[React-deep-dive]-14장.웹사이트 보안을 위한 리액트와 웹페이지 보안 이슈

리액트에서 발생하는 크로스 사이트 스크립팅 (xss)

크로스 사이트 스크립팅 (XSS)

웹사이트 개발자가 아닌 **제 3자가 웹사이트에 악성 스크립트를 삽입해 실행**할 수 있는 취약점

XSS 이슈 예시

- `dangerouslySetInnerHTML`
  - https://react.dev/reference/react-dom/components/common#dangerously-setting-the-inner-html
  - 특정 브라우저 DOM의 innerHTML을 특정한 내용으로 교체할 수 있는 방법
  - 사용자가 입력한 내용을 브라우저에 표시하는 용도로 사용
  - 인수로 받는 문자열에는 제한이 없다는 위험성 → 넘겨 받는 문자열 값은 한번 더 검증 필요

- useRef를 활용한 직접 삽입
  - DOM에 직접 접근할수 있으므로, 보안 취약점이 있는 스크립트를 삽입하면 동일한 문제가 발생

XSS 문제 피하기

결론: `sanitize` or `escape`로 제 3자가 삽입 할 수 있는 HTML을 안전한 HTML코드로 한번 치환 하자!

xss 위험성이 있는 스크립트등으로 post 요청을 직접 보낼시 서버에 바로 저장될 가능성이 있으므로, 치환 과정은 클라이언트가 아니라 서버에서 수행하는 것이 좋다.

- 관련 라이브러리
  - DOMpurity
    - https://github.com/cure53/DOMPurify
  - sanitize-html
    - https://github.com/apostrophecms/sanitize-html#readme
  - js-xss
    - https://github.com/leizongmin/js-xss

리액트의 JSX 데이터 바인딩

기본적으로 리액트는 XSS를 방어하기 위해 이스케이프 작업이 존재한다,

```
import { Remarkable } from 'remarkable';

export default function MarkdownPreview({ markdown }) {

  return <>
    <div>{markdown}</div>
    </>;
}
```

결론 : {html} 와 같이 HTML에 직접 표시되는 textContent와 HTML 속성 값에 대해서는 리액트가 기본적으로 이스케이프 작업을 해준다.

그러나 `dangerouslySetInnerHTML` , props로 넘겨받는 값의 경우 원본 값이 필요한 경우가 있기에 이러한 작업이 수행되지 않는다.

서버 컴포넌트에서 주의점

결론 : 서버에서 브라우져에 정보를 내려줄때는 조심하자 쿠키, 토큰등의 민감한 정보를 다루는 서버에서 토큰의 확인과, 리다이렉트를 하도록 권장 → getServerSideProps & middleware.ts

- getServerSideProps가 반환하는 props값은 모두 사용자의 HTML에 기록되고, 전역 변수로 등록되어 스크립트로 충분히 접근할 수 있는 보안 위협에 노출되는 값이 된다.

- 서버 컴포넌트가 클라이언트 컴포넌트에 반환하는 props는 **반드시 필요한 값으로만 철저하게 제한**되어야 한다.

https://nextjs.org/docs/app/guides/content-security-policy

https://nextjs.org/blog/security-nextjs-server-components-actions

a 태그의 값에 적절한 제한 두기

`<a href='javascript:;'`

- 안티 패턴, 리액트에서도 비권장하는 방식이므로 경고문과 함께 렌더링됨
- 페이지 이동을 막고 특정 이벤트 핸들러를 작동시키기 위한 용도로 주로 사용
- `href`에 사용자가 입력한 주소를 넣을 수 있다면 피싱 사이트 같은 보안 이슈가 있을 수 있기 때문에 가능하다면 `origin`도 확인해 처리하는 것을 권장

```
//(상대경로만 허용)
const SafeInternalLink = ({ userInputUrl }: { userInputUrl: string }) => {
```

```
  const isSafe = userInputUrl.startsWith('/') && !userInputUrl.startsWith('//')

  return isSafe ? (
    <a href={userInputUrl}>페이지로 이동</a>
  ) : (
    <span>유효하지 않은 링크</span>
  )
}

//origin 체크

const SafeLinkButton = ({ userInputUrl }: { userInputUrl: string }) => {
  const handleClick = useCallback(() => {
    try {
      const url = new URL(userInputUrl)

      // 허용된 origin
      const allowedOrigins = [
        'https://yourdomain.com',
        'https://믿을만한 페이지~',
      ]

      if (allowedOrigins.includes(url.origin)) {
        window.location.href = userInputUrl
      }else{
      ...

  return <button onClick={handleClick}>이동하기</button>
}
```

HTTP 보안 헤더 설정하기

HTTP 보안 헤더

- 브라우저가 렌더링하는 내용과 관련된 보안 취약점을 미연에 방지하기 위해 브라우저와 함께 작동하는 헤더
- 웹사이트 보안에 **가장 기초적인 부분으로 HTTP 보안 헤더만 효율적으로 사용해도 많은 보안 취약점을 방지 가능**

Strict-transport-Security

모든 사이트가 HTTPS를 통해 접근해야 하며, 만약 HTTP로 접근할 경우 모든 시도는 HTTPS로 변경되게 하는 HTTP의 응답 헤더

```
Strict-Transport-Security: max-age=<expire-time>; includeSubDomains
```

- `<expire-time>`
  - 브라우저가 기억해야 되는 시간, 이 시간이 경과하면 HTTP로 로드한 후에 응답에 따라 HTTPS로 이동하는 등의 작업을 수행하고, 이 시간이 0으로 되어 있으면 헤더가 즉시 만료되고 HTTP로 요청 (권장값은 2년)
  - `includeSubDomains`가 있을 경우 이런 규칙이 모든 하위 도메인에도 적용됨.

X-XSS-Protection

→ (비표준 기술로 현재 사파리와 구형 브라우저에만 제공되는 기능)

- 페이지에서 XSS 취약점이 발견되면 페이지 로딩을 중단하는 헤더
- 그러나 전적으로 믿어서는 안되고 반드시 페이지 내부에서 XSS에 대한 처리가 필요

X-Frame-Options

- 페이지를 `frame`, `iframe`, `embed`, `object` 내부에서 렌더링을 허용할지를 나타낼 수 있는데, `X-Frame-Options`는 외부에서 자신의 페이지를 위와 같은 방식으로 삽입되는 것을 막아주는 헤더

```
X-Frame-Options: DENY
X-Frame-Options: SAMEORIGIN
```

- `DENY` : 위와 같은 프레임 관련 코드가 있으면 막음
- `SAMEORIGIN` : 같은 `origin` 에 대해서만 프레임 허용