CS 455 – Computer Security Fundamentals

Dr. Chen-Yeou (Charles) Yu

6 cryptography concepts every developer should know

- 6 cryptography concepts every developer should know
  - Hash
  - HMAC
  - Symmetric Encryption
  - Keypairs
  - Asymmetric Encryption
  - Signing
- This is the part of Cryptography 101. I’m not trying to go into detail. Instead, I will share with you in practices by using node.js (Javascript)

6 cryptography concepts every developer should know

- From the next lecture, I will briefly and systematically talk about the cryptography and this will last to the ending of this semester (TBD)
  - If we get some time, I really want to do this. But!?...
- Cryptography? We need to know lots of MATH? Absolutely.
  - But, did you now that we CS student, we are not required to take lots of MATH classes ☺
- First thing of all, as a developer, you don’t need to understand the entire MATH that goes into cryptography, but it’s absolutely essential to know key concepts.
- Some of the concepts are super useful when you are developing IT projects
- In the following, we are going to verify our idea in NodeJS

6 cryptography concepts every developer should know

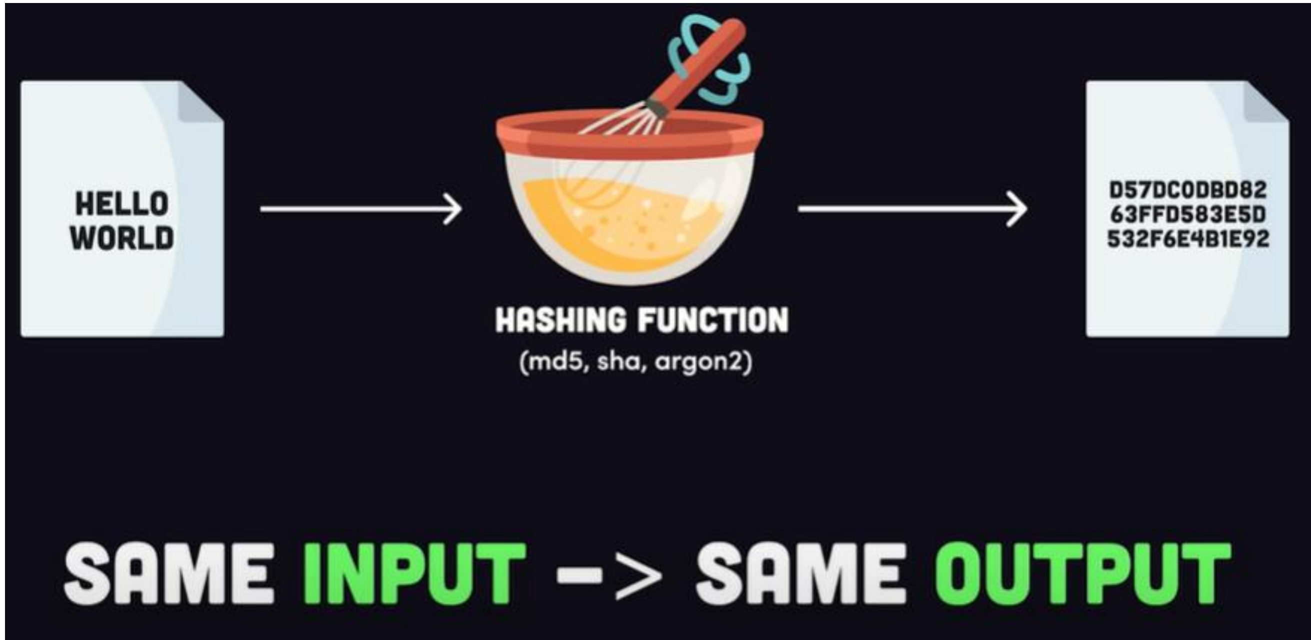
- For NodeJS, you can go to this website to download
  - https://nodejs.org/en

Hash

- Hash is not even the cryptography…not yet!
- It is just an approach to protect your data.
- It means to ***chop and mix*** and that perfectly describes what a hashing function does. (some of the culinary related)
- It starts with an input to a “hashing function”, and this function returns a fixed length value
- Hashing function? It can be md5, sha (family), argon2…

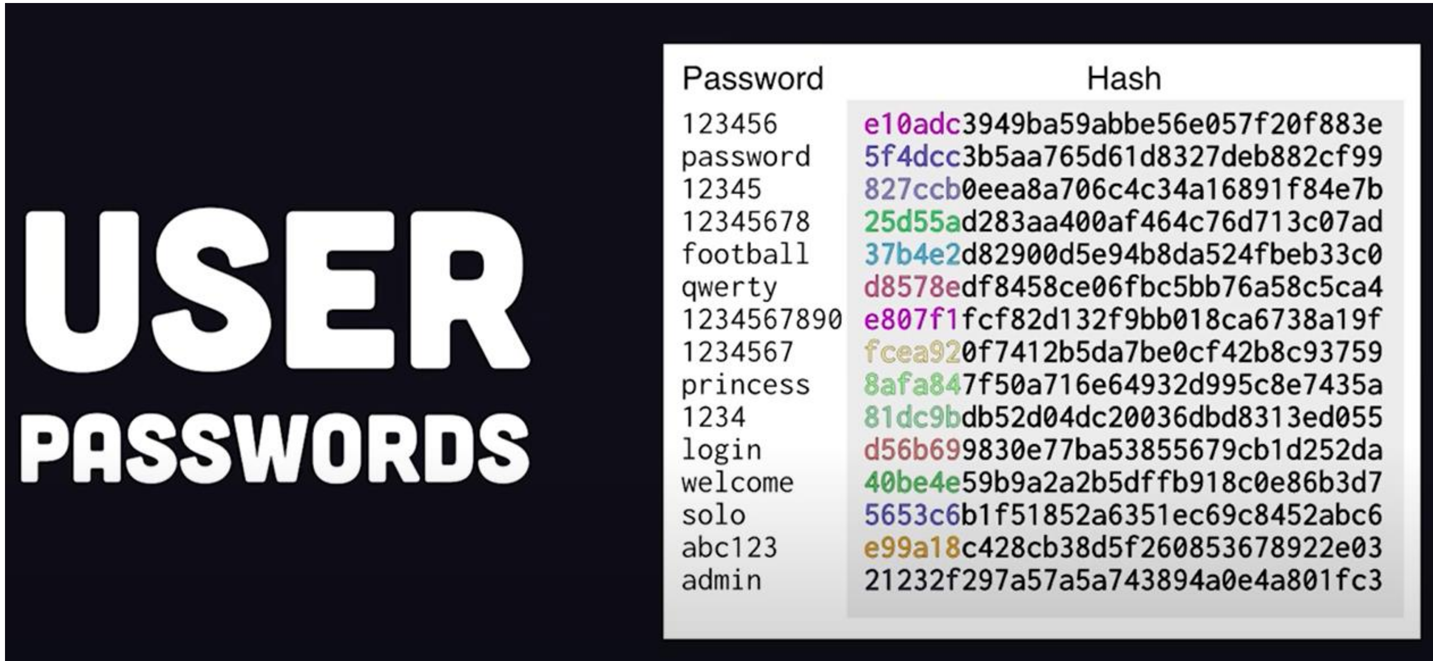
Hash

- The output just looks like garbage
- The function always produce the same output, if inputs are the same!
  - Yes! It is the vulnerability!
- The good thing is, it is infeasible to reverse the output to the original message was.



Hash

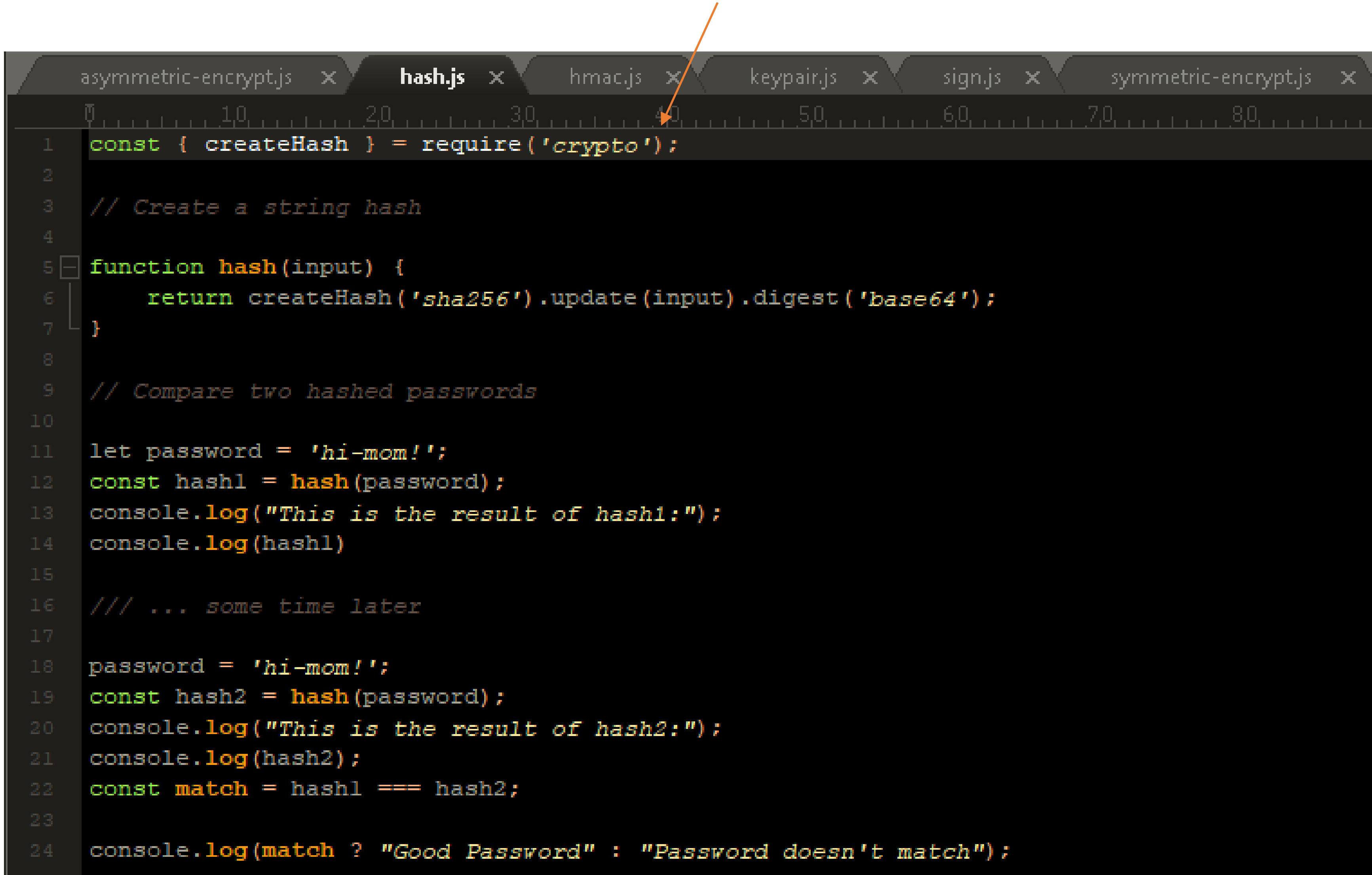
- The most commonly used application is to store the user’s login password
- Even though the hacker get the database, they still need to crack the hash to get the true password.


USER
PASSWORDS

Password	Hash
123456	e10adc3949ba59abbe56e057f20f883e
password	5f4dcc3b5aa765d61d8327deb882cf99
12345	827ccb0eea8a706c4c34a16891f84e7b
12345678	25d55ad283aa400af464c76d713c07ad
football	37b4e2d82900d5e94b8da524fbeb33c0
qwerty	d8578edf8458ce06fbc5bb76a58c5ca4
1234567890	e807f1fcf82d132f9bb018ca6738a19f
1234567	fcea920f7412b5da7be0cf42b8c93759
princess	8afa847f50a716e64932d995c8e7435a
1234	81dc9bdb52d04dc20036dbd8313ed055
login	d56b699830e77ba53855679cb1d252da
welcome	40be4e59b9a2a2b5dffb918c0e86b3d7
solo	5653c6b1f51852a6351ec69c8452abc6
abc123	e99a18c428cb38d5f260853678922e03
admin	21232f297a57a5a743894a0e4a801fc3

Hash

- Check the hash.js
- “crypto” is a kind of built-in module in NodeJS
- So, createHash() is a built-in function
- Hash() is our custom function



```
const { createHash } = require('crypto');

// Create a string hash

function hash(input) {
    return createHash('sha256').update(input).digest('base64');
}

// Compare two hashed passwords

let password = 'hi-mom!';
const hash1 = hash(password);
console.log("This is the result of hash1:");
console.log(hash1)

/// ... some time later

password = 'hi-mom!';
const hash2 = hash(password);
console.log("This is the result of hash2:");
console.log(hash2);
const match = hash1 === hash2;

console.log(match ? "Good Password" : "Password doesn't match");
```

