SECG4: Security Report

June 16, 2023

Contents

58412 Lecky-Thompson William

58744 Taha Ahmed

1. <u>Introduction</u>

This project aims to develop a secure client/server system for managing patient's medical records, similar to those used in hospitals. The application will be built using Django, a Python web framework known for its versatility and robustness.

The application will feature an intuitive user interface that allows authenticated users to upload, store, and manage medical records securely.

We will begin by providing a high-level overview of the project's features and their implementation. This section will cover the core functionalities of the application, such as user authentication, secure file storage, and file sharing among authenticated users.

The next section will present a comprehensive check-list of security measures that need to be implemented to ensure the confidentiality and integrity of patient data.

Following the security check-list, we will discuss potential improvements and enhancements that the platform could benefit from. This section will provide insights into additional security measures or functionality that can be incorporated to further strengthen the system.

Lastly, we will conclude the paper with a brief summary of the project.

2. <u>Implementation</u>

This section breaks down the choices made in the project as well as the technical details of which algorithms have been used and why

a. <u>Server</u>

For the current project, the django-sslserver package is being used to enable HTTPS and TLS requests during development and testing in Django. It simplifies the setup process by generating a self-signed certificate for secure communication.

However, when deploying the project to a production environment, it is crucial to use a different type of connection and obtain a signed SSL/TLS certificate from a trusted certificate authority (CA). Self-signed certificates, like the ones generated by OpenSSL, are not trusted by default by web browsers and can result in security warnings or errors.

SSL/TLS encryption provided by a signed certificate ensures that data transmitted between the web server and clients remains confidential. This protects sensitive information, such as user credentials, personal data, and other confidential information, from interception and eavesdropping.

SSL/TLS also ensures data integrity by detecting any tampering or modification of the transmitted data. The use of digital signatures and encryption mechanisms guarantees that the data remains unchanged during transit.

b. <u>Clients</u>

Regarding information encryption and decryption, we employ a two-set RSA key pair system. This involves using one key pair for encryption from the client to the server and another key pair for encryption from the server to the client. This approach allows us to verify if the data has been compromised during the communication process as well as ensuring safe data transit between both communication sides.

For security reasons, we only store the public key in a file, while the private key is kept in a local variable. This ensures that the private key remains confidential and limits the risk of unauthorized access.

By employing Django's built-in authentication mechanism, utilizing PBKDF2 for password security, and employing RSA encryption with separate key pairs for communication, we establish a secure and robust system for handling patient's medical records. These measures help protect sensitive data, maintain privacy, and ensure the integrity of the communication between the client and server.

c. **Files**

To ensure data integrity, each file is signed by its owner upon uploading as well as when it is edited. This signature allows us to verify that the data on the server has not been tampered with or corrupted. By validating the signature, we can ensure the authenticity and integrity of the files stored in the system.

To further protect the privacy of the data, we store only the encrypted content of the files and their encrypted file name in the database and in the database only. This ensures that even if the database is accessed without authorization, the data remains encrypted and secure, and thus useless to attackers.

d. **Features**

To ensure strict control over privileged accounts, a heightened security measure is implemented for newly registered doctors and other administrative personnel. They undergo a verification process by being placed in a waiting list, requiring confirmation from another admin. This careful validation ensures that only authorized individuals gain access to sensitive functionalities, maintaining the integrity of the system.

To safeguard the confidentiality of user files, our application employs a clever approach. When users access their files, only the file names are retrieved from the database, keeping the content itself secure. This step acts as an additional layer of protection for the file content stored in the system.

When a user downloads a file, an advanced encryption-decryption mechanism is in place. The server decrypts the file for transmission to the web interface, ensuring secure delivery. However, before storing the file on the user's device, it is promptly re-encrypted. This additional encryption step guarantees that the file remains protected during transmission and storage, significantly minimizing the risk of unauthorized access to sensitive information.

We have implemented a comprehensive waiting list system to regulate access for doctors modifying user files and patient-doctor associations. This system requires both parties to request the association, and the request is carefully reviewed for approval or denial. By doing so, we ensure that only authorized doctors can make modifications to the patient's files, and patients can associate themselves with approved doctors, upholding data integrity and privacy.

