Report SECG4

Medical Record

FERRAJ Yassine 52075 - TAHIRI Ibrahim 54563
HE2B ESI

Table of contents

1. Introduction

This project is about ensuring an appropriate security policy for a client/server file system storage where medical records are uploaded.

In the web world, it has always been particularly important to be able to secure any application, especially when it comes to transmitting sensitive data. To do so, we had to make sure to apply the global characteristics of a secure system. Those are methods that ensure confidentiality, integrity, authentication, and availability of a system. Of course, no application is one hundred percent secure, since new code is constantly committed, but along with Kerkhof's principles, we tried to build a system that IS secure as much as possible on our level.

In the following subsections, we will tackle the security choices we have made for the **SFTP**[1] web application we have created.
We will explain, the main reason(s) for each choice, which methods were used in order to ensure the wanted level of security as well as potential issues encountered while doing so.

[1] SFTP = SSH File Transfer Protocol or Secure File Transfer Protocol. SFTP is part of SSH or Secure Shell.

2. Security choices

2.1. A Secure File Transfer Protocol

To begin with, you might wonder why we chose SFTP instead of FTP.
It is quite self-explanatory that we used SFTP and not simply FTP since the S stands for « SSH » or « SECURE SHELL ». Since it is secure, SFTP is therefore a completely different protocol than FTP. It is indeed protected with cryptographic techniques, which means that all traffic between a client and a server is fully encrypted, from the identification process to the sending of files. Given this protection, SFTP is very suitable for the secure exchange of files on the internet.

2.2. Confidentiality

When it comes to sensitive data, our web application ensures confidentiality. Let us see how we decided to approach this specific characteristic of a secure system.

2.2.1. User's email and password

For the user's data storage in the database, we decided to only encrypt and salt the user's password using `argon2id`.
Argon2ID (Argon2 with ID): This variant is a combination of the Argon2i and Argon2d variants, offering protection against memory-time and side-channel attacks. It dynamically adjusts the algorithm's internal parameters to provide better protection against possible attacks. This is generally the recommended variant when you want optimum security for password hashing.

The reason we chose to encrypt only the passwords is because we do not wish database administrator(s) to have access to every user's password for privacy reasons obviously. As for the email we store in plain. But the main reason is that encrypting user's data in the database is only adding complexity to admins working around with non-private data and certainly not going to stop attackers to get the raw data once they have access to the database. Hence, the security must be focused on defending the perimeter of the database to prevent any access to it.

2.2.2. *Médical Record*

When an patient or a doctor wants to add a file in a medical record to the server, the file is send trough TLS, then the cipher uncipher it temporally to make some operations:

- Create a symmetric key **on the client side.**
- Hash the file and send it to a SFTP server.
- Cipher the file.
- Send the file to a SFTP server.
- Cipher the symmetric key with the user public key and send it to a SFTP server.

When an patient or doctor wants to download a file from the medical record, the server:

- Get the ciphered file from a SFTP server.
- Get the symmetric key from a SFTP server.
- Get the private key **from the client side using the path where the key is store (the key should be protected by a password).**
- Decrypt the symmetric key using the **user private key given a path in local in client side.**
- Decrypt the file and hash it to check **integrity**, if there is a match, the user can get the file that will be send through TLS.

When a patient wants to share any file of his medical records, the server:

- Get the ciphered symmetric key of the corresponding file.
- Decrypt the symmetric key using the **user private key given by a path in local in client side.**
- Get the public key of the doctor with who the patient wants to share the file.
- Cipher the symmetric key using the public key of the doctor.
- Send the ciphered symmetric key for the doctor to a SFTP server.

2.2.3. *Sensitive requests sent to server*

In this project, we are also aware that sensitive requests will be sent to the server, and we made sure these are transmitted securely.
Every single interaction with the server requires the clients to be registered and logged in. To do so, we used the Laravel's middleware which is a mechanism for inspecting and filtering HTTPS requests entering the application. There are many uses of the middleware.

It can be used for authentication for example. The middleware will inspect whether the user is authenticated (thanks to the class "Authenticate") and will redirect him accordingly. If yes, it

allows the user to go further into the application, otherwise, it will redirect him to the login page.
Also, there is a middleware for CSRF protection included in Laravel. CSRF is an attack that will take the identity of a trusted user and will send unwanted commands to the application. Since this protection works with tokens, it can easily verify that the authenticated person is the person actually making the requests to the application. This verification is done by storing the user's session with this unique token and changes with each new session. Therefore, malicious applications are unable to access it.

However, we do not provide session id's timeouts which can be a problem because a user stays logged in even after having closed his browser (therefore an attacker is able to use a logged in member's account by using the same browser).

2.3. Integrity

To assure integrity, when a patient or doctor push a file, the server save also the hash of the file and when a patient or doctor wants to download the file, the server decipher the file (using symmetric key that was uncipher using **patient or doctor private key in client side**) then hash it and compare the original hash with the new hash.

