



# Common Security Mistakes in Laravel Applications

By **cyberDanda**

# Introduction

---

Most of the time security vulnerabilities are only the result of lack of awareness, not negligence. While we find that the majority of developers deeply care about security, sometimes they do not realize how a particular code pattern can lead to a vulnerability, so in the e-book, we decided to share most common security issues that we saw during many years of helping different startups to secure their Laravel applications. With every attack vector, we also will show best practices on how to protect your application from the attack. We hope you find the information useful for you and your dev team.

**CyberPanda Team**

<https://cyberpanda.la>

info@cyberpanda.la

# SQL Injections

Laravel provides a robust Query Builder and Eloquent ORM. And thanks to them most of the queries are protected in Laravel applications by default, so for example a query like


```
Product::where('category_id', $request->get('categoryId'))->get();
```

will be automatically protected, since under the hood Laravel will translate the code into a prepared statement and execute.

But developers usually make mistakes by assuming Laravel protects from all SQL injections, while there are some attack vectors that Laravel can't protect, here are the most common causes of SQL injections that we saw in modern Laravel applications during our security checks.

## 1. SQL Injection via column name

The first common mistake that we see is that a lot of people think that Laravel would escape any parameter that is passed to Query Builder or Eloquent. But in reality, it's not safe to pass user-controlled column names to the query builder. Here is a warning from Laravel's documentation.

 PDO does not support binding column names. Therefore, you should never allow user input to dictate the column names referenced by your queries, including "order by" columns, etc. If you must allow the user to select certain columns to query against, always validate the column names against a white-list of allowed columns.

So the following code will be vulnerable to a SQL injection

```
$categoryId = $request->get('categoryId');  
$orderBy = $request->get('orderBy');  
Product::where('category_id', $categoryId)  
->orderBy($orderBy)->get();
```

and if someone makes a request with the following “orderBy” parameter value

```
http://example.com/users?orderBy=id->test" ASC, IF((SELECT count(*)  
FROM users ) < 10, SLEEP(20), SLEEP(0)) DESC -- "'
```

under the hood the following query will be executed and we will get a successful SQL injection.

```
select  
  * from `users`  
order by `id`->'$. "test" ASC,  
        IF((SELECT count(*) FROM users ) < 10, SLEEP(20), SLEEP(0))  
DESC -- "" asc limit 26 offset 0
```

It’s important to mention that the demonstrated attack vector is fixed on the latest Laravel versions, but still, Laravel warns developers even in the latest documentation to not pass user-controlled column names to Query Builder without whitelisting.

In general, even if there is no possibility to turn a custom column to an injected SQL string, we still do not recommend allowing to sort the data by any user-provided column name, since it can introduce a security issue. Consider an example when a “users” table can have some secret column “secretAnswer”, a clever attacker possibly could deduce the value without ever needing SQL injection.

## 2. SQL Injection via validation rules

Let’s look at the following simplified validation code

```
$id = $request->route('id');  
  
$rules = [  
  'username' => 'required|unique:users,name,' . $id,  
];  
  
$validator = Validator::make($request->post(), $rules);
```

Since Laravel uses `$id` here to query that database and `$id` is not escaped, it will allow an attacker to perform an SQL injection. We will cover this attack in more details in Validation [Rule Injection -> 3. SQL Injection](#) section.

### 3. SQL Injection via raw queries

Another pattern worth mentioning here, but less common that we see in our security code reviews is just using old style `DB::raw` function and not escaping passed data. A pattern like this usually happens rarely, mostly in cases when there is a need to pass some custom query. If you have to use `DB::raw` function for some custom query, make sure you escape the passed data via `DB::getPdo()->quote` method.

## Validation Rule Injection

---

Let's look at following vulnerable code.

```
public function update(Request $request) {
    $id = $request->route('id');
    $rules = [
        'username' => 'required|unique:users,username,' . $id,
    ];

    $validator = Validator::make($request->post(), $rules);
    if ($validator->fails()) {
        return response()->json($validator->errors(), 422);
    }

    $user = User::findOrFail($id);
    $user->fill($validator->validated());
    $user->save();
    return response()->json(['user' => $user]);
}
```

If your eye caught the vulnerability in `'required|unique:users,username,' . $id` code, great job!

