Public-key Infrastructure - CRY Assignment #2

José Donato, donato@student.dei.uc.pt, 2016225043

***Abstract*—Public-key infrastructures (PKIs) is what make asymmetric cryptography possible. If there were no PKIs we couldn't be sure that the public key of a person really belongs to the person claiming that. However, there is a big problem: these infrastructures rely on a Centralized Certification Authority and we need to trust this Authority. Blockchain applications are trying to decrentralize products that were before centralized. It is happening with payments, supply chains and many others. What if we try to decentralize PKIs? [1]**

I Introduction

In this assignment I tried to provide a proof of concept of a blockchain-based Public-key infrastructure. I start by explaining, in a first part, some core fundamentals about PKIs and blockchain. Then, in a second part, I show the application developed with explanations about its implementation. The application is live on: https://pki-frontend.now.sh/.

II Fundamentals

II-A Public-key Infrastructures

This type of infrastructures makes public-key cryptography possible. It handles the authentication of identities over the internet, i.e., imagine a person Bob that claims to own a certain public key. It is the PKI that tells us if we can trust that this certain key belongs to Bob or not. Each PKI has one or more Certification Authorities and is this centralized component that emit and keep record of the certificates (more on certificates on the next section) [2].

II-B PKI - Important Components

Certificates are the component that contains all the information about a public key and its owner. This information includes the validity of the certificate, i.e., until when it is valid. It also includes other important details such as which is the usage of the certificate or which Certification Authority signed it.

As said before, the Certification Authorities are the entities that sign and emit the certificates.

Registration Authorities are the ones responsible for validating the identity of the users.

Finally, there are two big datastores. One contains all the active certificates and the other contains the ones revoked or non-active and other important information.

II-C Blockchain

A blockchain, as the name implies, is a chain of blocks. The first block is connected to the second one (and the second has an indication of the first block, normally the hash of the previous block), the second one is connected to the third (and the third block has an indication of the second) and so on. This way, they are connected in both ways. Because of this strong connection between the blocks, it is impossible to change them in a blockchain.

Transposing the PKI to this environment, each block could contain one or more certificates. This way, we would have a decentralized, immutable and trustable store of certificates. It is a possible implementation for PKIs.

III Application

III-A Introduction

The application is all done with a type-safe language: TypeScript (https://www.typescriptlang.org/).

As normal in web applications, it is separated in a backend and frontend:

- a backend server done with Apollo GraphQL (https://www.apollographql.com/), Express (https://expressjs.com/) and Typeorm (https://typeorm.io/). In the backend I developed a simple blockchain to store certificates.
- frontend was done using ReactJS, a frontend framework (https://reactjs.org/), and consumes an apollo graphQL API with the help of apollo boost client (https://www.apollographql.com/docs/react/get-started/).

Since the application deals with user credentials, I have deployed a docker container with the backend to heroku (https://www.heroku.com/) and the react app to zeit (https://zeit.co/). This way, the application got HTTPS without the need of dealing with SSL/TLS certificates myself.

The application is live on the following url: https://pki-frontend.now.sh/ and all the source code is available in https://gitlab.com/pki1.

III-B Blockchain Data Structures & Functions

Regarding the blockchain, I have three different structures: BlockChain, Block and Certificate.

```
class BlockChain {
  chain: Array<Block>;
  difficulty: number;
}
```

The BlockChain has an array of Blocks and a difficulty number. This difficulty number will tell how much hard will be to had a new block to the chain (more on this later).

```
class Block {
  timestamp: number;
  user: string;
  certificate: Object;
  hash: string;
  nonce: number;
  previousHash: string;
}
```

Each Block of the Chain has:

- the timestamp with the time of its creation
- an identifier of the user that created it
- an object of the class Certificate (each Block has a Certificate but the blockchain can be optimized where one block has multiple certificates like occurs in Bitcoin where one block has multiple transactions)
- a hash of the block that is calculated using the SHA3 hash function from cryptojs library (https://github.com/brix/crypto-js) by joining all attributes of the block:

```
calculateHash() {
    return crypto
    .SHA3(
      this.previousHash +
      String(this.timestamp) +
      this.user +
      JSON.stringify(this.certificate) +
      String(this.nonce)
      )
    .toString();
  }
```

- a nonce: it is a number that is incremented until the hash has the difficulty number of zeros in the beginning. For example, if the difficulty indicated in the BlockChain structure is 2, the nonce only stops being incremented when the hash of the block is "00"+rest_of_hash. This logic is achieved with mineBlock function:

```
mineBlock(difficulty: number) {
  while (
    this.hash.substring(0, difficulty) \
    !== Array(difficulty + 1).join("0")
  ) {
    this.nonce++;
    this.hash = this.calculateHash();
  }
}
```

- a previousHash: an indication of the hash of the last block in the chain

```
class Certificate {
  version: string;
  user_identity: string;
  algorithm: string;
  validity: number;
  valid: boolean,
  pub_key: string;
  issuer_id: string;
  usage: string;
}
```

