

Medical Records

Proposal

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 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 illegal 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 its 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.

Tool References

1. **[WELL-TESTED]** MariaDB/MySQL - Database to save users and medical records.
2. **[WELL-TESTED]** MySQL Connector/J - JDBC driver to use MySQL with Java.
3. **[WELL-TESTED]** Spring Framework - Web MVC framework.
4. **[WELL-TESTED]** Log4j - Logging library.
5. **[FOUND]** ALFA - Pseudocode Language to write policies that converts to XACML.
6. **[FOUND]** Authzforce - XACML Java API.

Work Plan

	Mafalda Ferreira	Leonardo Epifânio	Pedro Lopes	Observations
Week 7 29/10 - 4/11	Setup visual outline of the web site. Research AuthzForce API.	Setup Database. Research AuthzForce API.	Create git repository & setup base project. Research AuthzForce API.	Basic Solution Implementation
Week 8 5/11 - 11/11	Write policies in ALFA. Create the module to interact with the databases.	Start of the PEP module.	Create REST API endpoints for the requests.	Basic Solution Implementation
Week 9 12/11 - 18-11	Make extension point of PIP.	Finish PEP module.	Create the JSP for the response of the requests.	Basic Solution Implementation (Intensive Test)
Week 10 19/11 - 25/11	Setup Java HTTPS server in the front-end server.	Create the certificates. Check client’s certificates.	Setup a CA. Code review and testing.	Basic Delivery Intermediate Solution Implementation (Intensive Test)
Week 11 26/11 - 2/12	Code review and testing.	Check client’s certificates. Code review and testing.	Set encryption outline in the database.	Intermediate Solution Implementation (Intensive Test)
Week 12 3/12 - 9/12	Implement backup server API (Update and “I’m alive”).	Implement “I’m alive” and update logic.	Implement front-end server API and request parsing.	Intermediate Delivery Advanced Solution Implementation
Week 13 10/12 - 12/12	Final Tests and report.	Final Tests and report.	Final Tests and report.	Submission Project (Code & Report)