Hash

- In the software engineering, this guy, the hash() is called the "wrapper" because it makes use of createHash()
- In our case, we use 'sha256'

in our hashing algorithm, which

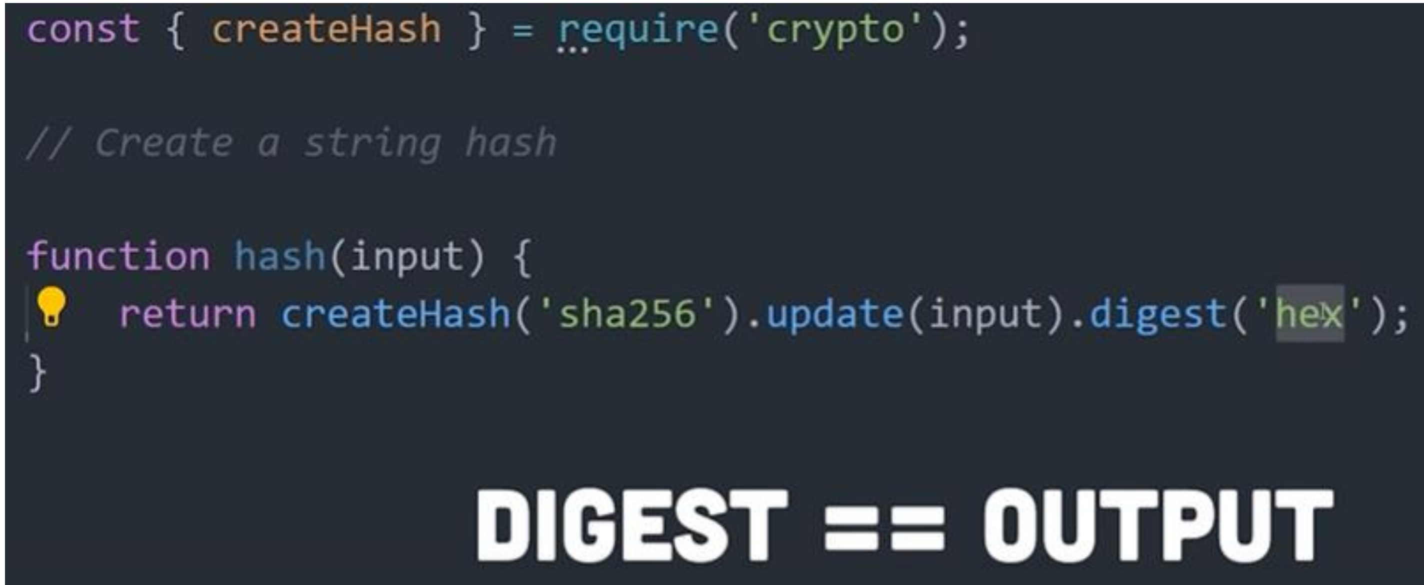
returns 256 bit digest

- We can also use 'md5', but as

the computer is becoming

faster, it is not safe anymore –

easily get cracked!

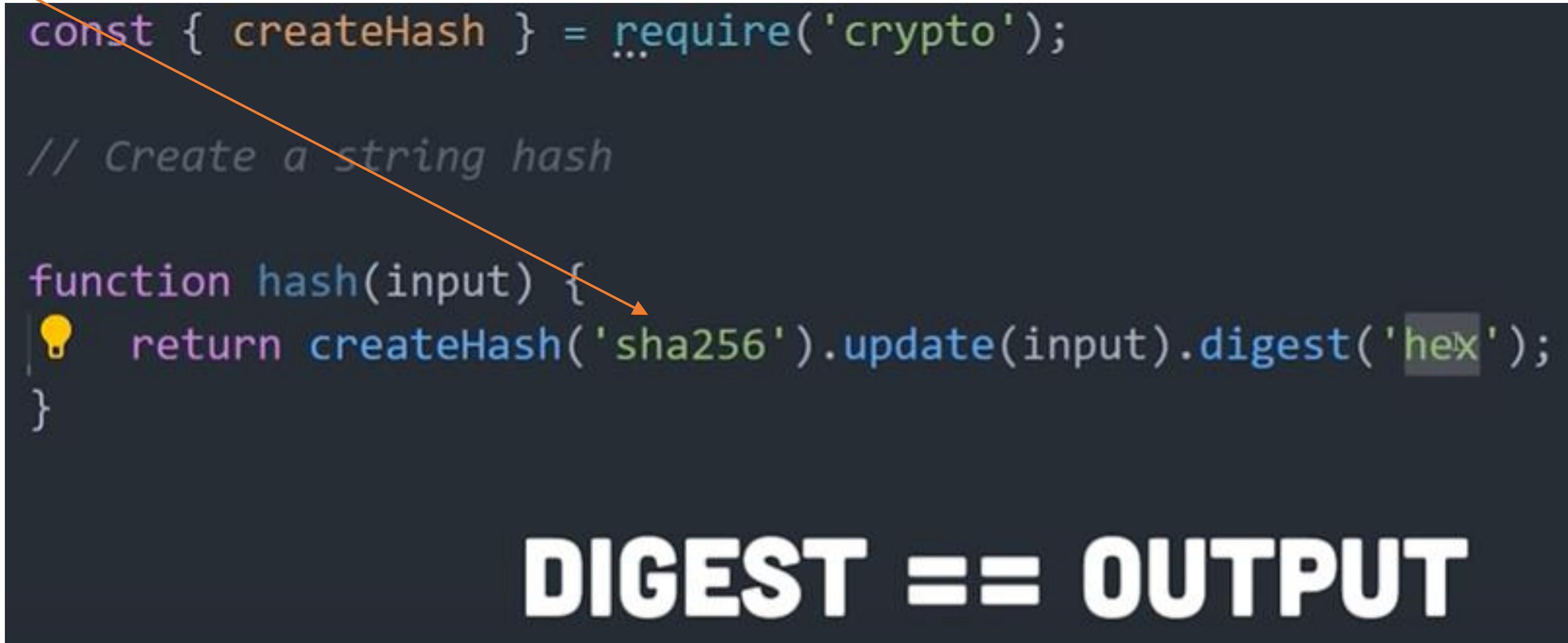


```
const { createHash } = require('crypto');

// Create a string hash

function hash(input) {
  return createHash('sha256').update(input).digest('hex');
}
```

DIGEST == OUTPUT

Hash

- ‘Argon2’ is good but is not built in the node crypto library
- When we get the hash ready, we can call the update() on our input and to output the result in the digest by specifying its format (hex)
- Now, its about the time to check our inputs!

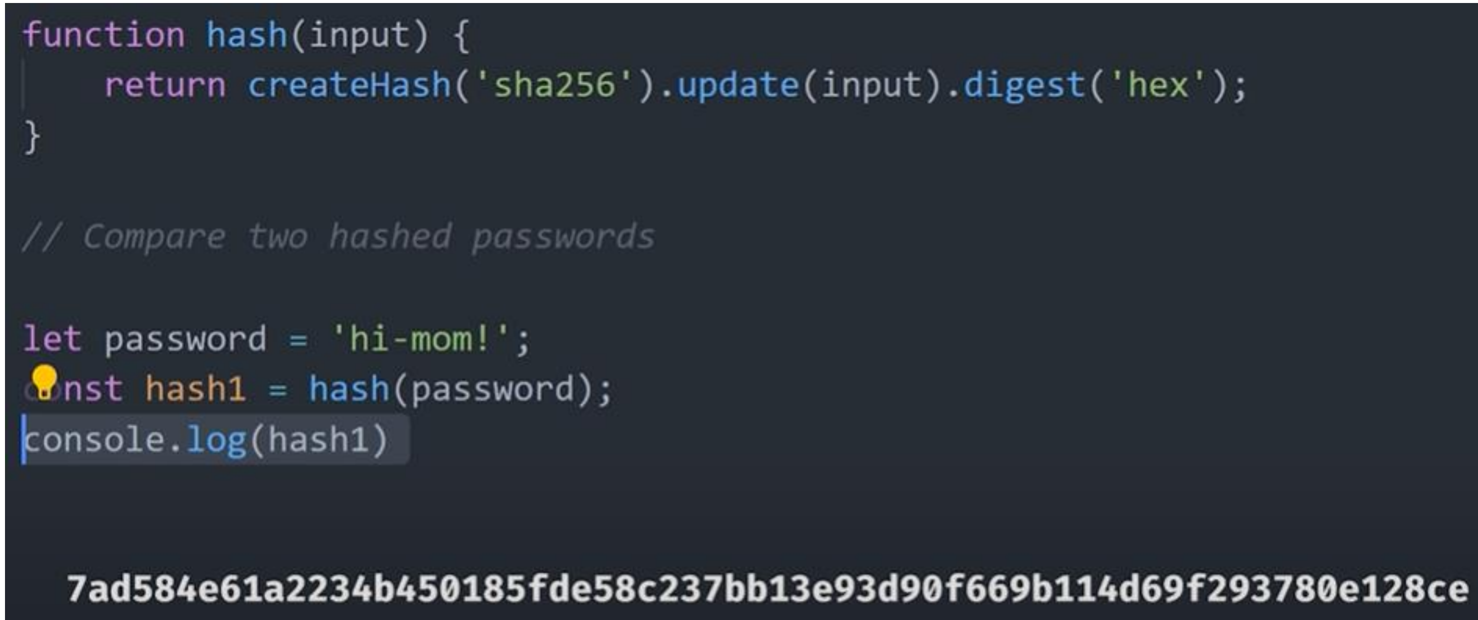


```
const { createHash } = require('crypto');

// Create a string hash

function hash(input) {
    return createHash('sha256').update(input).digest('hex');
}
```

DIGEST == OUTPUT

Hash

- We can use this string, ‘hi-mom!’ ,as an input.
- ‘let’ for a variable is ‘block scoped’ but ‘var’ is for normal varaibles
- console.log() is like C++’s “cout” or Java’s System.out.println()

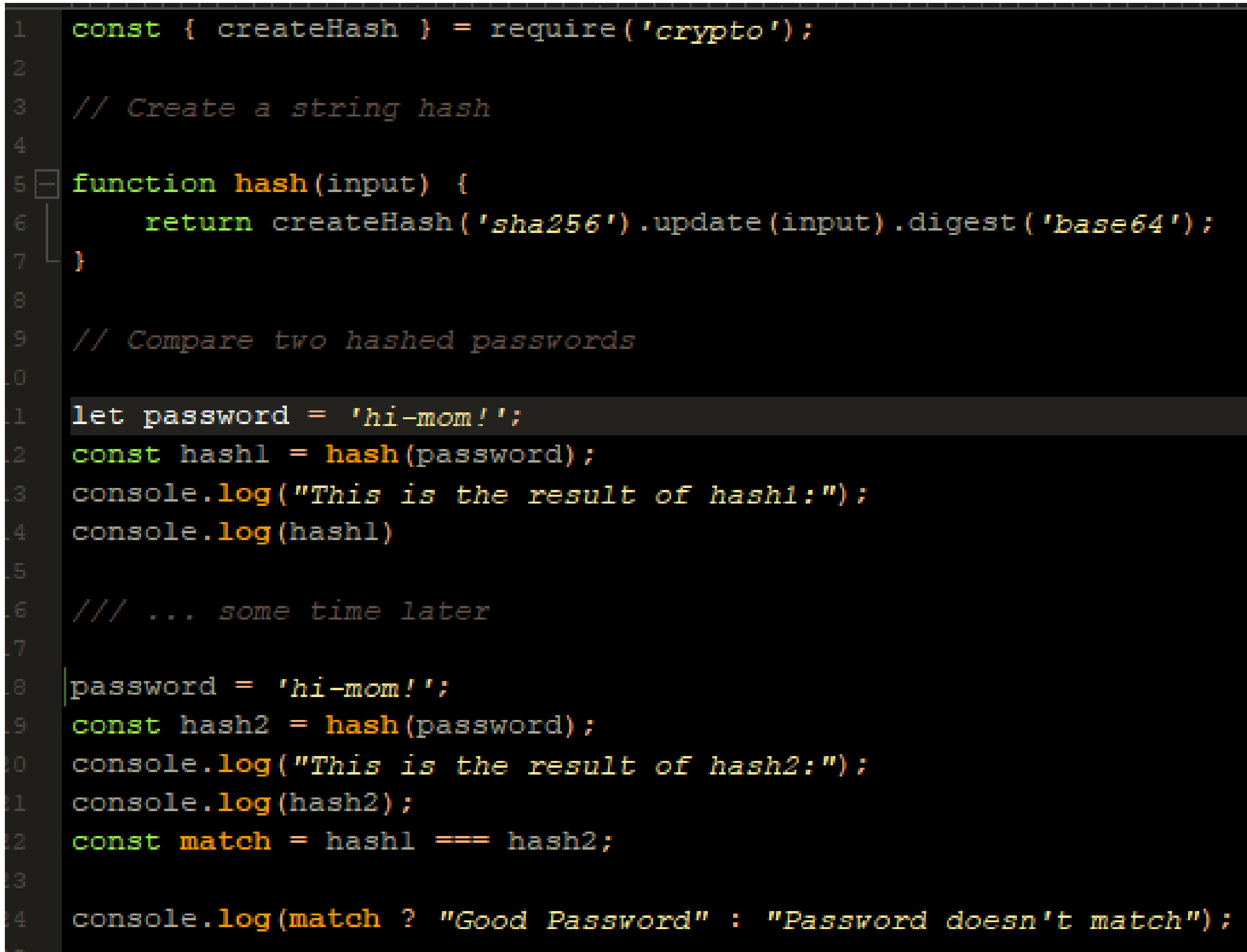


```
function hash(input) {
    return createHash('sha256').update(input).digest('hex');
}

// Compare two hashed passwords

let password = 'hi-mom!';
const hash1 = hash(password);
console.log(hash1)
```

```
7ad584e61a2234b450185fde58c237bb13e93d90f669b114d69f293780e128ce
```

