Halan Pinheiro
Posted on Sep 17, 2019 • Updated on Oct 12, 2019

4 ways of Symmetric Cryptography in JavaScript / How to AES with JavaScript

#cryptography #javascript #security

For the most part, secure systems of the Internet just use HTTPS as a protocol (HTTP over SSL/TLS), so all data sent from the browser to the server, including paths will be encrypted up to the server side, where it will be decrypted. Also, all data coming from the server side will be encrypted and decrypted on the browser. These kinds of systems protect us against interception between the connection and this use to be enough.

But, imagine that you can’t store plain text on the database. You want to encrypt from the browser before sending because you don’t want to touch or be responsible for the plain text. Or, just to decrypt a file during uploading before sending, or even to encrypt or decrypt an email on the browser. There are countless cases for this kind of encryption.

Here I want to cover how to use the most common symmetric cryptographic algorithm using JavaScript on the browser or even on the server side with Node.js. I can just write about asymmetric cryptography later, but this current article is already big enough.

4 platforms

Currently there are at least 4 important platforms for building cryptographic systems using JavaScript.

- A native Node implementation since version 0.10.x around 2015 and updated up to the latest versions (check the latest documentation: https://nodejs.org/api/crypto.html#crypto_crypto);
- A native implementation of an API called Web Cryptography API recommended by W3C since 2012 up until today (check the latest recommendation from 2017: https://www.w3.org/TR/WebCryptoAPI/) and already supported by all browsers https://caniuse.com/#feat=cryptography (you can also check the implementation details in your browser here https://diafygi.github.io/webcrypto-examples/). This is the

recommended solution so far and it solves some classic issues when using JavaScript to deal with cryptosystem;
- A very nice and complete implementation in pure JavaScript first released in 2009! It was abandoned in 2013 and readopted 4 months later. It’s called CryptoJS and it’s still used by ~92k projects on GitHub;
- And there is also a very robust and modern implementation in pure JavaScript called Forge. First released in 2013 and still being updated today, it’s used by 1965k projects on GitHub!
- Also there is a large list of crypto libraries in JavaScript maintained on that gist: https://gist.github.com/jo/8619441; and that page: http://cryptojs.altervista.org/.

In general cryptography and JavaScript is a very recent subject in terms of releases, mainly when compared with other languages and their openSSL wrapper normally built in their standard libraries. I found a great summary of all main discussions related to JavaScript Cryptography for the last 10 years, it's worthy taking a look at: http://blog.kotowicz.net/2014/07/js-crypto-goto-fail.html.

When dealing with cryptosystems it's extremely important to know exactly what you are doing and to know the most recent vulnerabilities and recommendations. Cryptographic systems are like a chain, the strength of the system is always the strength of the weakest part.

In this article, I'l demonstrate how to use each of these tools in comparison, explaining some concepts of symmetric cryptography with JavaScript.

Dealing with data, not text

When dealing with cryptography, we’re dealing with data, not text. Eventually that data must be transmitted through text-only fields, so it needs to be represented as text too. A UTF-8 character is made of 1 to 4 bytes, and there is also a huge bunch of bytes without representation on UTF-8 (like control characters), so UTF-8 is not efficient to represent data. Hexadecimal is the most readable way to handle data but it’s convenient for sharing, because it uses 2 characters per byte! Base64 is the best way to share data as characters so far.

Let’s take a look on how to navigate through data representation using JavaScript with Node.js tooling, the browser Forge and CryptoJS.

Node.js has a nice interface to handle these formats, it’s called Buffer:

```
Buffer.from('hello world')
// <Buffer 68 65 6c 6c 6f 20 77 6f 72 6c 64>

Buffer.from('hello world').toString('hex')
// '68656c6c6f20776f726c64'

Buffer.from('hello world').toString('base64')
// 'aGVsbG8gd29ybGQ='

Buffer.from('aGVsbG8gd29ybGQ=', 'base64').toString()
// 'hello world'

Buffer.from('68656c6c6f20776f726c64', 'hex').toString()
// 'hello world'

[...Buffer.from('hello world')]
// [ 104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100 ]
```

