Secure software development and web security - Report (SECG4)

Nathan Furnal & Rehan Duran

September 6, 2022

Contents

1 Introduction 1

2 Implementation 2
- 2.1 Non-features . . . 2
- 2.2 System characteristics . . . 2
  - 2.2.1 Server . . . 2
  - 2.2.2 Clients and users . . . 2
  - 2.2.3 Files . . . 3
- 2.3 Features . . . 3

3 Security check-list 3
- 3.1 Confidentiality . . . 3
- 3.2 Integrity of stored data . . . 4
- 3.3 Non-repudiation . . . 4
- 3.4 Authentication scheme . . . 4
- 3.5 Security features relying on secrecy . . . 4
- 3.6 Vulnerability to injections . . . 4
- 3.7 Vulnerability to data remanence attacks . . . 4
- 3.8 Vulnerability to replay attacks . . . 4
- 3.9 Vulnerability to fraudulent request forgery . . . 5
- 3.10 Monitoring . . . 5
- 3.11 Use of vulnerable components . . . 5
- 3.12 System update status . . . 5
- 3.13 Access control . . . 5
- 3.14 Authentication . . . 5
- 3.15 General security features . . . 5

4 Possible improvements 5

5 Conclusion 6

6 References 6

1 *Introduction*

This project consists in a secure file sharing application among authenticated users. It's been built with Laravel, a PHP web framework, using a virtual box which contains the web server (see Homestead for more details).

In more details, it provides an easy interface for users to authenticate, stores their files securely and share them with other users. The technical details to build the project can be found in the `README.md` file while this paper will focus on the implementation of the project. First, with a high-level overview of its features and their implementation. Second, with a detailed check-list delving into the security details. Then, we'll go over the possible improvements the platform could benefit from and finally end with a short review of the project.

2 *Implementation*

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

2.1 Non-features

First and foremost, this was entirely a software oriented project. There is very little to no consideration to the hardware used and hardware related security exploits. The goal of the project was to emulate a NAS or FTP server on a local network but no actual NAS was used, everything lives inside a Linux virtual machine. The virtual machine runs a recent LTS version of Ubuntu (20.04) with a recent install of PHP (8.1) and Laravel (9.24), the exact version number of each packages used can be found in the `composer.json` and `composer.lock` files. The web server is nginx, a commonly used and battle-tested server.

2.2 System characteristics

2.2.1 Server

When the web server is first created in the virtual machine, a certificate and a set of keys is generated as well. This is where the trust comes in, Homestead effectively acted as a trusted third-party that both the server and us have to rely on.

Once this is done, a TLS connection is established and information transport is considered secured and any subsequent keys sent to the server (for example Laravel's) can be expected to truly live on the server and not a malicious third-party.

Additionally, at the software level, we use Laravel middleware to re-route all requests to HTTPS.

2.2.2 Clients and users

To avoid the common pitfall of implementing known strategies ourselves, we relied on Laravel Breeze, which is an implementation of Laravel authentication features, to create an authentication portal with a password. Passwords are hashed with the `bcrypt` algoritm which includes a salt directly in the hash to compensate the deterministic nature of hashes. Only hashes are ever stored in the database, not plain text passwords.

Next, two pairs of keys are generated for each user, thanks to the Halite library. We choose this library for multiple reasons: it is managed by the Paragon Initiative who specializes in web security, it is easy to use and it relies on `libsodium`, a very broadly used cryptography library. The reason for using two set of keys is that one is used for encrypting information and the other is used to sign the files.

In our setup, the private keys are stored in a folder for ease of access but in a real setup, the users should generate the keys themselves with tools such as `openssl`. Both public keys (for encryption and signing) are stored on the server, in a database.

The contact system lives in a database as well with no sensitive information stored. We only keep track of the tuple of friends through their ID's on the database.

2.2.3 Files

The files are stored encrypted in the server, their name is encrypted as well. Each file is signed by its owner when it is uploaded as well as when it is edited. An interface is also provided to check the signature of a file owned by the user with a file living on the server, provided that the user is authenticated and has the permission to access the file on the server.

2.3 Features

In the current setup, users register and have their credentials (keys) generated for them but they are not directly able to connect to the platform. An admin (us in this case) must first change a field in the database before a user is allowed to authenticate. Once a user is authenticated, they can upload, edit, share and delete files on the server as well as share them with a willing contact. It's also possible for a user to delete their contact and completely remove all of their files and credentials.

When an authenticated user uploads a file on the server, the file is signed with the user private signature key. Then, a new symmetric encryption key is generated and used to encrypt the content and the name of the file. After that, the symmetric encryption key is encrypted in an asymmetric fashion with the file owner's private key and stored on the DB. When a file is shared with a contact, the symmetric encryption key is unsealed and encrypted anew with the public key of the receiver, which lets them decrypt it on their end to finally decrypt the file. This has the advantage of never requiring to move the files around except for upload/download allow the users to only deal with the encrypted symmetric key during the sharing process. Signatures are also stored in the database and updated on each edit, this lets the user upload files they want to check the ownership of.

When the files are displayed on the user pages, they are not actually decrypted, only the names are decrypted and shown to the user. Moreover, file access is defined at the database level in a "file sharing" table, which means that unauthorized users never get to see the file names they are not privy to, even in the source code of the page. Files a user owns or is privy to can be downloaded, they are then decrypted and sent back to the user.

N.B: Note that only a restricted selection of file extensions are allowed to be uploaded. Common types like text and images files (without SVG) are accepted but nothing else, this is to avoid potential executable files.

1. Details about the cryptography primitives

   Above, we have mentioned symmetric encryption, signing and asymmetric encryption, the specific encryption used are detailed on the Halite project page (*Halite/Primitives.Md at Master · Paragonie/Halite*, n.d.). They are quoted here under for the sake of exhaustivity.

