Cryptographic operations in the browser

Inventive Designers / Nick Van den Bleeken / @nvbleek
About me

• R&D Manager at Inventive Designers
• Member of the W3C Web Cryptography Working Group
• @nvdbleek
• https://github.com/nvdbleek
Agenda

- Why do we need cryptography on the web?
- What is the Web Cryptography API?
- What can you do with it?
- Who is implementing it?
- Demo (Chrome, IE, Firefox, Safari)
- What's next?
Why do we need cryptography on the web?

- Easy and secure authentication
- Signing/verifying messages or documents
- Encrypting/decrypting messages or documents
- ...

.
What JS Crypto implementations are out there?

- Stanford Javascript Crypto Library (SJCL)
- CryptoJS
- JSBN
- OpenPGP.js
- Closure Library (crypt)
- DojoX (Crypto module)
- ...
What is the problem with JS Crypto implementations?

- Hard to protect against timing attacks (Optimizations in JS engine change)
- `Math.random()` is not secure
- It is hard to get it correct (need for cryptographers due to complex specifications)
- Performance is hard due to JS limitations (e.g.: number is 64 bit float)
What other options are there?

- Java Browser plugin
  - [Java security vulnerabilities](#) (and disabled by default in some browsers)
  - Department of homeland security recommends turning Java off in your browser
  - Not available on mobile
- Native extensions
  - Separate install
  - Browser specific (some browsers are dropping support)
  - Not available on mobile
The Web Cryptography API

- JavaScript API for web developers
- Natively implemented in the browser
- Provide low level crypto operations to web applications
- Defined by web Cryptography W3C Working Group
  (Google, Mozilla, Microsoft, Apple, Netflix, Inventive Designers, ...)
- Available on desktop, mobile, televisions, ...
What can you do with it?
Real Life Use Cases
Banking transactions
Use case

John opens a bank account and wants to:

- Authenticate
- Sign documents and transactions
- Send documents that only the bank can read
- Receive data and documents that only he can read
Setup keys

1. Bank provides temporary verification code (SMS, Token Generator, ...)
2. Bank asks if used browser is not at a kiosk
3. Generates a public key/private key pair
4. Store key pair in client-side storage
5. Send our public key to Bank
6. Accept public key of Bank
Setup keys (Code generate keys)

```javascript
// Algorithm Object
var algorithmKeyGen = {
    name: "RSASSA-PKCS1-v1_5",

    // RsaKeyGenParams
    modulusLength: 2048,
    publicExponent: new Uint8Array([0x01, 0x00, 0x01]), // Equivalent to 65537
};

window.crypto.subtle.generateKey(algorithmKeyGen, /* extractable */ false, ["sign"],
    function(keyPair) {
        /* Store the key pair in IndexedDB or localStorage */
        /* Send the public key to the bank */
    },
    console.error.bind(console,
        "Unable to generate keys -- call your Banker or retry later."));```
Setup keys (Code import key bank)

```javascript
// jwkKeyOfBank is the public key JSON Web Key embedded in a JWT
var jwkKeyOfBank = retrievePublicKeyOfBank();
var jwkKeyObject = JSON.parse(jwkKeyOfBank);

window.crypto.subtle.importKey(  
  "jwk",
  jwkKeyObject,
  "RSAES-PKCS1-v1_5",
  /* extractable */ "true",
  ["encrypt", "verify"]
)

.then(  
  function(pubKey) {  
    /* Store the public key of the bank in IndexedDB or localStorage */
  }, console.error.bind(console,  
    "Unable to import key -- call your Banker or retry later."));
```
Authenticate using keys

Subsequent access to the bank website -- always over TLS -- may use keypairs to authenticate.

1. Bank sends message that only you can decipher (encrypted with your public key) and is signed by the bank
2. Verify signature (with public key of bank) and decipher message with your private key
3. Sign deciphered message with your private key
4. Send back signed deciphered message to bank
5. Bank can verify the signature with your public key
Authenticate using keys (Code)

```javascript
// This could be in a parsable JWT format such that portions with signature 
// and message are easily understood
var cat = "qANQR1DBw04Dk2uPpEjcJT8QD/0VCfFK2XDM5Cg4iTRwmXrB+Pp8SMK5x09WkYgc...";

var data = createArrayBufferView(cat);
var signature = extractSignature(data);
var pMessage = extractMessage(data);

var pubGBKeySign = retrievePublicKeyOfBankFromLocalStorage();
var mRSARFC3447 = {
    name: "RSASSA-PKCS1-v1_5",
    hash: "SHA-256"
};
```
Authenticate using keys (Code)

```javascript
window.crypto.subtle.verify(mRSA RFC3447, pubGBKeySign, signature)

  .then(
    function(verified) { // Signature from bank is verified
      prvKeyEncrypt = retrievePrivateKeyFromLocalStorage();
      return window.crypto.subtle.decrypt("RSAES-PKCS1-v1_5", prvKeyEncrypt, pMessage);
    },
    console.log.bind(console, "Verification Error! Contact Bank."))

