Pragmatic Web Security

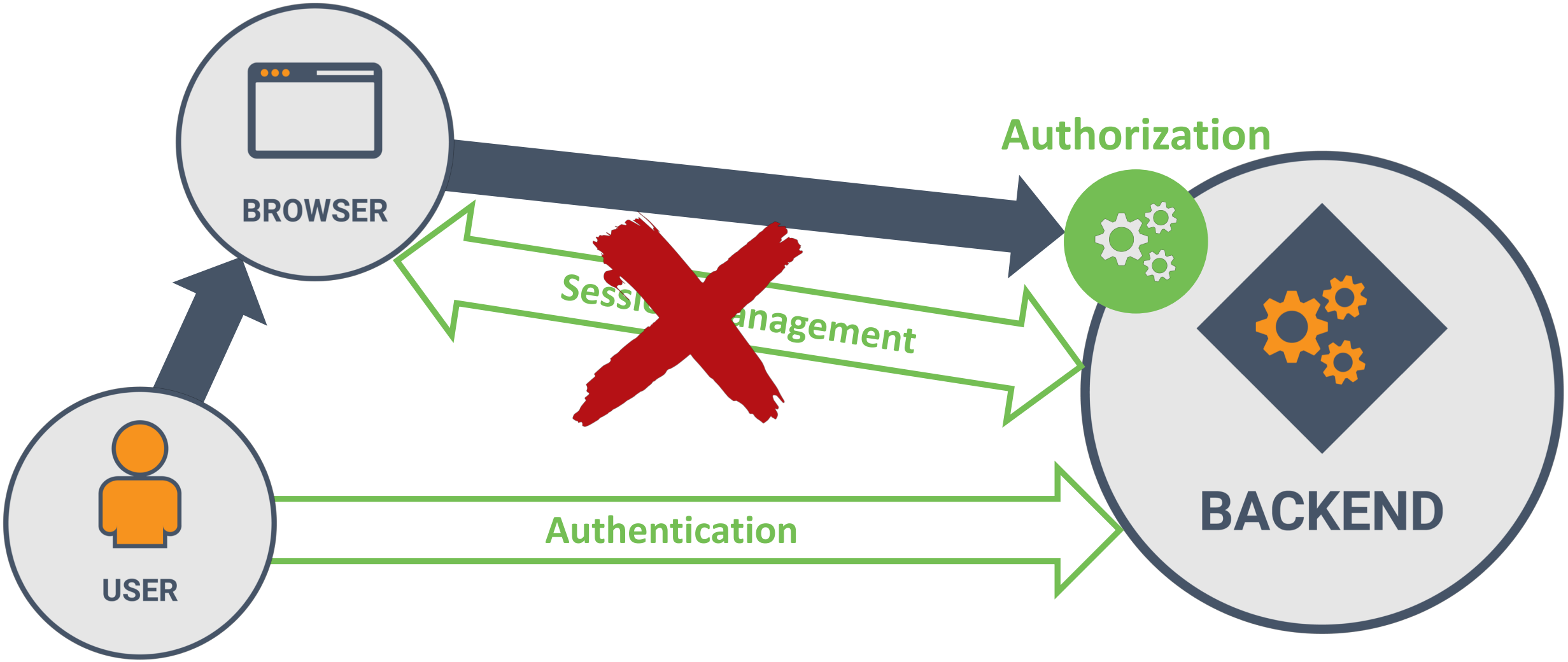
Security training for developers

Cookies vs Tokens: a paradoxical choice

Philippe De Ryck

@PhilippeDeRyck – philippe@PragmaticWebSecurity.com

SHOULD YOU EVER USE COOKIES FOR YOUR API?


BROWSER
Authorization
Session Management
BACKEND
USER
Authentication

- Founder of **Pragmatic Web Security**
  - In-depth web security training for developers
  - Covering web security, API security & Angular security
- 15+ years of security experience
  - Web security instructor and conference speaker
  - Author of ***Primer on client-side web security***
  - Creator of ***Web Security Fundamentals*** on edX
- Course curator of the **SecAppDev course**
  - Yearly security course targeted towards developers
  - More information on ***https://secappdev.org***
- Foodie and professional chef