   > Symmetric key encryption uses XChaCha20 then BLAKE2b-MAC
   >
   > Asymmetric key encryption uses X25519 then HKDF-BLAKE2b followed by symmetric-key authenticated encryption.
   >
   > Asymmetric-key digital signatures use Ed25519

   Another post describes more in depth why the files are signed before encryption but the message authentication code (MAC) is added after encryption, (*Public Key - Why Do We Encrypt-Then Mac but Sign-Then-Encrypt? - Cryptography Stack Exchange*, n.d.).

3 *Security check-list*

3.1 Confidentiality

Confidentiality is ensured as sensitive data is always transmitted over TLS, thus avoid "Man-in-the-middle" (MITM) attacks. Also, since passwords are hashed and files are encrypted following the scheme described above, no data is accessible in plain text nor is it possible to do anything with the files if the storage is compromised. Since an admin is required to validate the registration though, a user could be logged out of the system, but no sensitive data would leak.

3.2 Integrity of stored data

The symmetric key encryption used is recent and used in recent cryptography libraries, the files are also signed to make sure that any alteration would then create a mismatch under a signature check.

3.3 Non-repudiation

Non-repudiation is ensured in theory as every uploaded file is signed, and signed again when they are edited, using the private signing key of the file owner. In practice, users have to submit the data they downloaded to the platform again to check for the signature, it is not automatically. The reason for it, is that we chose a sign then encrypt logic, meaning that a decrypt then verify scheme must follow. Verifying the signature on the platform would then require to store the file in plain text somewhere. So, asking the user for an additional step was considered an acceptable trade-off in order to not ever store the decrypted file in storage.

3.4 Authentication scheme

Clients are authenticated with a password using at least 8 characters without strong requirements on the case or symbol used, this could be improved. It is otherwise satisfying as the passwords are hashed with the `bcrypt` algorithm (and thus salted as well). Only the hashes are stored on the database.

A stronger authentication scheme could use 2 factor authentication for example but we did not implement it.

3.5 Security features relying on secrecy

Except for the plain text password that the user must remember as well as the private keys that can't be shared, there is no reliance on secrecy.

3.6 Vulnerability to injections

The website is not vulnerable to SQL injections because Laravel provides builtin protection again such injections, it's mentioned directly in the docs.

Laravel also provides protection against such injections since the blade templating language automatically escape special characters.

We also added a specific validation against too large files to avoid denial of service attacks.

Finally, we are missing out on some specific security headers so adding dedicated middleware to tune those headers would improve security.

3.7 Vulnerability to data remanence attacks

Since users never access the physical hardware nor is the storage manipulated, data remanence should not be an issue. That being said, no specific measure was taken to avoid data remanence attacks so in the case of storage theft, it could present an issue.

3.8 Vulnerability to replay attacks

Though Laravel does not explicitly provide protection against replay attacks, the starter kits such as Breeze do, as it is specified in the documentation. Since we use Breeze, replay attacks are not an issue.

Erratum (28/08): TLS also protects password acquisition because data is securely transferred. With respect to replay attacks, sessions tokens protect against replays across session but a within session attack could still be possible.

3.9 Vulnerability to fraudulent request forgery

Laravel uses a special token (`@csrf`), that we used in all our forms and which protects against request forgery out of the box.

3.10 Monitoring

Invalid input is not monitored but as explained at the end of the `README.md`, we can read and use Laravel and nginx logs to get a better idea of the code called and requests made and received.

3.11 Use of vulnerable components

No component is particularly vulnerable but we didn't harden Laravel to the maximum. Mainly because it is a large framework and it is difficult to know all the options, for example the headers configurations that we missed.

3.12 System update status

The system is up-to-date and runs on a LTS version of Ubuntu (20.04) with mostly up to date elements as well: PHP, Laravel, composer, Halite, etc. All the most recent version possible.

3.13 Access control

In our website, all routes use the "auth" middleware that require authentication for any action to be taken. This greatly reduces access control issues because users can't access data or routes they are not allowed to.

Edit (06/09): There was an issue (now fixed) of broken access control where the ID of the owner of a file was not checked against the current session user's ID; when downloading one's own files.

3.14 Authentication

Identifier are not exposed through the URL and up to standard hashing and encrypting strategies have been used. As mentioned above, sessions' ID's are regenerated as well to avoid replay attacks. We are missing multiple factor authentication or check against weak passwords though.

3.15 General security features

Overall the configuration is decent because we use a framework such as Laravel with a starter-kit such as Breeze, which implement basic and well-configured defaults. We also use modern cryptographic algorithms.

We could improve on the more specific options of the server and the framework and provide better guarantee of non-repudiation by verifying files' digital signatures by default, since for now, users must take action themselves. Note that all files are still signed at any time though.

4 *Possible improvements*

A better guarantee on first contact with the server would have been possible, with an official trusted entity. We could have improved on the security configuration.

A thorough list of improvements, most of which we didn't implement, can be found in a list made by Paragonie on hardening websites and best practice around websites.

5 Conclusion

Overall, the website provides good encryption and protection of the files themselves as well as decent defaults but could benefit from more involved design choices.

6 References

Halite/Primitives.md at master · paragonie/halite. (n.d.). Retrieved August 11, 2022, from https://github.com/paragonie/halite/blob/master/doc/Primitives.md

Public key - Why do we encrypt-then mac but sign-then-encrypt? - Cryptography Stack Exchange. (n.d.). Retrieved August 11, 2022, from https://crypto.stackexchange.com/questions/15485/why-do-we-encrypt-then-mac-but-sign-then-encrypt