Project 2: Medical Records and Secure Connections

Yazan Al-Aswad
ya4652al-s@student.lu.se

Alvin Karlsson
al2810ka-s@student.lu.se

Adrian Steene
ad3122st-s@student.lu.se

Daniel Stoopendahl
da4136st-s@student.lu.se

February 2024

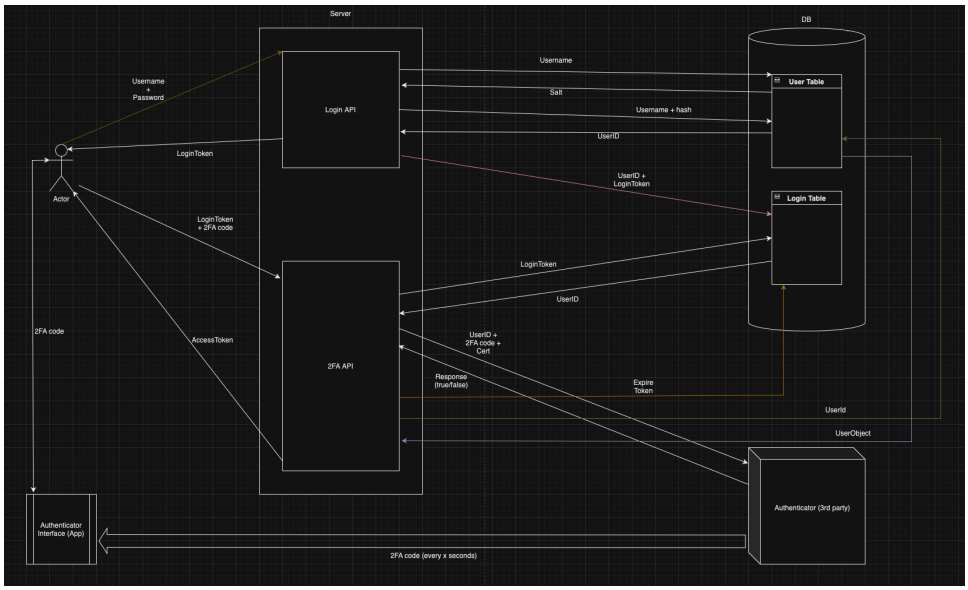

Figure 1: Login Procedure

1 Architectural Overview

1.1 Tech-Stack

The system will use Java21 for the programming language in combination with Springboot (and necessary Springboot configuration) to handle HTTPS connections with TLS to both client and database, which will be a MySQL instance. These choices were made for multiple reasons. Primarily, all of these technologies are very well documented, widely used, and thus well studied for both security and implementation techniques. The choice for Java with Springboot specifically was made to reduce human error, as Springboot comes bundled with modules to set up and handle HTTPS with TLS connections to both client and database, input validation, authentication, authorization, logging, and other security functions needed to perform and maintain the functionality necessary for the system requirements. Java21 was chosen for its usage of records and simple asynchronous/multi-threaded workflow, further reducing human error through its simplified syntax while still allowing the developers to use their previously acquired knowledge within the language. Additionally, Java itself provides additional safety by running in an isolated Virtual Machine while also providing more memory safety than a traditionally more low-level language like C, by managing the memory automatically.

1.2 Database Structure

The database will be composed of 4 tables: User, MedicalRecord, Division, and Login. The tables will contain the following entries:

- User: userId (int), username (string), passwordhash (string), salt (string), role (Enum: patient, nurse, doctor, govorg), divisionId (int) (nullable)
- MedicalRecord: recordId (int), patientId (int), nurseId (int), doctorId (int), divisionId (int), note (string) The following bindings exists between the aforementioned tables:
  - User.userId (one to many) MedicalRecord.patientId
  - User.userId (one to many) MedicalRecord.nurseId
  - User.userId (one to many) MedicalRecord.doctorId
  - User.divisionId (one to many) MedicalRecord.divisionId
- Division: divisionId (int), divisionName (string)
- Login: loginId (int), userId (int), loginToken (string), created (datetime), expired (boolean)

1.3 Login Procedure and Permissions

The login procedure for the system is rather complicated. It involves a username, password, a second factor of authentication, and a final access token in the form of JWT. The login procedure is as follows: The client sends an HTTP post request with their username and password via HTTPS to the server, the server retrieves the necessary salt from the database (using the username), the salt is then appended to the password, hashed, then the resulting hash and username is used to find a matching entry in the database. If a matching entry is not found, 401 (Bad Request) is returned, otherwise a login-Token is generated and posted together with the UserId and the creation time to the Login Table. That same login token is then sent back to the client to be used within a specified amount of time. The user is then asked to verify themselves through the second authentication method. In a UI this would be done by redirecting the user to a new page that requests the 2FA code. This can also be done by making an HTTP request to a different endpoint. The client is then expected to send in the 2FA code as well as the login token. The 2FA code can be obtained from a mobile app provided by a trusted third party (e.g Google Authenticator), through an email or SMS depending on the implementation. The different implementations have different pros and cons in terms of security, implementation difficulty, and maintainability. However, this implementation will aim to implement 2FA through Google Authenticator for the reasons discussed below. Once the 2FA code and the login token are sent to the server, the server attempts to find a matching entry in the database. If a matching entry is found and the timestamp is within the correct time limit, the

