1v0 Ship v1.0 Faster

How to Prevent CSRF, XSS, and SQL Injection in Laravel Apps

Every modern web application faces security threats. Three of the most common and dangerous ones are **CSRF (Cross-Site Request Forgery)**, **XSS (Cross-Site Scripting)**, and **SQL Injection**. If left unprotected, attackers can hijack sessions, steal data, or even take control of your application.

The good news: **Laravel 12 comes with built-in protection** against these attacks. In this article, we'll break down what each attack means, why it's dangerous, and how to prevent it in Laravel step by step. You'll also see code examples and best practices so you can secure your apps confidently.

1 – What is CSRF (Cross-Site Request Forgery)?

CSRF happens when an attacker tricks a logged-in user into making an unwanted request to your app. For example, they might embed a hidden form on a malicious website that submits a `POST` request to your app's `/delete-account` endpoint. If the user is logged in, the request could succeed — unless you protect against it.

How Laravel Prevents CSRF

Laravel includes a CSRF token with every form. This unique token must match the one stored in the session; otherwise, the request is rejected.

```
<!-- Example Blade form with CSRF protection -->
<form method="POST" action="/profile/update">
    @csrf
    <input type="text" name="name">
    <button type="submit">Save</button>
```

```
</form>Code language: PHP (php)
```

The `@csrf` directive generates a hidden token field. Laravel validates this automatically, protecting you from CSRF attacks.

Best Practices for CSRF Protection

- Always include `@csrf` in forms.
- For SPAs, send the CSRF token in the `X-CSRF-TOKEN` or `X-XSRF-TOKEN` header.
- Do not disable CSRF middleware unless absolutely necessary.

2 – What is XSS (Cross-Site Scripting)?

XSS allows attackers to inject malicious JavaScript into your app. For example, if you display user-submitted comments without escaping, an attacker could inject:

```
<script>alert('Hacked!')</script>Code language: HTML, XML (xml)
```

If another user loads the page, the script executes in their browser. This could be used to steal cookies, capture keystrokes, or redirect users to malicious sites.

How Laravel Prevents XSS

By default, Laravel escapes all Blade output. For example:

```
{{ $comment }}Code language: PHP (php)
```

If `$comment` contains `<script>alert('XSS')</script>`, Laravel will escape it to:

```
&lt;script&gt;alert('XSS')&lt;/script&gt;Code language: HTML, XML (xml)
```

This prevents execution. If you want to render HTML, you must explicitly use `{!! $comment !!}` — but do this only with trusted content.

Best Practices for XSS Protection

- Keep default escaping (use `{{ }}` not `{!! !!}`).
- Sanitize user input before displaying it.
- Use libraries like `DOMPurify` on frontend if you must allow limited HTML.

3 – What is SQL Injection?

SQL Injection happens when user input is inserted directly into SQL queries without proper escaping. For example:

```
// ❌ Vulnerable raw query
$user = DB::select("SELECT * FROM users WHERE email = '$email'");Code language: PHP (php)
```

If an attacker sets `$email` to `' OR 1=1 --`, the query becomes:

```
SELECT * FROM users WHERE email = '' OR 1=1 --'Code language: JavaScript (javascript)
```

This would return all users — a catastrophic data leak.

How Laravel Prevents SQL Injection

Laravel's **Eloquent ORM** and **Query Builder** use PDO parameter binding, which automatically escapes input values:

```
// ✅ Safe query with parameter binding
$user = DB::table('users')
          ->where('email', $email)
          ->first();Code language: PHP (php)
```

This ensures user input never breaks SQL syntax, eliminating SQL injection risks.

1v0 Ship v1.0 Faster

Best Practices for SQL Injection Prevention

- Always use Eloquent or Query Builder instead of raw queries.
- If raw queries are unavoidable, use bindings: `DB::select('... where email = ?', [$email])`.
- Validate and sanitize inputs before using them in queries.

Wrapping Up

We covered the three most common web attacks: CSRF, XSS, and SQL Injection. Laravel 12 protects you against these by default, but only if you use its features correctly. Always include `@csrf` in forms, let Blade escape your output, and stick to Eloquent or Query Builder for database access. Combine these best practices, and your app will already be resilient against many real-world threats.

What's Next

- How to Expire User Sessions Automatically in Laravel — strengthen security with session timeouts.
- Implementing Two-Factor Authentication in Laravel — add another layer of login protection.
- Best Practices for Storing API Keys Securely in Laravel — handle secrets the right way.