# Security

## Reporting Vulnerabilities

When a vulnerability is reported, it immediately becomes our top concern, with a full-time contributor dropping everything to work on it. To report a vulnerability, please email security@vuejs.org.

While the discovery of new vulnerabilities is rare, we also recommend always using the latest versions of Vue and its official companion libraries to ensure your application remains as secure as possible.

## Rule No.1: Never Use Non-trusted Templates

The most fundamental security rule when using Vue is **never use non-trusted content as your component template**. Doing so is equivalent to allowing arbitrary JavaScript execution in your application - and worse, could lead to server breaches if the code is executed during server-side rendering. An example of such usage:

```js
Vue.createApp({
  template: `<div>` + userProvidedString + `</div>` // NEVER DO THIS
}).mount('#app')
```

Vue templates are compiled into JavaScript, and expressions inside templates will be executed as part of the rendering process. Although the expressions are evaluated against a specific rendering context, due to the complexity of potential global execution environments, it is impractical for a framework like Vue to completely shield you from potential malicious code execution without incurring unrealistic performance overhead. The most straightforward way to avoid this category of problems altogether is to make sure the contents of your Vue templates are always trusted and entirely controlled by you.

## What Vue Does to Protect You

### HTML content

Whether using templates or render functions, content is automatically escaped. That means in this template:

```template
<h1>{{ userProvidedString }}</h1>
```

if `userProvidedString` contained:

```js
'<script>alert("hi")</script>'
```

then it would be escaped to the following HTML:

```template
&lt;script&gt;alert(&quot;hi&quot;)&lt;/script&gt;
```

thus preventing the script injection. This escaping is done using native browser APIs, like `textContent`, so a vulnerability can only exist if the browser itself is vulnerable.

### Attribute bindings

Similarly, dynamic attribute bindings are also automatically escaped. That means in this template:

```template
<h1 :title="userProvidedString">
  hello
</h1>
```

if `userProvidedString` contained:

```js
'" onclick="alert(\'hi\')'
```

then it would be escaped to the following HTML:

```template
&quot; onclick=&quot;alert('hi')
```

thus preventing the close of the `title` attribute to inject new, arbitrary HTML. This escaping is done using native browser APIs, like `setAttribute`, so a vulnerability can only

## Potential Dangers

In any web application, allowing unsanitized, user-provided content to be executed as HTML, CSS, or JavaScript is potentially dangerous, so it should be avoided wherever possible. There are times when some risk may be acceptable, though.

For example, services like CodePen and JSFiddle allow user-provided content to be executed, but it's in a context where this is expected and sandboxed to some extent inside iframes. In the cases when an important feature inherently requires some level of vulnerability, it's up to your team to weigh the importance of the feature against the worst-case scenarios the vulnerability enables.

### HTML Injection

As you learned earlier, Vue automatically escapes HTML content, preventing you from accidentally injecting executable HTML into your application. However, **in cases where you know the HTML is safe**, you can explicitly render HTML content:

- Using a template:

```template
<div v-html="userProvidedHtml"></div>
```

- Using a render function:

```js
h('div', {
  innerHTML: this.userProvidedHtml
})
```

- Using a render function with JSX:

```jsx
<div innerHTML={this.userProvidedHtml}></div>
```

> ⚠ WARNING
>
> User-provided HTML can never be considered 100% safe unless it's in a sandboxed iframe or in a part of the app where only the user who wrote that HTML can ever be exposed to it. Additionally, allowing users to write their own Vue templates brings similar dangers.

In a URL like this:

```template
<a :href="userProvidedUrl">
  click me
</a>
```

There's a potential security issue if the URL has not been "sanitized" to prevent JavaScript execution using `javascript:`. There are libraries such as sanitize-url to help with this, but note: if you're ever doing URL sanitization on the frontend, you already have a security issue. **User-provided URLs should always be sanitized by your backend before even being saved to a database.** Then the problem is avoided for *every* client connecting to your API, including native mobile apps. Also note that even with sanitized URLs, Vue cannot help you guarantee that they lead to safe destinations.

### Style Injection

Looking at this example:

```template
<a
  :href="sanitizedUrl"
  :style="userProvidedStyles"
>
  click me
</a>
```

Let's assume that `sanitizedUrl` has been sanitized, so that it's definitely a real URL and not JavaScript. With the `userProvidedStyles`, malicious users could still provide CSS to "click jack", e.g. styling the link into a transparent box over the "Log in" button. Then if `https://user-controlled-website.com/` is built to resemble the login page of your application, they might have just captured a user's real login information.

You may be able to imagine how allowing user-provided content for a `<style>` element would create an even greater vulnerability, giving that user full control over how to style the entire page. That's why Vue prevents rendering of style tags inside templates, such as:

```template
<style>{{ userProvidedStyles }}</style>
```

To keep your users fully safe from clickjacking, we recommend only allowing full control over CSS inside a sandboxed iframe. Alternatively, when providing user control through a style binding, we recommend using its object syntax and only allowing users to provide values for specific properties it's safe for them to control, like this:

```
  :style="{
    color: userProvidedColor,
    background: userProvidedBackground
  }"
>
  click me
</a>
```

### JavaScript Injection

We strongly discourage ever rendering a `<script>` element with Vue, since templates and render functions should never have side effects. However, this isn't the only way to include strings that would be evaluated as JavaScript at runtime.

Every HTML element has attributes with values accepting strings of JavaScript, such as `onclick`, `onfocus`, and `onmouseenter`. Binding user-provided JavaScript to any of these event attributes is a potential security risk, so it should be avoided.

> ⚠ WARNING
>
> User-provided JavaScript can never be considered 100% safe unless it's in a sandboxed iframe or in a part of the app where only the user who wrote that JavaScript can ever be exposed to it.

Sometimes we receive vulnerability reports on how it's possible to do cross-site scripting (XSS) in Vue templates. In general, we do not consider such cases to be actual vulnerabilities because there's no practical way to protect developers from the two scenarios that would allow XSS:

1. The developer is explicitly asking Vue to render user-provided, unsanitized content as Vue templates. This is inherently unsafe, and there's no way for Vue to know the origin.
2. The developer is mounting Vue to an entire HTML page which happens to contain server-rendered and user-provided content. This is fundamentally the same problem as #1, but sometimes devs may do it without realizing it. This can lead to possible vulnerabilities where the attacker provides HTML which is safe as plain HTML but unsafe as a Vue template. The best practice is to **never mount Vue on nodes that may contain server-rendered and user-provided content**.

---

## Best Practices

actually holds true whether using Vue, another framework, or even no framework.

Beyond the recommendations made above for Potential Dangers, we also recommend familiarizing yourself with these resources:

- HTML5 Security Cheat Sheet
- OWASP's Cross Site Scripting (XSS) Prevention Cheat Sheet

Then use what you learn to also review the source code of your dependencies for potentially dangerous patterns, if any of them include 3rd-party components or otherwise influence what's rendered to the DOM.

## Backend Coordination

HTTP security vulnerabilities, such as cross-site request forgery (CSRF/XSRF) and cross-site script inclusion (XSSI), are primarily addressed on the backend, so they aren't a concern of Vue's. However, it's still a good idea to communicate with your backend team to learn how to best interact with their API, e.g., by submitting CSRF tokens with form submissions.

## Server-Side Rendering (SSR)

There are some additional security concerns when using SSR, so make sure to follow the best practices outlined throughout our SSR documentation to avoid vulnerabilities.