On the browser side we have TextEncoder to transit to and from text formats, and we have the functions atob and btoa to transit to and from Base64. To handle hexadecimals unfortunately we have to map it rudimentary with a little help of toString and parseInt:

```
new TextEncoder().encode('hello world')
// Uint8Array(11) [104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100]

new TextDecoder().decode(new Uint8Array([104, 101, 108, 108, 111, 32, 119, 111, 114, 108,
// "hello world"

[...(new TextEncoder().encode('hello world'))]
  .map(b => b.toString(16).padStart(2, "0")).join('')
// "68656c6c6f20776f726c64"

"68656c6c6f20776f726c64".match(/.{1,2}/g)
  .map(e => String.fromCharCode(parseInt(e, 16))).join('')
// 'hello world'

btoa('hello world')
// "aGVsbG8gd29ybGQ="

atob('aGVsbG8gd29ybGQ=')
// "hello world"
```

CryptoJS uses an interface very similar to Node.js' Buffer. It's pretty easy to transit between all representations. At the end CryptoJS uses an internal representation for dealing with an array of words (32 bits):

```
var CryptoJS = require('crypto-js')

CryptoJS.enc.Utf8.parse('hello world')
// { words: [ 1751477356, 1864398703, 1919706112 ], sigBytes: 11 }

CryptoJS.enc.Utf8.parse('hello world').toString()
// '68656c6c6f20776f726c64'

CryptoJS.enc.Utf8.parse('hello world').toString(CryptoJS.enc.Base64)
// 'aGVsbG8gd29ybGQ='

CryptoJS.enc.Base64.parse('aGVsbG8gd29ybGQ=').toString(CryptoJS.enc.Utf8)
// 'hello world'

CryptoJS.enc.Hex.parse('68656c6c6f20776f726c64').toString(CryptoJS.enc.Utf8)
// 'hello world'
```

Forge uses the native Uint8Array to represent the data and it's pretty simple to transit between the formats as well:

```
var forge = require('node-forge')

forge.util.text.utf8.encode('hello world')
// Uint8Array [ 104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100 ]

forge.util.binary.hex.encode('hello world')
// '68656c6c6f20776f726c64'

forge.util.binary.base64.encode(new Uint8Array([ 104, 101, 108, 108, 111, 32, 119, 111,
// aGVsbG8gd29ybGQ=

forge.util.binary.base64.decode('aGVsbG8gd29ybGQ=')
// Uint8Array [ 104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100 ]

forge.util.binary.hex.decode('68656c6c6f20776f726c64')
// Uint8Array [ 104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100 ]
```

As we can see, on the browser without any help from tools it’s not so trivial, mainly when we want to convert to or from hexadecimals. By the way, when dealing with data it’s important to have in mind how to convert bases easily and understand which format is required at each step. We’ll use these concepts throughout this article.

Password is not a key

Looking through the open issues on https://github.com/brix/crypto-js I had found several people with very similar questions about symmetric encryption and how to handle the cryptography elements. Actually those questions have strongly inspired me to write this article. I want to start explaining what are those cryptography elements and what kind of care we have to take with them and how to use them on our systems. Specifically the confusion about key and password.

All cryptographic systems have at least one key. Symmetric encryption uses the same key to encrypt and decrypt, and asymmetric encryption uses two keys, one to encrypt and another to decrypt. There are also authentication systems based on key, where using a key we can ensure the authenticity of a chunk of data. Hash algorithms are very important pieces in cryptographic systems and they don’t use keys (despite them being used to compose systems that use keys, see the next section).

A length of a key is not about character count, it’s about bits, always. All cryptographic keys have a series of bits that do not necessarily correspond with characters, meanwhile password length is about characters and normally passwords are made from characters. Cryptographic systems use very strict lengths of keys, because the length of keys interacts directly with the implementation of the algorithm, increasing or decreasing rounds, steps or even changing the length of blocks. Passwords normally have minimum and maximum sizes and that is just related with storage fields or brute-force concerns, because passwords are normally used to feed a hash algorithm and act completely different than a cryptographic key.

