Encrypt And Decrypt Data In Node.js With The Crypto Library

January 22, 2018 Nic Raboy JavaScript, Node.js

As you've probably noticed from the previous few articles, I've been doing a lot of development around cryptocurrency wallets using Node.js. Up until now, I've only been writing about interacting with different currencies. However, I haven't discussed how to safely store your wallet information.

When it comes to storing anything sensitive, whether it be cryptocurrency secrets or something else, you must do so safely and securely. For example, the data must be encrypted at rest and decrypted when used.

We're going to see how to encrypt data with a passphrase using Node.js and decrypt it using that same passphrase when necessary.

Creating a New Node.js Project with Crypto Dependencies

To keep this example simple, we're going to create a fresh project to work with. Somewhere on your computer, execute the following command:

```
npm init -y
```

The above command will create a new **package.json**file and initialize our project. We're going to be making use of the Node.js Crypto library for any and all cipher and decipher logic.

To install the dependency, execute the following from the command line:

```
npm install crypto --save
```

Finally, we need somewhere within our project to add code. For project cleanliness, we're going to create a custom class for all encryption and decryption and a driver file that will instantiate our class.

Within your project, create the following two files:

```
touch safe.js
touch app.js
```

If you're using an operating system that doesn't have the `touch` command, create the above files manually. With these two files created, we can focus on the development of our application.

Encrypting and Decrypting Data with an AES Cipher Algorithm

We're going to be using modern JavaScript for this example which means we're going to create an ES6 class for our encryption and decryption logic.

Open the project's **safe.js** file and include the following code:

```
const FileSystem = require("fs");
const Crypto = require("crypto");

class Safe {

    constructor(filePath, password) { }

    encryptAsync(data) { }

    encrypt(data) { }

    decryptAsync() { }

    decrypt() { }
```

```
}

exports.Safe = Safe;
```

As you can probably guess from the above code, we're going to be offering synchronous and asynchronous methods for encrypting and decrypting a file on the hard disk.

Let's take a look at the `constructor` method to start:

```
constructor(filePath, password) {
    this.filePath = filePath;
    this.password = password;
}
```

For this example, we're going to keep record of the file location and the password for the instance of the class. You may or may not want to do this in your own production version.

Now let's say we want to encrypt a JavaScript object on disk. If we wanted to do this synchronously, we would look at the `encrypt` function:

```
encrypt(data) {
    try {
        var cipher = Crypto.createCipher('aes-256-cbc', this.password);
        var encrypted = Buffer.concat([cipher.update(new Buffer(JSON.strir
        FileSystem.writeFileSync(this.filePath, encrypted);
        return { message: "Encrypted!" };
    } catch (exception) {
        throw new Error(exception.message);
    }
}
```

In the above example we are stating that we want to use an AES algorithm. Because we want to save to a file, we want to create a buffer from our plaintext data. Once we have an encrypted buffer, we can write to the file system and return a message.

You might be wondering why we're creating a cipher within the function rather than once within the `constructor` method. After we call `cipher.final()` we are no longer able

to use the cipher. Because of this, we'll get strange results if we create a class variable for it and try to use it multiple times.

To decrypt this file, we would call the `decrypt` method:

```
decrypt() {
    try {
        var data = FileSystem.readFileSync(this.filePath);
        var decipher = Crypto.createDecipher("aes-256-cbc", this.password)
        var decrypted = Buffer.concat([decipher.update(data), decipher.fir
        return JSON.parse(decrypted.toString());
    } catch (exception) {
        throw new Error(exception.message);
    }
}
```

In the above function we are reading the file into a buffer, decrypting it with the AES algorithm, and returning the decrypted object back to the client.

Synchronous JavaScript has its perks, but in most scenarios you'll want to read and write to the disk asynchronously. For this reason let's have some asynchronous alternatives.

Take a look at the `encryptAsync` function:

```
encryptAsync(data) {
    return new Promise((resolve, reject) => {
        try {
            var cipher = Crypto.createCipher('aes-256-cbc', this.password)
            var encrypted = Buffer.concat([cipher.update(new Buffer(JSON.s
        } catch (exception) {
            reject({ message: exception.message });
        }
        FileSystem.writeFile(this.filePath, encrypted, error => {
            if(error) {
                reject(error)
            }
            resolve({ message: "Encrypted!" });
        });
    });
}
```

The above `encryptAsync` function is similar to the previous. However this time around we are returning a promise and using the asynchronous version of the file system commands.

As you probably guessed, the `decryptAsync` function will be similar as well:

```
decryptAsync() {
    return new Promise((resolve, reject) => {
        FileSystem.readFile(this.filePath, (error, data) => {
            if(error) {
                reject(error);
            }
            try {
                var decipher = Crypto.createDecipher("aes-256-cbc", this.p
                var decrypted = Buffer.concat([decipher.update(data), deci
                resolve(JSON.parse(decrypted.toString()));
            } catch (exception) {
                reject({ message: exception.message });
            }
        });
    });
}
```

Not so difficult right?

With the Crypto library we can encrypt and decrypt streams, files, and strings depending on what you want to accomplish.

So let's take a look at the driving logic that will use the `Safe` class that we had just created.

Using the Cipher Class in a Driver File

We've already done the heavy lifting, but now we want to use what we've created. Open the project's **app.js** file and include the following code:

```
const { Safe } = require("./safe");
```

```
var safe = new Safe("safe.dat", "my-password");
var data = {
    "private_keys": [
        "9F86D081884C7D659A2FEAA0C55AD015A3BF4F1B2B0B822CD15D6C15B0F00A08"
        "5E884898DA28047151D0E56F8DC6292773603D0D6AABBDD62A11EF721D1542D8"
        "3FC9B689459D738F8C88A3A48AA9E33542016B7A4052E001AAA536FCA74813CB"
    ]
};
safe.encryptAsync(data).then(result => {
    return safe.decryptAsync();
});
```

The above code will create a file called **safe.dat** that will be encrypted using the passphrase provided. The JavaScript object will be asynchronously encrypted and then decrypted after. The file will remain present on the hard drive in the end.

Conclusion

You just saw how to encrypt and decrypt data with the Node.js Crypto library. This is very useful if you need to encrypt sensitive data in a file for a local application. For example, let's say we wanted to create an Electronapplication and store sensitive information. We could in this scenario.

If you've been keeping up, you'll remember I wrote an article titled, Send and Manage DigiByte DGB Coins with Node.js. That example made use of private keys which could easily be stored in an encrypted wallet on the hard disk. While we didn't use a secret key in my Ripple XRP wallet example, we could have and made use of encryption as well.