So what the “unique” rule is doing here, it’s making sure the username is unique inside the users table and it will also ignore the row with the given ID during the check. But the issue here is that we got the \$id from the request and without validating it, we used it to create a validation rule based on the user’s input. So using that we can customize the validation rule and create some attack vectors, let's look at the following examples.

## 1. Making the validation rule optional

The simplest thing that we can do here is to send a request with ID = ``10|sometimes``, which will alter the validation rule to

``required|unique:users,username,10|sometimes`` and will allow us to not skip the username in the request data, depending on your application business logic, a bypass like this might create a security issue.

## 2. DDOS the server by creating an evil REGEX validation rule

Another attack vector here could be to create an evil Regex validation, that is vulnerable to ReDoS attack and DDOS the app. For example, the following request would consume a lot of CPU and if multiple requests sent concurrently can cause a big CPU spike on the server.

```
PUT /api/users/1,id,name,444|regex:%23(.*){100}%23
{
  "username": "aaaaa.....ALOT_OF_REPETED_As_aaaaaaaaaa"
}
```

## 3. SQL Injection

The simplest SQL injection here would be to just add an extra validation rule that is querying the database, for example

```
PUT /api/users/1,id,name,444|unique:users,secret_col_name_here
{
  "username": "secret_value_to_check"
}
```

But important to mention since using unique we are able to provide both custom column name and values (values are not going through PDO parameter binding) possibilities of SQL injection here could be not limited to just a simple attack vector that is mentioned above. For more details, check out Laravel Blog's post [here](#).

#### Prevention tips:

1. The best prevention here is to not use user-provided data to create a validation rule
2. If you have to build your validation rule based on provided data(ID in our example), make sure to cast or validate the provided value before putting it into the validation rule.

# XSS(Cross-Site Scripting) in Laravel Blade

XSS attacks have been reported and exploited since the 1990s but still sometimes we see cases when developers underestimate how dangerous the attack can be because of the fact that it's executed on the browser not on the server.

But it can be very dangerous, for example an XSS attack in the admin panel can allow an attacker to inject a code like this.

```
Some text
<input onfocus='$.post("/admin/users", {name:"MaliciousUser", email:
"MaliciousUser@example.com", password: "test123", });' autofocus />
test
```

Which will allow an attacker to create an admin user with his credentials and take over the admin panel. And even restricting the admin panel by IPs will not prevent the attack, since the code will be executed in the user's browser who has an access to the network/application.

Now lets see what are the possible XSS attack vectors in Laravel.

If you are using Laravel Blade, it protects you from most of XSS attacks, so for example an attack like this will not work

```
// $name = 'John Doe <script>alert("xss");</script>';
<div class="user-card">
  <div> ... </div>

  <div>{{$name}}</div>

  <div> ... </div>
</div>
```

because the Blade `{{ }}` statement automatically encodes the output. So the server will send the following properly encoded code to the browser



```

<div class="user-card">
  <div> ... </div>

  <div>John Doe
  &lt;script&gt;alert(&quot;xss&quot;);&lt;/script&gt;</div>

  <div> ... </div>
</div>

```

which will prevent the xss attack.

But not everything Laravel(or any other framework) can handle for you, here are some XSS attack vectors that we found most common during our security checks

## 1. XSS via `{!! \$userBio !!}` Statement

Sometimes you need to output a text that contains HTML, and for it you will use `{!! !!}`

```

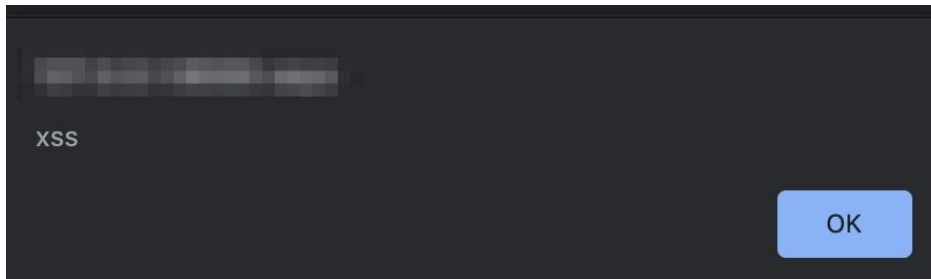
// $userBio = 'Hi, I am John Doe <script>alert("xss");</script>';
<div class="user-card">
  <div> ... </div>

  <div>{!!$userBio}</div>

  <div> ... </div>
</div>

```

In this case Laravel can't do anything for you and if the `$userBio` contains JavaScript code, it will be executed as-is and we will get an XSS attack.



