보안

취약점 보고

취약점이 보고되면 즉시 최대 관심사가 되며, 이를 해결하기 위해 모든 작업을 중단할 수 있는 풀타임 기여자가 있습니다. 취약점을 보고하려면 **security@vuejs.org**로 이메일을 보내주십시오.

새로운 취약점이 발견되는 경우는 드물지만, 항상 최신 버전의 Vue 및 공식 동반 라이브러리를 사용하여 앱의 보안을 최대한 유지할 것을 권장합니다.

필수 규칙: 신뢰할 수 없는 템플릿 절대 사용 금지

Vue를 사용할 때 가장 기본적인 보안 규칙은 **신뢰할 수 없는 컨텐츠를 컴포넌트 템플릿으로 사용하지 않는 것**입니다. 이렇게 하는 것은 앱에서 임의의 JavaScript 실행을 허용하는 것과 동일하며, 서버 측 렌더링 중에 코드가 실행되는 경우, 서버 보안사고로 이어질 수 있습니다. 그러한 사용의 예:

```js
Vue.createApp({
  template: `<div>` + userProvidedString + `</div>` // 절대 이렇게 하지 마십시오!
}).mount('#app')
```

Vue 템플릿은 JavaScript로 컴파일되고 템플릿 내부의 표현식은 렌더링 프로세스의 일부로 실행됩니다. 표현식은 특정 렌더링 컨텍스트를 기준으로 평가되지만, 전역 실행 환경의 복잡성으로 Vue와 같은 프레임워크가 성능 오버헤드 없이 잠재적인 악성 코드 실행으로부터 사용자를 완전히 보호하는 것은 불가능합니다. 이 범주의 문제를 모두 피하는 가장 간단한 방법은 Vue 템플릿의 내용이 항상 신뢰할 수 있고 전적으로 개발을 하는 당신이 제어하는 것인지 확인하는 것입니다.

Vue가 당신을 보호하기 위해 하는 일

HTML 컨텐츠

템플릿을 사용하든 렌더링 기능을 사용하든 컨텐츠는 자동으로 이스케이프됩니다. 이것은 아래 템플릿에서 다음을 의미합니다:

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

만약 `userProvidedString` 이 아래와 같다면:

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

다음과 같이 HTML로 이스케이프됩니다:

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

결과적으로 스크립트 주입을 방지합니다. 이 이스케이프는 `textContent` 와 같은 기본 브라우저 API를 사용하여 수행되므로, 브라우저 자체가 취약한 경우에만 취약성이 존재할 수 있습니다.

속성 바인딩

마찬가지로 동적 속성 바인딩도 자동으로 이스케이프됩니다. 이것은 아래 템플릿에서 다음을 의미합니다:

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

만약 `userProvidedString` 이 아래와 같다면:

```js
'" onclick="alert(\'안녕\')'
```

다음과 같이 HTML로 이스케이프됩니다:

```template
&amp;quot; onclick=&amp;quot;alert('안녕')
```

따라서 새로운 임의의 HTML을 삽입하기 위해 `title` 속성을 닫는 것을 방지합니다. 이 이스케이프는 `setAttribute` 와 같은 기본 브라우저 API를 사용하여 수행되므로, 브라우저 자체가 취약한 경우에만 취약성이 존재할 수 있습니다.

잠재적 위험

모든 웹 앱에서 정제되지 않은 사용자 제공 컨텐츠를 HTML, CSS, JavaScript로 실행하는 것은 잠재적으로 위험하므로 가능한 한 피해야 합니다. 그래도 어느 정도 위험을 감수할 수 있는 경우가 있습니다.

예를 들어, CodePen 및 JSFiddle과 같은 서비스를 사용하면 사용자 제공 컨텐츠를 실행할 수 있지만 이는 iframe 내에서 어느 정도 예상되고 샌드박싱되는 컨텍스트에 있습니다. 중요한 기능이 본질적으로 어느 정도의 취약성을 필요로 하는 경우, 취약성이 가능하게 하는 최악의 시나리오에 대해 기능의 중요성을 평가하는 것은 팀의 몫입니다.

HTML 주입

앞에서 배운 것처럼 Vue는 HTML 컨텐츠를 자동으로 이스케이프하여 실수로 실행 가능한 HTML을 앱에 주입하는 것을 방지합니다. 그러나 **HTML이 안전하다는 것을 알고 있는 경우** HTML 컨텐츠를 명시적으로 렌더링할 수 있습니다:

템플릿 사용:

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

렌더 함수 사용:

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

JSX와 함께 렌더 함수 사용:

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

WARNING

사용자 제공 HTML은 샌드박스 처리된 iframe이나 해당 HTML을 작성한 사용자에게만 노출되는게 아닌 한 100% 안전한 것으로 간주될 수 없습니다. 또한 사용자가 Vue 템플릿을 작성할 수 있도록 허용하면 유사한 위험이 따릅니다.

URLs 주입

다음과 같은 URL에서:

template

```
<a :href="userProvidedUrl">
  클릭해봐요
</a>
```

javascript: 를 사용하는 자바스크립트 실행을 방지하기 위해 URL을 "살균(sanitized)"하지 않은 경우 잠재적인 보안 문제가 발생할 수 있습니다. **sanitize-url**과 같은 라이브러리가 있지만, 프론트엔드에서 URL 살균을 수행했다면 이미 보안 문제가 있는 것입니다. **사용자가 제공한 URL은 데이터베이스에 저장되기 전에 항상 백엔드에서 살균 처리해야 합니다.** 그러면 네이티브 모바일 앱을 포함하여 API에 연결하는 모든 클라이언트에 대해 문제를 방지할 수 있습니다. 또한 위생 처리된 URL을 사용하더라도 Vue는 안전한 목적지로 연결된다고 보장할 수 없습니다.