Hash

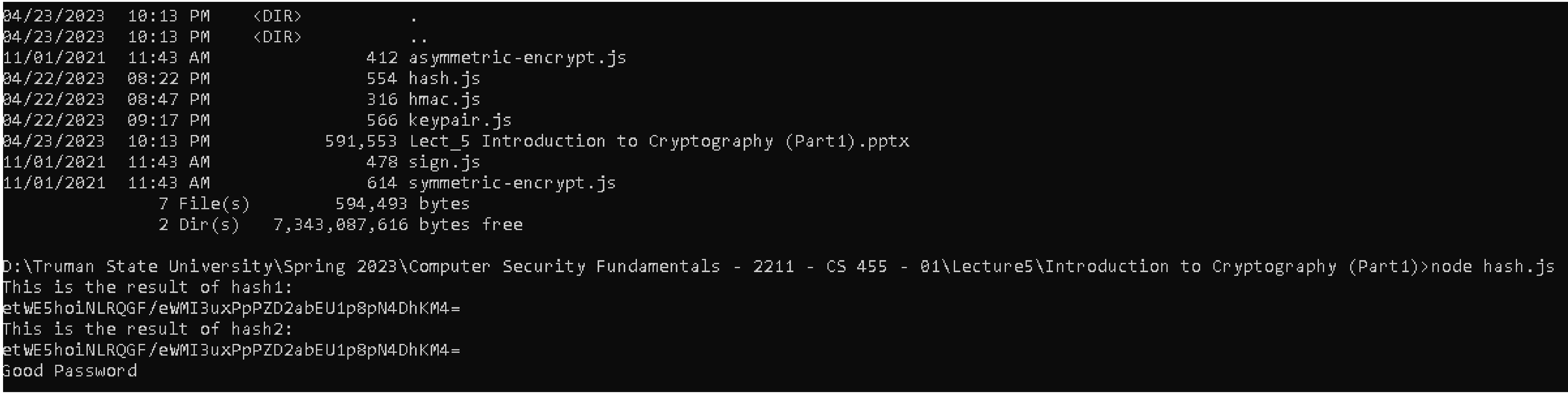
- Now we input the 2nd string with the same content and compare these 2 strings
- “==“ in Javascript is used to compare the value, while “===“ is used to compare the value and the type
- So, in this case, since the generated digest would be the same (input is the same), so “Good Password” will be printed



```
const { createHash } = require('crypto');

// Create a string hash

function hash(input) {
    return createHash('sha256').update(input).digest('base64');
}

// Compare two hashed passwords

let password = 'hi-mom!';
const hash1 = hash(password);
console.log("This is the result of hash1:");
console.log(hash1)

/// ... some time later

password = 'hi-mom!';
const hash2 = hash(password);
console.log("This is the result of hash2:");
console.log(hash2);
const match = hash1 === hash2;

console.log(match ? "Good Password" : "Password doesn't match");
```

Hash

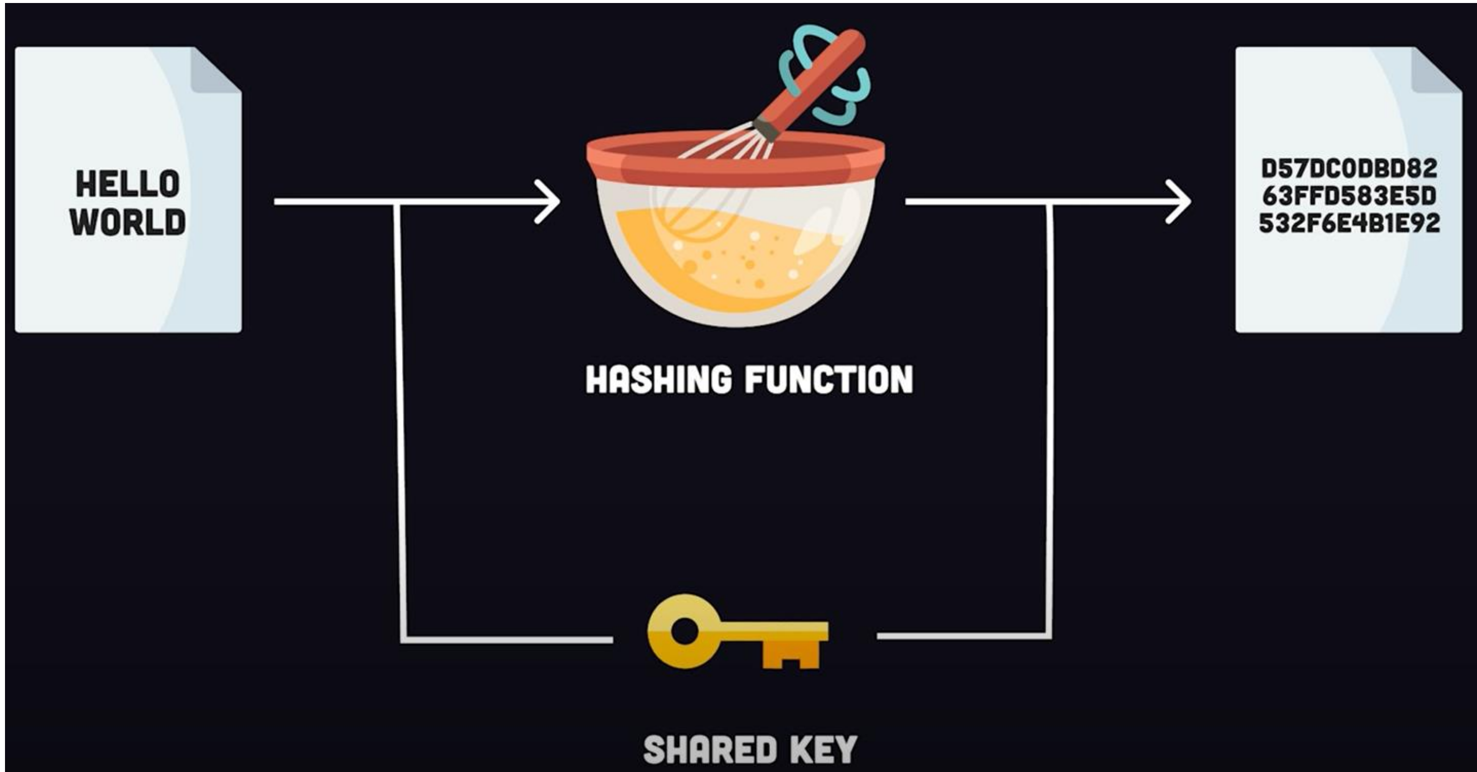
- Here is the execution results



```
04/23/2023  10:13 PM    <DIR>          .
04/23/2023  10:13 PM    <DIR>          ..
11/01/2021  11:43 AM               412 asymmetric-encrypt.js
04/22/2023  08:22 PM               554 hash.js
04/22/2023  08:47 PM               316 hmac.js
04/22/2023  09:17 PM               566 keypair.js
04/23/2023  10:13 PM           591,553 Lect_5 Introduction to Cryptography (Part1).pptx
11/01/2021  11:43 AM               478 sign.js
11/01/2021  11:43 AM               614 symmetric-encrypt.js
               7 File(s)        594,493 bytes
               2 Dir(s)   7,343,087,616 bytes free

D:\Truman State University\Spring 2023\Computer Security Fundamentals - 2211 - CS 455 - 01\Lecture5\Introduction to Cryptography (Part1)>node hash.js
This is the result of hash1:
etWE5hoiNLRQGF/eWMI3uxPpPZD2abEU1p8pN4DhKM4=
This is the result of hash2:
etWE5hoiNLRQGF/eWMI3uxPpPZD2abEU1p8pN4DhKM4=
Good Password
```

HMAC

- Hash-Based Message Authentication Code
- You can see it as the kind of hashing function with a key, this is an addition compared with the pure HASH
- The owner of message or data, he must have the password or key as well.
- You can see it as a kind of “Advanced hash”



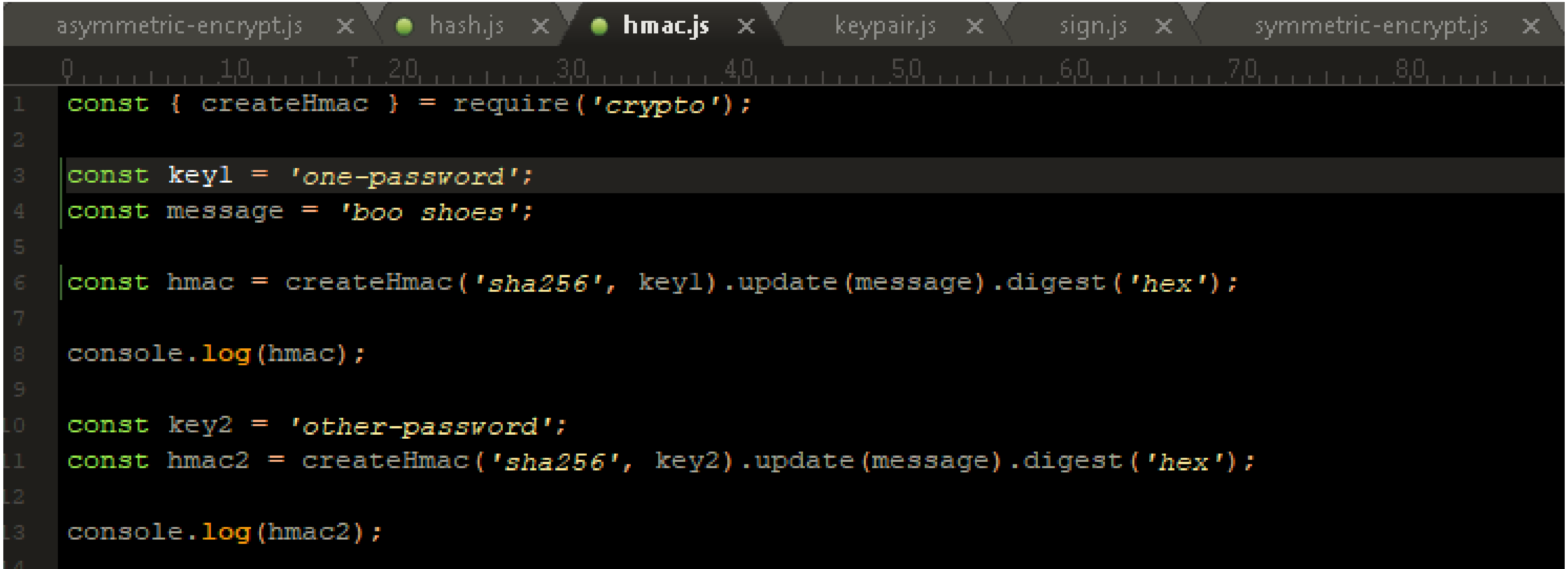
HMAC

- A commonly seen example: A JSON web token for authentication

- Here is the process, roughly speaking:
  - A user use the browser login to server (some website) →
  - Server generate the token with a key (send back to user) →
  - The user use the key to generate HMAC (send back to the server) →
  - The server trust the user because the server knows that there is only the user who can generate the hashed message

HMAC

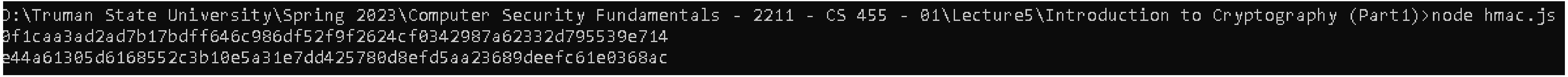
- If the same key (password) is used, it still generates the same hash.
- But if the key is different, it generates different hash



```
const { createHmac } = require('crypto');

const key1 = 'one-password';
const message = 'boo shoes';

const hmac = createHmac('sha256', key1).update(message).digest('hex');

console.log(hmac);

const key2 = 'other-password';
const hmac2 = createHmac('sha256', key2).update(message).digest('hex');

console.log(hmac2);
```

HMAC

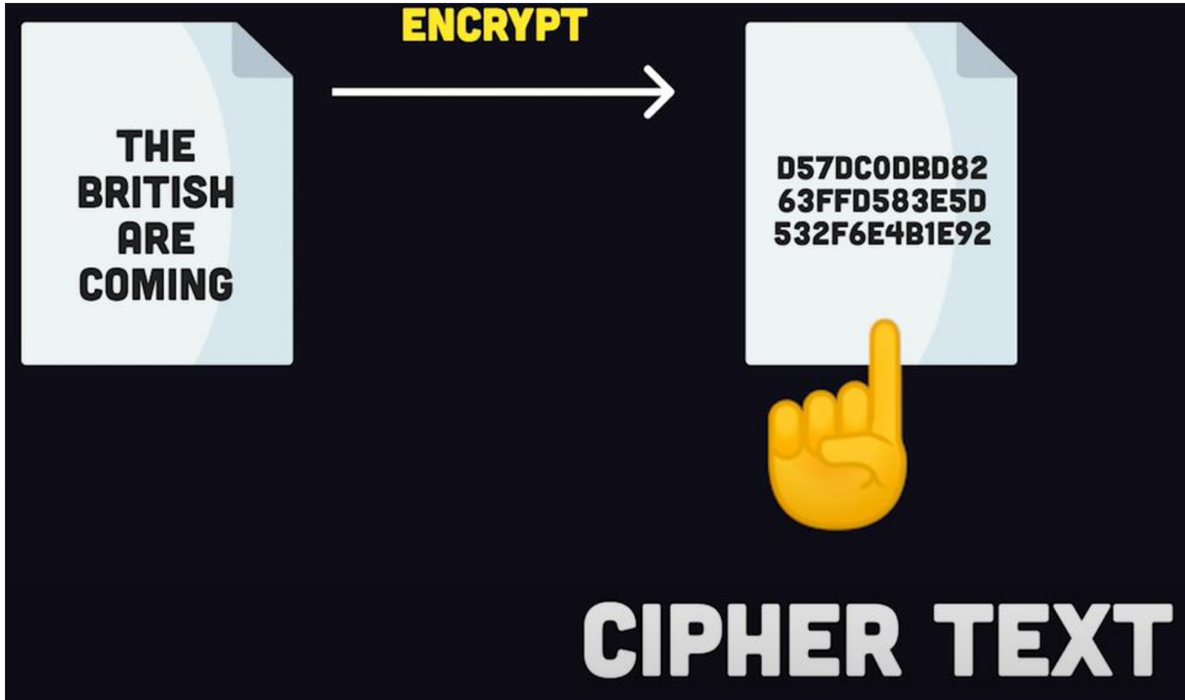
- See? Based the code in the previous slides, the output hash is different (slightly improved)



```
D:\Truman State University\Spring 2023\Computer Security Fundamentals - 2211 - CS 455 - 01\Lecture5\Introduction to Cryptography (Part1)>node hmac.js
0f1caa3ad2ad7b17bdff646c986df52f9f2624cf0342987a62332d795539e714
e44a61305d6168552c3b10e5a31e7dd425780d8efd5aa23689deefc61e0368ac
```