server obtains the corresponding userId (or other user-identifying data requested by the 3rd party) and sends it, combined with the 2FA code and the server SSL certificate to the 3rd party authenticator, at which point the authenticator responds with Valid/Invalid. If a corresponding entry for the Login token is not found or the authenticator finds the 2FA code to be invalid, 401 is returned to the client. Otherwise, the server obtains the necessary information from the database to create an HMACSHA256-encrypted JWT. The JWT token is created by creating a JSON object that includes userId, role, and, in the case of doctors and nurses, divisionId. This JSON object is prepended with a header that contains a notation to describe the encryption algorithm (e.g. HS256) as well as the type of token (JWT). A secret is then postfixed to the JSON object, which will be used to verify any consequent request.

2 Ethical discussion

There are several difficulties when implementing a system that requires both high security and usability and often compromises have to be made. The product needs to comply with the requirements in the commission but in many cases, there are also more responsibilities that a security expert or developer has from an ethical standpoint even though they are not explicitly stated.

In our case in delivering a journaling system, our main focus has been to protect confidentiality. While this is important, in a real-world hospital the number one priority is to provide good health care. Thus if confidentiality is applied to the point where it detriments availability, patients may come to harm and the product works worse. Therefore engineers have to see a broader picture than just the requirements of the commission. Likewise, many commissions might not explicitly state security requirements, and neither have hired security experts. They then put a lot of trust in their hired developers and it is the developers' responsibility to implement a secure system or at the very least inform about security flaws.

In a live hospital environment that has requirements on both confidentiality, integrity, and availability access can be managed with an Attribute-Based Access Control (ABAC) with a log-in procedure with multiple steps of authentication. After users have logged in and authenticated themselves, their read access to journals is based on their profession, division, and patients. Unlike in our solution, doctors and nurses should be able to see the full journal of a patient they are assigned to, including those provided by other divisions. Write access is not as crucial and should be based on whether the doctor or nurse is assigned to the journal entry just like in our solution. The added availability is required to provide good health care but it also reduces confidentiality. Therefore logging should be conducted for all operations and should be reviewed regularly to catch unwarranted access.

When it comes to access for patients, availability is less prioritized. A log-in system is not necessary. Instead, the patients can make individual requests to fetch their journals.

The players involved in developing and using a healthcare system have different priorities which can collide. The hospital staff most probably values usability most while the hospital administration might look more into economic aspects. Just like confidentiality can impede availability so can budget and time cuts undermine the system's usability and security. Here it is up to the administration to commission a job with a feasible timeline and budget but the engineers also must produce a secure system and look beyond the stated requirements as they are responsible for a lot of patients.

General statistics can be handy for the hospital but pose a risk to confidentiality. Just anonymizing data by removing names is not sufficient, instead, confidentiality can be acquired through adding noise to the data. [4] Using this approach, personal data can not be inferred from general data whilst the general data still can provide overall statistics.

3 Security evaluation

3.1 Two factor authentication

Two-factor authentication, usually called 2FA increases security significantly. 2FA adds a security layer to an application in the form of an additional step a user must do to gain access to their account. Usually, a password is the only credential needed to gain access but when using 2FA and a second layer is added, the second layer can consist of an email, text messages, etc. This means that if a password is compromised the hacker still can't access the account, this is especially useful nowadays when passwords are often easily compromised. 2FA is thereby safe against brute force and dictionary attacks because of the second layer of security consists of something different than a password. 2FA is however vulnerable to man-in-the-middle attacks where hackers intercept the communication between the user and the server capturing the user's 2FA's code. Our application uses 2FA with a third party, this ensures that if our application is compromised users our still safe due to the third-party authentication. 2FA also increases trust and a feeling of commitment to security among users, this is especially important when dealing with personal data such as medical records. The problem with 2FA is that users can feel that it is inconvenient, especially when a device or an email address is lost making it impossible for the user to access the account. Security is constantly a delicate balance between security and convenience, here 2FA exceeds by adding a heavily increased security while still making it somewhat convenient.

3.2 Guessing Attacks

There are several guessing attacks hackers can use to gain access to users' accounts. As discussed in the previous section, 2FA decreases the risk of successfully guessing attacks however by making the password less susceptible to guessing attacks, security can be increased. In this section, we will discuss sev-

eral sorts of guessing attacks and how are application aims to minimize the risk of successful attacks.

