SECG4 Project Security

Project of secure client/server manage handling patient's medical records

Authors:

Younes Oudahya(54314)

Aimen El Mahsini(56149)

Billal Zidi (54637)

Teacher:

Romain Absil

Group

D121

Academic Year:

2022 - 2023

Table of contents

Introduction

The purpose of this project is to develop a secure client/server system for managing patient medical records. The system utilizes the Laravel framework and adheres to the security principles defined by the Open Web Application Security Project (OWASP). This report outlines the implementation details and security measures employed in the project.

The system follows a client/server architecture, where the server handles user registration, login, and the management of medical records. The server is not considered a trusted entity, requiring secure data transmission and storage mechanisms. Clients, representing administrators, doctors, and patients, interact with the server to access and modify medical records. Patients can appoint doctors who have access to their records.

Additional Information

For this project, the Laravel framework was chosen as it is a widely recognized and well-supported framework with a large developer community. Laravel offers numerous built-in security features and optimizations, making it a suitable choice for implementing a secure system. Regular updates and bug fixes contribute to maintaining a solid and secure codebase.

The project utilizes PHP as the programming language and requires a server and a database for data storage. Additionally, Composer and NodeJS are necessary for seamless development and modification of the Laravel project.

Safety information

To ensure the security of the system, the following measures have been implemented:

1. Certificate Usage:

To establish secure communication, a self-signed certificate is employed. This ensures the confidentiality and integrity of data exchanged between clients and the server.

2. Password encryption:

User passwords are encrypted using the "password_hash()" function with the argon2id algorithm. Regular expressions (regex) are utilized to enforce strong password policies. Passwords must meet certain complexity requirements, such as a minimum length and a combination of uppercase and lowercase letters, numbers, and special characters. This approach enhances the security of user accounts and mitigates password-related attacks.

3. Captcha for Brute Force Protection:

To prevent brute force attacks on user login credentials, a captcha mechanism is implemented. After a certain number of failed login attempts, users are required to solve a captcha challenge, which adds an additional layer of protection against automated login attempts.

4. Secure Patient File Management:

The management of patient files is designed with security in mind. Files are stored in encrypted format on the server, ensuring the confidentiality of sensitive information. Only authorized users, such as the patient and appointed doctors, can access these files.

5. Laravel authentication model:

The authentication model built into the Laravel framework is used for user registration and login functionality. This model provides secure authentication mechanisms that we have subsequently edited to make it much more secure.

6. Secure File Uploads:

Implement strict file validation and restrict file types to prevent malicious file uploads. Apply server-side checks on file content and enforce strict file size limits to prevent denial-of-service attacks and storage abuse.
Utilize secure file storage mechanisms, such as storing files outside the web root directory or using hashed file names, to prevent unauthorized access.

7. Input Validation and Sanitization:

Implement robust input validation and sanitization techniques to prevent common vulnerabilities such as SQL injection, cross-site scripting (XSS), and command injection attacks. Use server-side validation with regular expressions (regex) to enforce strong password policies and prevent weak passwords.

8. Protection of downloaded files:

When adding a medical file request, the code performs several security checks on the downloaded file. It checks the file extension to make sure it matches a predefined list of permitted extensions. This helps prevent malicious or potentially dangerous file downloads.

9. Encrypting medical files before storage:

When the doctor adds a medical file for a patient, the code performs the following operations:

It checks the file extension to ensure that it is authorized.
Generates a random symmetrical encryption key and a random initialization vector (IV).
Encrypts the contents of the file using the encryption key and the IV.
Encrypts the symmetric encryption key with the doctor's public key.

It stores the encrypted file, the IV and the encrypted encryption key in the file system. This encryption process guarantees the confidentiality of medical files, even if the security of the storage system is breached. Only doctors with the corresponding private key can decrypt the files.

10. Medical file consent requests:

When the doctor sends a medical file request to a patient, the code creates a new consent request. This consent request is linked to the doctor, the patient and the medical file. This approach ensures that only authorized patients can access and manage their own medical files.

11. Deleting temporary files:

A temporary file is created to store the decrypted contents of the medical file. After downloading, the temporary file is deleted to avoid leaving sensitive data traces on the system.

12. Error handling:

The function handles errors appropriately. If the requested file is not found, the user is redirected back with an error message. This contributes to security by preventing the disclosure of sensitive information.

13. Use of prepared queries:

When using prepared queries in many project methods, it is recommended to use prepared queries to avoid SQL injection attacks. This ensures greater security by ensuring that input data is correctly escaped.

14. Session management:

Codes use sessions to maintain user authentication status. Sessions are stored on the server side and associated with a unique identification token for each user. This secures authentication information and prevents attacks such as session theft. Laravel handles session management securely and the code seems to use the built-in features for this.

15. Secure deletion of medical files:

When deleting a medical file, the code uses the Storage::delete() function to delete the associated binary, iv and key files. This ensures that the files are correctly deleted from the file system.

Before deleting the files, the code also checks whether the current user is authorized to delete the file. It retrieves the associations between patient and doctors via the DoctorPatient model and checks whether the current user is an authenticated patient before deleting the key files associated with the doctors. This reinforces security by ensuring that only authorized patients can delete the corresponding medical files.

16. Using migrations to create the user table:

The code uses Laravel migrations to create the user table in the database. Migrations enable secure versioning and management of database schema modifications.

17. Secure downloading of medical files:

When downloading a medical file, the code checks whether the file exists and is accessible in storage. It uses the Storage::exists() function to check whether the file is present before downloading it.

The code then uses encryption operations to decrypt the contents of the medical file. It extracts the encrypted data from the corresponding binary, iv and key files, then uses the authenticated user's private key to decrypt the symmetric key stored in the key file. It then uses this decrypted key and the IV to decrypt the contents of the medical file.

