TÉCNICO LISBOA

Medical Records

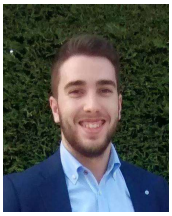
Final Report

Network and Computer Security

Alameda

Group 39

Mafalda Ferreira 81613



Leonardo Epifânio 83496

Pedro Lopes 83540

Problem

A patient's medical history is highly sensitive data. Medical records ease the process of diagnostics and ensure its quality, which helps the clinical staff to treat fast and accordingly. This information must be kept private and only responsible staff can access it. However, every health care institution must have access to this kind of information so that the patients can receive healthcare anywhere, at all times.

Therefore, the data should be protected from outside agents (outside of medical institutions) and from unauthorized personnel inside the institutions, on the other hand, the medical records should always be available.

Requirements

To help identify security requirements we first specify the application and cloud service requirements (functionality).

Client Application Requirements

- As a user, I must be able to read/write medical records, therefore:
  - I must be able to send requests to the system.
  - I must be able to receive responses from the system.
- As a system administrator, I must be able to change/adjust user privileges.

Cloud Service Requirements

- As a system, it must be able to receive client requests.
- As a system, it must be able to respond to client requests.
- As a system, it must log the requests.
- As a system, it must store, access and be able to modify medical records.
- As a system, it must be able to temporarily block an account if wrong credentials were given.

Security Requirements

- As a system, it must guarantee the confidentiality and integrity of the medical records.
- As a system, it must guarantee that only working personnel from medical institutions and citizens have an account.
- As a system, it must guarantee that there is only one account per citizen.
- As a system, it must authenticate users.
- As a system, it must mitigate brute force attacks to the authentication system.
- As a system, it must minimize the impact of faults within the system.
- As a system, it must minimize the impact of attacks inside of the system.
- As a system, it must stop the access to the medical records if the user doesn't have privileges.
- As a system, it must only accept that user A changes user B's privileges, if user A is a system administrator.
- As a system, it must verify that the users have the privileges they claim to have.
- As a system, it must guarantee the confidentiality and integrity of the communications with the web application.
- As a system, it must prevent web attacks such as XSS, CSRF, BO, replay attack and code injection (i.e. SQL).
- As a user, I cannot repudiate my actions.

Proposed Solution

We designed a solution that tries to mitigate all the possible security threats, taking in account the time we have available to develop the system, therefore, this is not a perfect solution, but it is as robust as we can make it.

We're trying to build a system that is scalable and that can be further developed, also acknowledging the problems that are left unsolved by this solution in the following paragraph.

The system will be composed by the client applications that communicate through the internet to the cloud service. Throughout the solution we have the assumptions that the communication channels within the cloud service are secure, that no byzantine faults occur in the servers, and that the servers never fail at the same time. Any of these assumptions represent unsolved problems, which resolution would be infeasible given the time restraints for the development.

In the Basic, there is also the assumption that the communications between the client apps and the cloud service are done through secure channels.

The final solution is shown in *Figure 1* of the appendix.

Basic solution:

In this first phase our goal is to design the system to have full functionality, disregarding network security mechanisms, so we will use HTTP in the communication between the client and the cloud. Through the client app, the user will be able to send requests in the system. This app will be the interface to all users, independently of their privilege. However, some functionality may not appear in the interface if some actions don't match with the user's privileges.

The service parses the request and generates a XACML authorization request, that is fed into the AuthzForce PDP engine. The PDP evaluates the request against the policies it is configured with, and, if needed, also retrieves other attribute values from the database, in order to perform its decision. This decision is then sent back from the engine, and depending on the result, the server generates a response and sends it to the client app. This interaction is shown in the *Figure 2* of the appendix.

We chose to use ABAC as the access control model, since it is a fine-grained model that can control access based on situational conditions, such as time, place, relations etc..

Intermediate solution:

In the intermediate phase, we will improve the communication between the Client and the Cloud by replacing the HTTP protocol with HTTPS, which consists in HTTP over TLS. When the client initiates an HTTPS session with the Front End Server, they perform a TLS handshake, starting with the hello phase, in which they agree (among other things) with the cipher algorithm that they will use later. After that, they exchange SSL certificates. These certificates are verified by a trusted CA (which we might need to setup one for testing). The client verifies the server's certificate with the public key of the trusted CA and checks its validity. The server also does the same for the client, to verify if the client is who he claims to be. The client's certificate issues his role (eg. if he's a doctor). At last, the client generates a random key according to the algorithm agreed upon in the hello phase and sends it encrypted with the server's public key, and from this point on, the communications will be encrypted using the symmetric algorithm and signed with the private keys of each one.