3.2.1 Brute Force Attacks

A brute force attack is where a hacker uses trial and error to try to gain access to the user's password. Working through all possible combinations until the correct password is guessed. [1] A brute force attack works especially well when users have short passwords due to the number of possible passwords being much smaller. Adding a character to the password increases the number of possible passwords exponentially. Locking the account after several unsuccessful login attempts works very well against brute force attacks and make them impossible to execute. This is the reason behind our choice to set a maximum attempt counter on the login page. We also have decided to implement requirements on length and special characters on passwords, to reduce the risk of a successful brute force attack.

3.2.2 Time Memory Trade-off Attacks

A time-memory trade-off attack involves a precomputed set of password hashes, that are compared to the password hash that the hacker has accessed. The set of passwords hashes in the time memory trade-off attacks are tried against the password until the correct password is found. An effective way of complication for hackers is to add a salt to the password before hashing. This means that the same passwords will have different hashes, this means that the hacker is forced to create a new Time memory trade-off table for every salt. [6] For this reason, we have added a unique salt for each password to increase the difficulty for potential hackers. A common problem with salts is that they are not random this is due to a seed that is used during the salting process. To ensure that our seed indeed is random, our seed will be generated with random weather data. This will lead to the seed being as close to random as possible, thereby making the salt unique. This complicates time memory trade-off attacks even further.

3.2.3 Dictionary

A dictionary attack resembles a brute force attack, but instead of testing all combinations, a dictionary attack uses common passwords and word combinations to guess the password. To prevent a dictionary attack, forcing users to use special characters, uppercase, and numbers works quite well. Our system implements this by forcing users to have a password of set length and enforcing at least one capital letter, one number, and one special character to be included in the password.

3.3 Interceptive Attacks

Beyond guessing and brute-forcing systems, it is also possible to gain access to confidential data through direct data interception (sniffing) of active com-

munications. The two main types of attacks that use this approach are listed below.

3.3.1 Man In The Middle (MITM)

In an MITM attack, an attacker maliciously intercepts and possibly alters the communication between two parties without their knowledge. The attacker can eavesdrop on the conversation and may even pose as any of the two independent parties in an attempt to gain access to confidential information or inject malware into the communication stream. [2]

The typical way to protect against these kinds of attacks is to use strong encryption as well as good message authentication. This is accomplished in our design by using TLS (Transport Layer Security), with CA (Certificate Authority) and third-party authentication. Between the user and the server, third-party authentication mitigates the risk of malicious entities impersonating users, and the TLS keeps the transmission secure. CA along with TLS is used between the server and the database to ensure authority in this crucial step.

3.3.2 Replay Attacks

Replay attacks are similar to MITM attacks, with some crucial differences. In a replay attack, instead of acting as a real-time intermediary intercepting and modifying messages, the attacker attempts to sniff legitimate traffic and use it later on to attempt to bypass security mechanisms. The goal is to trick the receiver into accepting the sniffed message as valid and thus gaining access to a system. [5]

Crucial protection against these types of attacks is already implemented in the TLS protocol, including the usage of Initialization Vectors (IV:s) for handshakes to ensure unique cryptographic keys for each session, hash functions for generation of MAC:s to prevent tampering with messages and sequence numbers to preserve order of messages to prevent malicious resending. Since TLS is used across our entire stack, the system is sufficiently protected against these attacks.

3.4 Cipher suites

A cipher suite describes the set of algorithms that act as security in a network connection. They contain methods for encryption, authentication, key exchange, and message authentication. A network connection's security is therefore determined by the security of the algorithms included in a suite for a given connection. What algorithms to use, in the case of TLS, is decided by the server and client in the TLS-handshake phase. Since the security hinges on the choice of the cipher suite, it is often a trade-off for the client and server to pick algorithms that are compatible with both hosts and also provide optimal security. [3]

When setting up a manual Certificate Authority for a server, one can specify what suites to prefer (or one gets chosen as default), and the server will then

"suggest" suites to connecting hosts. An equivalent process occurs behind the scenes in our implementation, but since we no longer use the manual server we can't be sure what cipher suite is used every time since this is handled automatically.

References

[1] Brute Force Attack: Definition and Examples, June 2023. [Online; accessed 23. Feb. 2024].

[2] What is MITM (Man in the Middle) Attack | Imperva, December 2023. [Online; accessed 23. Feb. 2024].

[3] Contributors to Wikimedia projects. Cipher suite - Wikipedia, January 2024. [Online; accessed 23. Feb. 2024].

[4] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating Noise to Sensitivity in Private Data Analysis. In *Theory of Cryptography*, pages 265–284. Springer, Berlin, Germany, 2006.

[5] William Stallings and Lawrie Brown. *Computer Security: Principles and Practice*. Pearson, London, England, UK, August 2017.

[6] François-Xavier Standaert and Jean-Jacques Quisquater. Time-Memory Trade-offs. In *Encyclopedia of Cryptography and Security*, pages 1297–1299. Springer, Boston, MA, Boston, MA, USA, 2011.