Hashing data

Hashing algorithms are functions that transform a chunk of data into a pre-sized chunk of non predictable data. Once hashed, the content can never be reverted to the original. Also, hash algorithms must have a collision resistance, in a way that must be impracticable to find two matching contents.

The first widely used hashing algorithms were the MD (Message Digest), it was replaced by MD2, MD3, MD4 and finally MD5, which was first broken at the beginning of this century (here is a demonstration of that weakness: https://www.mscs.dal.ca/~selinger/md5collision/). Then the SHA1 (Secure Hash Algorithm) was created based on MD4, and was broken too (here you can check some vulnerabilities: https://shattered.io/). Currently we use SHA2, which is a family of algorithms able to produce hashes of 224, 256, 384 or 512 bits. All the most important cryptographic systems today work using the security of SHA2!

Hash functions are used in almost all crypto systems. Also there are some uses which are not related with cryptography, for example: git uses SHA1 over the parameters and body of one commit to act as a kind of commit reference. Bitcoin uses SHA2 in 256 mode to hash the entire block of transactions twice appending a nonce (an arbitrary data) in order to ensure a proof of work. When storing passwords within a database, it is a must to store the password hashed and not as plain text.

The most common attack against hashes is rainbow tables. They are pre-computed tables of values and corresponding hashed results. For example, try to type this hash `8BB0CF6EB9B17D0F7D22B456F121257DC1254E1F01665370476383EA776DF414` within this hash table: https://md5decrypt.net/Sha256. We get the answer in 0.1 seconds! The defense consists in appending a chunk of random data at the end of the content and hashing it together.

There are two main techniques to protect against rainbow tables: salt and pepper. While salt is a non secret random data appended to original content, pepper is random data appended to original content also, but in this case, the data is secret. Salt must be unique for each hash and is normally stored together with the content because it isn't a secret, while pepper could be reused in the same application, but it needs to be stored outside of the database where we put the salts and hash results. By adding a pepper, brute force will be impracticable since the pepper data isn't known.

All 4 platforms mentioned in this article implement the most relevant hashing functions: SHA1 and SHA2 of all possible lengths. MD5 was never supported by web crypto because of its vulnerabilities.

From password to key!

Usually we use passwords to generate keys, that operation is called KDF (Key Derivation Function). Basically a password passes through some hash algorithms or some symmetric encryption repeatedly.

Before I talk about KDF functions, let me introduce another concept: MAC (Message Authentication Code). Basically it's a code appended to a content acting as a proof of the content's authenticity. HMAC is Hash-based Message Authentication Code. It uses a primary hashing function internally, normally SHA1, and in the process it'll hash the password and a key in a very specific way separately and together. That way, knowing the key we can calculate the HMAC of a message and just compare with a given MAC, this is enough to prove the integrity and authenticity of the content. We'll use HMAC soon, but not with this original purpose, we'll use it in order to generate some bytes from a given password and a salt.

One of the most commonly used and secure KDF algorithms today is PBKDF2 (Password-Based Key Derivation Function 2, described and specified by RFC-8018: https://tools.ietf.org/html/rfc8018#section-5.2), it can increase significantly their strength just by increasing the iterations of hashing. Normally it uses HMAC to hash, using the password as a content and the salt as a key. The iterations are the times that each block will pass through the hash (HMAC) before outputting and beginning to hash the next block in the chain and hash several iterations again until we derive sufficient blocks. This way PBKDF2 can generate any amount of data apparently random but reproducible once you know the password and the salt.

Let's generate a key of 256 length using Node.js:

```
var crypto = require('crypto');
derivedKey = crypto.pbkdf2Sync('my password', 'a salt', 1000, 256/8, 'sha1');
console.log(derivedKey.toString('hex'));
// 8925b9320d0fd85e75b6aa2b2f4e8ecab3c6301e0e2b7bd850a700523749fbe4
```

And CryptoJS:

```
var CryptoJS = require('crypto-js');
CryptoJS.PBKDF2('my password', 'a salt', { keySize: 256/32, iterations: 1000 }).toString(
// 8925b9320d0fd85e75b6aa2b2f4e8ecab3c6301e0e2b7bd850a700523749fbe4
```

