Medical Test Records

Segurança Informática em Redes e Sistemas

Taguspark G02

Autores Rafaela Jorge Timóteo, nº 90773
Ana Marta Sousa Contente, nº 98643
Alexandru Pena, nº 98742

december 11, 2020

1 Problem

The Medical test Records project approaches two main issues that hospital information systems face.

The first issue, is controlling access to a patient's profile. A patient's profile has associated with it a group of information with different levels of sensitivity, such as age, name, blood type, diseases, family records and others. A patient's name is not extremely sensitive so most employees can access it. However, for instance,the patient's family records do not need to be accessed nor should be accessible to most of the hospital staff, for example to volunteers.

Given this scenario, the access control mechanism must be able to correctly determine, and enforce the correct policies so that different parts of a patient's profile are only available to whom actually requires them to fulfil one's obligations to the patient, regarding the performed role and in the correct context.

The second problem comes from medical tests being analysed in different facilities, like partner labs, that are not part of the hospital infrastructure, but require that both are interconnected in a way that enables all sides to identify each other and communicate in a secure way.

1.1 Requisites

Given the previous problems, the implementation of this project requires the following security guarantees:

- A fine grained access control and secure authentication mechanism.
- Authenticity of all communications with the information systems.
- Confidentiality of all communications with the information systems.
- Integrity of all communications with the information systems.
- Non-repudiation of the communications with the partner labs.

1.2 Trust assumptions

There are three main entities in this project:

- The Hospital System

  The system that will store all the patient data and tests results received from the labs.
- The Partner Labs System

  The partner lab can be internal, or external to the Hospital network. The internal lab is in the same network as the Hospital, but as the external lab, they will have different infrastructures from the hospital, which means they will have their own certificates and machines.
- The users of those systems, the staff.
- The certificate authority.

In case of the staff there is a full trust assumption that they won't have their credentials stolen, shared or compromised in any way, and as such we will not provide any mechanism to rapidly revoke those credentials.

Regarding the certificate authority, there is a full trust assumption that the private key used to issue/sign certificates won't be stolen and as such, we won't have a mechanism to revoke certificates.

Concerning the access control components which will be explained further (PDP,PEP...), there is a full trust assumption that those components will be correctly and logically well implemented, and won't allow any type of unauthorized access, but at the same time correctly serving good requests.

All communications with the partner labs (internal or external) will use HTTP requests with a custom security protocol, and will be considered secure and resistant against attacks like replay, tampering and eavesdropping.

There is also a full trust assumption, that all the libraries and frameworks used are error free and won't give any type of access to an attacker.

2 Proposed solution

2.1 Overview

The implementation will be separated in three API's, an hospital API, a laboratory REST API and an Access Control REST API, all developed using the framework Spring MVC[1].
All the components involved in this project, will be developed using the Java[6] programming language.

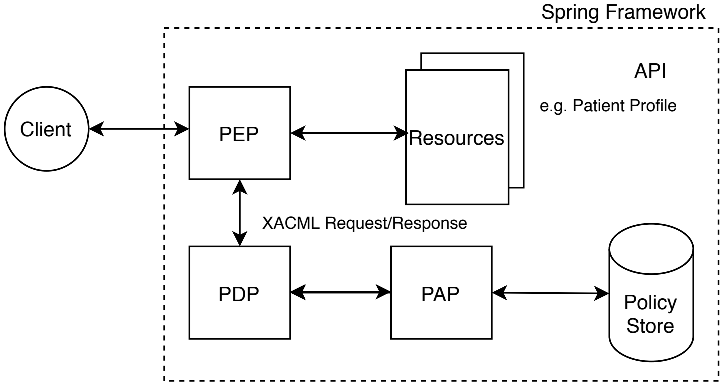
All requests received by the Hospital API will be intercepted by an access control mechanism and redirected to the access control API for validation. This access control API, for requests like getting a patient treatment will require a group of extra information to determine if the requests should be permitted or denied (e.g. role of the person making the request). This set of information is added to the access control request by the Hospital API based on the presence (or lack of) of a token, this token is provided by the Hospital API on authentication to an end-user (e.g. Doctor) if the correct set of credentials (username:password) were supplied during the "login" action. The token is transported in the "Authorization" header of the HTTP request from the end-user to the Hospital API. In reality the Hospital API, for optimization, will know that if the request is not to the "login" action and doesn't possess a token, the PDP will never PERMIT the action to continue, so it denies the request before interrogating the PDP for authorization, to avoid a useless trip.

The database will store the hash version of the password using the Bcrypt[2] algorithm, which already salts the passwords and avoids rainbow table attacks. The API will use a set of methods already provided by the spring-security-crypto[7] module to validate the received credentials by hashing them and comparing to the value already stored in the database. BCrypt provides a set of mechanisms that allows to increase the complexity of the algorithm to use, to increase the time of processing and delay brute-force attacks. This complexity can be increased with time to adapt to the evolution of processing power on computers.

