey	deriveBit
RSAES-PKCS1-v1_5***	✓	✓				✓		
RSASSA-PKCS1-v1_5			✓	✓		✓		
RSA-PSS			✓	✓		✓		
RSA-OAEP	✓	✓				✓		
ECDSA*			✓	✓		✓		
ECDH*						✓	✓	✓
AES-CFB	✓	✓				✓		
AES-CTR	✓	✓				✓		
AES-CBC	✓	✓				✓		
AES-GCM	✓	✓				✓		
AES-KW						✓		
HMAC			✓	✓		✓		

SHA-1***	✓	
SHA-224	✓	
SHA-256	✓	
SHA-384	✓	
SHA-512	✓	
HKDF	✓	✓
PBKDF2	✓	✓

** WebKit doesn't support P-521 yet, see bug 169231.*

*** WebKit doesn't check or produce any hash information from or to DER key data, see bug 165436, and bug 165437.*

**** RSAES-PKCS1-v1_5 and SHA-1 should be avoided for security reasons.*

Transition to the New SubtleCrypto Interface

This section covers some common mistakes that web developers have made when they have tried to maintain compatibility to both `WebKitSubtleCrypto` and `SubtleCrypto`, and then we present recommended fixes to those mistakes. Finally, we summarize those fixes into a de facto rule to maintain compatibility.

Example 1:

```
// Bad code:
var subtleObject = null;
if ("subtle" in self.crypto)
    subtleObject = self.crypto.subtle;
if ("webkitSubtle" in self.crypto)
    subtleObject = self.crypto.webkitSubtle;
```

This example wrongly prioritizes `window.crypto.webkitSubtle` over `window.crypto.subtle`. Therefore, it will overwrite the subtleObject even that the `subtle` object actually exists. A quick fix for it is to prioritizes `window.crypto.subtle` over `window.crypto.webkitSubtle`.

```
// Fix:
var subtleObject = null;
if ("webkitSubtle" in self.crypto)
    subtleObject = self.crypto.webkitSubtle;
if ("subtle" in self.crypto)
    subtleObject = self.crypto.subtle;
```

Example 2:

```
// Bad code:
(window.agcrypto = window.crypto) && !window.crypto.subtle && window.crypto.webkitSubtle && (
var h = window.crypto.webkitSubtle ? a.utils.json2ab(c.jwkKey) : c.jwkKey;
agcrypto.subtle.importKey("jwk", h, g, !0, ["encrypt"]).then(function(a) {
    ...
});
```

This example incorrectly pairs `window.agcrypto` and the latter `jwkKey`. The first line prioritizes `window.crypto.subtle` over `window.crypto.webkitSubtle`, which is correct. However, the second line prioritizes `window.crypto.webkitSubtle` over `window.crypto.subtle` again.

```
// Fix:
(window.agcrypto = window.crypto) && !window.crypto.subtle && window.crypto.webkitSubtle && (
var h = window.crypto.subtle ? c.jwkKey : a.utils.json2ab(c.jwkKey);
agcrypto.subtle.importKey("jwk", h, g, !0, ["encrypt"]).then(function(a) {
    ...
});
```

A deeper analysis of these examples reveals they both assume `window.crypto.subtle` and `window.crypto.webkitSubtle` cannot coexist and therefore wrongly prioritize one over the other. In summary, developers should be aware of the coexistence of these two interfaces and should always prioritize `window.crypto.subtle` over `window.crypto.webkitSubtle`.

Feedback

In this blog post, we reviewed WebKit's update to the WebCrypto API implementation which is available on macOS, iOS, and GTK+. We hope you enjoy it. You can try out all of these improvements in the latest Safari Technology Preview. Let us know how they work for you by sending feedback on Twitter (@webkit, @alanwaketan, @jonathandavis) or by filing a bug. ∎

Older Post
Release Notes for Safari Technology Preview 35

Newer Post
Adobe Announces Flash Distribution and Updates to End

