보안

취약점 보고

취약점이 보고되면, 즉시 우리의 최우선 관심사가 되며, 전담 기여자가 모든 작업을 중단하고 해당 문제에 집중합니다. 취약점을 보고하려면 security@vuejs.org로 이메일을 보내주세요.

새로운 취약점이 발견되는 일은 드물지만, 애플리케이션의 보안을 최대한 유지하기 위해 항상 Vue와 공식 보조 라이브러리의 최신 버전을 사용하는 것을 권장합니다.

규칙 1: 신뢰할 수 없는 템플릿을 절대 사용하지 마세요

Vue를 사용할 때 가장 기본적인 보안 규칙은 **신뢰할 수 없는 콘텐츠를 컴포넌트 템플릿으로 절대 사용하지 않는 것**입니다. 이렇게 하는 것은 애플리케이션에서 임의의 JavaScript 실행을 허용하는 것과 같으며, 더 나아가 서버 사이드 렌더링 중에 코드가 실행된다면 서버 침해로 이어질 수 있습니다. 다음은 그러한 사용 예시입니다:

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

Vue 템플릿은 JavaScript로 컴파일되며, 템플릿 내의 표현식은 렌더링 과정의 일부로 실행됩니다. 표현식은 특정 렌더링 컨텍스트에서 평가되지만, 잠재적인 전역 실행 환경의 복잡성 때문에 Vue와 같은 프레임워크가 비현실적인 성능 저하 없이 잠재적인 악의적 코드 실행으로부터 완전히 보호해 주는 것은 실질적으로 불가능합니다. 이러한 문제를 완전히 피하는 가장 간단한 방법은 Vue 템플릿의 내용이 항상 신뢰할 수 있고 전적으로 여러분이 제어하는 것임을 보장하는 것입니다.

Vue가 여러분을 보호하기 위해 하는 일

HTML 콘텐츠

템플릿이든 렌더 함수이든, 콘텐츠는 자동으로 이스케이프됩니다. 즉, 이 템플릿에서:

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

`userProvidedString` 에 다음과 같은 값이 들어 있다면:

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

다음과 같이 이스케이프된 HTML로 변환됩니다:

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

따라서 스크립트 삽입을 방지할 수 있습니다. 이 이스케이프는 `textContent` 와 같은 브라우저의 네이티브 API를 사용하여 수행되므로, 취약점은 브라우저 자체에 취약점이 있을 때만 존재할 수 있습니다.

속성 바인딩

마찬가지로, 동적 속성 바인딩도 자동으로 이스케이프됩니다. 즉, 이 템플릿에서:

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

`userProvidedString` 에 다음과 같은 값이 들어 있다면:

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

다음과 같이 이스케이프된 HTML로 변환됩니다:

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

`setAttribute` 와 같은 브라우저의 네이티브 API를 사용하여 수행되므로, 취약점은 브라우저 자체에 취약점이 있을 때만 존재할 수 있습니다.

잠재적 위험

어떤 웹 애플리케이션이든, 정제되지 않은 사용자 제공 콘텐츠가 HTML, CSS, JavaScript로 실행되도록 허용하는 것은 잠재적으로 위험하므로 가능한 한 피해야 합니다. 다만, 때로는 약간의 위험이 허용될 수 있는 경우도 있습니다.

예를 들어, CodePen이나 JSFiddle과 같은 서비스는 사용자 제공 콘텐츠의 실행을 허용하지만, 이는 기대되는 상황이며 어느 정도 iframe 내에서 샌드박스 처리됩니다. 중요한 기능이 본질적으로 어느 정도의 취약성을 필요로 하는 경우, 해당 기능의 중요성과 취약점이 초래할 수 있는 최악의 시나리오를 팀에서 신중히 저울질해야 합니다.

HTML 인젝션

앞서 배운 것처럼, Vue는 HTML 콘텐츠를 자동으로 이스케이프하여 실수로 실행 가능한 HTML이 애플리케이션에 삽입되는 것을 방지합니다. 하지만 **HTML이 안전하다고 확신하는 경우**에는 명시적으로 HTML 콘텐츠를 렌더링할 수 있습니다:

템플릿을 사용할 때:

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

렌더 함수를 사용할 때:

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

JSX를 사용하는 렌더 함수에서:

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

⚠ WARNING

은 앱의 일부가 아닌 이상 100% 안전하다고 간주할 수 없습니다. 또한, 사용자가 직접 Vue 템플릿을 작성하도록 허용하는 것도 유사한 위험을 초래합니다.

URL 인젝션

다음과 같은 URL에서:

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

`javascript:` 를 사용한 JavaScript 실행을 방지하기 위해 URL이 "정제(sanitized)"되지 않았다면 잠재적인 보안 문제가 있습니다. sanitize-url과 같은 라이브러리가 이를 도와줄 수 있지만, 주의하세요: 프론트엔드에서 URL 정제를 하고 있다면 이미 보안 문제가 있는 것입니다. **사용자 제공 URL은 데이터베이스에 저장되기 전에 반드시 백엔드에서 정제되어야 합니다.** 그러면 네이티브 모바일 앱을 포함한 모든 API 클라이언트에서 이 문제가 사전에 방지됩니다. 또한, URL이 정제되었다고 해도 Vue가 해당 URL이 안전한 목적지로 연결되는지 보장해 줄 수는 없습니다.