Styles 주입

이 예를 보면:

template

```
<a
  :href="sanitizedUrl"
  :style="userProvidedStyles"
>
  클릭해봐요
</a>
```

sanitizedUrl 이 살균 처리되어 자바스크립트가 아닌 실제 URL이라고 가정해 봅시다. userProvidedStyles 을 사용하면 악의적인 사용자가 "로그인" 버튼 위에 링크를 투명한 상자로 스타일링하는 등 여전히 "클릭 잭"을 위한 CSS를 제공할 수 있습니다. 그런 다음 https://user-controlled-website.com/ 이 애플리케이션의 로그인 페이지와 유사하게 작성된 경우 사용자의 실제 로그인 정보를 캡처했을 수 있습니다.

<style> 엘리먼트에 대해 사용자 제공 컨텐츠를 허용하면, 페이지 스타일 전체를 완전히 제어해 더 큰 취약점이 발생할 수 있습니다. 그래서 Vue는 다음과 같은 템플릿 내부의 스타일 태그 렌더링을 방지합니다:

template

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

사용자를 클릭 재킹으로부터 완전히 보호하려면 샌드박스 처리된 iframe 내에서 CSS에 대한 제어만 허용하는 것이 좋습니다. 또는 스타일 바인딩을 통해 사용자 제어를 제공할 때 객체 구문을 사용하고, 다음과 같이 사용자가 제어하기에 안전한 특정 속성에 대한 값만 제공하도록 허용하는 것이 좋습니다:

template

```
<a
  :href="sanitizedUrl"
  :style="{
    color: userProvidedColor,
    background: userProvidedBackground
  }"
>
  클릭해봐요
</a>
```

JavaScript 주입

템플릿과 렌더 함수에 부작용이 있어서는 안 되므로 Vue에서 `<script>` 앨리먼트를 렌더링하는 것을 강력히 권장하지 않습니다. 하지만 런타임에 자바스크립트로 평가되는 문자열을 포함하는 유일한 방법은 아닙니다.

모든 HTML 앨리먼트에는 `onclick`, `onfocus`, `onmouseenter` 과 같은 자바스크립트 문자열을 허용하는 값이 있는 속성이 있습니다. 사용자가 제공한 JavaScript를 이러한 이벤트 속성에 바인딩하는 것은 잠재적인 보안 위험을 초래할 수 있으므로 피해야 합니다.

> **WARNING**
>
> 사용자가 제공한 자바스크립트는 샌드박스가 적용된 아이프레임에 있거나 해당 자바스크립트를 작성한 사용자만 노출될 수 있는 앱의 일부에 있지 않는 한 **100%** 안전하다고 볼 수 없습니다.

때때로 Vue 템플릿에서 크로스 사이트 스크립팅(XSS)이 가능하다는 취약성 보고서를 받기도 합니다. 일반적으로 XSS를 허용하는 두 가지 시나리오로부터 개발자를 보호할 수 있는 실질적인 방법이 없기 때문에 이러한 경우를 실제 취약점으로 간주하지 않습니다:

1. 개발자가 Vue에 사용자가 제공한 살균되지 않은 콘텐츠를 Vue 템플릿으로 렌더링하도록 명시적으로 요청하고 있습니다. 이는 본질적으로 안전하지 않으며 Vue가 그 출처를 알 수 있는 방법이 없습니다.

2. 개발자가 서버 렌더링 및 사용자 제공 콘텐츠가 포함된 전체 HTML 페이지에 Vue를 마운트하는 경우. 이는 근본적으로 #1과 동일한 문제이지만 개발자가 이를 인지하지 못하고 실행하는 경우가 있습니다. 이로 인해 공격자가 일반 HTML로는 안전하지만 Vue 템플릿으로는 안전하지 않은 HTML을 제공하는 취약점이 발생할 수 있습니다. 가장 좋은 방법은 **서버 렌더링 및 사용자 제공 콘텐츠를 포함할 수 있는 노드에 Vue를 마운트하지 않는 것**입니다.

모범 사례

기본 규칙은 가공되지 않은 사용자 제공 컨텐츠(HTML, JavaScript 또는 CSS)를 실행하도록 허용하면 공격에 노출될 수 있다는 것입니다. 이것은 Vue를 사용하든 다른 프레임워크를 사용하든 프레임워크를 사용하지 않든 동일하게 적용됩니다.

잠재적 위험에 대해 위에서 설명한 권장 사항 외에도 다음 리소스를 숙지하는 것이 좋습니다:

- **HTML5** 보안 치트 시트
- **OWASP**에서 제공하는 **XSS(**교차 사이트 스크립팅**)** 방지 치트 시트

그런 다음 학습한 내용을 사용하여 잠재적으로 위험한 패턴의 소스 코드를 검토합니다. 이러한 패턴이 타사 컴포넌트나 DOM에 렌더링되는 내용에 영향을 미치는 경우가 있습니다.

백앤드와의 협업

크로스 사이트 요청 위조(CSRF/XSRF) 및 크로스 사이트 스크립트 포함(XSSI)과 같은 HTTP 보안 취약성은 주로 백엔드에서 해결해야하므로 Vue의 문제는 아닙니다. 그러나 양식과 CSRF 토큰을 제출하여 API와 가장 잘 상호 작용하도록 백엔드 팀과의 논의하는 것이 좋습니다.

서버 사이드 렌더링 (SSR)

SSR을 사용할 때 몇 가지 추가적인 보안 문제가 있으므로 **SSR** 문서 전반에 걸쳐 요약된 모범 사례(예제)를 따라 취약점을 방지하십시오.