To guarantee confidentiality of saved files and users' attributes, the database system will encrypt the columns, each with a different symmetric key. These keys will be stored in a key wallet, and the server will hold a master key which will be used to encrypt/decrypt the key wallet.

To guarantee the integrity of the policies (since confidentiality here is not a major issue), we calculate the SHA-256 hash of the XACML file after each change, and then, whenever it is accessed, we calculate the hash again and compare them to verify if no ilegal changes were made.

Advanced solution:

In the last phase, we will implement fault tolerance with a primary-backup (passive replication) model. For this architecture, we will need to separate the server functionality. There will be a front-end server, a primary server with it's own database and a backup server with other database.

The front-end server parses the requests and forwards them to the primary server which executes the rest of the process described in the basic and intermediate solutions. In the end of that process, the primary server sends an update to the backup server (that writes the changes into its database) and then sends the response to the front-end server, which, in turn, formats it into an HTTPS response and sends it to the client.

At a constant time rate, "I'm alive" messages are sent from the primary to the backup. If the backup doesn't receive this message before a timeout, it assumes that a fault occurred in the primary server. The backup server then sends a message to the front-end stating that he is the new primary, and from this moment on, all the requests are forwarded to this server, which will now behave as the primary.

Results

The basic solution was fully implemented as planned. In the intermediate solution, the encryption in the communications between the Client and the Cloud was developed as expected, however, the encryption in the database system has a few differences. We implemented data at rest encryption, using a symmetric key for each table. The keys are stored in the database server file system, encrypted with the master key. This master key file will be needed at the start-up of the database server, but should be removed right away.

Each citizen has a Public/Private key pair, which are stored in a password-protected Java Keystore file. When a medical record is created or updated, it is signed with the doctor's private key, to assure integrity of the file, and to enforce non-repudiation.

In the other actions applied on the system data (role attribution, citizen creation, etc...), non-repudiation is itself assured by saving the name of the author along with the corresponding data in the database. For example, when a superuser adds a citizen, the identity of this superuser is saved along with the citizen that was created. Since the users are authenticated when they execute these actions, they cannot repudiate said actions.

Due to time constraints, we didn't manage to implement the XACML SHA-256 hash verification, since this was not our priority.

In the advanced solution, there were changes in the replication model. The front-end server, as it receives the request, will call a remote method (using Java RMI), on the first server that is alive in a list of known addresses. The front-end tries each address on the list, until he successfully calls the remote method. The request is then processed and the response is sent back to the front-end which will, in turn, send it to the client. There is no communication between the replica servers.

Evaluation

We implemented our access control model using ABAC, since we not only wanted to control access based on roles, but also based on situational conditions. We defined a strong set of policies that allowed us to control access to resources, using the principle of least privilege. These policies cover all the endpoints, all the roles we allow on our program and all the actions. The core of the project are the policies and their enforcement. The XACML architecture is very robust, making the access control the cornerstone of our implementation.

Our solution is resistant to several well-known attacks in the web:

- **Stack and Buffer overflow**
  Since our solution as build using Java, the memory allocation is done dynamically and with pointers, so it's not vulnerable to this attacks.

- **XSS (Cross Site Scripting)**
  Before any user-controlled input is displayed to the client, its made a sanitation (escaping html tags).

- **SQL injection**
  There are prepared statements and many characters are forbidden avoiding or, at least, minimize the possibility of second order SQL injection.

Security-wise, our focus was the users' data, especially the medical records. Since each table is encrypted with a different key, and these keys are themselves encrypted in the database server, hacking into the users' data would be a very hard job.

Contrasting with the database, the weakest point in our system is the worker server. The policies file isn't encrypted nor do we calculate its hash, so its confidentiality and integrity could be compromised if someone manages to have access to the server. In this case, although it is hard to change the policies file to do what one

pretends, a malicious agent, with the necessary knowledge could change the policies, and have access to confidential data.