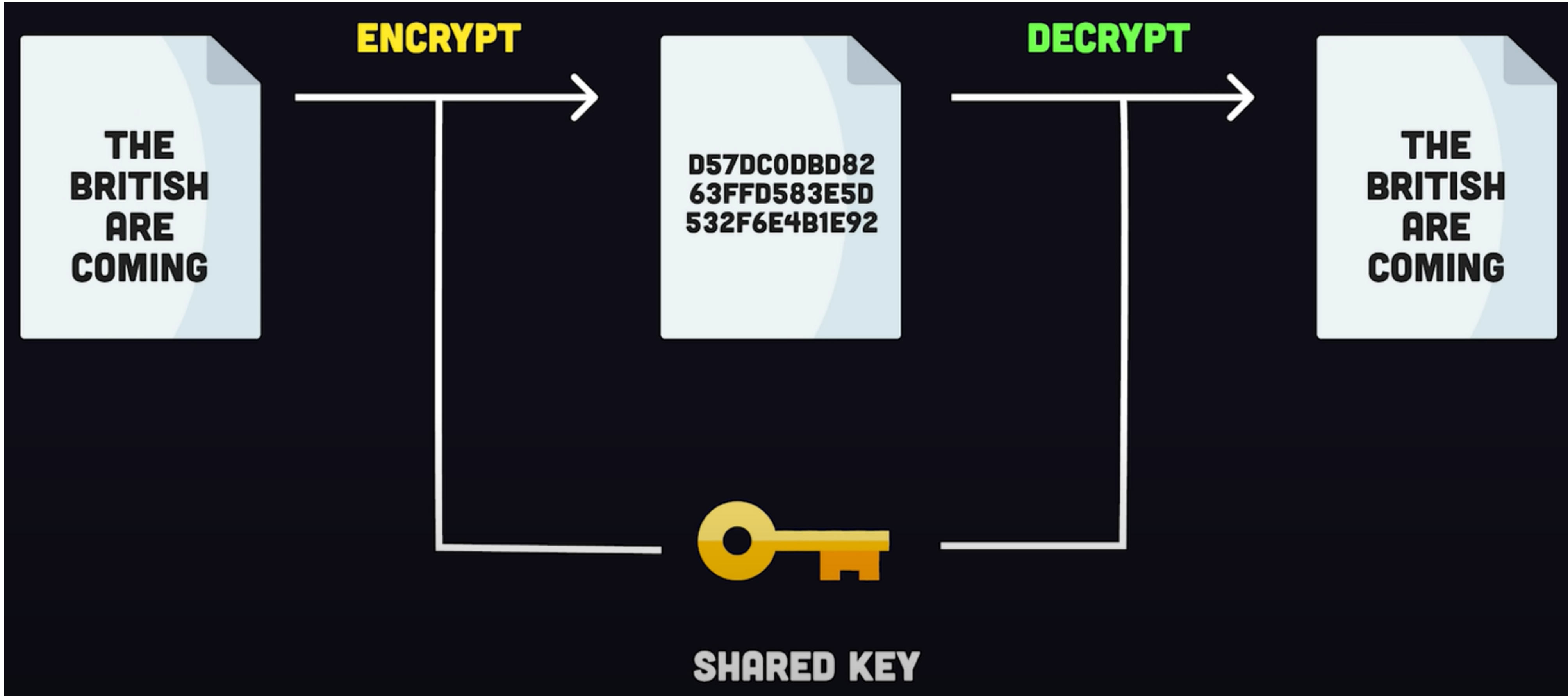
Symmetric Encryption

- What if you want to share the secret (encrypted message) with someone? And allow someone the read the original message
- What is encryption?
    - We take the message, scrambled with bytes and make it unreadable. It is called cipher text



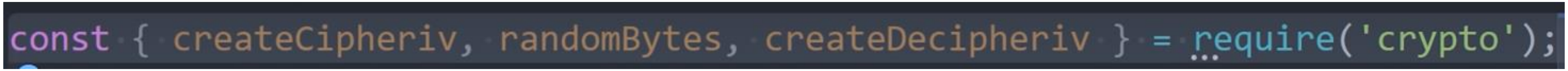
Symmetric Encryption

- Then? You want to reverse it? Want to see the original message? We need a key!



Symmetric Encryption

- The good news is, the cipher is randomized!
- Each time you encrypt, you get the entirely different output, even the key + original message are THE SAME!
- The reason we call it as symmetric is because the sender and the receiver **SHARE** the same key
- To implement these, we need to require the ‘crypto’ library. We also need “createCipheriv” and “createDecipheriv”

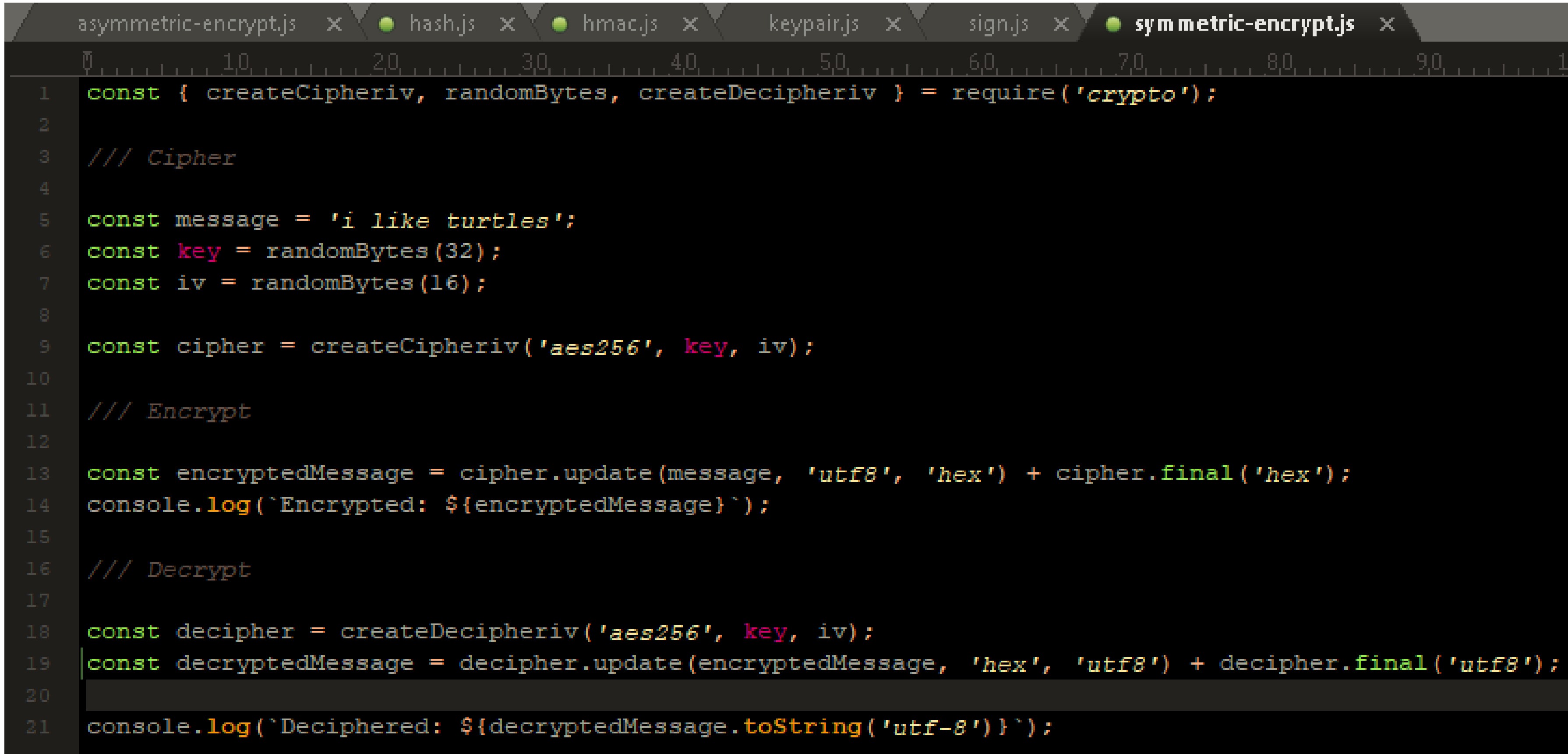


```
const { createCipheriv, randomBytes, createDecipheriv } = require('crypto');
```

- IV stands for Initialization Vector

Symmetric Encryption

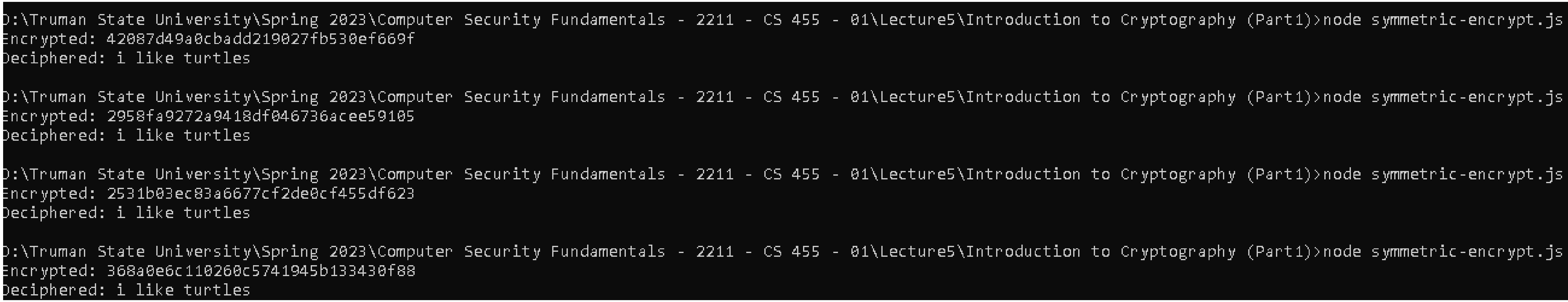
- Here is the code
- Encrypt is to encrypt the message into hex format.
- Decrypt is to do something in the reverse direction
    - Take something in hex format and output the into utf8 format



```
const { createCipheriv, randomBytes, createDecipheriv } = require('crypto');

/// Cipher

const message = 'i like turtles';
const key = randomBytes(32);
const iv = randomBytes(16);

const cipher = createCipheriv('aes256', key, iv);

/// Encrypt

const encryptedMessage = cipher.update(message, 'utf8', 'hex') + cipher.final('hex');
console.log(`Encrypted: ${encryptedMessage}`);

/// Decrypt

const decipher = createDecipheriv('aes256', key, iv);
const decryptedMessage = decipher.update(encryptedMessage, 'hex', 'utf8') + decipher.final('utf8');

console.log(`Deciphered: ${decryptedMessage.toString('utf-8')}`);
```

Symmetric Encryption

- See? Every time, it is different!



```
D:\Truman State University\Spring 2023\Computer Security Fundamentals - 2211 - CS 455 - 01\Lecture5\Introduction to Cryptography (Part1)>node symmetric-encrypt.js
Encrypted: 42087d49a0cbadd219027fb530ef669f
Deciphered: i like turtles

D:\Truman State University\Spring 2023\Computer Security Fundamentals - 2211 - CS 455 - 01\Lecture5\Introduction to Cryptography (Part1)>node symmetric-encrypt.js
Encrypted: 2958fa9272a9418df046736acee59105
Deciphered: i like turtles

D:\Truman State University\Spring 2023\Computer Security Fundamentals - 2211 - CS 455 - 01\Lecture5\Introduction to Cryptography (Part1)>node symmetric-encrypt.js
Encrypted: 2531b03ec83a6677cf2de0cf455df623
Deciphered: i like turtles

D:\Truman State University\Spring 2023\Computer Security Fundamentals - 2211 - CS 455 - 01\Lecture5\Introduction to Cryptography (Part1)>node symmetric-encrypt.js
Encrypted: 368a0e6c110260c5741945b133430f88
Deciphered: i like turtles
```

