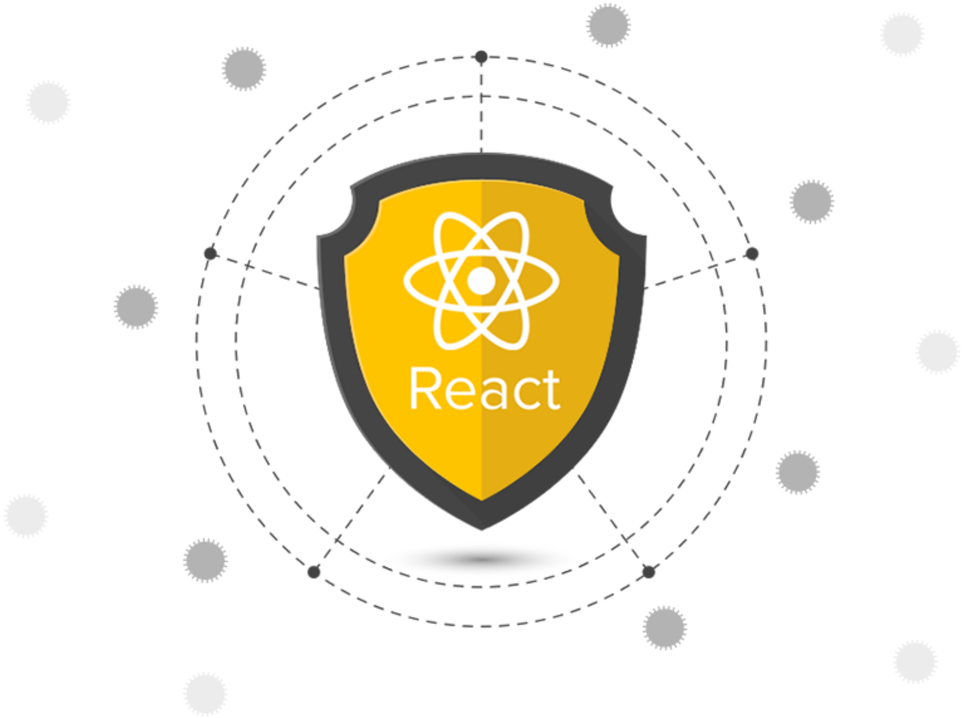
웹사이트 보안을 위한 리액트와 웹페이지 보안 이슈



2025.03.03
정상영

웹 사이트의 보안

1. 웹 사이트의 성능만큼 중요한 것이 **웹 사이트의 보안**
2. 개발자들은 높은 완성도와 성능을 가진 웹 사이트를 만드는 것과 동시에 **안전한 웹 사이트를 만들어야 할 책임감**도 가져야 함
3. 프론트엔드에서 해야 할 일이 많아질 수록 코드의 규모가 증가하며 이는 **필연적으로 보안 취약점에 노출될 확률도 증가**

1. Cross-Site Scripting(XSS)

- 제 3자가 악성 스크립트를 삽입해 실행할 수 있는 취약점

```
<p>사용자가 글을 작성했습니다</p>

<script>
  alert('XSS')
</script>
```

1.1 dangerouslySetInnerHTML prop

- 브라우저 DOM의 innerHTML을 특정한 내용으로 교체할 수 있는 방법

```
function App() {
  // 다음 코드의 결과물은 <div>First · Second</div>
  return <div dangerouslySetInnerHTML={{ __html: 'First &middot; Second' }} />
}
```

- dangerouslySetInnerHTML의 문제점은
  dangerouslySetInnerHTML이 **인수로 받는 문자열에 제한이 없는 것**

```
const dangerHtml = `<span><svg/onload=alert(origin)></span>`

function App() {
  return <div dangerouslySetInnerHTML={{ __html: dangerHtml }} />
}
```

=> dangerouslySetInnerHTML은 사용에 주의를 기울여야 함

1.2 useRef를 활용한 직접 삽입

- useRef를 활용하면 DOM에 접근할 수 있으므로 동일한 문제가 발생

```
const dangerHtml = `<span><svg/onload=alert(origin)></span>`

function App() {
  const divRef = useRef<HTMLDivElement>(null)

  useEffect(() => {
    if(divRef.current) {
      divRef.current.innerHTML = dangerHtml
    }
  })

  return <div ref={divRef} />
}
```