## Prevention tips:

1. If you can, avoid outputting user supplied data without html encoding.
2. If in some cases you know that the data can contain HTML, use a package like <http://htmlpurifier.org/> to clean the HTML from JS and unwanted tags before outputting the content.

## 2. XSS via a.href Attribute

If you are outputting user provided value as a link, here are some examples on how it can turn into an XSS attack.

### Example 1: Using javascript:code

```
// $userWebsite = "javascript:alert('Hacked!');";  
  
<a href="{{ $userWebsite }}" >My Website</a>
```

### Example 2: Using base64 encoded data:

Note this one will work only for non top-level frames

```
// $userWebsite =  
"data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGFja2VkaSIp0zwvc2NyaXB0Pg  
==";  
  
<a href="{{ $userWebsite }}" >My Website</a>
```

the `alert('Hacked!')` code` will get executed when a user clicks on the My Website link in both cases..

### Prevention tips:

1. Validate user provided links, in most cases, you need only to allow http/https schemas
2. As an extra layer of security, before outputting you can replace any link that is not starting with http/https schema with some “#broken-link” value.

### 3. XSS via Custom Directive

If you have custom blade directives in your code, you need to manually escape any output that should not be rendered as HTML in the code. For example the following custom directive is vulnerable because the name variable is not encoded and Laravel can't do anything about it.

```
// Registering the directive code
Blade::directive('hello', function ($name) {
    return "<?php echo 'Hello ' . $name; ?>";
});

// user.blade.php file
// $name = 'John Doe <script>alert("xss");</script>';

@hello($name);
```

## Prevention tips:

Use Laravel's `htmlspecialchars` function to escape any code that is user provided.

The above mentioned 3 vulnerabilities are the most common that we saw in different Laravel applications during our security checks. And as you see, this section addresses XSS attacks in Laravel's scope, but to fully prevent XSS you also will need to make sure you front-end code which might be a React.js, Vue.js, vanilla javascript or old fashioned jQuery is also protects from XSS attacks.

## Mass Assignment Vulnerabilities in Laravel

---

Eloquent like many other ORMs have a nice feature that allows assigning properties to an object without having to assign each value individually, this is a nice feature that saves a lot of time and code lines but can lead to a vulnerability if used incorrectly.

For example here is a simplified example of an insecure code that we found during a security code review for one of our clients.

```
// app/Models/User.php file
class User extends Authenticatable {
    use SoftDeletes;
    const ROLE_USER          = 'user';
    const ROLE_ADMINISTRATOR = 'administrator';

    protected $fillable = ['name', 'email', 'password', 'role'];

    // ... rest of the code ...
}
```

```
// app/Http/Requests/StoreUserRequest.php file
class StoreUserRequest extends Request {
    public function rules() {
        return [
            'name'           => 'string|required',
            'email'          => 'email|required',
            'password'       => 'string|required|min:6',
            'confirm_password' => 'same:password',
        ];
    }
}
```

```
// app/Controllers/UserController.php file
class UserController extends Controller {
    public function store(StoreUserRequest $request) {
        $user = new User();
        $user->role = User::ROLE_USER;
        $user->fill($request->all());
        $user->save();
        return response()->json([
            'success' => true,
        ],201);
    }
    // ... rest of the code ...
}
```

The issue here is if a malicious attacker submits the payload below, he will be able to register an admin user with higher privileges and probably take over the admin panel of the application.

```
{
    "name"           : "Hacker",
    "email"          : "hacker@example.com",
    "role"           : "administrator",
    "password"       : "some_random_password",
    "confirm_password" : "some_random_password"
```

```
}
```

Also you might wonder here why “role” is in the `$fillable` attribute, if the “role” wasn't there, there would not be any issue. The reason that it was added there is that there was another API that would allow to change role too, this is a common pattern that we see in Laravel applications that we review or pen-test, while `$fillable` is great if you have a simple application, it gets hard to manage when application grows and there are multiple APIs that are updating/creating a same type of model with different ACL roles.