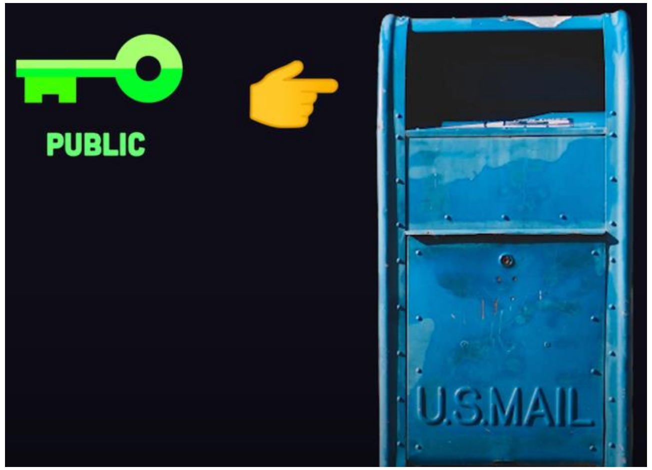
Keypairs

- Private key has to be kept secret

- Public key can be shared with other people

Keypairs

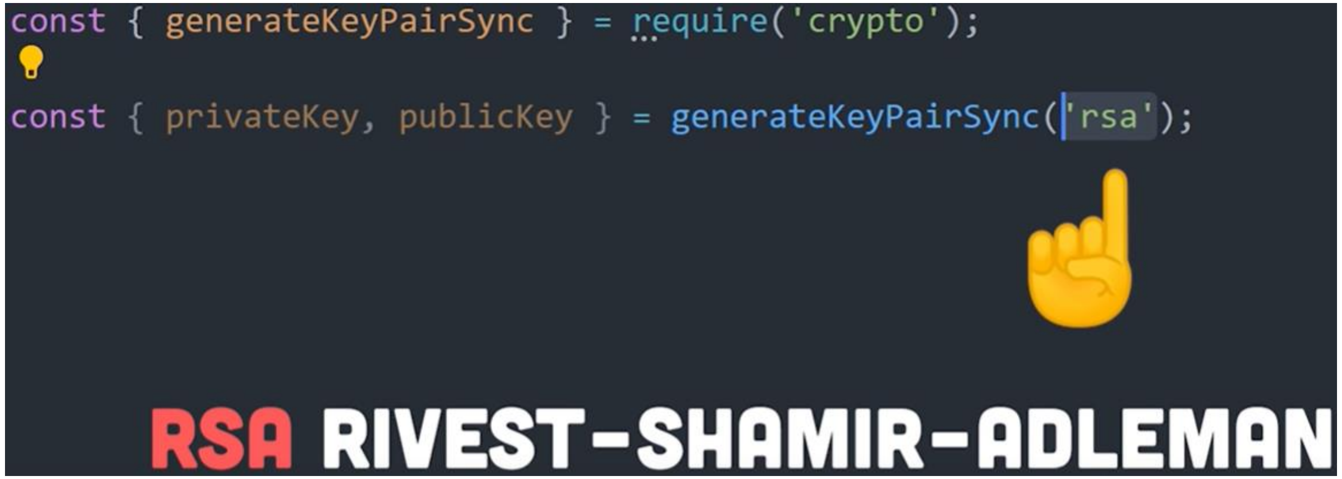
- The postman can use the public key to open the mailbox to add the mail in
- It could be more than one postman
- To get the mail out, you need the private key



PRIVATE
U.S.MAIL

Keypairs

- First thing, we need to require the ‘crypto’ and its built-in function, ‘generateKeyPairSync’
- RSA stands for “Rivest-Shamir-Adleman’



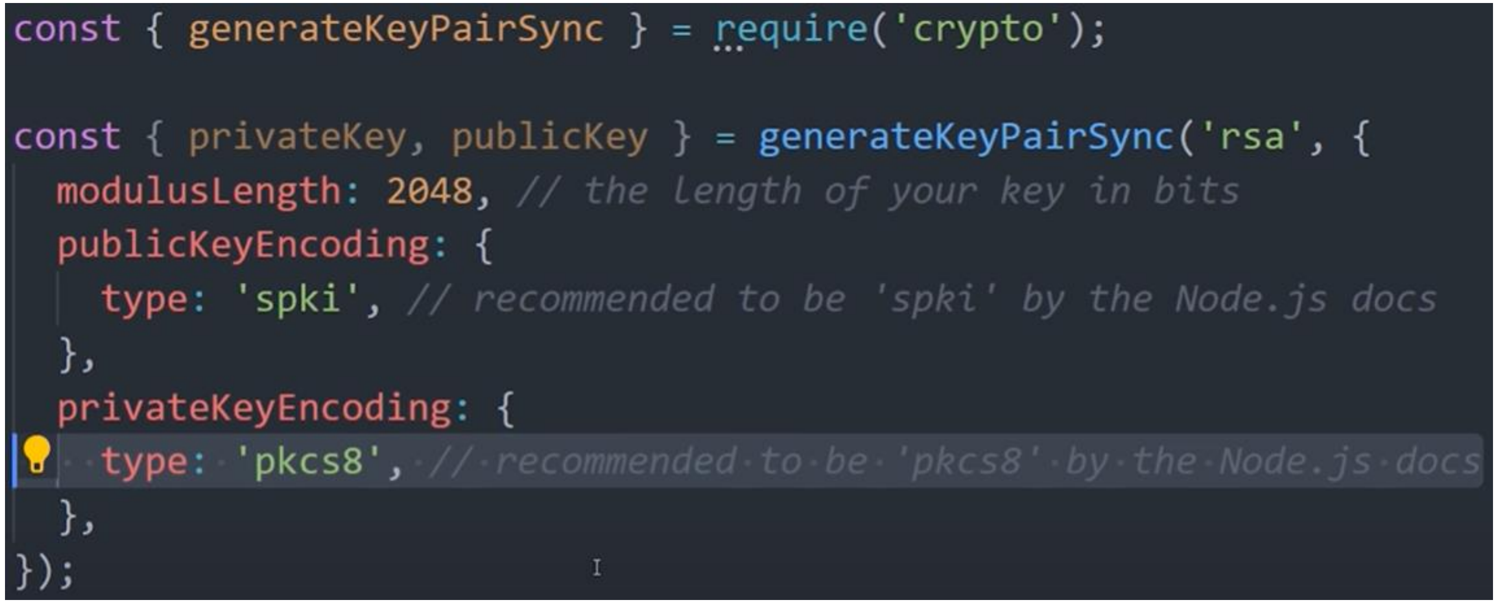
```
const { generateKeyPairSync } = require('crypto');

const { privateKey, publicKey } = generateKeyPairSync('rsa');
```

RSA RIVEST-SHAMIR-ADLEMAN

- Then, finish a bunch of the settings for 2 keys

Keypairs

- We need to finish the section for public key and private key.
- Note that their encoding are different. Generated format are different

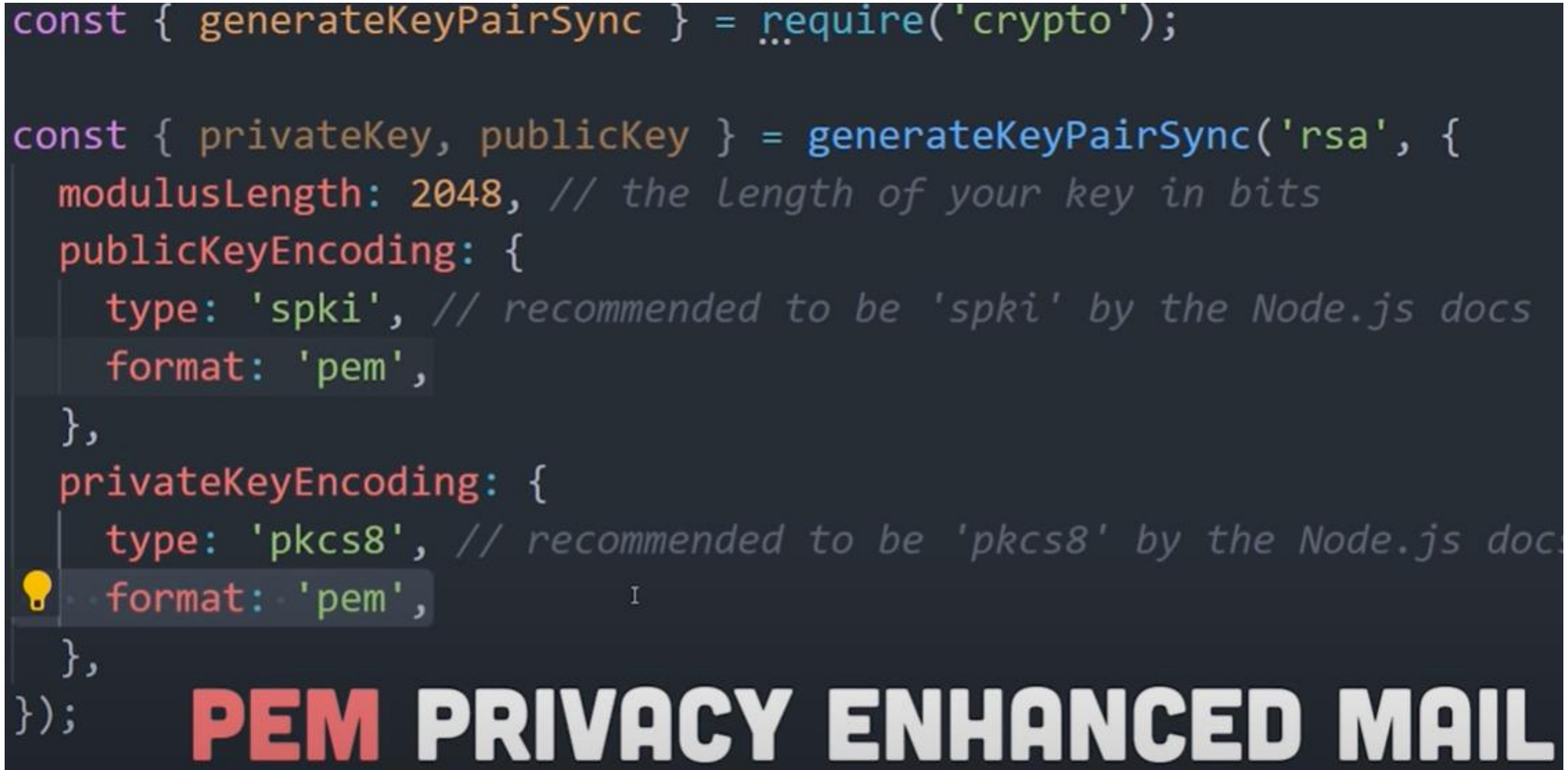


```
const { generateKeyPairSync } = require('crypto');

const { privateKey, publicKey } = generateKeyPairSync('rsa', {
  modulusLength: 2048, // the length of your key in bits
  publicKeyEncoding: {
    type: 'spki', // recommended to be 'spki' by the Node.js docs
  },
  privateKeyEncoding: {
    type: 'pkcs8', // recommended to be 'pkcs8' by the Node.js docs
  },
});
```

Keypairs

PEM stands for Privacy Enhanced Mail

Form the name we know it is used for email communications originally.

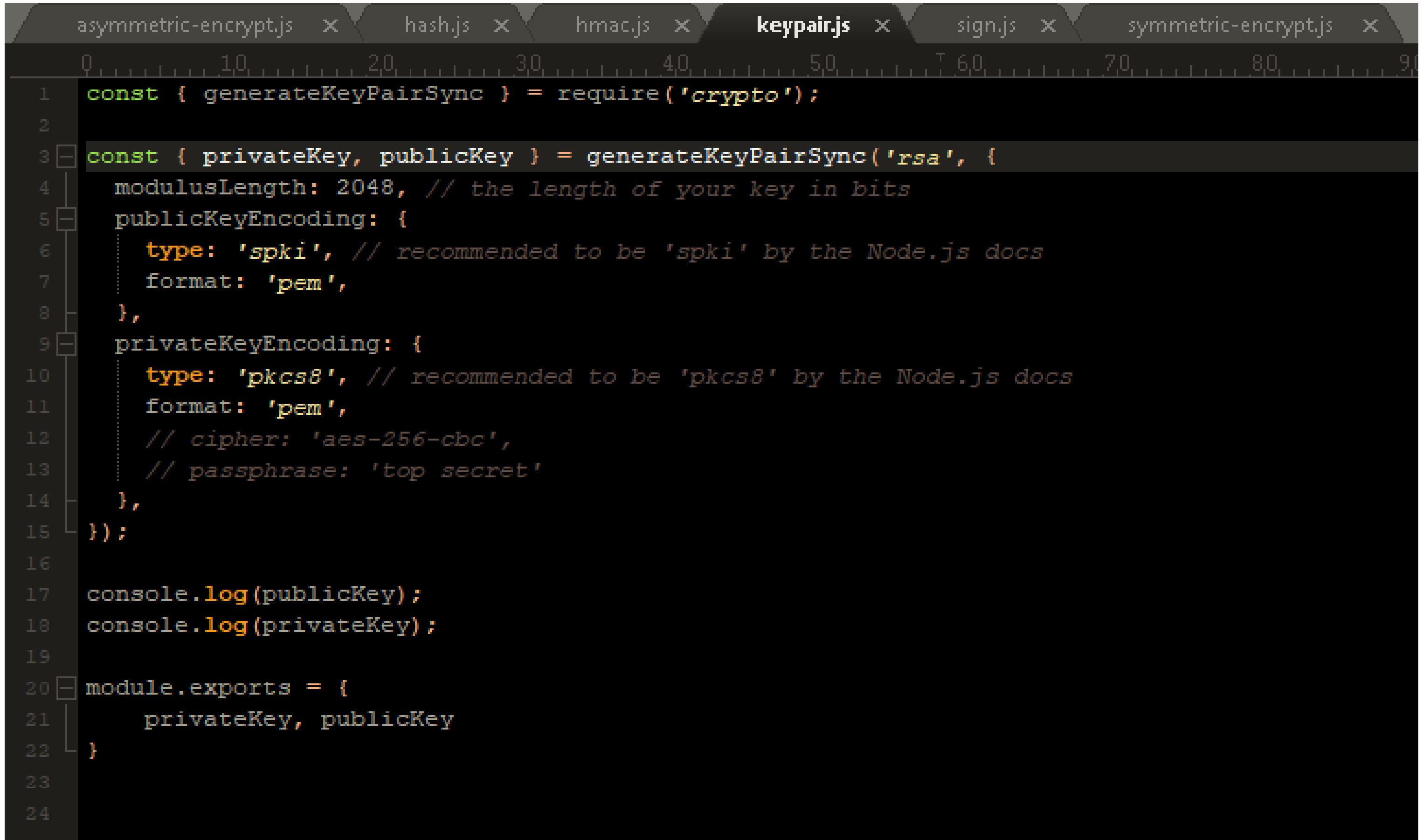


```
const { generateKeyPairSync } = require('crypto');

const { privateKey, publicKey } = generateKeyPairSync('rsa', {
  modulusLength: 2048, // the length of your key in bits
  publicKeyEncoding: {
    type: 'spki', // recommended to be 'spki' by the Node.js docs
    format: 'pem',
  },
  privateKeyEncoding: {
    type: 'pkcs8', // recommended to be 'pkcs8' by the Node.js doc
    format: 'pem',
  },
});
```