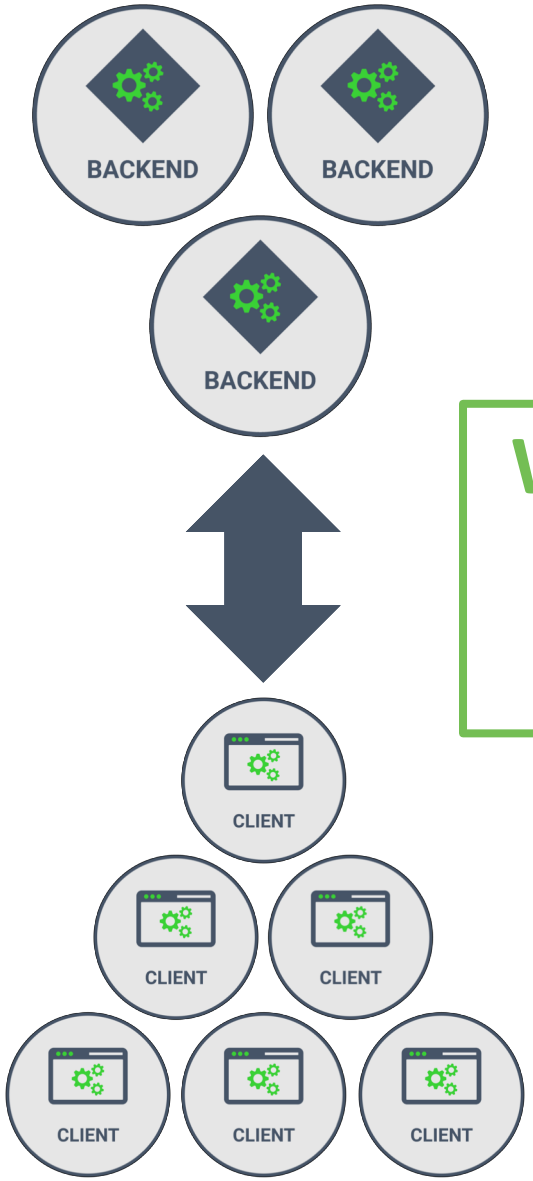
Dr. Philippe De Ryck

Ph.D. in web security

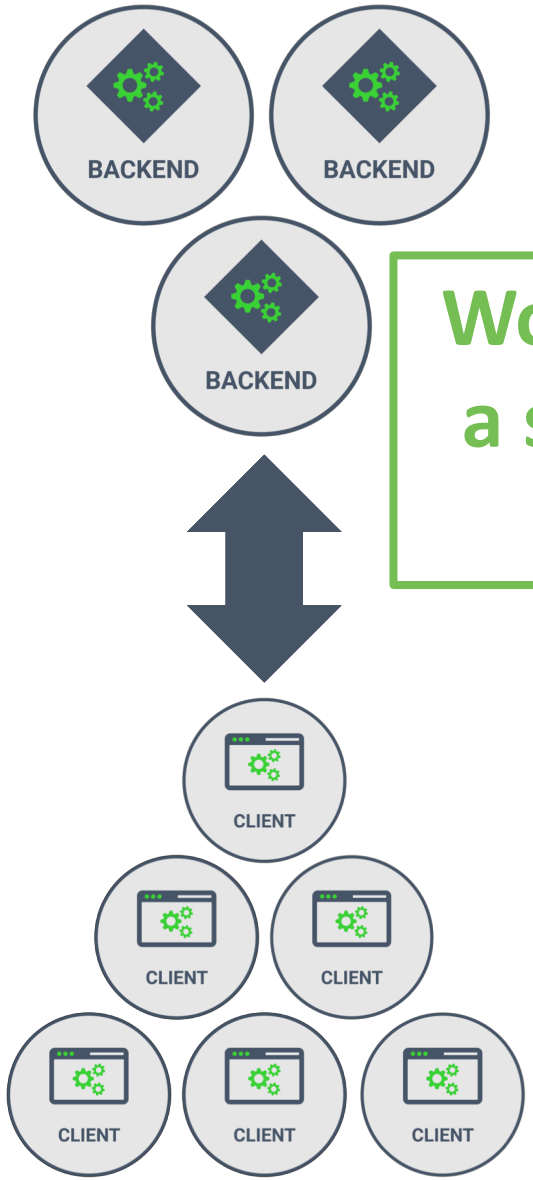
Google developer expert

@PhilippeDeRyck

https://PragmaticWebSecurity.com

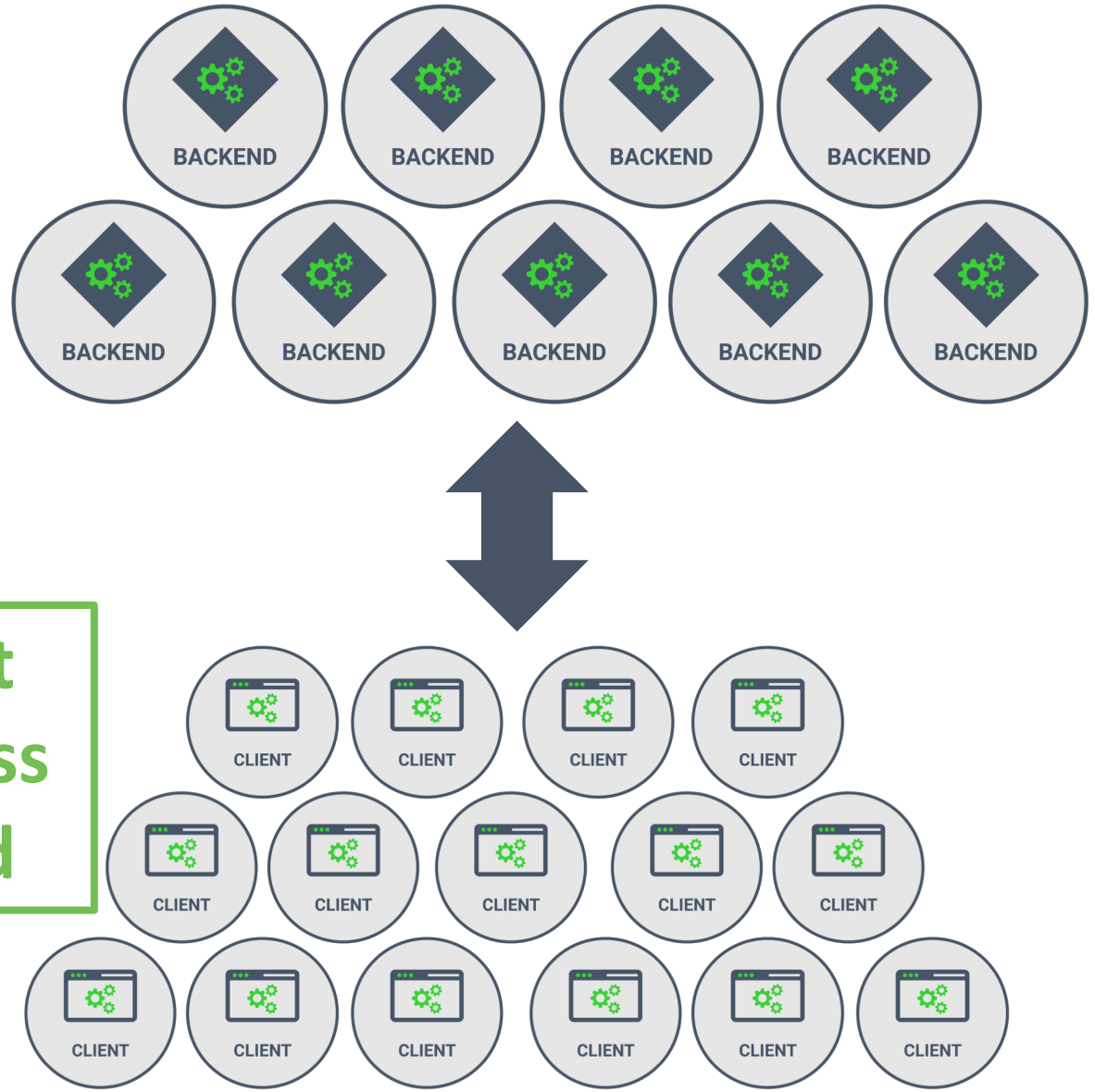


Works fine with a stateful ~~REST~~ backend



Works fine with a stateful ~~REST~~ backend

Might benefit from a stateless REST backend

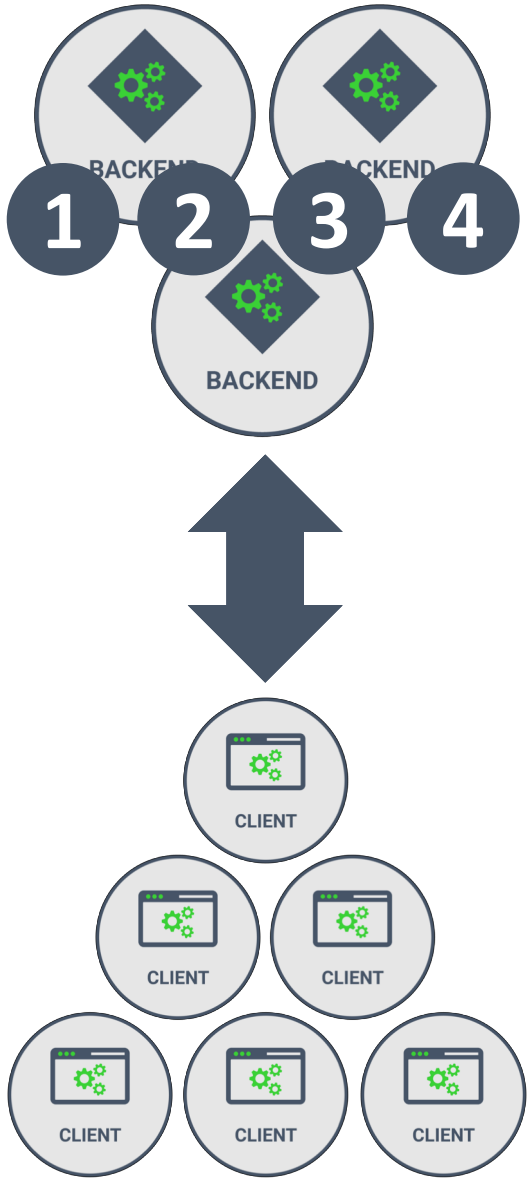