The access control mechanism is composed by 4 components:

- PEP - Policy Enforcement Point
- PDP - Policy Decision Point
- PAP - Policy Administration Point
- Policy Store

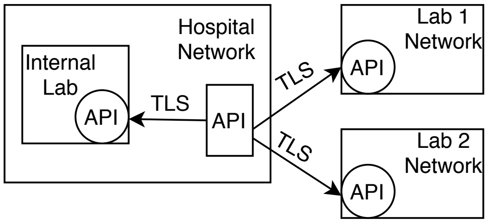
The PEP will send a XACML[3] request to the PDP, the PDP will then based on the policies defined by the system return a XACML response which will dictate the action (ACCEPT or REFUSE) that PEP will enforce. If the result is ACCEPT, the request will continue and read or write the requested resource, otherwise the request will be dropped and replied with a 403 UNAUTHORIZED.



The PEP will be contained on the Hospital API and the rest of the components will be part of the Access Control API, this way the access control mechanism is externalized providing a more robust security in case the Hospital API gets somehow compromised.

2.2 Deployment

The following figure includes the distinct machines of this system, and as mentioned they will be interconnected through secure channels (TLS or custom protocol).



A staff member of the hospital , for example a doctor, will start by sending a request to the Hospital API requesting the test results of a given patient. The Hospital API will then make a request to the Lab API requesting the actual test results data.

The deployment scripts will use Vagrant[4] to manage and deploy the VM's and Ansible[**?**] for infrastructure providing (POSTGRESQL, JDK, CA installation...).

In this project there will be used only one external lab, but thats only for ease of management, as adding more lab's is just a matter of editing the deployment scripts. Also, in the diagram where there is an API circle, there are actually two machines, one with the Hospital API which contains the PEP and the other with the rest of the access control components (PDP,PAP...), as explained previously.

2.3 Secure Channels

All communications between the end-users (staff) and respective API's will run with TLS to provide confidentiality, integrity and freshness. TLS, between those entities will be provided by the Spring framework[1], which is the framework and language that will be used to develop the project. As Spring default cipher suites already use by default the key exchange algorithm Diffie-Hellman, we won't override this configuration, as we want the Perfect Forward Secrecy property that comes from it.

The communications between the Hospital API and partner lab API will be under a custom made protocol which will provide the same properties as a TLS channel (confidentiality, integrity and freshness).

The secure channels will require the definition of a certificate for each entity, we will consider that the certificates are installed manually by an administrator (during the provisioning process) on each respective machine. We will have a made-up CA, which we will use to issue and sign the hospital, lab and access control API's certificates (including labs inside the hospital)(certificates created with OpenSSL[5]). Those certificates alongside with the secure communication channel, will be used to digitally sign the sent tests data to guarantee its authenticity at any time. In order to validate the certificates received by each API during the TLS Handshake, each machine will have the root CA certificate already installed, manually, by an administrator (again, during the provisioning process).

2.4 Secure Protocol to Develop

The custom protocol to develop, will enable secure communications between the Hospital API and partner lab, providing almost all of the same properties as TLS. This protocol will work on top of the HTTP protocol and not under it like TLS, this means that the handshakes, message exchanges will all happen with HTTP requests.

In our case we will only support one cipher suite, the protocols to be used will be Diffie-Hellman[10] for key exchanging, AES-256 for block cipher and hmac-sha256 for message authentication.

When a doctor makes a test data request, the Hospital API will be in charge of starting the handshake with the Lab API, by making a request to a special endpoint with its certificate and its Diffie-Hellman parameters (which is the public key of its Diffie-Hellman KeyPair Java Object) , before exchanging any data. After the lab receives the certificate, it will verify it by checking its hostname (to see if it is actually a valid person in the hospital side making the request), expiration date and validity with the CA. If it's all ok, the client will use the hospital's DH public key to generate its own DH KeyPair, and to generate the secret following the rest of the Diffie-Hellman protocol. It will then send its own Diffie-Hellman parameters to the hospital and add a TAG (for integrity) using the same secret key. The hospital will now finish its part of the DiffieHellman's protocol, by generating the secret key and confirm the integrity of the received data.

Finished this handshake, the hospital will now make a request to the lab API encrypting the request with the secret key and adding a TAG (for integrity) using the same secret key, for more security the key shouldn't be the same, but the group didn't have time to complete that step. The Lab will then answer with the test results data encrypted and protected using the same generated secret key. Finally, all the messages will also have an NONCE combined with a TIMESTAMP to protect from message replay attacks.

3 Results

The group started with developing the deployment and provisioning scripts using the mentioned tools (Vagrant and Ansible), then creating the necessary cryptographic material (certificates, keys and keystores) with OpenSSL and configuring the necessary services to run the API's automatically with the systemd system.