With Forge:

```
var forge = require('node-forge');

forge.util.binary.hex.encode(forge.pkcs5.pbkdf2('my password', 'a salt', 1000, 256/8))
// '8925b9320d0fd85e75b6aa2b2f4e8ecab3c6301e0e2b7bd850a700523749fbe4'
```

Let’s try it using webcrypto on the browser:

```
// firstly we need to importKey
window.crypto.subtle.importKey(
    //the format that we are input
    "raw",
    //the input in the properly format
    new TextEncoder().encode("my password"),
    //the kind of key (in that case it's a password to derive a key!)
    {name: "PBKDF2"},
    //if I permit that this material could be exported
    false,
    //what I permit to be processed against that (password to derive a) key
    ["deriveBits", "deriveKey"]
  // the derive key process
  ).then(keyMaterial => window.crypto.subtle.deriveKey(
    {
      "name": "PBKDF2",
      salt: new TextEncoder().encode("a salt"),
      "iterations": 1000,
      "hash": "SHA-1"
    },
    // it should be an object of CryptoKey type
    keyMaterial,
    // which kind of algorithm I permit to be used with that key
    { "name": "AES-CBC", "length": 256},
    // is that exportable?
    true,
    // what is allowed to do with that key
    [ "encrypt", "decrypt" ]
  )
// exporting...
).then(key => crypto.subtle.exportKey("raw", key)
).then(key => console.log(
// finally we have a ArrayBuffer representing that key!
  [...(new Uint8Array(key))]
    .map(b => b.toString(16).padStart(2, "0"))
    .join("")
));
//8925b9320d0fd85e75b6aa2b2f4e8ecab3c6301e0e2b7bd850a700523749fbe4
```

As you can see, when using webcrypto directly on the browser there is a bunch of concerns and permissions involving the key and what it can do. It is important to protect the keys, but that’s not user-friendly.

These information are safe to share:

- salt
- interactions
- key length
- hashing algorithm

Increasing the interactions will increase how many basic hashes the algorithm has to do, considering HMAC, each interaction will hashing at least 2 SHA1 (or whatever you have set up). That can make the process slow, it has to be slow enough to be ok to run one or two times, but very hard to brute-force, try not to freeze your browser haha!

A good salt must be chosen randomly, we can do it on the 4 platforms as well:

Node.js:

```
const crypto = require('crypto');
crypto.randomBytes(8);
```

CryptoJS:

```
const CryptoJS = require('crypto-js');
CryptoJS.lib.WordArray.random(8);
```

Forge:

```
const forge = require('node-forge');
forge.random.getBytesSync(8);
```

WebCrypto (browser):

```
window.crypto.getRandomValues(new Uint8Array(8));
```

What is an Operation Mode?

The most used symmetric cryptography algorithm today is AES (Advanced Encryption Standard). AES is a cipher block system able to use 128, 192 and 256 key length where that key operates over blocks of 128 bits of plain text to generate 128 bits of encrypted text.

AES is used pretty much everywhere. To protect ebooks bought on Amazon, encrypting connections through SSL, protecting session cookies stored in your browser, encrypting the data on your mobile phone... everywhere!

When using a cipher block system as AES, we should pad the plain text in a way that the padding could be removed from the plain text when decrypted. The most usual padding is the PKSC#5/PKSC#7 (also published as RFC-8018 https://tools.ietf.org/html/rfc8018). Given a hexadecimal of 11 bytes with a padding of 16 bytes:

```
 h  e  l  l  o     w  o  r  l  d  —  11 bytes
68 65 6c 6c 6f 20 77 6f 72 6c 64
68 65 6c 6c 6f 20 77 6f 72 6c 64 05 05 05 05 05  —  16 bytes
                                |___padding____|
```

We just pad it by printing the number of bytes that we should concatenate repeatedly. (Check the implementation that I did: https://github.com/halan/aes.js/blob/master/src/padding.js)

By the way, when using block based cipher we need to split the plain text into blocks of the same size (128 bits for AES) and choose an operation mode to handle those blocks and encrypt it against a key. Because of that, sometimes the last block won't have the right size to go through.