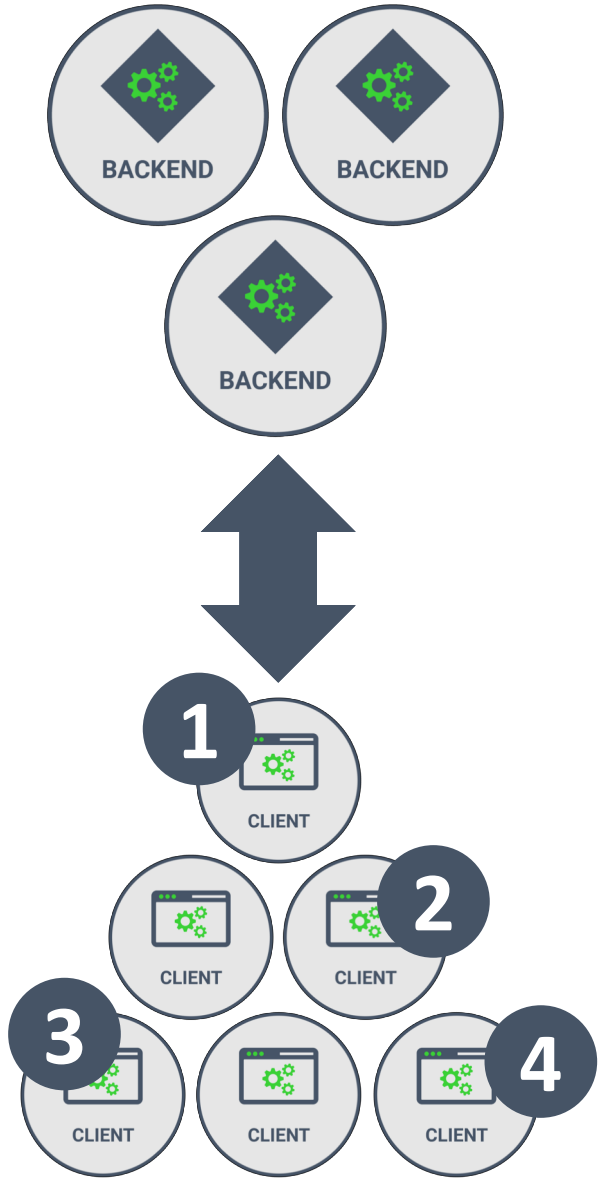
THE TRUTH IS A LOT MORE COMPLICATED

- Pure REST APIs should be stateless
  - The server is stateless, and the client provides all the required information
  - A valid argument for stateless backends is flexible scalability
- Purity is rarely a good argument to throw working solutions overboard
  - An API can just as well keep session state on the server
  - Works perfectly well with small to medium-scale applications
  - Makes scalability harder, but not impossible
    - We have been doing this for 20 years with sticky sessions, session replication, ...
- OAuth 2.0 is commonly used in both a stateful and stateless manner
  - The debate on reference tokens vs self-contained tokens is essentially the same issue

SESSION DATA REPRESENTATION AND LOCALITY

How will you represent session data?
Do you keep the data on the client or the server?