Here are a few tips on how you can prevent mass assignment vulnerabilities in Laravel.

Prevention tips:

### 1. Pass to model only fields that have been validated.

This is probably the most effective method of dealing with mass assignment attacks, instead of passing the full data from request, you can pass only fields that have been validated. In the above code example that will be name, email and password. To do so, Laravel provides you with a `$request->validated()` method, that returns you only fields that have been validated. So in the code above, replacing `$request->all()` with `$request->validated()` would fix the issue.

```
public function store(StoreUserRequest $request) {
    $user = new User();
    $user->role = User::ROLE_USER;
    $user->fill($request->validated());
    $user->save();

    return response()->json([
        'success' => true,
    ],201);
}
```

If you are not using Laravel's request validation, you can also use `$request->validate()` or `$validator->validated()` which also returns only data that has been validated.

This is a good technique since it not only protects you from mass assignment but also prevents saving properties that we forgot to add validation for.

## 2. Use whitelisting instead of blacklisting

We encourage using `$fillable`(which is whitelisting only columns that can be mass assigned) instead of `$guarded`(defines properties that can't be mass assigned) because you may easily forget to add a new column to a `$guarded` array, leaving it open for mass assignment by default.

## 3. Use `$model->forceFill($data)` method with caution

`$model->forceFill` method ignores all the config that is set in `$forceFill` and `$fillable` properties and saves passed data as it is into the model. If you have to use `forceFill`, make sure passed data can not be manipulated by the user.

# Not Protecting from Credential Stuffing Attacks

---

So first of all what is a credential stuffing attack? It is a type of brute force attack where attackers send pairs of username/passwords that they got from other data breaches. So for example an attacker will get millions of username/passwords from an existing [data breach](#) and will automatically try on the target's website. And because many people use the same username/password on different websites, according to reports on average around 0.1-0.2% of attempts will succeed. So if a website has around 1M users, and an attacker is using 10M of username/passwords pairs from other data breach, around 10000-20000 accounts will be hacked, the bigger is the attacker's database, bigger is the number of accounts they will potentially hack.

Credential stuffing is one of the most popular ways of attacks since it requires just a small effort on the hacker's side.

The default Laravel Auth has a basic protection from brute force where it will check if multiple requests are coming from the same IP for the same user and will block it but there is no protection against credential stuffing. You will need to implement the protection yourself and here are few tips for you.

Prevention tips:

1. **Implement Multi-Factor Authentication**

This is probably one of the most effective measures to improve your application security posture. Symantec estimates that as many as 80% of data breaches could be prevented by implementing 2FA.

2. **Limit amount of request to login endpoint from one IP**

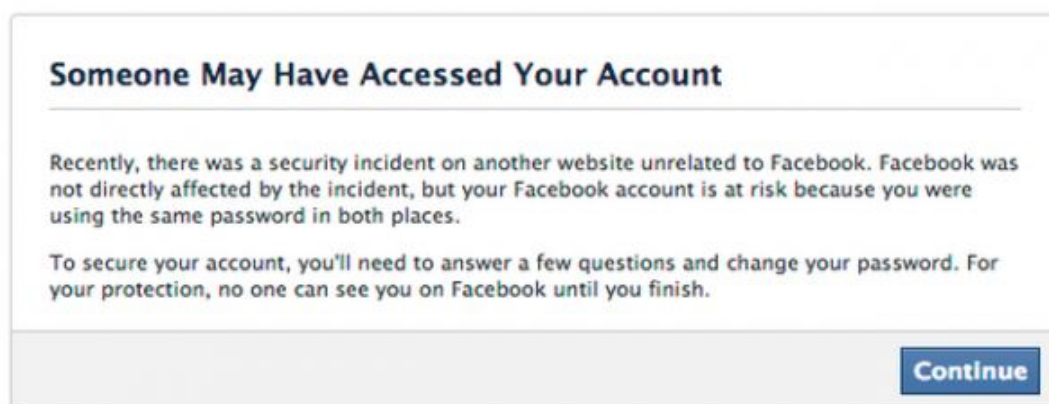
This will not fully prevent an attack if the attacker has access to a wide range of IPs, but definitely will make it harder.