In this article I'll show you an Operation Mode called CBC.

CBC starts doing an XOR (Special OR) between the first block of plain text and a special block called IV (initialization vector), then it's encrypted against the key to generate the first encrypted block. So, that first encrypted block is used to make an XOR with the second plain text block, then it's encrypted against the key to generate the second encrypted block and so on... Changing one block will cause an avalanche over the next blocks, so when ensuring a random and unpredictable IV, it'll have a totally different result even with the same key and plain text.

To decrypt, it'll do the inverse process. First decrypt the first block, then make an XOR with the IV to get the first plain text block. The second plain text block is made from a decryption of the second encrypted block XORed with the first encrypted block and so on...
Note, IV must be unpredictable, it could be random and doesn't need to be secret. Normally it's pre concatenated with the encrypted data or stored close. And the size of the IV is ALWAYS the same length of the block. (Check that implementation that I did: https://github.com/halan/aes.js/blob/master/src/opModes.js#L12-L24)

Let's encrypt something

Finally we can join these concepts together and encrypt/decrypt any data from the browser or from the Node.js.

Our cryptographic system will use the following scheme:

- Encryption AES using CBC mode with a 256 key
- Key generated by PBKDF2 hashing with HMAC-SHA512, using 100k interactions and a random salt of 16 bytes
- IV randomly generated
- Final format: base64(salt + IV + data)
- Part of that schema I have just copied from the real implementation of Enpass that I found here: https://www.enpass.io/docs/security-whitepaper-enpass/EnpassSecurityWhitepaper.pdf

Note that this schema is not compatible with openssl enc command-line, unless you pass the raw key derived from PBKDF2 directly. As we discussed above, openssl enc uses EVP_BytesToKey to derive the key and IV from a salt prefixing the encrypted data.

Node.js

```
const crypto = require('crypto');

salt = crypto.randomBytes(16);
iv = crypto.randomBytes(16);
key = crypto.pbkdf2Sync('my password', salt, 100000, 256/8, 'sha256');

cipher = crypto.createCipheriv('aes-256-cbc', key, iv);

cipher.write("That is our super secret text");
cipher.end()

encrypted = cipher.read();
console.log({
  iv: iv.toString('base64'),
  salt: salt.toString('base64'),
  encrypted: encrypted.toString('base64'),
  concatenned: Buffer.concat([salt, iv, encrypted]).toString('base64')
});

/*

{ iv: 'JaTFWNAEiWIPOANqW/j9kg==',
  salt: '4DkmerTT+FXzsr55zydobA==',
  encrypted: 'jE+QWbdsqYWYXRIKaUuS1q9FaGMPNJko9wOkL9pIYac=',
  concatenned:
   '4DkmerTT+FXzsr55zydobCWkxVjQBIliDzgDalv4/ZKMT5BZt2yphZhdEgppS5LWr0VoYw80mSj3A6Qv2khh

*/
```

Simple and easy, let's decrypt from `4DkmerTT+FXzsr55zydobCWkxVjQBIliDzgDalv4/ZKMT5BZt2yphZhdEgppS5LWr0VoYw80mSj3A6Qv2khhpw==`. Knowing that this data is salt + IV + encrypted data:

```
const crypto = require('crypto');

encrypted = Buffer.from('4DkmerTT+FXzsr55zydobCWkxVjQBIliDzgDalv4/ZKMT5BZt2yphZhdEgppS5L
const salt_len = iv_len = 16;

salt = encrypted.slice(0, salt_len);
iv = encrypted.slice(0+salt_len, salt_len+iv_len);
key = crypto.pbkdf2Sync('my password', salt, 100000, 256/8, 'sha256');

decipher = crypto.createDecipheriv('aes-256-cbc', key, iv);

decipher.write(encrypted.slice(salt_len+iv_len));
decipher.end();

decrypted = decipher.read();
console.log(decrypted.toString());
// That is our super secret text
```

There are some concerns about that API:

- All data can be represented as buffer, string, typed array or data view. The second argument of the write() function would be used to define the input format: utf8, hex, base64. The first argument of read() would be used to define the output format as well.
- end() will add the padding and encrypt the cipher’s last block, calling read() before that will output all blocks, except the last one. final() will act similar to end() but it’ll also output the last block. If you run read() before or after final() it’ll output all blocks, except the last. The first argument of final() would be used to define the output format as we saw in read().
- There is an update() function, and it acts by adding the input and returning the output. It doesn't output any data previously encrypted using write(). But if the data inserted through update is less than one block, it will output an empty buffer and join that data with the next update() or the final(). The 2nd and 3rd arguments of update() are about the input and output formats.
- Cipher and Decipher also support events through on(). We can listen to ‘readable’ and ‘end’ events.
- All step have an async function equivalent (except write()/read(), final()/end() and update()), check the documentation for more details.

Forge

```
const forge = require('node-forge');

const salt = forge.random.getBytesSync(16);
const iv = forge.random.getBytesSync(16);

const key = forge.pkcs5.pbkdf2('my password', salt, 100000, 256/8, 'SHA256');

const cipher = forge.cipher.createCipher('AES-CBC', key);

cipher.start({iv: iv});
cipher.update(forge.util.createBuffer('That is our super secret text'));
cipher.finish();

const encrypted = cipher.output.bytes();

console.log({
  iv: forge.util.encode64(iv),
  salt: forge.util.encode64(salt),
  encrypted: forge.util.encode64(encrypted),
  concatenned: forge.util.encode64(salt + iv + encrypted)
});
```

```
/*

{ iv: '2f0PCR5w/8a4y/5G4SGiLA==',
  salt: 'sYoCiGLJ9xuH3qBLoBzNlA==',
  encrypted: '9LYfj1wUrkro8+a+6f6rglHlVX9qj8N4EMC8ijMjp7Q=',
  concatenned:
   'sYoCiGLJ9xuH3qBLoBzNlNn9DwkecP/GuMv+RuEhoiz0th+PXBSuSujz5r7p/quCUeVVf2qPw3gQwLyKMyOn


*/
```

And then:

```
const forge = require('node-forge');

const encrypted = forge.util.binary.base64.decode('sYoCiGLJ9xuH3qBLoBzNlNn9DwkecP/GuMv+Ru
);

const salt_len = iv_len = 16;

const salt = forge.util.createBuffer(encrypted.slice(0, salt_len));
const iv = forge.util.createBuffer(encrypted.slice(0+salt_len, salt_len+iv_len));

const key = forge.pkcs5.pbkdf2('my password', salt.bytes(), 100000, 256/8, 'SHA256');
const decipher = forge.cipher.createDecipher('AES-CBC', key);

decipher.start({iv: iv});
decipher.update(
  forge.util.createBuffer(encrypted.slice(salt_len + iv_len))
);
decipher.finish();

console.log(decipher.output.toString());
// That is our super secret text
```

Important notes:

- pbkdf2() expects strings as password and salt. So, if you have a forge buffer, you have to call bytes() before.
- cipher.update()/decipher.update() expects a buffer.

CryptoJS

```
const CryptoJS = require('crypto-js');

const salt = CryptoJS.lib.WordArray.random(16);
const iv = CryptoJS.lib.WordArray.random(16);

const key = CryptoJS.PBKDF2('my password', salt, { keySize: 256/32, iterations: 10000, ha

const encrypted = CryptoJS.AES.encrypt('That is our super secret text', key, {iv: iv}).ci

const concatenned =  CryptoJS.lib.WordArray.create().concat(salt).concat(iv).concat(encry

console.log({
  iv: iv.toString(CryptoJS.enc.Base64),
  salt: salt.toString(CryptoJS.enc.Base64),
  encrypted: encrypted.toString(CryptoJS.enc.Base64),
  concatenned: concatenned.toString(CryptoJS.enc.Base64)
});

/*

{ iv: 'oMHnSEQGrr04p8vmrKU7lg==',
  salt: 'OkEt2koR5ChtmYCZ0dXmHQ==',
  encrypted: 'jAOb0LwpmaX51pv8SnTyTcWm2R14GQj0BN7tFjENliU=',
  concatenned:
   'OkEt2koR5ChtmYCZ0dXmHaDB50hEBq69OKfL5qylO5aMA5vQvCmZpfnWm/xKdPJNxabZHXgZCPQE3u0WMQ2WJ

*/
```