To maintain a secure environment, our application has restricted file uploads to text files only. By prohibiting the upload of executable files or potentially malicious content, we mitigate the risk of unauthorized code execution and enhance the overall security of the system. Furthermore, the files uploaded to the server have a maximum size to ensure even more secure transactions.

Because we employ an asymmetric encryption system, every user needs his own key pair. Those keys are securely stored on the user's device in hidden folders that the user can't easily access. In windows devices, those keys are stored in the AppData hidden directory and on Linux devices, they are stored in the .config hidden directory. Note however that those directories are not password protected, implementing this protection step furthers security of those files, but we considered hidden folders to be secure enough for this project.

3. Security check-list

a. Confidentiality

To safeguard sensitive data from "Man-in-the-Middle" (MITM) attacks, we enforce the use of Transport Layer Security (TLS) for all data transmissions. By utilizing TLS, we establish encrypted communication channels that protect the confidentiality of information exchanged between the parties involved. This encryption ensures that even if an attacker intercepts the communication, they are unable to decipher or manipulate the data. The encrypted passwords and files also ensure confidentiality as they cannot be found as plain text anywhere on the server. This also means that nothing can be done with that information in case the storage is compromised.

b. Integrity of stored data

Non-repudiation is a vital aspect of security that ensures individuals cannot deny their involvement or the authenticity of their actions in a transaction or communication. In our system, we employ multiple measures, including digital signatures, encryption, and log files, to achieve non-repudiation and strengthen the overall security of our application.

We utilize digital signatures to establish proof of origin and integrity. When a user performs an action, such as uploading or modifying a file, we generate a digital signature using their private key.

To ensure the confidentiality of sensitive information, we employ encryption techniques. When a file is uploaded, its content is encrypted using a secure encryption algorithm. This ensures that even if the data is accessed without authorization, it remains unreadable and inaccessible.

Alongside digital signatures and encryption, we maintain detailed log files. These log files capture and record critical information about user actions, system events, and access attempts. By maintaining a comprehensive audit trail, we can reconstruct the sequence of events, trace user activities, and detect any suspicious or unauthorized actions. Its also worth noting that users who have admin role on the website have no access to the database as they are only administrator of users, not the application itself, thus ensuring data integrity in case an admin goes rogue.

c. Authentication scheme

When clients authenticate, they are required to create a password that meets specific criteria. The password must be at least 8 characters in length and include a combination of alphanumeric characters, including both letters and numbers. We enforce these requirements to enhance the complexity of passwords and reduce the likelihood of easily guessable or commonly used passwords being employed.

Furthermore, we implement an additional layer of security through Two-Factor Authentication (2FA). After entering their password, clients receive a unique verification code. This verification code acts as a second factor of authentication. To successfully log in, clients must enter the correct verification code in the application, ensuring that only authorized individuals with access to their email can complete the authentication process. However, due to the small scale of this project and its basically non-existant budget, we do not have an email address to contact our users with, so for the time being, verification codes are printed on the console that started the application. We still left it as it still behaves like 2FA.

d. Security features relying on secrecy.

Only the user’s login password and the private keys that are kept secret, there is no reliance on secrets.

e. Vulnerability to injections

To prevent SQL injection attacks, Django incorporates robust security measures. One key protection mechanism is the use of query parametrization in constructing query sets. This approach separates the SQL code of a query from its parameters. As user-provided parameters can be potentially unsafe, Django's underlying database driver automatically escapes and sanitizes these parameters. By implementing query parametrization, Django ensures that user input is treated as data rather than executable code, effectively mitigating the risk of SQL injection. In addition to that, files are limited in size, thus avoiding DOS attacks.

f. Vulnerability to data remanence attacks

Since users do not have direct access to the physical hardware where the data is stored. This helps protect against data remanence attacks, as the users cannot physically tamper with the storage medium or attempt to recover deleted data. The files are also never stored as files on any system, in fact, files are only stored in the database and are separated into file name and file content, both securely encrypted.