The Certificate class is what is stored inside each Block. It contains:

- version: indicates the version of the certificate, can be useful to later implementations to revoke/change certificates (remember blockchain is immutable so blocks cannot be changed or removed, a possible solution is to add a new Certificate with a new version and go through the blockchain to see which Certificate has the latest version)
- user_identity: which user the certificate belongs to
- algorithm: a string containing the algorithm used to create the pair of public and private keys. In my implementation I am using one elliptic curve algorithm ECDH with the curve P256
- validity: contains the date in seconds until the certificate is valid
- valid: boolean that tells whether the Certificate is valid or not (to use with the logic explained before in version item)
- pub_key: a string containing the public key compressed generated by the client using ECDH (the private key is stored in the computer of the user)
- issuer_id: not currently being used in this implementation but may be used later. Tells who issues the certification
- usage: a string that explains for what this certificate will be used (for example "VPN")

There are two more important functions in my implementation of the blockchain:

- function to add a block to the blockchain

```
addBlock(newBlock: Block) {
 newBlock.previousHash = this     \
 .getLatestBlock().hash;
 newBlock.mineBlock(this.difficulty);
 this.chain.push(newBlock);
}
```

- function to get all the certificates from the blockchain or only the ones from a user (if a username is provided)

```
const getCertificates =       \
(blockchain: BlockChain,      \
username?: string) => {
  return blockchain.chain
    .filter(block =>
      username
        ? block.user === username    \
        && !isEmpty(block.certificate)
        : !isEmpty(block.certificate)
    )
    .map(({ certificate })      \
    => certificate);
};
```

All this code regarding this part can be found in the following url: https://gitlab.com/pki1/pki-server/blob/master/src/blockchain.ts.

III-C Sign Up & Authentication & Session

To store the users' password in the database the winner of password hashing competition was used: Argon2 (https://www.npmjs.com/package/argon2) [3]. Instead of storing the plain text password (very bad practice) or using solutions with known vulnerabilities like bcrypt [4], I store a hash of that password using the hash function named Argon2 from the library node-argon2 (https://github.com/ranisalt/node-argon2).

To create the hash is as simple as:

```
const hash = await argon2.hash("password");
```

To compare the plaintext password to perform login operations we just need to use verify function from argon2 that return a boolean where the password provided by the unauthenticated user matches the hash stored in the database or not.

```
const IS_VALID = await argon2     /
.verify(hash, "password");
//check if plaintext password
//is equivalent to hash
```

Session is an important topic in web applications. It is the set of data structures that are used to track the state of the user's interaction with the application [5]. To handle session, I used JSON Web Tokens (JWTs).

Right after user signs in, it is sent from the server a cookie (this cookie has the httpOnly flag activated to prevent XSS attacks) that contains a JWT which includes the users' username, id and name. This way, we can identify a request and match it to the user that sends it.

If a request is sent without the cookie we already know that the user isn't logged in and redirect him directly to the login page. Protected pages, such as Homepage, can only be seen if the user is authenticated, i.e., sends the request to the server with a valid JWT token.

Also, there is the case where a request has an invalid JWT (for example, a expired JWT) can happen. To prevent this, the jsonwebtoken library (https://www.npmjs.com/package/jsonwebtoken) has a method called verify that checks if a JWT is valid or not. When user logs out, the jwt cookie is cleared.

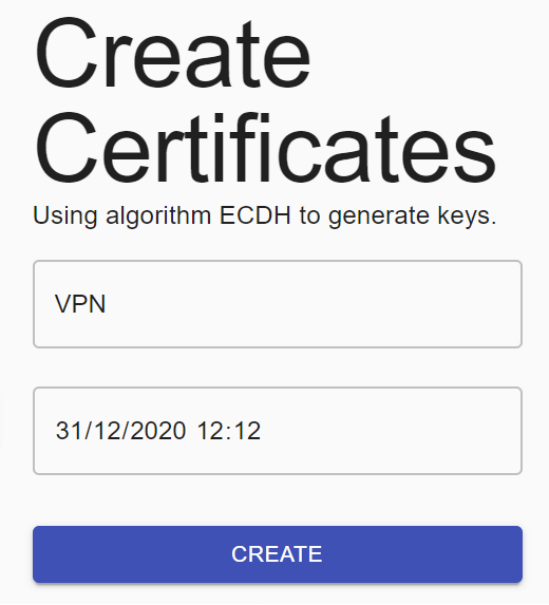
III-D Applications Features

In the time of this report, there are three features for the application after users sign in:

1) Create certificates
2) Search other users certificates
3) See own certificates

1) Create certificates

Users can create their own certificates in the homepage.



In the first field the user needs to write the usage of the certificate, i.e., for what purpose it will be used. In the other field, is the validity, i.e., until when the certificate will be valid.