Decrypting:

```
const CryptoJS = require('crypto-js');

const encrypted =  CryptoJS.enc.Base64.parse('OkEt2koR5ChtmYCZ0dXmHaDB50hEBq69OKfL5qylO5a

const salt_len = iv_len = 16;

const salt = CryptoJS.lib.WordArray.create(
  encrypted.words.slice(0, salt_len / 4 )
);
const iv = CryptoJS.lib.WordArray.create(
  encrypted.words.slice(0 + salt_len / 4, (salt_len+iv_len) / 4 )
);
```

```
const key = CryptoJS.PBKDF2(
  'my password',
  salt,
  { keySize: 256/32, iterations: 10000, hasher: CryptoJS.algo.SHA256}
);

const decrypted = CryptoJS.AES.decrypt(
  {
    ciphertext: CryptoJS.lib.WordArray.create(
      encrypted.words.slice((salt_len + iv_len) / 4)
    )
  },
  key,
  {iv: iv}
);


console.log(decrypted.toString(CryptoJS.enc.Utf8));
// That is our super secret text
```

Important notes:

- If you pass a string as a key on encrypt() it will enter into a password based mode compatible with OpenSSL (assuming that the first 8 bytes is the string “Salted__” and the next 8 bytes will be a salt to be used to derive the IV and the key. That derivation is not compatible with PBKDF and uses MD5 as core hasher function, so it’s not secure!). Given a key as string encrypt() will ignore the iv sent as option.
- That interface is so confusing and I figured several issues on Github all related with that magic.
- To decrypt we need to send an object with an attribute ciphertext containing a WordArray (a type provided by CryptoJS.lib).
- WordArray is exactly array of numbers of 4 bytes. We can access that array directly through ‘words’. So, the slices is always divided by 4, because the length of each word.

Web Crypto API

```
const encoder = new TextEncoder();

const toBase64 = buffer =>
  btoa(String.fromCharCode(...new Uint8Array(buffer)));
```

```
const PBKDF2 = async (
  password, salt, iterations,
  length, hash, algorithm =  'AES-CBC') => {

  keyMaterial = await window.crypto.subtle.importKey(
    'raw',
    encoder.encode(password),
    {name: 'PBKDF2'},
    false,
    ['deriveKey']
  );


  return await window.crypto.subtle.deriveKey(
      {
        name: 'PBKDF2',
        salt: encoder.encode(salt),
        iterations,
        hash
      },
      keyMaterial,
      { name: algorithm, length },
      false, // we don't need to export our key!!!
      ['encrypt', 'decrypt']
    );
}


const salt = window.crypto.getRandomValues(new Uint8Array(16));
const iv = window.crypto.getRandomValues(new Uint8Array(16));
const plain_text = encoder.encode("That is our super secret text");
const key = await PBKDF2('my password', salt, 100000, 256, 'SHA-256');

const encrypted = await window.crypto.subtle.encrypt(
  {name: "AES-CBC", iv },
  key,
  plain_text
);

console.log({
  salt: toBase64(salt),
  iv: toBase64(iv),
  encrypted: toBase64(encrypted),
  concatennated: toBase64([
    ...salt,
    ...iv,
    ...new Uint8Array(encrypted)
  ])
});

/*

{ salt: "g9cGh/FKtMV1LhnGvii6lA==",
  iv: "Gi+RmKEzDwKoeDBHuHrjPQ==",
  encrypted: "uRl6jYcwHazrVI+omj18UEz/aWsdbKMs8GxQKAkD9Qk=",
  concatennated:

"g9cGh/FKtMV1LhnGvii6lBovkZihMw8CqHgwR7h64z25GXqNhzAdrOtUj6iaPXxQTP9pax1soyzwbFAoCQP1CQ=

*/
```