Conclusion

Time management and planning are important in a project, and even though we stepped over our planning, we can conclude that, overall, all the requirements that we set in the proposal were accomplished. The system can be considered secure (for a definition of secure) and does its job.

We built a solution that can be easily scaled with high availability, and resilient against web attacks. The policies we made for the system are strict, and therefore, the main goal was achieved.

References

ALFA
ALFA is a pseudocode language to write policies that converts to XACML.We used the ALFA plugin for Eclipse by Axiomatics, which converted our ALFA policies into XACML in a xml file that could be interpreted by the authzforce tool.

Authzforce
We used the AuthzForce Core PDP engine to build the PDP engine in order to evaluate the request against the policies it is configured with. We also added a PIP, with the help of this tool, in order to fetch more attributes if needed for the policy evaluation and a Request Builder that receives the request attributes, parses the request and generates a XACML authorization request.

Spring Boot RMI
We used the Spring Boot RMI remoting package to accomplish the fault tolerance we proposed to do. We used the exporter for making beans available to the RMI clients and the proxyfactory for accessing the available RMI services.

MariaDB
We used the open source MySQL MariaDB for the database.

Java Keytool
Used for managing the Private/Public key pairs used in medical record certification.

Appendix

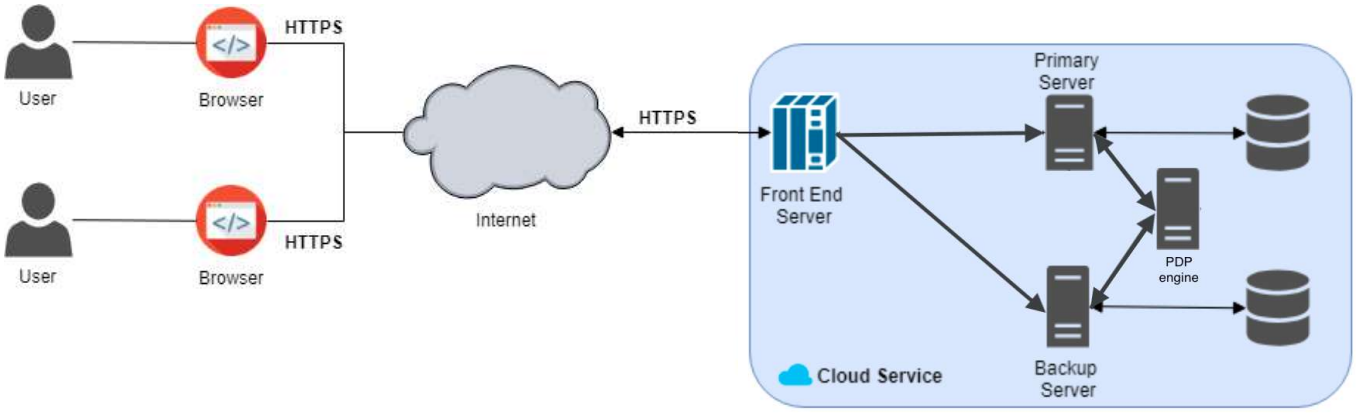


Figure 1: System Architecture

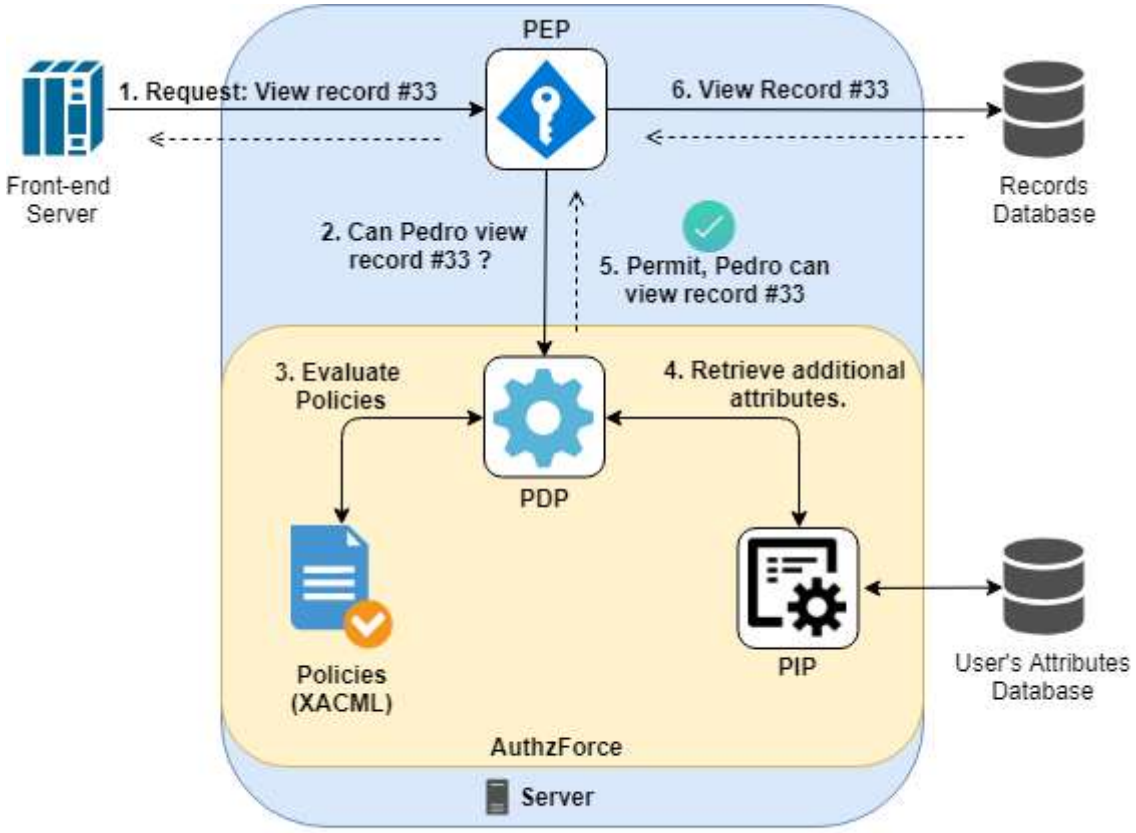


Figure 2: Control Access Architecture of the system.

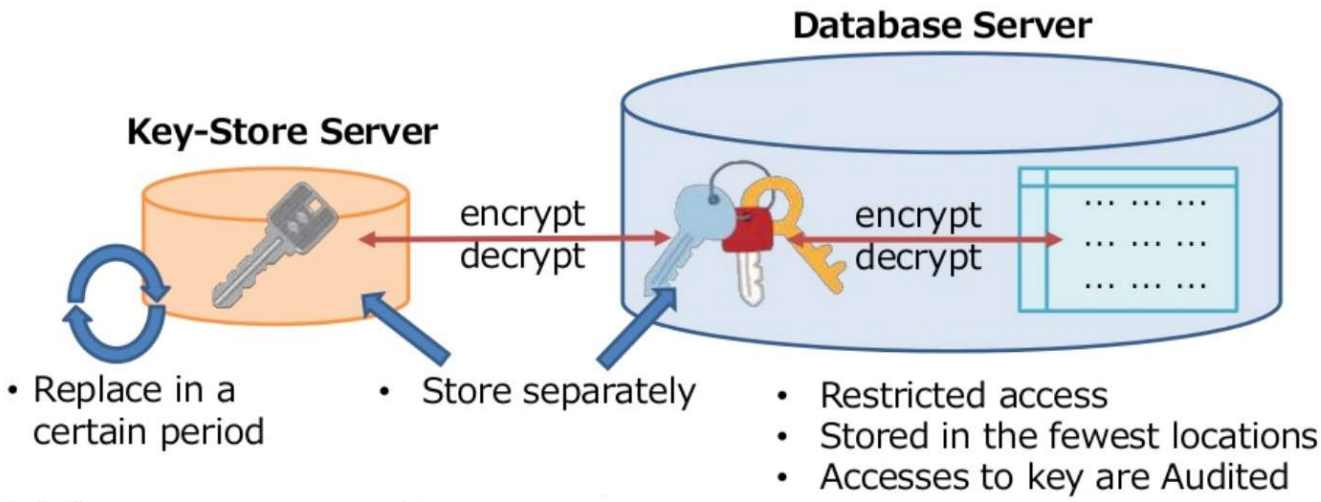


Figure 3: Data Encryption Key-management Outline.