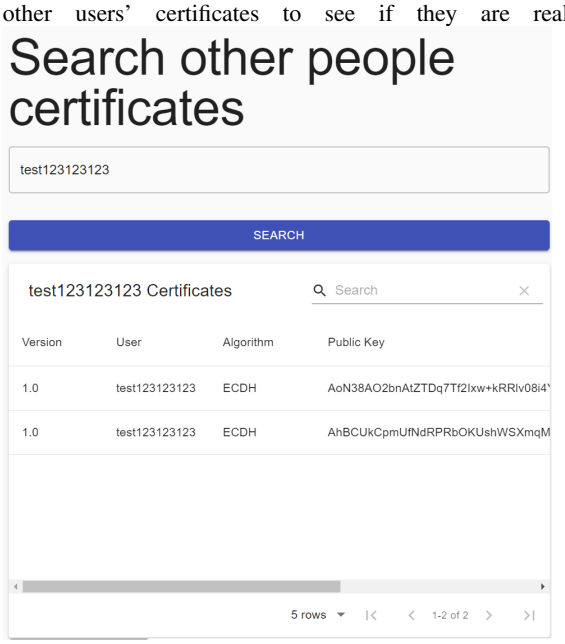
After the user clicks the create button, the pair of public and private keys are created with the help of the crypto module present in the browser (window.crypto.subtle). This module supports ECDH algorithm with the P-256 elliptic curve.

```
var ecdh = await Crypto.generateKey
  ({ name: 'ECDH', namedCurve: 'P-256' },
  true, ['deriveBits'])
```

After this, we have a public and a private key that we can send to the server and store locally, respectively. With a GraphQL mutation, the certification information is sent to the server, validated and added to the blockchain if everything is correct. On the client side, a ".txt" file is created with all the certificate information (including the private key that must stay safe and private).

2) Search other users certificates

Other thing authenticated users can do is to search other users' certificates to see if they are real or not.



After the user types a username and search for a user's certificates, another GraphQL mutation is called. The client requests to the server all the certificates from a certain user in the blockchain (in the server the function named "getCertificates" explained in the blockchain section is called). If there are matches, they are displayed back to the user.

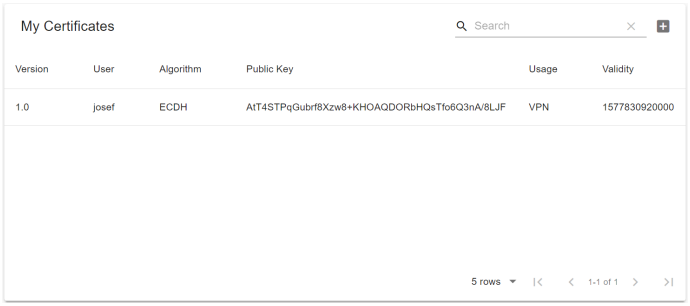
The following code is a example of the mutation that is called in the server from the client. First checks if the user is authenticated and only runs if so. If successful, searches the certificates from a certain username provided by the client in the blockchain and returns an array of CertificateS.

```
@Mutation(() => [CertificateS])
@UseMiddleware(isAuthenticated)
searchCertificates(
 @Arg("username") username: string,
 @Ctx() { blockchain }: any
) {
 const certificates = getCertificates
  (blockchain, username);
 return certificates;
}
```

All the code regarding GraphQL resolvers can be found in the following file https://gitlab.com/pki1/pki-server/blob/master/src/Resolvers.ts.

3) See own certificates

Finally, the users right after he logs in can see his own certificates.



As soon as the page loads, a GraphQL query is made to grab all the certificates from the username that is logged in. The code is similar to the search certificates presented before:

```
@Query(() => [CertificateS])
@UseMiddleware(isAuthenticated)
myCertificates(
@Ctx()
 { payload, blockchain }: any) {
 const { username } = payload;
 const certificates = getCertificates
  (blockchain, username);
 return certificates;
}
```

In this one, instead of using the username provided by the user, the server uses the username stored in the JSON Web Token payload (the username of the user logged in).

IV Conclusion

To sum up this report, although this application is only a ”proof of concept”, we can see the potential of blockchain and its application in Public-Key Infrastructures. With this assignment, I was able to learn about some new technologies, frameworks and how to create a simple blockchain using no third-party libraries (only crypto-js to create the hash for the blocks). I also had the opportunity to increase my knowledge about Public-key Infrastructures and its components.

References

[1] Anders Wallbom David Sanda Alexander Yakubov, Wazen M. Shbair and Radu State. A Blockchain-Based PKI Management Framework. https://www.researchgate.net/publication/323692746_A_Blockchain-Based_PKI_Management_Framework.

[2] Fernando Boavida and Mário Bernardes. INTRODUÇÃO À CRIPTOGRAFIA. https://www.fca.pt/pt/catalogo/informatica/seguranca-ciberseguranca-protecao-de-dados/introducao-a-criptografia/.

[3] password hashing. Password Hashing Competition. https://password-hashing.net/.

[4] Snyk. bcrypt vulnerabilities. https://snyk.io/vuln/rubygems:bcrypt.

[5] Marcus Pinto Dafydd Stuttard. The Web Application Hacker’s Handbook. https://portswigger.net/web-security/web-application-hackers-handbook.