1.3 리액트에서 XSS 문제를 피하는 방법

- 가장 확실한 방법은 제 3자가 삽입할 수 있는 HTML을 **안전한 HTML로 치환하는 것**
- 이를 새니타이즈(sanitize) 혹은 이스케이프(escape)라고 함
- 직접 구현하거나 npm 라이브러리를 사용한다
- 대표적으로 **DOMpurity, sanitize-html, js-xss**가 있음

- sanitize-html 같은 **허용 목록(allow list) 방식**은 사용하기 번거롭지만 안전하다는 장점이 있음
- **사용자가 컨텐츠를 저장할 때도 이스케이프 과정을 거치는 것**이 보다 효율적이고 안전함, 애초에 데이터베이스에 저장하지 않는 것이 효율적
- 클라이언트의 요청은 모두 의심한다는 자세로 **치환 작업은 서버에서 수행하는 것이 좋음**

1.4 리액트의 JSX 데이터 바인딩

- dangerouslySetInnerHTML이 존재하는 이유가 뭘까?
- **리액트는 기본적으로 XSS를 방어**하기 위한 이스케이프 작업이 존재

```
const dangerHtml = `<span><svg/onload=alert(origin)></span>`

function App() {
  return <div id={dangerHtml}>{dangerHtml}</div>
}
```

=> 렌더링 시 그냥 텍스트로 나옴

2. getServerSideProps와 서버 컴포넌트를 주의하자

- SSR과 RSC는 성능 이점을 가져다 줌과 동시에 서버라는 개발 환경을 프론트엔드 개발자에게 일부 넘겨줌
- 서버에는 일반 사용자에게 노출되면 안되는 정보들이 있기 때문에 클라이언트에서 정보를 내려줄 때는 조심해야 함

문제점을 찾아보자

```tsx
export default function App({ cookie }: { cookie: string }) {
  if(!validateCookie(cookie)) {
    Router.replace(/*...*/)
    return null
  }

  /* do somthing */
}

export const getServerSideProps = async (ctx: GetServerSidePropsContext) => {
  const cookie = ctx.req.headers.cookie || ''
  return {
    props: {
      cookie,
    },
  }
}
```

문제점을 찾아보자

```tsx
export default function App({ cookie }: { cookie: string }) {
  if(!validateCookie(cookie)) {
    Router.replace(/*...*/)
    return null
  }

  /* do somthing */
}

export const getServerSideProps = async (ctx: GetServerSidePropsContext) => {
  const cookie = ctx.req.headers.cookie || ''
  return {
    props: {
      cookie,
    },
  }
}
```

1. 서버 사이드의 cookie 정보를 클라이언트 사이드에 문자열로 가져와 처리함
2. getServerSideProps가 반환하는 props는 사용자의 HTML에 기록되고, 전역 변수가 되어 스크립트로 접근 가능하기 때문에 보안 취약점임
3. 리다이렉트는 서버에서 처리할 수 있음에도 불구하고, 클라이언트 사이드에서처리해서 성능적 손해가 발생

```tsx
export default function App({ token }: { token: string }) {
  const user = JSON.parse(window.atob(token.split('.')[1]))
  const user_id = user.id

  /* do something */
}

export const getServerSideProps = async (ctx: GetServerSidePropsContext) => {
  const cookie = ctx.req.headers.cookie || ''

  const token = validateCookie(cookie)

  if(!token) {
    return {
      redirect {
        destination: '/404',
        permanent: false,
      },
    }
  }

  return {
    props: {
      token,
    },
  }
}
```

1. 클라이언트에서 필요한 token 값만 반환하도록 서버에서 처리
2. token이 없다면 리다이렉트하는 로직도 서버 사이드에서 처리

3. <a> 태그의 값에 적절한 제한을 둬야한다

```
function App() {
  function handleClick() {
    console.log('hello')
  }

  return (
    <>
      <a href="javascript:;" onClick={handleClick}>
        링크
      </a>
    </>
  )
}
```