Before downloading the file, the code creates a temporary file containing the decrypted content. This ensures that the downloaded file is dynamically generated and not permanently stored in the server's file system. The temporary file is deleted automatically after downloading.

18. Regular updates and bug fixes:

The project relies on the latest updates and bug fixes provided by Laravel. Staying up to date is crucial for maintaining a secure system.

19. OWASP security measures:

Common vulnerabilities, such as Injection, broken authentication, exposure of sensitive data, XML attacks, broken access control, XSS cross-site scripting, insecure deserialization, use of components with known vulnerabilities, insufficient logging and monitoring, are addressed and secured.

20. Brute force prevention:

After five unsuccessful login or registration attempts, a user is temporarily blocked for one minute, mitigating brute force attacks.

Unique user IDs: User IDs are generated using UUIDs, preventing unauthorized access by manipulating URLs.

21. Cross-Site Request Forgery (CSRF) protection:

All forms include the Laravel-specific @CSRF tag, safeguarding against CSRF attacks that may trick users into performing unintended actions.

22. In addition to the security measures mentioned earlier

the project also incorporates specific decision-making processes to ensure a secure environment. One such decision is the use of the Argon2i algorithm for password encryption, which offers significant advantages over other hashing algorithms.

- Argon2i for password encryption: The Argon2i algorithm is a modern, memory-hard, and resistant hashing function designed specifically for password hashing. It is considered one of the most secure algorithms available today. Here's why it is chosen over other options:

- Resistance against brute-force attacks: Argon2i is designed to be computationally expensive, making it highly resistant to brute-force attacks. It requires a significant amount of time and computational resources to verify each password attempt, thereby slowing down potential attackers.

- Memory-hardness: Argon2i utilizes a significant amount of memory, which further increases the difficulty and cost of performing parallel attacks, such as using GPUs or ASICs. This memory-hardness property makes it more resilient against hardware-based attacks.

23. Using middleware and its benefits

Middlewares are intermediate filters between HTTP requests and responses, used in the medical records management system to enhance security and improve functionality. Here are their advantages:

Authentication: Authentication middleware verifies user identity before granting access to protected resources, ensuring that only authenticated users can access sensitive functionality.

Authorization: Authorization middleware controls users' access rights according to their roles and permissions, restricting access to appropriate resources.

Data validation: Middleware enables incoming data to be validated and filtered, preventing code injection attacks or the use of non-compliant data.

Error handling: Error handling middleware intercepts exceptions and errors, ensuring proper handling and enhancing system robustness and security.

Logging: Middleware can perform logging operations, recording information on queries, responses and actions taken, facilitating system monitoring and auditing of user actions.

By using middleware, the medical records management system benefits from enhanced security, greater modularity and improved maintainability. They also enable the separation of specific concerns, and ensure flexible configuration of system functionalities.

24. Secure storage of private keys on encrypted external media

To further enhance the security of the private keys used in the medical records management system, we have decided to store these keys on an encrypted external medium.

The advantages of this approach are:

Enhanced key protection: Storing private keys on an encrypted external medium adds an extra layer of protection. Even if an attacker gains physical access to the external medium, he won't be able to use the keys without the appropriate password or decryption key.

Physical separation of keys: By storing private keys on an external medium, they are separated from the main system. This reduces the risk of the system being compromised, and gives greater control over key access.

Centralized key security management: By using encrypted external storage, you can set up a centralized key management process. This facilitates regular key rotation, access revocation and key authorization management.

Ease of backup and recovery: Storing private keys on an external medium facilitates regular key backup. In the event of loss or corruption of the main system, keys can be easily recovered from the external medium.

It is important to note that the external medium used to store private keys must be chosen with care. It must offer a high level of security, such as strong data encryption and mechanisms to protect against unauthorized access. It is also essential to store the password or decryption key of the external medium securely, to prevent unauthorized access.

25. File size limitation

User experience: It's important to consider the user experience when downloading files. Files that are too large can take a long time to download, which can be frustrating for users with slow Internet connections.

Storage capacity: Make sure you have enough storage space to store downloaded files. If you allow large files, you'll need to allocate more storage space to manage them.

Server performance: Handling large files can have an impact on server performance, particularly during downloading and further processing. Make sure your server is capable of efficiently handling files of the size you authorize.

Security: Uploaded files may present security risks. Large files could potentially contain vulnerabilities or be used for DDoS attacks. Limiting file size can reduce the associated risks.

To prevent executables: you limit the possibility of uploading executable files, malicious scripts or other types of potentially dangerous files. This reduces the risk of a File

Upload Attack, where an attacker attempts to upload a malicious file to your server to execute code or compromise the security of your application.

General conclusion

In conclusion, the client/server system developed for managing patient medical records demonstrates a strong commitment to security and follows the security principles defined by OWASP. The implementation incorporates numerous measures to ensure the confidentiality, integrity, and availability of sensitive data.

The use of Laravel as the framework provides a solid foundation for building a secure system, with its built-in security features and regular updates. The system employs various security measures such as certificate usage for secure communication, dedicated database accounts with restricted privileges, password encryption with strong complexity requirements, and captcha for brute force protection.

Additional security measures include secure patient file management, Laravel's authentication model, secure file uploads and validation, input validation and sanitization, encrypted storage of medical files, consent requests for accessing files, secure deletion of files, and secure downloading of files.

The project also adheres to secure coding practices such as prepared queries, session management, and secure handling of errors. Regular updates and bug fixes, along with the implementation of OWASP security measures, contribute to a robust and secure system.

Overall, the implemented security measures and adherence to best practices ensure that the client/server system for managing patient medical records prioritizes data protection and reduces the risk of unauthorized access, manipulation, or disclosure of sensitive information.