2.4. Non-repudiation

In our case, SFTP does not address the non-repudiation. To be able to ensure this principle, we generate a SSH key pair for user authentication5. This generates a public and a private key. The public key is copied to "authorized_files" on the server. As for the private key, it should generated (on the client side). Therefore, the server will let users that have the private key to connect and register on the platform. Once an administrator approves his registration, he can from now on log in. When he logs in, the client proves he is in possession of the private key by digitally signing the key exchange.

Secondly, each hash of the original file are signed with the user private key then at the download instance, the server check the signature by using the owner public key.

2.5. Authentication scheme and availability

Since we are using Laravel's Middelware, we are making sure that at any moment, for any request, the user MUST be logged in (authenticated) otherwise he does not have access to any of the functionalities of the platform.

To log in, a user must be registered and approved by an administrator. The user provides an email and a password to log in (or register) as well as a captcha[2].
Moreover, authenticated users can give **permissions[3]** to other authenticated users regarding shared files thanks to Laravel's Middleware. If an authenticated user does not have permissions on a file (meaning he is not the owner), he cannot access it at all. *(Figure 3)*

```php
$user = auth()->user();
$file = File::where('reference','=', $reference)->first();
//checker si user est proprietaire du fichier
if($file->file_owner != $user->id){
    return redirect('/files')->with('error', 'Access denied');
}
```

Figure 3

2.6. Other security choices

2.6.1. URL, SQL, JavaScript and dedicated parser injections

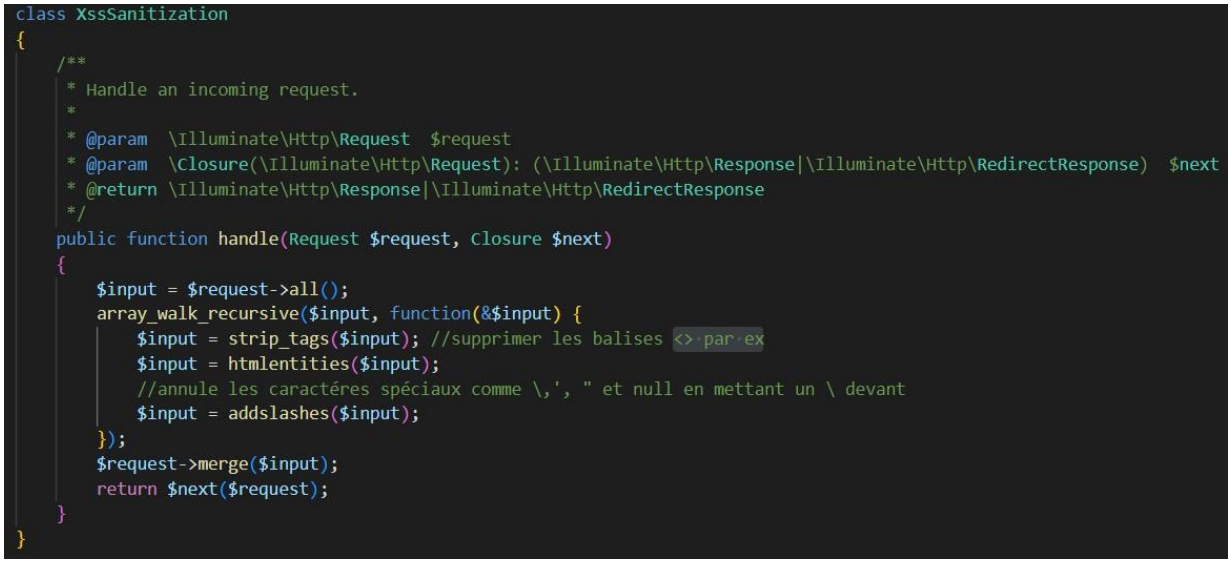
We are protected from raw SQL injections because we use Eloquent included with Laravel which provides an ActiveRecord[4] implementation to work with our database. There is a model for each database table used for the interaction with that table.

As for a safe user entry, we use the syntax "{{ }}" provided by Blade which "escapes" the data to the display. In this way, we avoided using the following syntax {!! ... !!} with data coming from outside the application which is not secure.

For futher security, we also used some XSS sanitization *(Figure 5)* like the following methods:

- Strip_tags() -> deletes all html and php tags .
- Htmlentities() -> converts characters to html entities.
- Addslashes() -> cancels the special characters like backslash, single and double quotes as well as null by putting a \ in front of it.