To further enhance the security of data deletion, we have implemented the django-secure-delete library. This library provides a secure mechanism for deleting data . When a deletion is triggered, django-secure-delete overwrites the data multiple times before performing the actual deletion. By doing so, it makes it significantly more difficult to recover the overwritten data. This process is used to delete the client's keys from his device when the client's account is removed from the database. In addition to his keys, any data created by said user in the database will be securely deleted along his account so no trace of those data can be found.

By combining these measures, we ensure that sensitive data is properly protected and securely deleted from the database. The combination of restricted user access to hardware and the utilization of django-secure-delete significantly reduces the risk of data remanence and helps maintain the confidentiality and integrity of our application's data.

g. Vulnerability to replay attacks and request forgery

Firstly, we employ CSRF (Cross-Site Request Forgery) protection, which is built into Django, on all of our forms. This mechanism involves generating and validating unique tokens for each user session. By including CSRF tokens in our forms and validating them on the server side, we can guarantee that requests originating from our application are legitimate and not the result of replayed requests from external sources. This safeguard prevents malicious actors from exploiting vulnerabilities through unauthorized replayed requests.

Secondly, we enforce the use of TLS/SSL (Transport Layer Security/Secure Sockets Layer) encryption in our application. TLS/SSL provides a secure communication channel by encrypting the data exchanged between the client and the server. This encryption prevents eavesdropping, tampering, and replay attacks by ensuring that the transmitted data remains confidential and unaltered during transit. By using TLS/SSL, we establish a trusted and secure connection, further safeguarding against replay attacks.

h. Monitoring

With the help of logs, we can monitor any action made on the application and quickly trace any suspicious activity.

i. Use of vulnerable components

No component used is particularly vulnerable as Django takes care of those out of the box.

j. System update status

All the elements used in our project are up-to-date with the most recent version possible.

k. Access control

we have implemented robust access control mechanisms to ensure that users can only access the appropriate data and functionalities based on their authorization levels. Two key components of our access control strategy are the use of login decorator and specific decorators corresponding to user groups for each view.

To begin, we utilize login tokens by incorporating the @login_required decorator in our views. This ensures that only authenticated users can access certain pages or perform specific actions. By specifying the login_url parameter, we can redirect unauthenticated users to the login page, thereby enforcing authentication before granting access to protected areas of our application. This login decorator helps ensure that only authorized users with valid credentials can access the desired resources.

Furthermore, we employ specific decorators corresponding to each user group for different views within our application. These decorators are designed to validate the user's permissions and group affiliations before granting access to specific data or functionalities. By associating each view with a specific token, we prevent users from accessing data or performing actions that they are not authorized to perform based on their user group. This granular level of access control ensures that users can only interact with the resources that are relevant to their assigned roles and responsibilities. Furthermore, cookies and the session itself have a life time after which they die and regenerate. Users are also logged out of the application if they close the web page or stay idle for a certain amount of time.

I. Authentication

To begin with, we enforce the use of strongish passwords to enhance the security of user accounts. We require our users to create passwords that meet certain criteria, such as a minimum length and a combination of alphanumeric characters. Additionally, we utilize hashing functions with salting to encrypt and store passwords securely. Hashing functions transform the user's password into an irreversible string of characters, ensuring that even if the stored data is compromised, the original passwords cannot be easily retrieved. This approach safeguards user accounts from unauthorized access and provides an additional layer of protection against password-related vulnerabilities.

In addition to password security, we prioritize the protection of user identifiers, such as user names or email addresses. We ensure that these identifiers are not exposed in the URL or any other publicly accessible areas of our application. Exposing user identifiers in URLs can pose a risk by potentially allowing unauthorized individuals to gather sensitive information or attempt malicious activities. By keeping user identifiers hidden, we reduce the likelihood of unauthorized access or data breaches related to user identity. To do so, all of our forms data are passed to the application through POST.

4. Possible improvements

Having an actual email address to contact our users with and thus enabling actual 2FA. We could also add an AES key for quicker decryption/encryption of the database and encrypt this AES key with our RSA key pair.
If we had to deploy our application, we would also have a key management service to store our keys.
We might also store the client's keys in a password protected folder to further enhance security.

5. Conclusion

Overall, the application is very functional and secure.