스타일 인젝션

다음 예시를 살펴보세요:

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

`santizedUrl` 이 정제되어 실제 URL임이 확실하다고 가정합시다. 하지만 `userProvidedStyles` 를 통해 악의적인 사용자가 링크를 투명한 박스로 만들어 "클릭재킹(click jack)"을 할 수 있습니다. 예를 들어, 해당 링크가 "로그인" 버튼 위에 투명하게 배치된다면, `https://user-controlled-website.com/` 이 여러분의 애플리케이션 로그인 페이지와 유사하게 만들어져 있다면 실제 사용자의 로그인 정보를 탈취할 수 있습니다.

사용자 제공 콘텐츠를 `<style>` 요소에 허용한다면 전체 페이지 스타일을 완전히 제어할 수 있으므로 훨씬 더 큰 취약점이 발생할 수 있습니다. 그래서 Vue는 다음과 같이 템플릿 내에서 style 태그 렌더링을 방지합니다:

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

iframe 내에서만 허용하는 것이 좋습니다. 또는 스타일 바인딩을 통해 사용자 제어를 제공할 때는 객체 문법을 사용하고, 사용자가 제어해도 안전한 특정 속성에 대해서만 값을 제공하도록 제한하는 것이 좋습니다. 예를 들면 다음과 같습니다:

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

JavaScript 인젝션

Vue로 `<script>` 요소를 렌더링하는 것은 강력히 권장하지 않습니다. 템플릿과 렌더 함수는 부작용이 없어야 하기 때문입니다. 하지만 런타임에 JavaScript로 평가되는 문자열을 포함시키는 방법은 이것만이 아닙니다.

모든 HTML 요소에는 `onclick`, `onfocus`, `onmouseenter` 와 같이 JavaScript 문자열을 값으로 받는 속성이 있습니다. 사용자 제공 JavaScript를 이러한 이벤트 속성에 바인딩하는 것은 잠재적인 보안 위험이 있으므로 피해야 합니다.

> ⚠ WARNING
>
> 사용자 제공 JavaScript는 샌드박스된 iframe 안에 있거나 해당 JavaScript를 작성한 사용자만 노출될 수 있는 앱의 일부가 아닌 이상 100% 안전하다고 간주할 수 없습니다.

가끔 Vue 템플릿에서 교차 사이트 스크립팅(XSS)이 가능하다는 취약점 보고를 받기도 합니다. 일반적으로, 이러한 경우는 실제 취약점으로 간주하지 않습니다. 왜냐하면 XSS를 허용하는 두 가지 시나리오로부터 개발자를 실질적으로 보호할 방법이 없기 때문입니다:

1. 개발자가 명시적으로 Vue에 사용자 제공, 정제되지 않은 콘텐츠를 Vue 템플릿으로 렌더링하도록 요청하는 경우. 이는 본질적으로 안전하지 않으며, Vue가 그 출처를 알 방법이 없습니다.

2. 개발자가 서버 렌더링 및 사용자 제공 콘텐츠가 포함된 전체 HTML 페이지에 Vue를 마운트하는 경우. 이는 본질적으로 1번과 동일한 문제이지만, 때로는 개발자가 이를 인지하지 못한 채로 할 수 있습니다. 이로 인해 공격자가 일반 HTML로는 안전하지만 Vue 템플릿으로는 안전하지 않은 HTML을 제공할 수 있는 취약점이 발생할 수 있습니다. **서버 렌더링 및 사용자 제공 콘텐츠가 포함될 수 있는 노드에는 절대 Vue를 마운트하지 않는 것**이 모범 사례입니다.

모범 사례

일반적인 규칙은, 정제되지 않은 사용자 제공 콘텐츠가 실행되도록 허용하면(HTML, JavaScript, CSS 모두 해당) 공격에 노출될 수 있다는 것입니다. 이 조언은 Vue, 다른 프레임워크, 심지어 프레임워크를 사용하지 않는 경우에도 마찬가지로 적용됩니다.

잠재적 위험에서 제시한 권장 사항 외에도, 다음 자료를 숙지하는 것을 권장합니다:

- HTML5 보안 치트시트
- OWASP의 교차 사이트 스크립팅(XSS) 방지 치트시트

그리고 배운 내용을 바탕으로, 의존성의 소스 코드도 검토하여 3rd-party 컴포넌트가 포함되어 있거나 DOM에 렌더링되는 내용에 영향을 미치는 위험한 패턴이 있는지 확인하세요.

백엔드 협업

교차 사이트 요청 위조(CSRF/XSRF), 교차 사이트 스크립트 포함(XSSI)과 같은 HTTP 보안 취약점은 주로 백엔드에서 다루는 문제이므로 Vue의 관점에서는 걱정할 필요가 없습니다. 하지만, 폼 제출 시 CSRF 토큰을 함께 전송하는 등 백엔드 팀과 소통하여 API와 최선의 방식으로 상호작용하는 방법을 배우는 것이 좋습니다.

서버 사이드 렌더링(SSR)

SSR을 사용할 때는 추가적인 보안 문제가 있으므로, SSR 문서에 나와 있는 모범 사례를 반드시 따라 취약점을 방지하세요.