Appendix

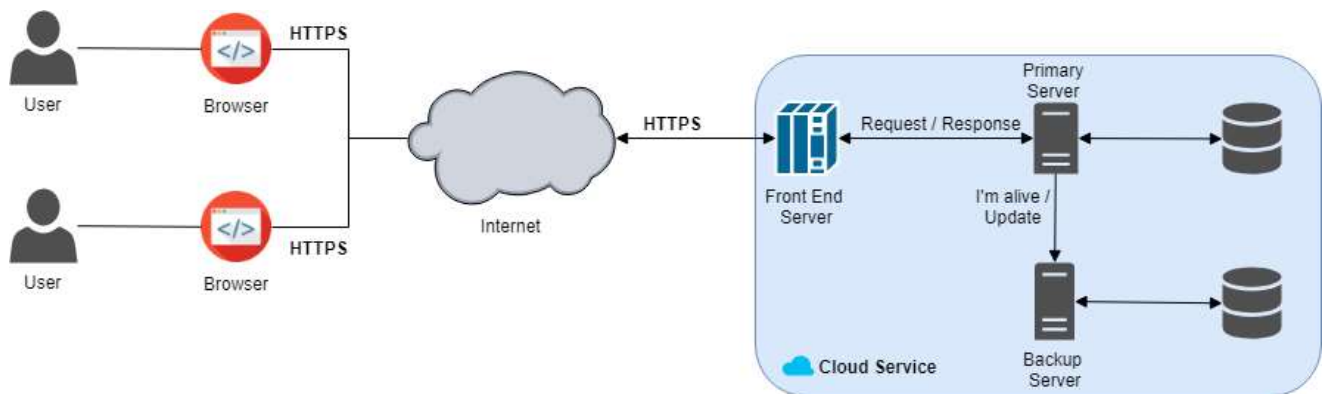


Figure 1: System Architecture

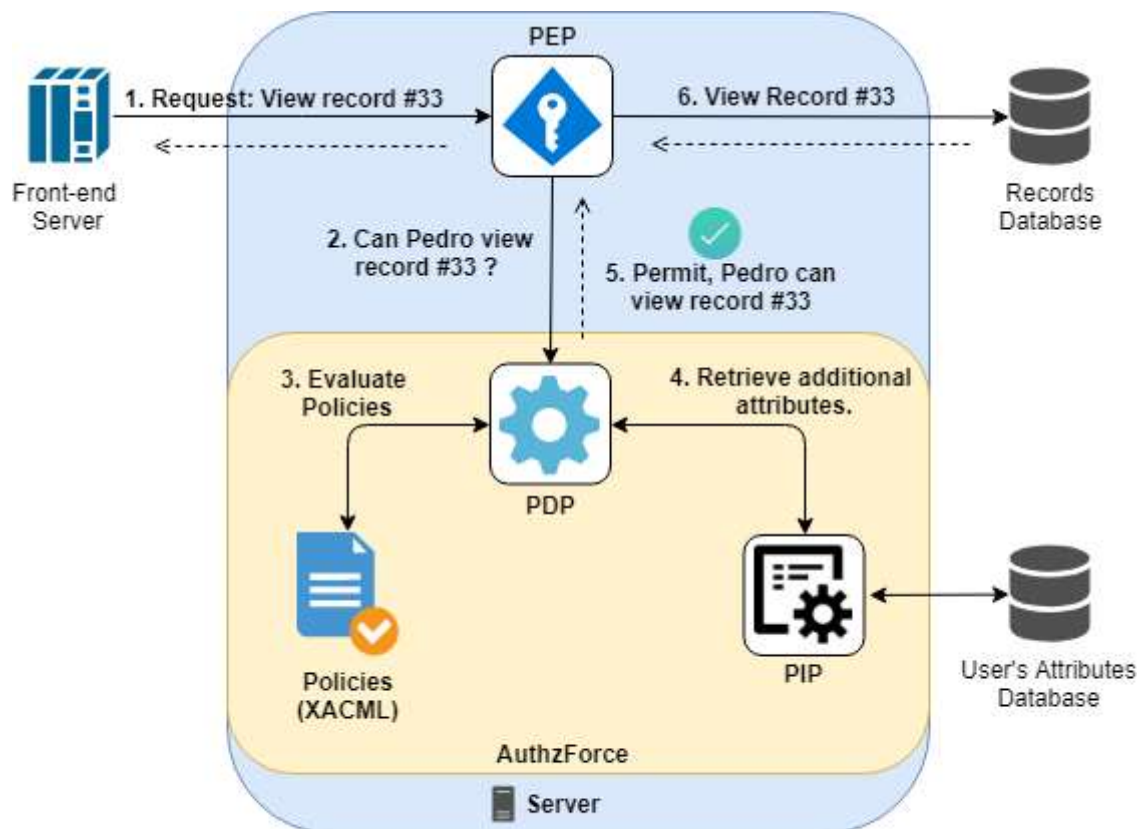


Figure 2: Control Access Architecture of the system.

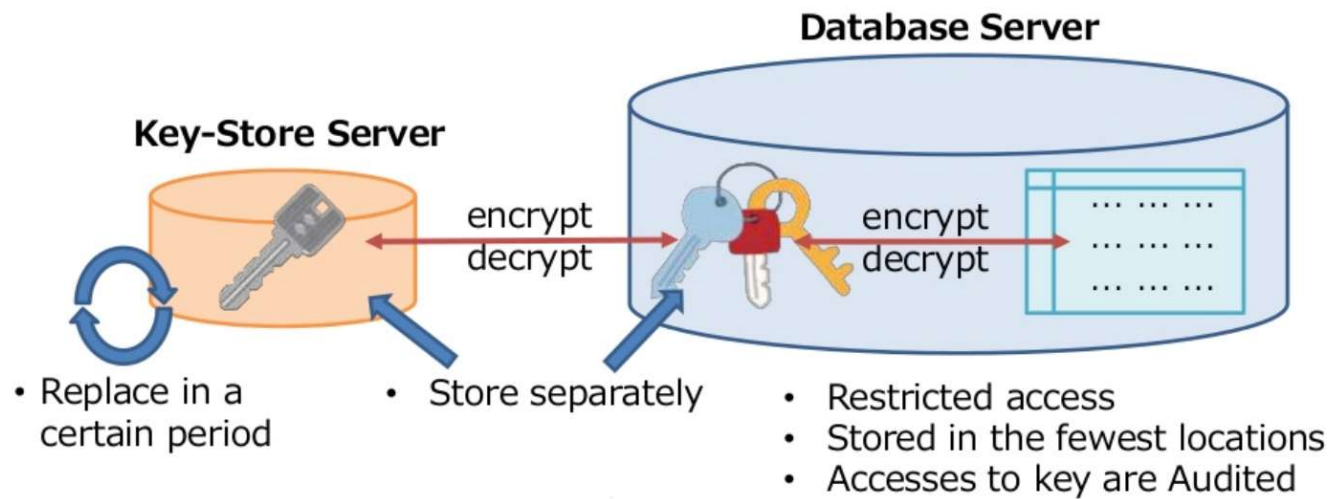


Figure 3: Data Encryption Key-management Outline.