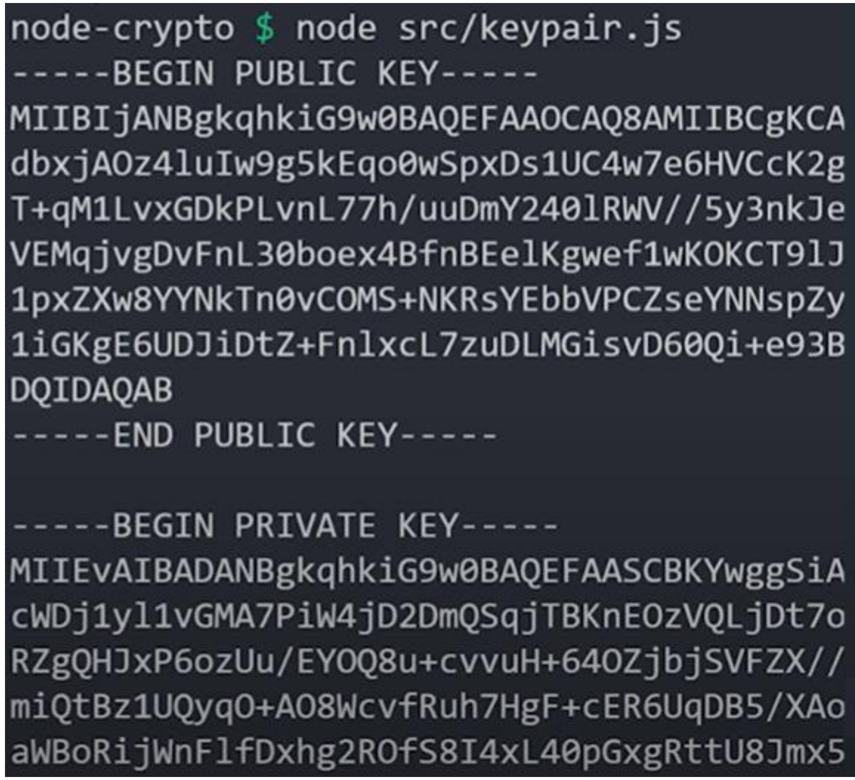
PEM PRIVACY ENHANCED MAIL

Keypairs

- You can also add some cypher or password for more security, it is allowable. Check the code for commented out lines



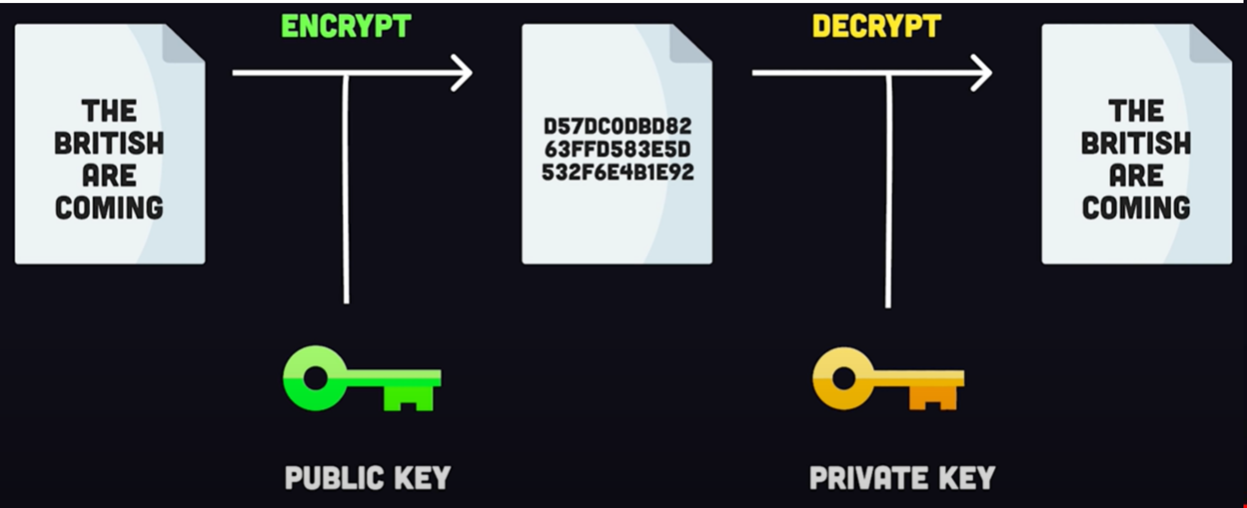
```
const { generateKeyPairSync } = require('crypto');

const { privateKey, publicKey } = generateKeyPairSync('rsa', {
  modulusLength: 2048, // the length of your key in bits
  publicKeyEncoding: {
    type: 'spki', // recommended to be 'spki' by the Node.js docs
    format: 'pem',
  },
  privateKeyEncoding: {
    type: 'pkcs8', // recommended to be 'pkcs8' by the Node.js docs
    format: 'pem',
    // cipher: 'aes-256-cbc',
    // passphrase: 'top secret'
  },
});

console.log(publicKey);
console.log(privateKey);

module.exports = {
    privateKey, publicKey
}
```

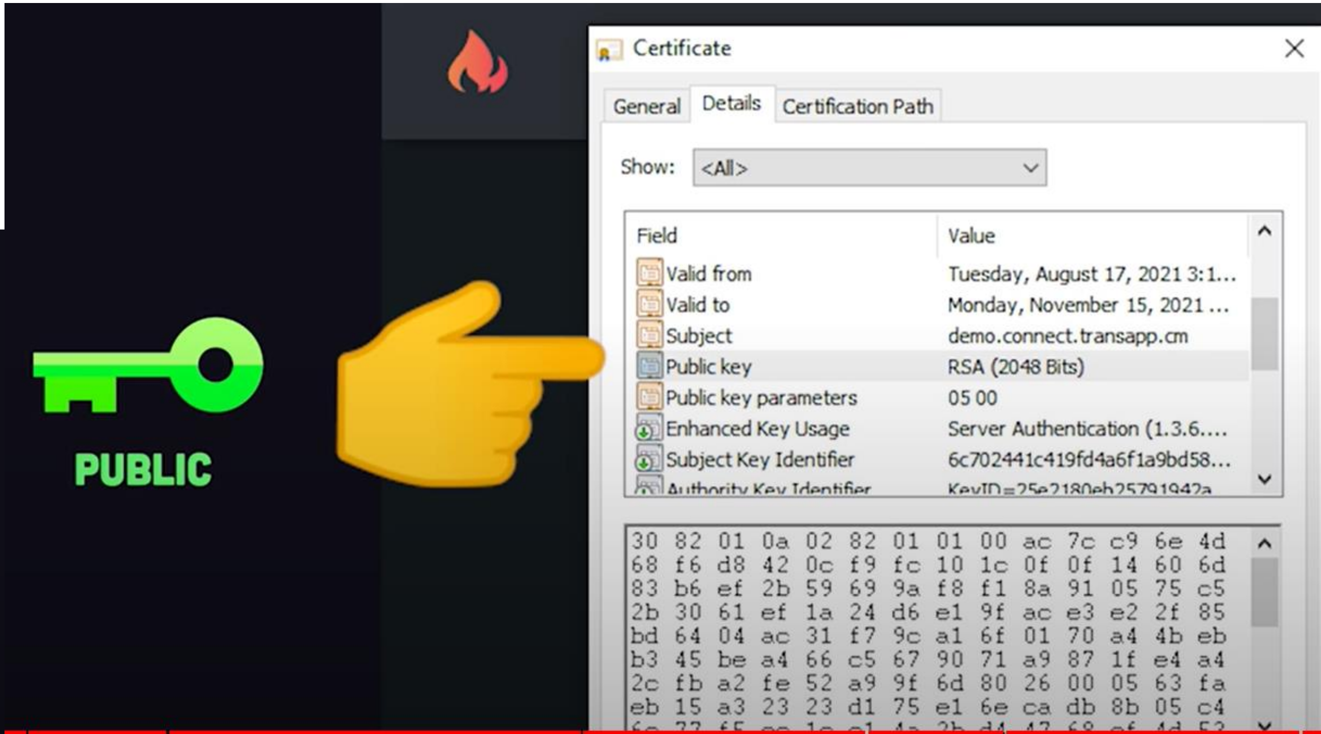


```
node-crypto $ node src/keypair.js
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCA
dbxjAOz4luIw9g5kEqo0wSpxDs1UC4w7e6HVCcK2g
T+qM1LvxGDkPLvnL77h/uuDmY240lRWV//5y3nkJe
VEMqjvgDvFnL30boex4BfnBEelKgwef1wKOKCT9lJ
1pxZXw8YYNkTn0vCOMS+NKRsYEbbVPCZseYNNspZy
1iGKgE6UDJiDtZ+FnlxcL7zuDLMGisvD60Qi+e93B
DQIDAQAB
-----END PUBLIC KEY-----

-----BEGIN PRIVATE KEY-----
MIIEvAIBADANBgkqhkiG9w0BAQEFAASCBKYwggSiA
cWDj1yl1vGMA7PiW4jD2DmQSqjTBKnEOzVQLjDt7o
RZgQHJxP6ozUu/EYOQ8u+cvvuH+64OZjbjSVFZX//
miQtBz1UQyqO+AO8WcvfRuh7HgF+cER6UqDB5/XAo
aWBoRijWnFlfDxhg2ROfS8I4xL40pGxgRttU8Jmx5
```

Asymmetric Encryption

- Every time you go to a website by using https, the browser will automatically receive the public key from the web site via SSL
- The public key is originally installed in the web site, now is sent to you, the client.