So dirty, but it works. Let’s decrypt it:

```
const encoder = new TextEncoder();
const decoder = new TextDecoder();

const fromBase64 = buffer =>
  Uint8Array.from(atob(buffer), c => c.charCodeAt(0));

const PBKDF2 = async (
  password, salt, iterations,
  length, hash, algorithm =  'AES-CBC') => {

  const keyMaterial = await window.crypto.subtle.importKey(
    'raw',
    encoder.encode(password),
    {name: 'PBKDF2'},
    false,
    ['deriveKey']
  );
  return await window.crypto.subtle.deriveKey(
    {
      name: 'PBKDF2',
      salt: encoder.encode(salt),
      iterations,
      hash
    },
    keyMaterial,
    { name: algorithm, length },
    false, // we don't need to export our key!!!
    ['encrypt', 'decrypt']
  );
};


const salt_len = iv_len = 16;

const encrypted = fromBase64('g9cGh/FKtMV1LhnGvii6lBovkZihMw8CqHgwR7h64z25GXqNhzAdrOtUj6

const salt = encrypted.slice(0, salt_len);
const iv = encrypted.slice(0+salt_len, salt_len+iv_len);
const key = await PBKDF2('my password', salt, 100000, 256, 'SHA-256');

const decrypted = await window.crypto.subtle.decrypt(
  { name: "AES-CBC", iv },
  key,
  encrypted.slice(salt_len + iv_len)
);
console.log(decoder.decode(decrypted));
```

There is some considerations:

- importKey(), deriveKey() and encrypt()/decrypt() are async functions. importKey() is used both to import key from their bytes and to import password to be used with deriveKey().
- deriveBits() also can be used to derive a key. It's often used if you want to derive an IV and a key together, actually you ask to derive several bytes and then take a chunk of that to seed a importKey in mode raw to be able to be used to encrypt or decrypt something.
- The last argument of deriveKey() or importKey() is a list of allowed functions to be chained.

That's it for a while. I hope to have introduced enough concepts to support how to encrypt plain text or stringified JSON objects using JavaScript.

kudos

- Luan Gonçalves for the good conversations while I was writing this article and for actively review this.
- Elias Rodrigues for the great reviewing including important fixes into the code examples

References and useful links

- Cryptography and Network Security: Principles and Practice by William Stallings - http://williamstallings.com/Cryptography/
- https://www.w3.org/TR/WebCryptoAPI/
- https://nodejs.org/api/crypto.html#crypto_crypto
- https://en.wikipedia.org/wiki/PBKDF2
- https://github.com/halan/aes.js - My didatic-purposes implementation of AES
- https://tonyarcieri.com/whats-wrong-with-webcrypto
- https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2011/august/javascript-cryptography-considered-harmful/
- https://tankredhase.com/2014/04/13/heartbleed-and-javascript-crypto/
- https://vnhacker.blogspot.com/2014/06/why-javascript-crypto-is-useful.html
- http://blog.kotowicz.net/2014/07/js-crypto-goto-fail.html?m=1
- https://hal.inria.fr/hal-01426852/document
- https://www.slideshare.net/Channy/the-history-and-status-of-web-crypto-api
- https://www.w3.org/wiki/NetflixWebCryptoUseCase

Discussion (2)

martinweihrauch • Mar 28

This is an excellent article, thank you for your meticulous work! In fact, implementing encryption is a piece of cake in C# thanks to clear typing of variables/objects, but can become a nightmare in JS (e. g. what is a "WORD" representation in CryptoJs).

For anyone interested encrypting between C# and JS, I solved the issue with this open source, MIT-licensed library:
github.com/smartinmedia/Net-Core-J...

Sendhuraan • Sep 14

Thank you for a detailed article !

Code of Conduct • Report abuse

Halan Pinheiro

A brazilian software developer working at Codeminer42. Specialist in Ruby and Javascript.

LOCATION
Brazil

WORK
Software Engineer at Codeminer42

JOINED
Sep 5, 2019

Trending on DEV Community 🔥

Who's participating in Hacktoberfest 2021?
#hacktoberfest #opensource #discuss

I Design, You Build! - Frontend Challenge #2
#idesignyoubuild #webdev #beginners #javascript

15 Developer Tools to Make You Super Productive
#codenewbie #productivity #programming #codequality