1. href에 자바스크립트 코드가 있다면 자바스크립트가 동작 **(안티패턴)**
2. 페이지 라우팅 없이 이벤트 핸들러만 동작시키고 싶다면 <button>의 사용이 적절하다
3. 사용자가 입력한 주소를 넣는다면 보안 취약점이 될 수 있다

```tsx
function isSafeHref(href: string) {
  let isSafe = false

  try {
    // javascript:가 오면 protocol이 javascript:가 된다
    const url = new URL(href)
    if (['http:', 'https:'].includes(url.protocol)) {
      isSafe = true
    }
  } catch {
    isSafe = false
  }

  return isSafe
}
```

1. url의 프로토콜에 http나 https가 있다면 안전한 href로 판단

2. 반환하는 isSafe 값으로 a 태그의 동작을 결정한다

4. HTTP 보안 헤더 설정하기

- HTTP 보안 헤더란?

  브라우저가 렌더링하는 내용과 관련된 보안 취약점을 미연에 방지하기 위해 브라우저와 함께 작동하는 헤더

4.1 Strict-Transport-Security

- 모든 사이트가 HTTPS를 통해 접근해야 하며, 만약 HTTP로 접근하면 HTTPS로 변경되게 한다

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

- expire-time: 이 설정을 브라우저가 기억하는 시간, 권장값은 2년
- includesSubDomain: 이 설정이 하위 도메인에도 적용

4.2 X-XSS-Protection

- 현재는 사파리와 구형 브라우저에서만 제공되는 기능 (비표준 기술)
- 페이지에서 XSS 취약점이 발견되면 페이지 로딩을 중단

```
X-XSS-Protection: 0
X-XSS-Protection: 1
X-XSS-Protection: 1; mode=block
X-XSS-Protection: 1; report=<reportin-uri>
```

- 0: XSS 필터링을 끈다
- 1: (기본값) XSS 필터링을 켠다, 만약 XSS 공격이 감지되면 관련 코드를 제거한 안전한 페이지를 보여준다
- mode=block: 1과 유사하지만, 코드를 제거하는 것이 아니라 접근 자체를 막아버린다
- report=<reporting-uri>: 크로미움 기반 브라우저에서만 작동하며, XSS 공격이 감지되면 보고서를 reporting-uri로 보낸다

4.3 X-Frame-Options

- 페이지를 frame, iframe, embed, object 내부에서 렌더링을 허용할지 결정
- url이 비슷한 사이트에서 iframe으로 렌더링하는 공격을 막을 수 있음

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

- DENY: 무조건 막는다
- SAMEORIGIN: 같은 origin에 대해서만 허용한다

4.4 Permission-Policy

- 웹 사이트에서 사용할 수 있는 기능과 사용할 수 없는 기능을 명시적으로 선언

```
# 모든 geolocation을 막는다
Permission-Policy: geolocation=()

# geolocation을 페이지 자신과 몇 가지 페이지에 대해서만 허용
Permission-Policy: geolocation=(self "https://...")

# 카메라는 모든 곳에서 허용
Permission-Policy: camera=*;

# pip 기능을 막고, geolocation은 자신과 특정 페이지만 허용하며,
# 카메라는 모든 곳에서 허용한다
Permission-Policy: picture-in-picture=(), geolocation=(self https://...), camera=*;
```

- 제어할 수 있는 기능은 MDN 문서에서 확인할 수 있다
- permissionspolicy.com에서 선택해서 편리하게 헤더를 만드는 것도 가능하다

4.5 X-Content-Type-Options

- Content-type 헤더에서 제공하는 MIME 유형이 브라우저에 임의로 변경되지 않게 하는 헤더
- MIME: Multipurpose Internet Mail Extensions의 약자, Content-type의 값으로 사용
- text/css 헤더가 없는 파일은 브라우저가 임의로 CSS로 사용할 수 없으며, text/javascript나 application/javascript 헤더가 없는 파일은 자바스크립트로 해석할 수 없다.

```
X-Content-Type-Options: nosniff
```

- 브라우저가 서버에서 명시한 컨텐츠 타입을 신뢰하여 처리하고, 타입 스니핑을 방지

4.6 Referrer-Policy

- Referer 헤더: 현재 요청을 보낸 페이지의 주소
- Referer 헤더는 사용자가 어디서 왔는지 인식할 수 있지만,
  반대로 원치 않는 정보가 노출될 위험도 존재한다
- Referrer-Policy는 Referer 헤더에서 사용할 수 있는 데이터를 나타냄

