

AppSec Walkthrough

Table of Contents

Introduction.....	1
Auth.....	1
Anyone can Register.....	1
Password Validation.....	3
Screaming at Login.....	5
Additional Considerations.....	6
Student Table.....	7
Create a Student.....	7
Creating an XSS Student.....	7
Unauth'd API point.....	8
Edit a Student.....	10
Delete a Student.....	10
Conclusion and Further Testing.....	11

Introduction

The following repo I mostly rewrote step by step watching the developers tutorials, but rest assured, I didn't just blindly fork it over. I did get a deeper understanding on how Laravel works. All said and done, I did some preliminary testing and auditing to boost it's security posture. This is just a first pass, and obviously this is somewhat of a simple application, but a lot of what's written here can be applied elsewhere.

Auth

It makes the most sense to start at auth/login page. For a lot of would-be attackers, this may be all they have access to. We'll take a glance at a few things in auth, starting with a passthrough of common problems we can detect using Burpsuite.

Password Validation

A big gripe I had with default registration is I was able to just use 'password' as a password. Actually, we did the same thing with the seeded user, but thats neither here nor there. Lets quickly change that – taking a look into our controller:

```

*/
public function store(Request $request): RedirectResponse
{
    $request->validate([
        'name' => 'required|string|max:255',
        'email' => 'required|string|lowercase|email|max:255|unique:'.User::class,
        'password' => ['required', 'confirmed', Rules\Password::defaults()],
    ]);

    $user = User::create([
        'name' => $request->name,
        'email' => $request->email,
        'password' => Hash::make($request->password),
    ]);
}

```

So it looks like Rules\Password::defaults() is likely causing problems here. Digging down into the guts of that provider I find this:

```

/**
 * Get the default configuration of the password rule.
 *
 * @return static
 */
public static function default()
{
    $password = is_callable(static::$defaultCallback)
        ? call_user_func(static::$defaultCallback)
        : static::$defaultCallback;

    return $password instanceof Rule ? $password : static::min(8);
}

```

Looks like the only rule is that there needs to be a minimum of 8 characters (which, 'password' is). While I was about to write my own implementation to add these in, I realized I can just flip these, so lets do that.

```

/**
 * If the password requires at least one uppercase and one lowercase letter.
 *
 * @var bool
 */
protected $mixedCase = false;

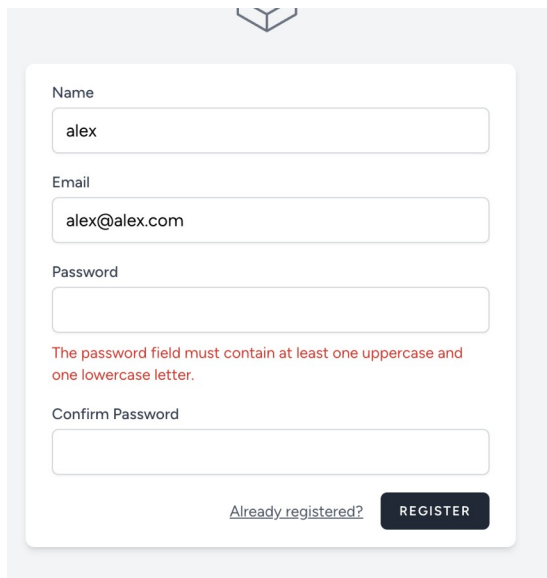
/**
 * If the password requires at least one letter.
 *
 * @var bool
 */
protected $letters = false;

/**
 * If the password requires at least one number.
 *
 * @var bool
 */
protected $numbers = false;

/**
 * If the password requires at least one symbol.
 *
 * @var bool
 */
protected $symbols = false;

```

Lets go give that a try now:



Name

alex

Email

alex@alex.com

Password

The password field must contain at least one uppercase and one lowercase letter.