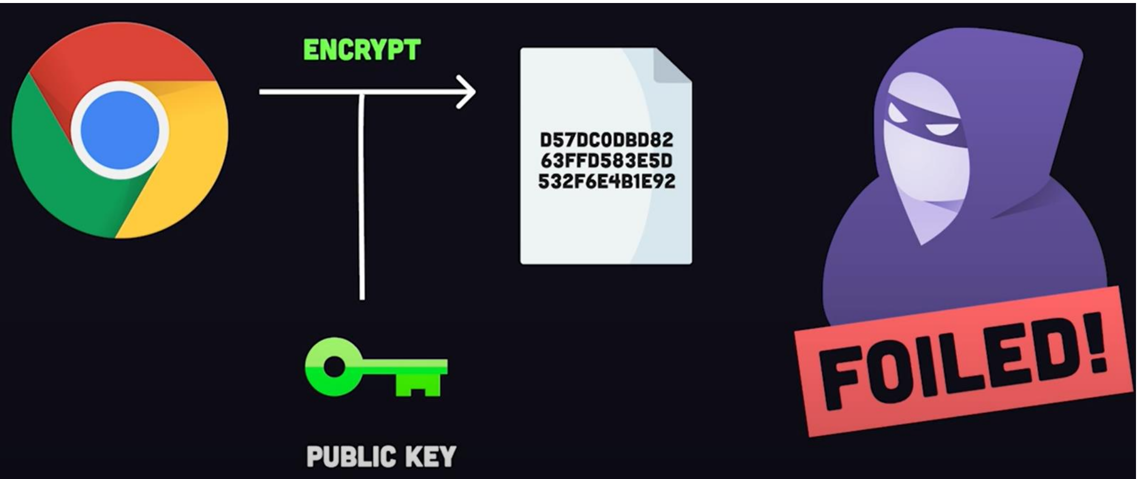
Show: <All>

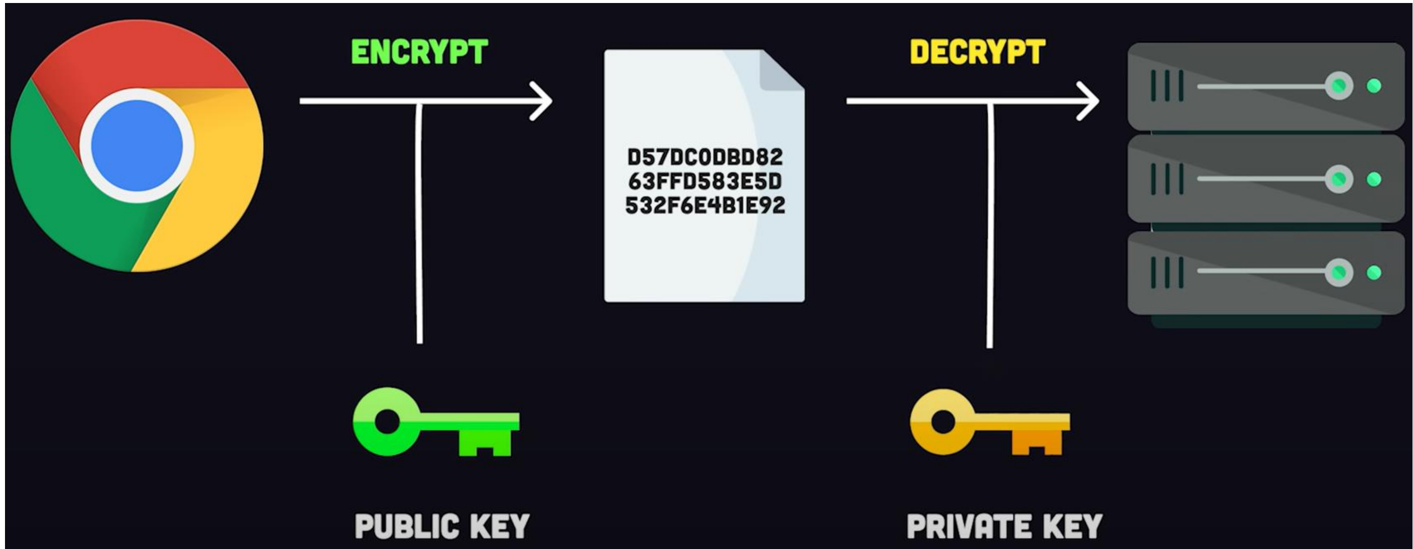
Field	Value
Valid from	Tuesday, August 17, 2021 3:1...
Valid to	Monday, November 15, 2021 ...
Subject	demo.connect.transapp.cm
Public key	RSA (2048 Bits)
Public key parameters	05 00
Enhanced Key Usage	Server Authentication (1.3.6....
Subject Key Identifier	6c702441c419fd4a6f1a9bd58...
Authority Key Identifier	KeyID=25e2180eb25791942a

```
30 82 01 0a 02 82 01 01 00 ac 7c c9 6e 4d
68 f6 d8 42 0c f9 fc 10 1c 0f 0f 14 60 6d
83 b6 ef 2b 59 69 9a f8 f1 8a 91 05 75 c5
2b 30 61 ef 1a 24 d6 e1 9f ac e3 e2 2f 85
bd 64 04 ac 31 f7 9c a1 6f 01 70 a4 4b eb
b3 45 be a4 66 c5 67 90 71 a9 87 1f e4 a4
2c fb a2 fe 52 a9 9f 6d 80 26 00 05 63 fa
eb 15 a3 23 23 d1 75 e1 6e ca db 8b 05 c4
```

Asymmetric Encryption

- Now, the public key in your side, the client is used to encrypt the data being transferred to the server that prevents the hacker stealing your data
- Your data is then decrypted in the server by using private key



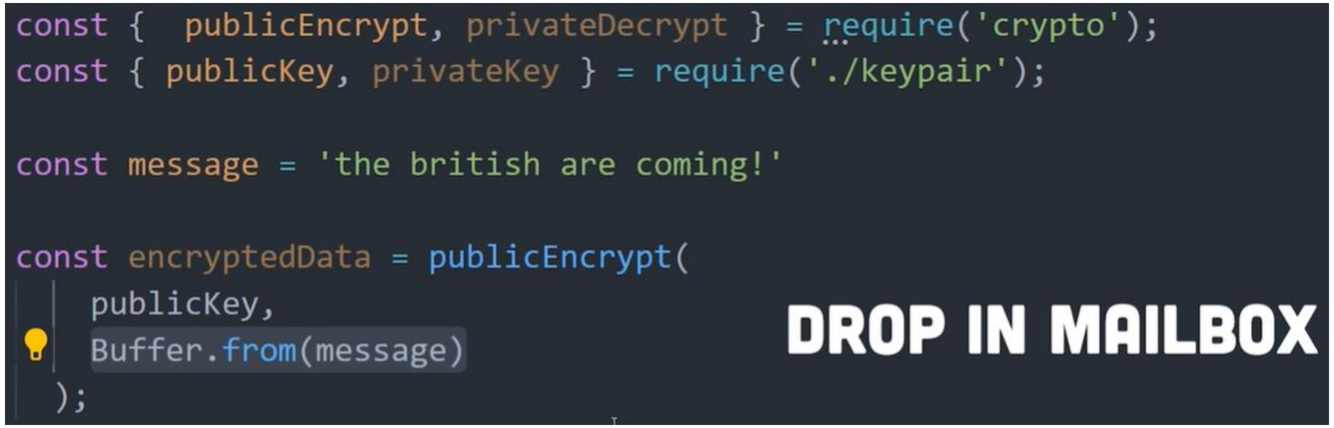


Asymmetric Encryption

- The encryption is simple, public key for encryption and private key for decryption.
- We require the keypair module we just exported in the previous ‘keypairs’ part.
- We just do something in the “module.exports”, right? Check this for details and you will know ‘exports’ mechanism in NodeJS
  - https://www.sitepoint.com/understanding-module-exports-exports-node-js/

Asymmetric Encryption

- We can then prepare the message or we can say to put that in the US mail box by using a call to the publicEncrypt()

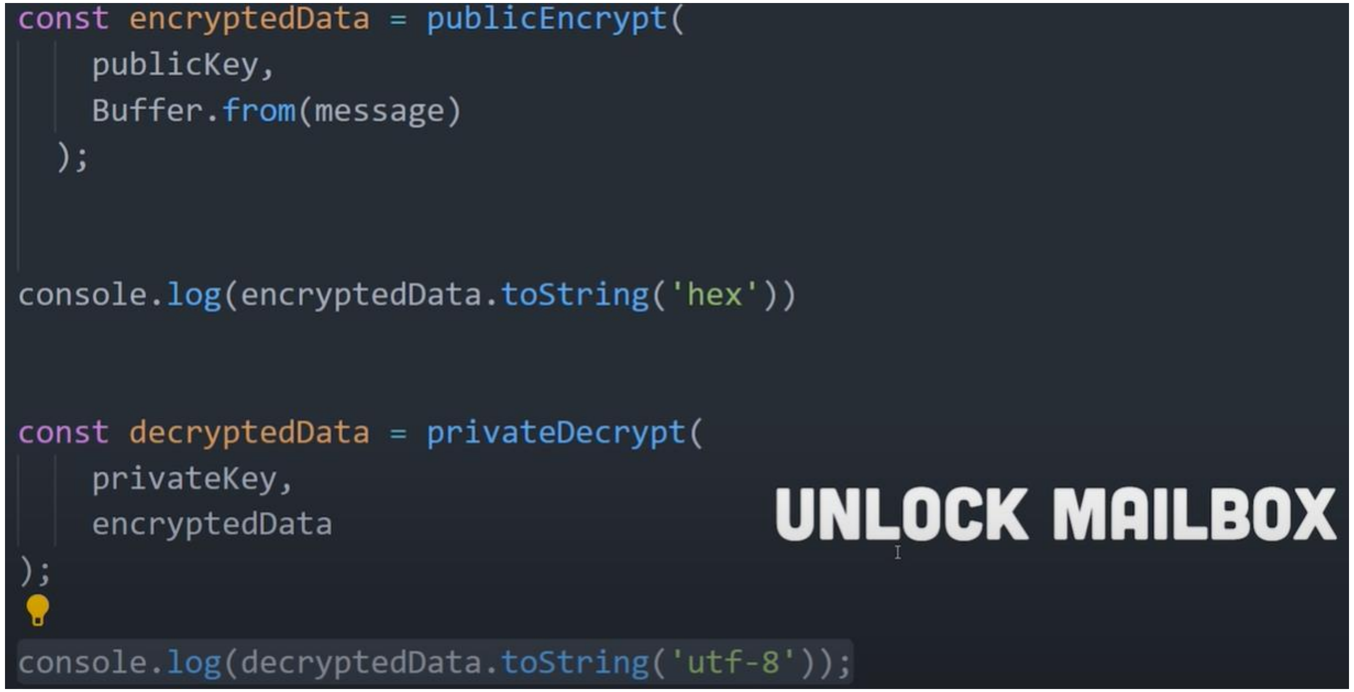


```
const {  publicEncrypt, privateDecrypt } = require('crypto');
const { publicKey, privateKey } = require('./keypair');

const message = 'the british are coming!'

const encryptedData = publicEncrypt(
    publicKey,
    Buffer.from(message)
  );
```

- So the owner with the private key open the mail box and read it

Asymmetric Encryption

- At some point, when the receiver want to read it, it need to have a private key



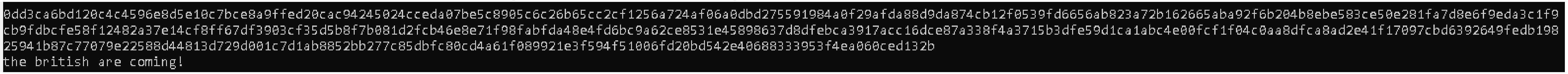
```
const encryptedData = publicEncrypt(
    publicKey,
    Buffer.from(message)
  );


console.log(encryptedData.toString('hex'))


const decryptedData = privateDecrypt(
    privateKey,
    encryptedData
);

console.log(decryptedData.toString('utf-8'));
```

- Sometimes, there is a more important job we need to do. We need to validate the data is from a trusted buddy.

- That is why we need the signing

Asymmetric Encryption

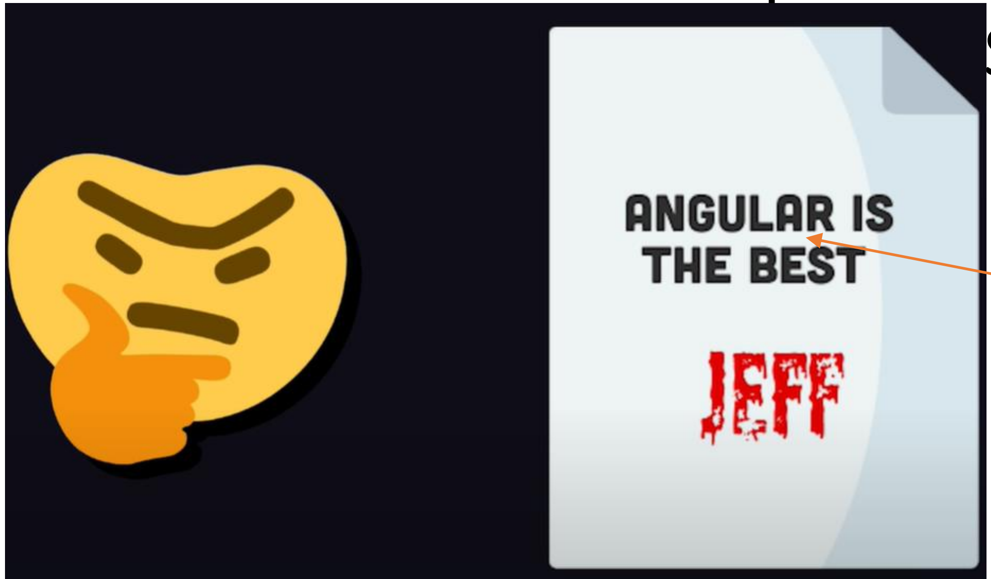
- See? This is the original message!



```
0dd3ca6bd120c4c4596e8d5e10c7bce8a9ffed20cac94245024cceda07be5c8905c6c26b65cc2cf1256a724af06a0dbd275591984a0f29afda88d9da874cb12f0539fd6656ab823a72b162665aba92f6b204b8ebe583ce50e281fa7d8e6f9eda3c1f9
cb9fdbcfe58f12482a37e14cf8ff67df3903cf35d5b8f7b081d2fcb46e8e71f98fabfda48e4fd6bc9a62ce8531e45898637d8dfebca3917acc16dce87a338f4a3715b3dfe59d1ca1abc4e00fcf1f04c0aa8dfca8ad2e41f17097cbd6392649fedb198
25941b87c77079e22588d44813d729d001c7d1ab8852bb277c85dbfc80cd4a61f089921e3f594f51006fd20bd542e40688333953f4ea060ced132b
the british are coming!
```

Signing

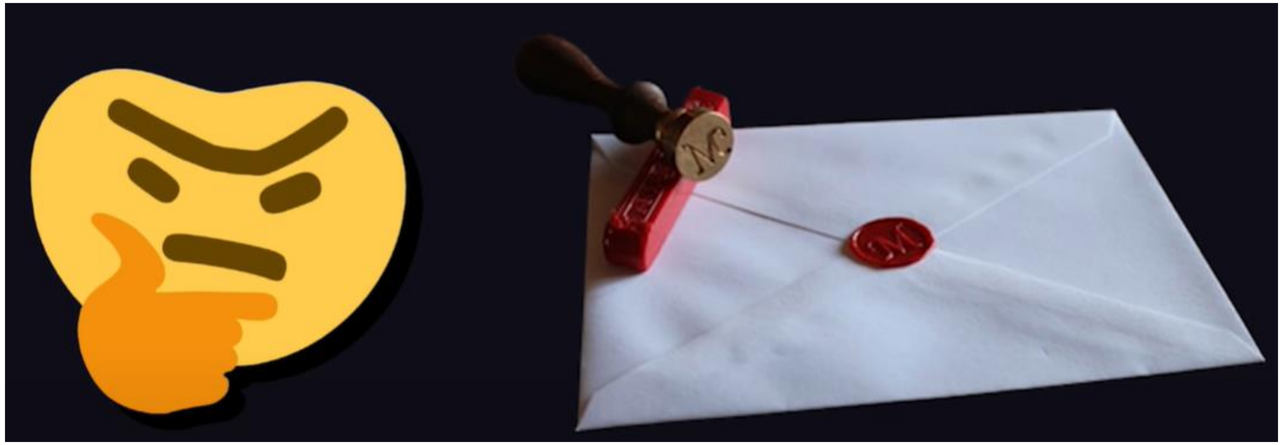
- What is the so-called digital signature?
- You want the mail is from a “trusted person”.
- So, you want them to sign the letter with their blood (their DNA is on it)
- But this still can be tempered with. (i.e. What if someone get killed? So, their ‘blood’ can be re-used?)