Then proceeding to create the API's and the basic functionalities described in the API's documentation file (docs/API.md), all the endpoints were implemented and their data access layer concluded. Thenceforth, is possible to make requests to the API for example to obtain a patient's name, diseases, treatment... but without any kind of limitation or access control, with that in mind the access control mechanisms and its API were implemented next.

As previously described the access control mechanism is structured into different points (Policy Enforcement Point, Policy Decision Point and Policy Administration Point), the PDP and PAP were implemented in the PDP/Access Control API, successfully externalizing the policies in another place. The PEP was also successfully implemented in the Hospital API and able to communicate in a secure way with the rest of the externalized components using the XACML framework rules. In our implementation of the XACML framework, we weren't extremely rigorous with the implementation of all individual components, specifically with the PIP, which is an additional component that the framework defines. This component is supposed to act as a source of attribute values (e.g. subject, environment...). In our case this component is blended in with the PEP, because the PEP is already providing all attributes based on the token presented and operation mode (pandemic or normal). If this project was scaled to a real world application where there can be dozens of attributes our solution wouldn't scale well without the Policy Information Point.

The group also took the freedom to adapt the ”language” of the policies file. In the XACML framework, more recent versions define different profiles for the structure of its messages, for example one of them is the JSON profile, this profile was used to adapt the structure of the REQUEST and RESPONSE messages from XML to JSON, this is done because in current world API's, JSON structure is very comfortable and easy to use. Unfortunately as the policies tend to be in the backend the XACML team didn't find the necessity to also define a JSON profile for it, but as we are already using the module Jackson [8] for parsing between JSON strings and input model objects, we decided that to facilitate the development we could define the POLICIES also using a JSON profile, even thought there is none officially defined. This doesn't mean we are disrespecting the XACML Framework rules, but simply translating the structure from XML to JSON, keeping always the original properties defined. An example of project who also does this is the implementation of Authz [9]

The last step to complete the planned basic version was using the provided TLS components by Spring to create secure channels between the end-users and API's and between the Hospital API and Access Control API. And so was done, using the default cipher suites because they already where using the Diffie-Hellman key exchange protocol.

In the next phase we created the necessary mechanisms to digitally sign test's data (by the lab), verify the authenticity of such tests at any time (by the Hospital when requested) and the use of Bcrypt to safely store the passwords in the database. As discussed in previous sections, this mechanism already salts passwords to avoid rainbow table attacks and slows down the process by using a certain number of rounds to make the process of guessing a password very slow.

The last step to complete the planned intermediate phase was the implementation of the custom security protocol. We used the JCA library to do this implementation. Our protocol works on top of HTTP, therefore the headers and URL are not protected, we only protect the body of the request and we do such by first doing an handshake process that exchanges cryptographic material like certificates, random strings for RSA or prime numbers for Diffie-Hellman, this random string is sent encrypted by the lab API playing as a server during the handshake, it's used to generate the session key used to encrypt data for confidentiality and to generate the TAG that provides integrity of the messages, same for the prime numbers. As previously mentioned, the group realizes that having only one key is not an adequate solution and we should have different keys for integrity and confidentiality, like for example TLS, that uses 4 different keys, 2 for confidentiality (Server-Write and Client-Write) and 2 for integrity (Server-MAC and Client-MAC). Our first key exchange mechanism was RSA based on the Hospital private and public key, but if the hospital ends up losing its private key, an attacker can generate all session key's and observe the traffic in plaintext that was sent in between the hospital and lab, if it was recorded. Therefore we decided to replace the RSA key exchange algorithm by a Diffie-Hellman implementation which grants Perfect-Forward Secrecy. Our implementation, as mentioned previously, uses AES-256-CBC for symmetrical key encryption and HmacSHA256 for integrity and is protected against replay attack's by using a combination of NONCES and Timestamps. To synchronize all the systems in play we used the protocol NTP (Network Time Protocol), it's configuration is deployed during the provisioning of the infrastructure.

Unfortunately, none of the goals in the advanced version were actually implemented because of time constraints.

Bibliography

[1] Pivotal Software, Spring MVC
`https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html`

[2] Niels Provos and David Mazières, bcrypt
`https://auth0.com/blog/hashing-in-action-understanding-bcrypt`

[3] OASIS, XACML
`https://www.oasis-open.org`

[4] HashiCorp, Vagrant
`https://www.vagrantup.com/`

[5] The OpenSSL Project, OpenSSL
`https://www.openssl.org/`

[6] Oracle, Java
`https://java.com/`

[7] Spring
`https://docs.spring.io/spring-security/site/docs/5.0.x/reference/html/crypto.html`

[8] FasterXML
`https://github.com/FasterXML/jackson`

[9] Authzforce
`https://github.com/authzforce/java-xacml-sdk`

[10] Diffie-Hellman Java Implementation
`https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html`