Confirm Password

[Already registered?](#) **REGISTER**

Screaming at Login

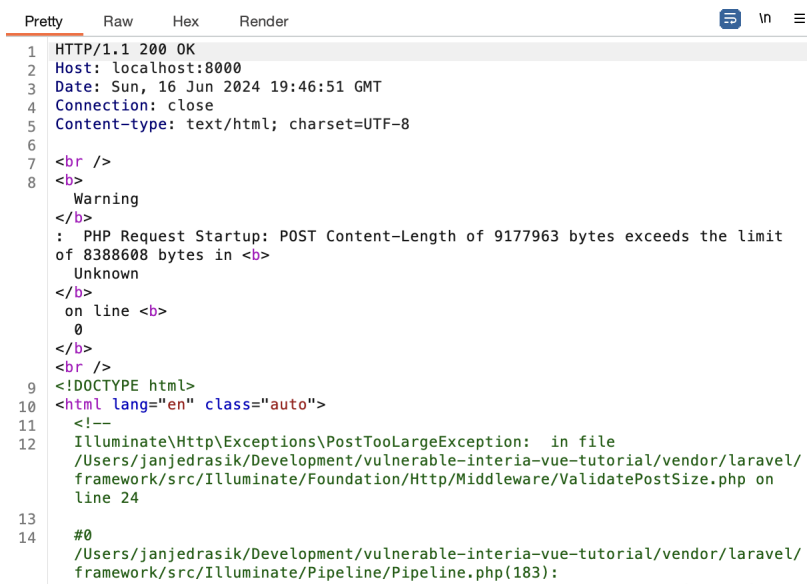
So lets go back to our original login page and this time actually capture some packets.

```
{
  "email": "admin@admin.com",
  "password": "password",
  "remember": false
}
```

Good a start as I've ever seen one – but we're running on localhost without a cert, so this is entirely expected, and not really indicative of the actual code. In a real deployment (i.e. not my laptop) – this wouldn't be the case. Anyways, logs in just fine.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="refresh" content="0;url='http://localhost:8000/dashboard'" />
    <title>
      Redirecting to http://localhost:8000/dashboard
    </title>
  </head>
  <body>
    Redirecting to <a href="http://localhost:8000/dashboard">
      http://localhost:8000/dashboard
    </a>
  </body>
</html>
```

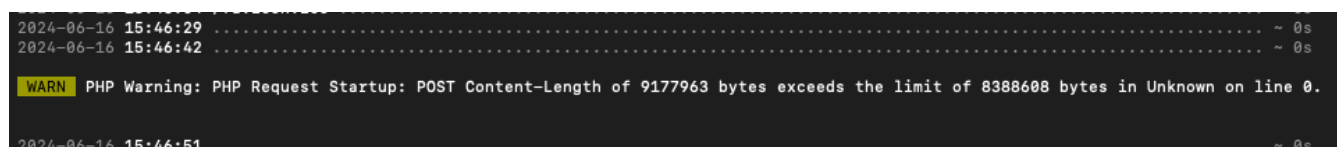
So we see a solid response. A lot of auth pages (and frankly API endpoints in general), I like to “scream” at, which is launch really long strings to see if it crashes out. Typically the front end would catch a string far too long before submitting it to the server. In this case, I’m gonna set my password to be ‘qwerty123’ written 10’000 times – I had chatGPT do it for me. Let’s launch that at our server:



```
1 HTTP/1.1 200 OK
2 Host: localhost:8000
3 Date: Sun, 16 Jun 2024 19:46:51 GMT
4 Connection: close
5 Content-type: text/html; charset=UTF-8
6
7 <br />
8 <b>
  Warning
</b>
  : PHP Request Startup: POST Content-Length of 9177963 bytes exceeds the limit
    of 8388608 bytes in <b>
      Unknown
    </b>
    on line <b>
      0
    </b>
  <br />
9 <!DOCTYPE html>
10 <html lang="en" class="auto">
11 <!--
12 Illuminate\Http\Exceptions\PostTooLargeException: in file
  /Users/janjedrasik/Development/vulnerable-interia-vue-tutorial/vendor/laravel/
  framework/src/Illuminate/Foundation/Http/Middleware/ValidatePostSize.php on
  line 24
13
14 #0
  /Users/janjedrasik/Development/vulnerable-interia-vue-tutorial/vendor/laravel/
  framework/src/Illuminate/Pipeline/Pipeline.php(183):
```

Looks like PHP does limit POST length (which we can edit smaller/larger if we want) – so this isn’t super concerning. If the limit was set much larger, we could in theory cause a denial of service but sending it increasingly longer strings to process.

In the case of this application – the front end doesn’t seem to catch it either, so we can even paste this into our password field:



```
2024-06-16 15:46:29 ..... ~ 0s
2024-06-16 15:46:42 ..... ~ 0s
WARN PHP Warning: PHP Request Startup: POST Content-Length of 9177963 bytes exceeds the limit of 8388608 bytes in Unknown on line 0.
2024-06-16 15:46:51 ..... ~ 0s
```

Additional Considerations

Since I’ve (re)written most of this, I do know that most of the querying goes on using Eloquent’s ORM – meaning I can paste in SQL injections and nothing will happen. Additionally, there is also the password reset link on breeze’s default auth screen. We could test that with SendGrid or SES – typically to capture the email via BurpSuite and reroute it, but I don’t think breeze would have such an obvious miss. I did also go ahead and checkout if pasting an XSS payload would yield anything – it did not.

Student Table

Students

Students

A list of all the Students.

ID	Name	Email	Class	Section	Created At	
1	Prof. Leo Harvey	oconnell.ralph@example.org	Class 1	Section A	Jun 16, 2024	Edit Delete
2	Hugh Hackett	marjory56@example.net	Class 1	Section A	Jun 16, 2024	Edit Delete
4	Mr. Virgil Hammes DVM	lesley.mclaughlin@example.net	Class 1	Section A	Jun 16, 2024	Edit Delete
5	Aiyana Weimann	mohammad.marquardt@example.com	Class 1	Section A	Jun 16, 2024	Edit Delete
6	Rosalind Konopelski DDS	keeling.earline@example.com	Class 1	Section B	Jun 16, 2024	Edit Delete

Showing 1 to 5 of 99 results

[« Previous](#) [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) [9](#) [10](#) [...](#) [19](#) [20](#) [Next »](#)

The actual student table doesn't have a ton of real functionality. It just displays the students using server-side pagination. Since we don't have any role-based access control implemented, we also can't try to impersonate users to see stuff we shouldn't be able to. We can do a quick packet inspection to see if anything shows up, but sure enough, it doesn't.

Create a Student

Student Information

Use this form to create a new student.

Name

Email Address

Class

Select a Class

▼

Section

Select a Section

▼

Cancel

Save

Creating an XSS Student

Again, I know that all the querying done is using an ORM, so I'm not going to spend time dropping injection payloads. That said, my knowledge of Vue is....lesser, so we can poke some XSS potential in these fields. Since only Name and Email are fillable, lets start with those.

Email Address

Unlock 1Password

<body onload=alert(1)>



Please include an '@' in the email address. '<body onload=alert(1)>' is missing an '@'.

We do have verification on the front end, but we can probably get around that...

```
1 HTTP/1.1 302 Found
2 Host: localhost:8000
3 Connection: close
4 X-Powered-By: PHP/8.3.8
5 Cache-Control: no-cache, private
6 Date: Sun, 16 Jun 2024 20:57:42 GMT
7 Location: http://localhost:8000/students/create
8 Vary: X-Inertia
9 Content-Type: text/html; charset=UTF-8
10 Set-Cookie: XSRF-TOKEN=
```

Actually not, looks like there's server side validation as well. If we look in the request for it...

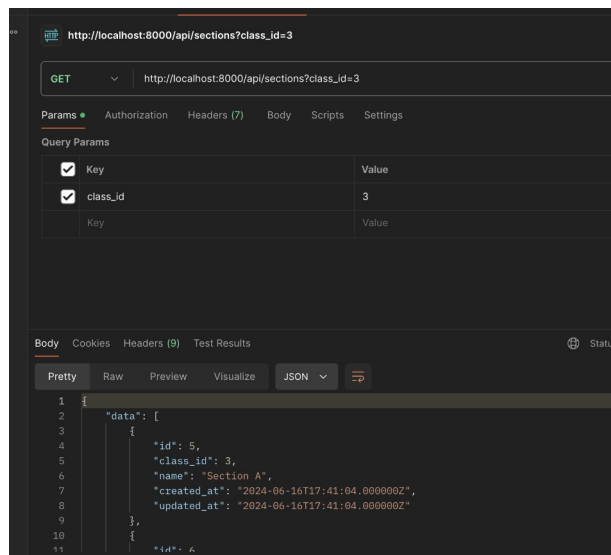
```
'email' => ['required', 'email', 'max:255', 'unique:students,email'],
```

Unauth'd API point

Taking a further look, we can also notice that when we pick a class, the dropdown populates – that's probably shooting out an api call to the ListSectionsRequest. What's interesting here is:

```
*/
public function authorize(): bool
{
    return true;
}
```

That's not great. I should be able to just shoot off an API call and get something back, lets try that.



While this isn't critical data by any stretch, the underlying issue it's an easy miss. Someone might have been debugging an endpoint and didn't want to paste in a token, and here we are. Regardless, we can fix this pretty easily with:

```

/**
 * public function authorize(): bool
 * {
 *     return auth()->check();
 * }
 */

```

And sure enough:

```

<title>Forbidden</title>

<style>
/*! normalize.css v8.0.1 | MIT License | github.com/necolas/normalize.css */
html {
  line-height: 1.15;
  -webkit-text-size-adjust: 100%
}

```

There is front end considerations:

```

const getSections = (class_id) => {
  axios.get("/api/sections?class_id=" + class_id).then((response) => {
    sections.value = response.data;
  });
};

```

In this case, since we're not passing in a user, we'll still get a 403. We'd need to append a token since we're using sanctum here. Alternatively, I'm struggling to find a reason this needs to be exposed into an API controller to begin with. Better recommendation would be to just move it into the web router and treat it like any other request behind the auth middleware.

In a more complex app, we might want to abstract this idea out further and really hammer what permissions each endpoint has. While it's unlikely someone was careless enough to just "auth->true", more complex auth logic might have holes that need to be poked.

Edit a Student

The edit student contains mostly the same functionality and forms as the create a student, so a lot of what we tested there is the same. That said, as in the last section, we should check the API endpoints to make sure they can't be hit unauth'd. At the same time, I can't imagine there is much point for allowing most endpoints to be public anyways.

```
class UpdateStudentRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     */
    public function authorize(): bool
    {
        return auth()->check();
    }

    /**
     * Get the validation rules that apply to the request.
     */
    /** @return array<string, \Illuminate\Contracts\Validation\ValidationRule|array<mixed>>|string */
    public function rules(): array
    {
        return [
            'name' => ['required', 'string', 'max:255'],
            'email' => ['required', 'email', 'max:255', 'unique:students,email,' . $this->student->id],
            'class_id' => ['required', 'exists:classes,id'],
            'section_id' => ['required', 'exists:sections,id'],
        ];
    }

    public function attributes()
    {
        return [
            'name' => 'name',
            'email' => 'email',
            'class_id' => 'class',
            'section_id' => 'section',
        ];
    }
}
```

Looks all fine to me.

Delete a Student

Same as above, delete is a relatively simple functionality. As we can see in the controller:

```
public function destroy(Student $student)
{
    $student->delete();

    return redirect()->route('students.index');
}
```

Again, since this is simply going through the controller routing, this can't be hit externally. There isn't much we can do with this. Actually, the fact that Laravel lets you put routing based on web or api is quite nice.

Conclusion and Further Testing

There's certainly a couple more things we could do to hammer in better security. Doing a quick npm/composer audit gives us some findings:

```
composer.json      jstconfig.json      postcss.config.js      tailwind.config.js
janjedrasik@Jans-MacBook-Pro vulnerable-interia-vue-tutorial % composer audit
No security vulnerability advisories found.
janjedrasik@Jans-MacBook-Pro vulnerable-interia-vue-tutorial % npm audit
# npm audit report

braces <3.0.3
Severity: high
Uncontrolled resource consumption in braces - https://github.com/advisories/GHSA-grv7-fg5c-xmjj
fix available via `npm audit fix`
node_modules/braces

follow-redirects <=1.15.5
Severity: moderate
follow-redirects' Proxy-Authorization header kept across hosts - https://github.com/advisories/GHSA-cxjh-pqwp-8mfp
fix available via `npm audit fix`
node_modules/follow-redirects

vite 5.1.0 - 5.1.6
Severity: moderate
Vite's 'server.fs.deny' did not deny requests for patterns with directories. - https://github.com/advisories/GHSA-8jhw-289h-jh2g
fix available via `npm audit fix`
node_modules/vite

3 vulnerabilities (2 moderate, 1 high)

To address all issues, run:
  npm audit fix

npm notice
npm notice New minor version of npm available! 10.7.0 -> 10.8.1
npm notice Changelog: https://github.com/npm/cli/releases/tag/v10.8.1
npm notice To update run: npm install -g npm@10.8.1
npm notice
janjedrasik@Jans-MacBook-Pro vulnerable-interia-vue-tutorial %
```

Since we'll likely want to deploy this over a container (as opposed to a pet vm) – the type of container this sits on is also important. While a simple alpine image would likely work - <https://github.com/wolfi-dev> is currently the new hotness. I haven't implemented it directly, but on paper it seems to be a more secure container image. Of course, also as mentioned above, we'd want to implement WAF for any sort of ingress.