  .then(
    function (decrypted) { // Message from bank is decrypted
      prvKeySign = retrievePrivateKeyFromLocalStorage();
      return window.crypto.subtle.sign(mRSA RFC3447, prvKeySign, decrypted);
    },
    console.error.bind(console, "Decryption Error! Contact Bank."))

  .then(
    function(signed) { /* Send signed message back to bank */ },
    console.error.bind(console, "Signing message failed! Contact Bank."));
```
Authenticate using keys (Notes)

As long as you use the same browser, within the validity period of the keys that were generated, you can use them as credentials by signing, verifying, encrypting and decrypting bits of data sent by the bank.
Additional operations using generated keys

- Authorize payments [SIGN]
- Sign Forms [SIGN]
- Submit documents that only the bank can read [SIGN | GENERATE-SYM | ENCRYPT-SYM | WRAP-ASYM]
- Receive encrypted documents from bank that only you can read, with the assurance that they have come from that bank [VERIFY | UNWRAP-ASYM | DECRYPT-SYM]
Documents in the cloud

1. Authenticate
2. Store encrypted documents
3. Retrieve encrypted documents and decrypt them
Documents in the cloud

- Authenticate (Could be the same as the banking use case)
- Store encrypted documents [HMAC | SHA-1 | DERIVE-SYM | ENCRYPT-SYM]
- Retrieve encrypted documents and decrypt them [HMAC | SHA-1 | DERIVE-SYM | DECRYPT-SYM]
Generate encryption key (PBKDF2)

```javascript
var mySalt = retrieveSaltFromServer(); // Salt generated by signup or change password
var iterations = retrieveNumberOfIterationsFromServer();

var algorithm = {
  name: "PBKDF2",
  salt: mySalt,
  iterations: iterations,
  password: myPassword,
  prf: { name: "HMAC", hash: { name: "SHA-1" } } } // pseudorandom function

window.crypto.subtle.deriveKey( algorithm,
  null,
  { name: "AES-GCM", length: 512 },
  false,
  ["encrypt", "decrypt"]

  .then(
    function(key) {
      // key used to encrypt and decrypt documents
      }, console.error.bind(console, "Unable to derive key."));
```
Off The Record Real Time Messaging
Use case

- John and Joe wish to have "Off The Record" chat in real time
- Primarily using text, but also share digital data (photographs)
- Server can not log their conversation
Off The Record Real Time Messaging

1. Generate asymmetric keys for John and Joe respectively, such that both get public and private keys. [KEYGEN-ASYM]
2. Engage in a key exchange, so that John and Joe get each other's public keys (e.g.: Diffie-Hellman key exchange). [RANDOM | KEYEX | KEYEX-DH]
3. John or Joe may now compose a message to each other. Each message exchange involves authentication, message authentication codes, further key derivation, and further key exchanges. [SIGN | VERIFY | MAC | RANDOM | DERIVE | KEYEX | KEYEX-DH]
Code Sanctity and Bandwidth Saver
Use case

Optimize website performance by:

- Storing JavaScript libraries that are served from a CDN in localStorage
- Prevent server roundtrips to the CDN
- Support library upgrades
Code Sanctity and Bandwidth Saver

1. Get Hash of latest JavaScript code from server
2. Calculate Hash of JavaScript code in localStorage [DIGEST]
3. If hashes match use local version, otherwise download code
Where is the high level API?

- Hard to define one high level API (there are so many different use cases).
- Based on NaCl, KeyCzar, SJCL or ... ?
- People already use frameworks like JQuery, Dojo, YUI and Prototype, which could provide their own high level APIs
Why is it implemented natively?

- Protect against timing attacks (Optimizations in JS engine change)
- Key material can be stored more securely
- Secure random number generator
- Correctness is hard
- Performance (e.g.: numbers are 64-bit floating points)
Who is implementing it?

- **Web Cryptography API**
  - Microsoft IE 11
  - Chromium (Flags)
  - Mozilla under development
  - Polyfills
- **WebCrypto Key Discovery**
  - Netflix
Demo

You can find the source of the demo at
https://github.com/nvdblek/web-crypto-samples
What is the implementation status?

- IE 11 and polycrypt implement an old version that still uses DOM events
- Chrome implements latest version but misses `importKey()`, `exportKey()` and some algorithms (user has to enable it)
- Firefox and Safari have no public implementation yet
What’s next?

- Get Spec to Last Call and CR *(we welcome your feedback)*
- Streaming API *(Waiting on the new Streams API)*
- Access to smart-cards and other secure elements?
References

- Slides at http://nick.bleeken.eu
- Demo at https://github.com/nvdbleek/web-crypto-samples
- Spec at http://www.w3.org/TR/WebCryptoAPI/
Thank You!

Q&A