3. **Add captcha to authentication endpoints**

Similar to IP blocking, will not get 100% protection but again will improve the security.

4. **Detect when a user using username/password from other data breaches**

If your company is someone like FaceBook, you can implement something like this but definitely will require more resources.



5. **Implement strong monitoring and alerting system**

Implement strong monitoring and alerting system that will alert you when there is unusual traffic coming to authentication endpoints.



## Broken Access Control

---

While a good framework like Laravel can provide you with a lot of out of box protection from attacks like SQL injection and XSS, it can't protect you from access control bypass attacks, since the ACL logic should live in the application code. And that's why ACL bypass is one of the most common issues that we see in modern application. Laravel provides [gates and policies](#) that you can use to restrict access based on user role.

## Missing HTTP Security Headers

---

Security headers are HTTP response headers(HTTP Strict Transport Security, X-Frame-Options, X-Content-Type-Options, X-XSS-Protection, Content Security Policy, etc...) that your application can use to increase the security. Once set, these HTTP response headers can restrict modern browsers from running into easily preventable vulnerabilities. For example *HTTP Strict Transport Security* header will force the application to always use HTTPS and will prevent man in the middle attacks(note that redirecting HTTP to HTTPS is not enough to prevent man in the middle attacks). Or setting proper *X-Frame-Options* will prevent clickjacking attacks.

During our penetration-tests we still see many Laravel applications that are missing HTTP Security Headers or not properly configured. And at this point you might wonder too, if security headers are so important why they are not turned on by default in browsers? The reason is that if browsers turn these settings on, they will break a lot of legacy applications, and that's the only reason that browsers leave to developers to turn them on.

By default Laravel does not have the headers so you will need to implement yourself. And here are few resources that might be helpful.

## Resources:

1. [How To Secure Your Web App With HTTP Headers](#)
2. [bepsvpt/secure-headers - Security Headers Package for Laravel](#)
3. [SecurityHeaders.com - Online tool to check your headers](#)
4. [Content Security Policy evaluator from Google](#)
5. [Content Security Policy Generator](#)

## Not Tracking Vulnerable Packages in the Application

---

Each year tens of thousand new [vulnerabilities](#) are reported in open source code. And for example one of the vulnerabilities like that costed a [fine of \\$700M](#) and reputation damage to Equifax. What happened is that hackers discovered that one of the Equifax servers contained a vulnerable version of Apache Struts that was reported 2 months ago but wasn't patched on the server. They used the vulnerability to get into Equifax servers and stole private records of 147.9 million Americans.

From that perspective, Laravel applications are no different from other apps and it's important to implement proper monitoring and alerting tool that will keep track of all packages that are used in your code and will alert you when a new vulnerability is discovered before attacks get to it.

# Conclusion

---

While Laravel is one the most secure web frameworks by design, security of web applications is a shared responsibility. A framework like Laravel will provide you a high level API(like Eloquent for example) which will by default mitigate a lot of security issues, but there are some attack vectors within Laravel and outside of its scope that should be handled by application code.

Also important to mention that in this ebook we covered only the most common issues that we saw in modern Laravel applications, but there are other issues too. Comparably less common, but can occur in an application depending on the features set of the application, here are few of them that are important to understand so proper mitigation can be provided.

1. [Race Condition](#)
2. [Server Side Request Forgery](#)
3. [PHP Type Juggling/](#)
4. [Unrestricted File Upload](#)
5. [Path Traversal](#)
6. [Sensitive Data Exposure](#)
7. [Server Side Template Injection](#)
8. [Insecure Deserialization](#)
9. [Session Fixation](#)
10. [XML External Entity \(XXE\) Processing](#)
11. [JWT Security](#)

Want to check  
the security of your  
Laravel application?



Contact us at  
[info@cyberpanda.la](mailto:info@cyberpanda.la)  
to start the conversation

cyberpanda