[2] Captcha is used to protect the user's account from spam and to prevent attempts to decrypt his password by subjecting him to a simple test to verify that it is a human and not a computer trying to register and/or access the account.
[3] Download permissions only.
[4] https://en.wikipedia.org/wiki/Active_record_pattern



```php
class XssSanitization
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure(\Illuminate\Http\Request): (\Illuminate\Http\Response|\Illuminate\Http\RedirectResponse)  $next
     * @return \Illuminate\Http\Response|\Illuminate\Http\RedirectResponse
     */
    public function handle(Request $request, Closure $next)
    {
        $input = $request->all();
        array_walk_recursive($input, function(&$input) {
            $input = strip_tags($input); //supprimer les balises <> par ex
            $input = htmlentities($input);
            //annule les caractéres spéciaux comme \,', " et null en mettant un \ devant
            $input = addslashes($input);
        });
        $request->merge($input);
        return $next($request);
    }
}
```

Figure 5

2.6.2. *Data remanence attacks*

To prevent against data remanence attack, before deleting a file the server overwrites the content.

Another way to prevent data recovery tools to get a grip of these remaining data is either to install some data wiping software (like BCWipe) - which is used to shred the contents of your sensitive files or disk space beyond recovery - , either physically destroy the disk (by degaussing it, destroy it with a hammer, burn it, cut it into pieces..whatever destroying solution you find pleasing to protect your sensitive data).

2.6.3. *Prevention methods against malicious intents*

We are using some prevention methods to control and/or analyze traffic for any malicious intent.

For example, we apply the RFC and DNS validators on the email address. If the email doesn't answer to these validators, the user is notified that he cannot register until a valid email is filled in.
As for the password, we strengthen it by applying a few constraints when created. This will protect the application from dictionary attacks. If these fail, the user cannot register, therefore will not be able to log in. *(Figure 6)*

We also implemented a specific interface for the administrator. He can view all registered members and activate their **status**[5]**.** In this, way, the admin can easily see if there was a

[5] Indeed, a registered member must be approved by the administrator to be able to start logging in on the platform.

malicious intent if he sees a registration like <script>alert('hey')</script>) as username. But that should not happen since a registration goes through the email validators and password constraints written above.

```php
if (auth()->user()->isAdmin) {
    $this->validate($request, [
        'name' => 'required|string|max:255',
        //email validate avec rfc,dns
        'email' => 'required|string|email:rfc,dns|max:255|unique:users',
        'password' => ['required', 'string', 'confirmed',
        Password::min(8)
            ->letters()
            ->mixedCase()
            ->numbers()
            ->symbols()
            ->uncompromised()
        ],
    ]);
```

Figure 6

2.6.4. Vulnerable components and security misconfigurations

At all times, we made sure all components, dependencies, and frameworks used in our application are supported, patched and up to date. Consequently, the application is protected against attacks on the known vulnerabilities for all these above.
We could take a step ahead and we could use an automated scanner (like SonarQube) to analyze OWASP's top vulnerabilities and handle the security issues/misconfigurations in our application.

3. Conclusion

As you could clearly see, web application security is a key point in such a project. It protects the users and their data from any cybercrimes such as data theft and other vulnerabilities.

We have tried to apply the five principles that a secure web application should have, namely integrity, authentication, availability and confidentiality along with Kerkhoff's principles.

All of this was done thanks to some security features:

1) The SSH File transfer Protocol which uses keys and signatures to make any file transfer and authentication secure.

2) Different packages or frameworks provided by Laravel such as:
    a. Middleware => for inspecting and filtering HTTP requests entering the application, CSRF protection as well as authentication mechanism
    b. Validator => a class that prevents unauthentified users to modify the URL
    c. Eloquent => an object relational mapper that protects from raw SQL injections
    d. XSS Sanitization thanks to methods like htmlentities(), strip_tags() and addslashes()
    e. Password => a class which strengthens the passwords' creation
3) Creation of an administrator interface to manage users that register, therefore nonregistered users are not able to use the interface as long as the administrator doesn't allow it or sees something is wrong (like a username being an html injection or any other intruders)
4) We kept all our components and their dependencies up to date so we are protected against any known vulnerabilities for those.

We are aware that any web application is never 100% secure and will always need supervision and improvements.
We could use some more features such as an automated security scanner to detect any security breaches or some data wiping software to provide protection against data remanence attacks. We also need to keep all our components up to date and keep ourselves informed of new vulnerabilities and how to protect ourselves against them.

Overall, we tried our best to secure this web applications on our level and are eager to learn more on how to improve it.

4. References

- SFTP user manual https://www.socialsecurity.be/site_de/general/helpcentre/batch/document/pdf/manuel_d_utilisateur_sftp_F.pdf

- Laravel's middleware https://laravel.com/docs/9.x/middleware

- Laravel's "streamDownload" method https://laravel.com/docs/6.x/responses#file-downloads

- About SFTP integrity of files https://hstechdocs.helpsystems.com/manuals/globalscape/cuteftp9/verifying_integrity_of_transferred_files.htm

- XSS Sanitization
  Laravel Validation & Sanitization to Prevent XSS Exploits (cloudways.com)

- *Laravel's Eloquent (ORM) https://laravel.com/docs/7.x/eloquent*

- *Laravel's "Validator" https://laravel.com/docs/5.0/validation*

- Data remanence *https://www.sciencedirect.com/topics/computer-science/data-remanence*