4.6 Referrer-Policy

- origin(출처): scheme, hostname, port의 조합
- https://naver.com 는 다음과 같이 구성되어 있다
  - scheme: HTTPS 프로토콜
  - hostname: naver.com
  - port: 433 포트 (433이 기본값이기 때문에, port가 명시되어 있지 않다면 433을 의미한다)
- same-origin과 cross-origin의 구분은 이러한 origin을 기준으로 **scheme, hostname, port가 같을 때 same-origin를 의미한다**

4.6 Referrer-Policy

Referrer-Policy의 설정값	No Data (전송 X)	Origin Only (Origin만 전송)	Full URL (URL 전체 전송)	브라우저 기본값
no-referrer	✅			
origin		✅		
unsafe-url (권장 X)			✅	
strict-origin (프로토콜 보안 수준이 동일할 때만 Origin 전송)	낮은 scheme	동일한 scheme		
no-referrer-when-downgrade (보안 수준이 동일하거나 더 높을 때 Full URL 전송)	낮은 scheme		동일한 scheme	Microsoft Edge
origin-when-cross-origin (동일한 보안 수준의 same-origin Full URL, cross-origin과 낮은 보안 수준은 Origin)		cross-origin	same-origin	
same-origin (same-origin일 때만 Full URL 전송)	cross-origin		same-origin	
strict-origin-when-cross-origin (same-origin은 Full URL, cross-origin과 동일한 보안 수준은 Origin, 낮은 보안 수준은 전송 X)	낮은 scheme	cross-origin 동일한 scheme	same-origin	Chrome, Firefox, Safari

4.6 Referrer-Policy

```
<meta name="referrer" content="origin" />
```

- Referrer-Policy는 응답 헤더뿐만 아니라 페이지의 <meta> 태그로도 설정할 수 있다

```
<a href="http://..." referrerpolicy="origin" />
```

- 페이지 이동 시나 이미지 요청, link 태그 등에도 사용할 수 있다

4.7 Content-Security-Policy (CSP)

- XSS 공격이나 데이터 삽입 공격과 같은 다양한 보안 위협을 막기 위해 설계
- 사용할 수 있는 지시문이 굉장히 많음
- https://www.w3.org/TR/CSP3/ 참고

4.7 Content-Security-Policy (CSP)

1. ***-src**:
   font-src, img-src, script-src 등 다양한 src를 제어할 수 있는 지시문

```
Content-Security-Policy: font-src <source>;
```

- <source> 이외의 모든 폰트 소스는 차단된다

2. **form-action**:
   submit URL을 제한할 수 있다

4.8 & 4.9 보안 헤더 설정 및 확인하기

1. **Next.js**
   Next.js는 HTTP 경로별로 보안 헤더를 적용할 수 있으며,
   next.config.js에서 추가할 수 있다

2. **NGINX**
   경로별로 add_header 지시자를 사용해 원하는 응답 헤더를
   추가할 수 있다

3. https://securityheaders.com 을 방문하면 헤더를 확인하고 싶은
   웹 사이트의 보안 헤더를 확인할 수 있다

5. 취약점이 있는 패키지의 사용을 피하자

1. npm 프로젝트 구동을 위해서는 수많은 패키지에 의존해야 하는데, 개발자가 package.json 정도는 파악할 수 있지만 **pagkage-lock.json의 모든 의존성을 파악하기란 불가능**에 가깝다

2. 따라서 깃허브의 Dependabot과 같은 **의존성 관리 도구나 적절한 테스트 코트**를 통해 패키지의 업데이트에도 취약점이 발생하지 않도록 관리해줘야한다

6. OWASP Top 10

- OWASP는 **Open Worldwide (Web) Application Security Project**를 의미한다
- 주로 웹에서 발생할 수 있는 정보 노출, 악성 스크립트, 보안 취약점 등을 연구하며 주기적으로 10대 취약점을 공개하는데 이를 OWASP Top 10이라고 한다
- https://owasp.org/www-project-top-ten/