A Javascript framework developed by Google

Signing

- Also, they put a special seal on it.
- If it is broken, the letter has been tampered with



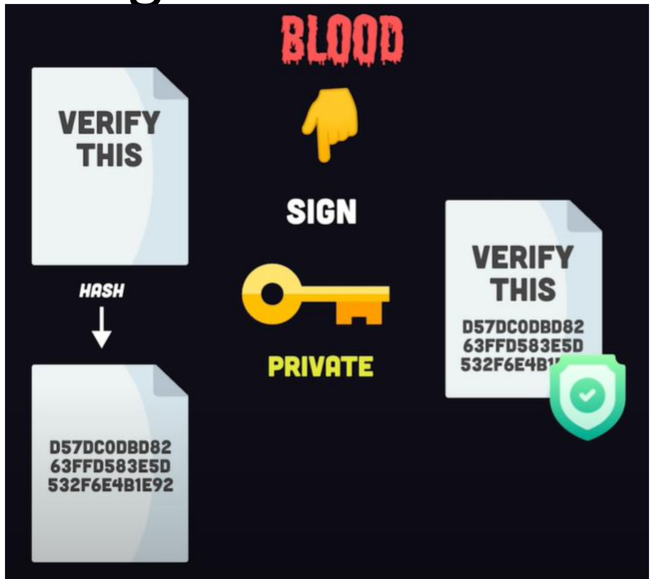
- Oh! Yes! Then we have dual protections!

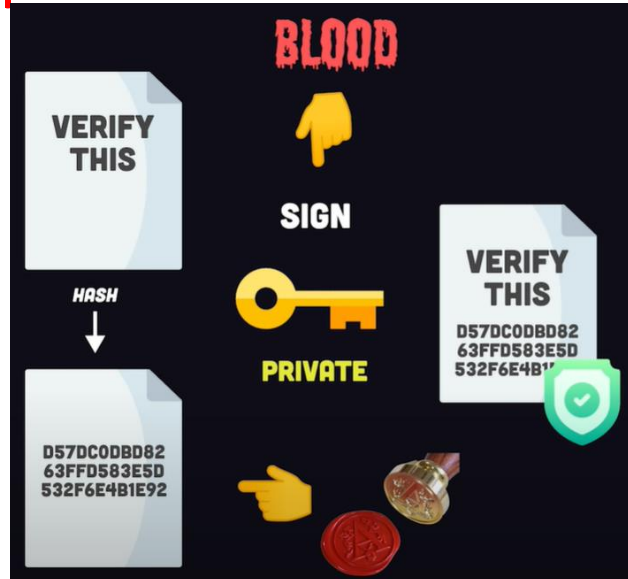
Signing

- Digital signature works in the similar way.
- The sender of the message will use their private key to sign a hash to the original message.
- The private key guarantees the authenticity like the blood
- And hash (seal) guarantees the message cannot be tampered with
- If the message is tampered,

it will produce the entire different

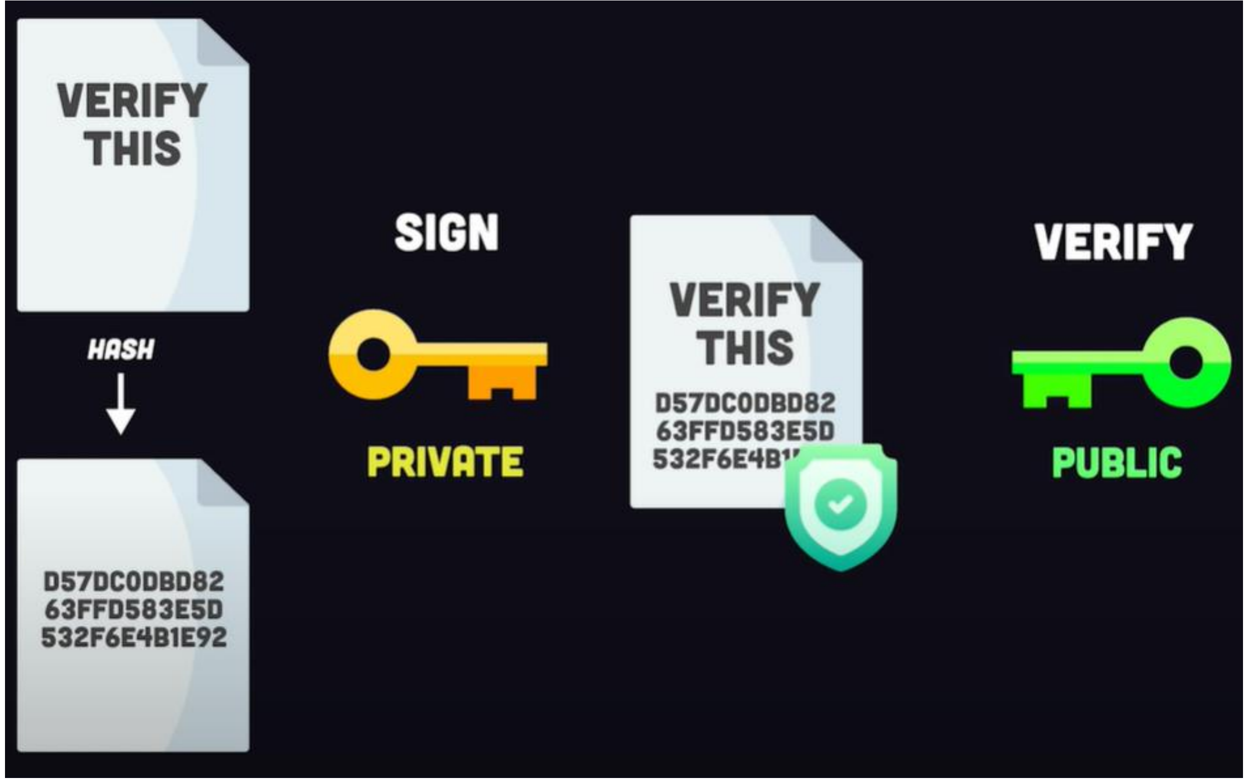
signature





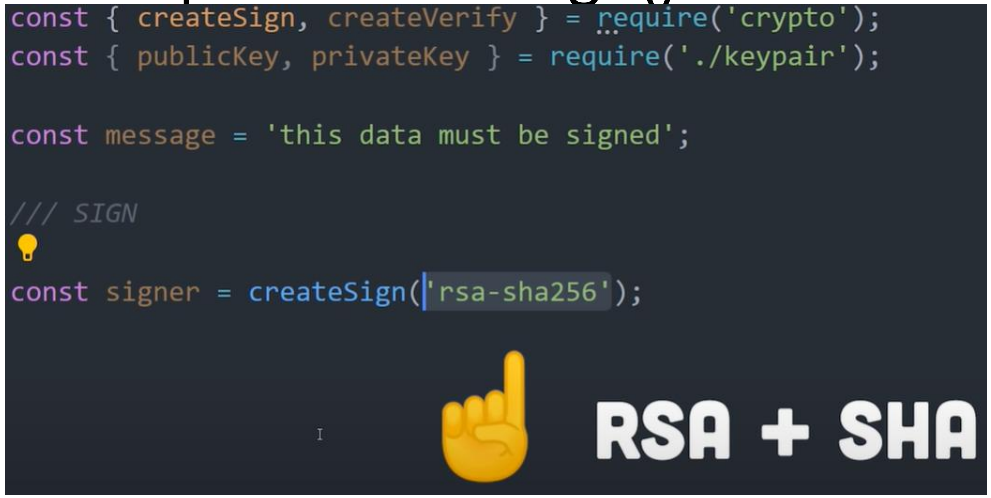
Signing

- The recipient can use the public key to verify the authenticity of the message

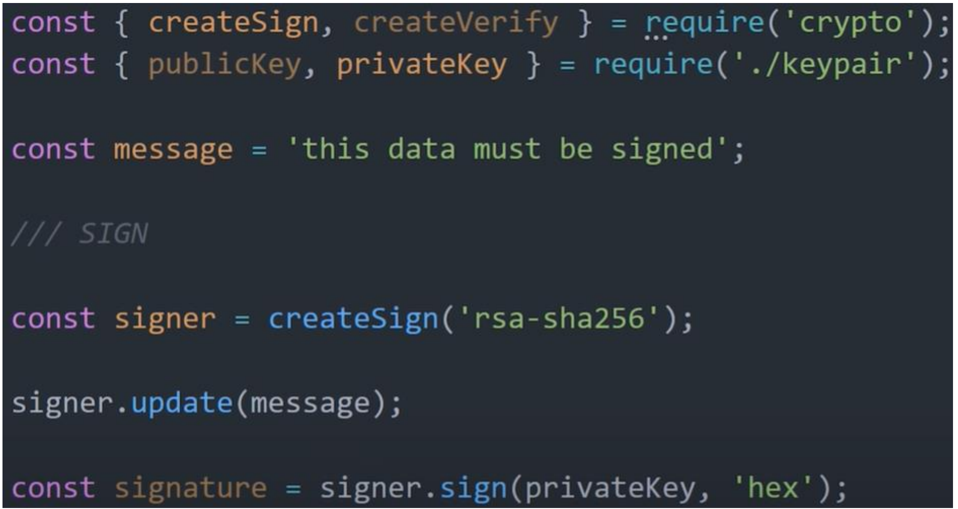


Signing

- Let’s see the code! First thing of all, we need to have createSign() and createVerify() these 2 functions.
- Setup our createSign() function by using ‘rsa-sha256’



```
const { createSign, createVerify } = require('crypto');
const { publicKey, privateKey } = require('./keypair');

const message = 'this data must be signed';

/// SIGN

const signer = createSign('rsa-sha256');
```

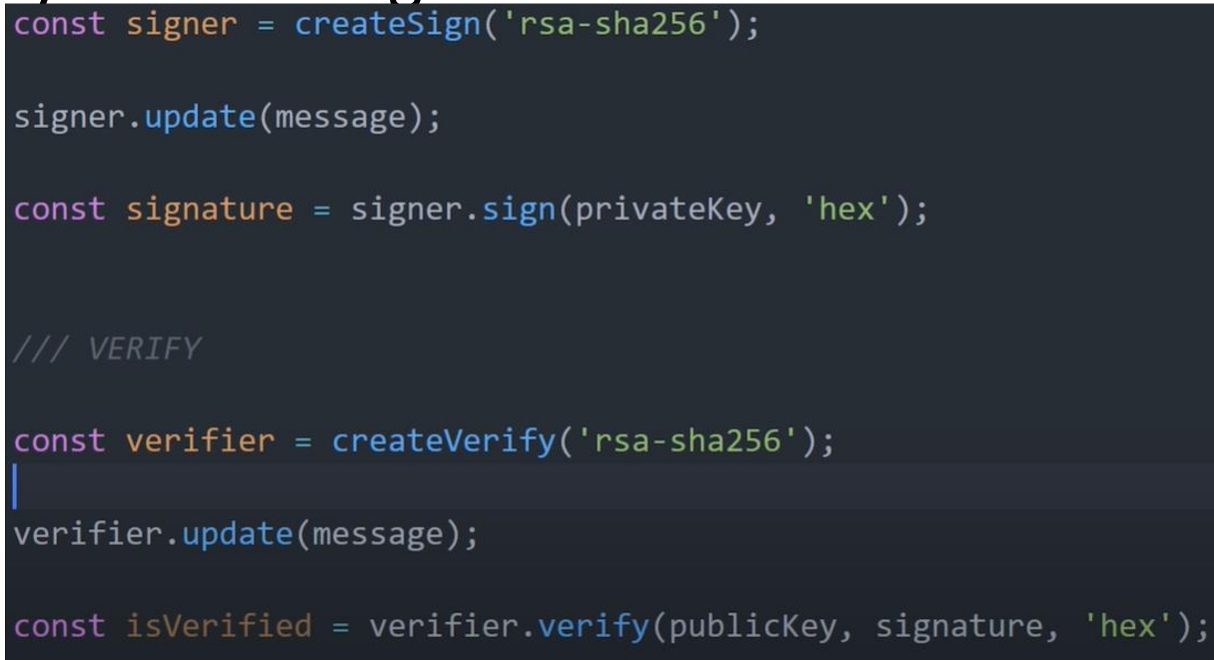


```
const { createSign, createVerify } = require('crypto');
const { publicKey, privateKey } = require('./keypair');

const message = 'this data must be signed';

/// SIGN

const signer = createSign('rsa-sha256');

signer.update(message);

const signature = signer.sign(privateKey, 'hex');
```

- With the signer to update the original message we want to sign
- Then, do the signing action!

Signing

- We can now sign the message and send to someone.
- When the recipient get it, they can create a verifier.
- Use the verifier to update the message just received
- Then, the verifier will be used to verify the message with “sender’s public key”
- If the message changed, the verifier will fail → That means, the signature is changed



```
const signer = createSign('rsa-sha256');

signer.update(message);

const signature = signer.sign(privateKey, 'hex');


/// VERIFY

const verifier = createVerify('rsa-sha256');

verifier.update(message);

const isVerified = verifier.verify(publicKey, signature, 'hex');
```