BACKEND
BACKEND
1
2
3
4
BACKEND
CLIENT
CLIENT
CLIENT
CLIENT
CLIENT
CLIENT


BACKEND
BACKEND
BACKEND
1
CLIENT
CLIENT
CLIENT
2
3
CLIENT
CLIENT
CLIENT
4

THE LOCALITY OF SESSION DATA IMPACTS SECURITY

- Server-side sessions share an ID with the client and store data on the server
  - Attacks on session management focus on guessing or stealing the ID
  - The data stored in the server-side session object can be considered trusted
- Client-side sessions are a completely different paradigm
  - The actual data is stored on the client, so it can be easily accessed
  - The data comes in from the client, and is untrusted by default
- Client-side sessions require additional data protection measures
  - Mandatory integrity checks to detect tampering with the data
  - Optional confidentiality mechanisms to prevent disclosure of information

Encoded PASTE A TOKEN HERE


```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IlBoaWxpcHBlIERlIFJ5Y2siLCJyb2xlcyI6InVzZXIgcmVzdGF1cmFudG93bmVyIiwiaWF0IjoxNTE2MjM5MDIyfQ.KPjhyE9oi83uehgw6Lm_0yAZzRuJhcUqXETD2AIrF2A
```


Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE


```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

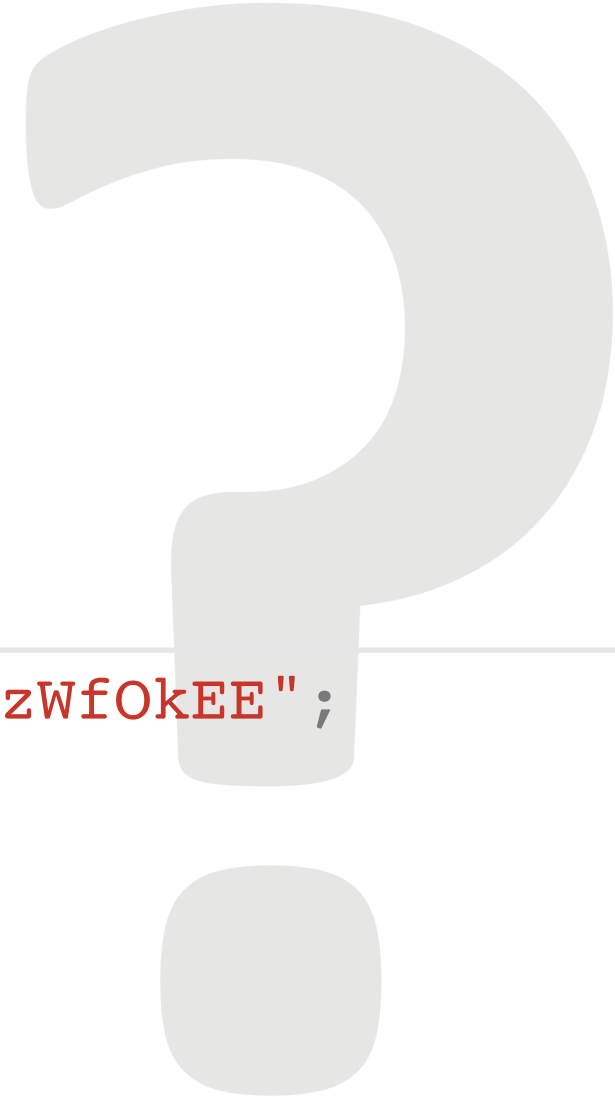

PAYLOAD: DATA


```
{
  "sub": "1234567890",
  "name": "Philippe De Ryck",
  "roles": "user restaurantowner",
  "iat": 1516239022
}
```


VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  SuperSecretHMACKey
) ☐ secret base64 encoded
```

CAN YOU SPOT A PROBLEM HERE?



```
1  String token = "eyJhbGciOiJIUzI1NiIsInR5c...zWfOkEE";
2  try {
3      DecodedJWT jwt = JWT.decode(token);
4  } catch (JWTDecodeException exception){
5      //Invalid token
6  }
```

```
String token = "eyJhbGciOiJIUzI1NiIsInR5c...zWfOkEE";
try {
    DecodedJWT jwt = JWT.decode(token);
} catch (JWTDecodeException exception){
    //Invalid token
}
```

Decoding only

```
String token = "eyJhbGciOiJIUzI1NiIsInR5c...zWfOkEE";
try {
    Algorithm algorithm = Algorithm.HMAC256("secret");
    JWTVerifier verifier = JWT.require(algorithm)
        .build(); //Reusable verifier instance
    DecodedJWT jwt = verifier.verify(token);
} catch (JWTVerificationException exception){
    //Invalid signature/claims
}
```

Signature verification

Encoded PASTE A TOKEN HERE


```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IlBoaWxpcHBlIERlIFJ5Y2siLCJyb2xlcyI6InVzZXIgcmVzdGF1cmFudG93bmVyIiwiaWF0IjoxNTE2MjM5MDIyfQ.KPjhyE9oi83uehgw6Lm_0yAZzRuJhcUqXETD2AIrF2A
```


Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE


```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

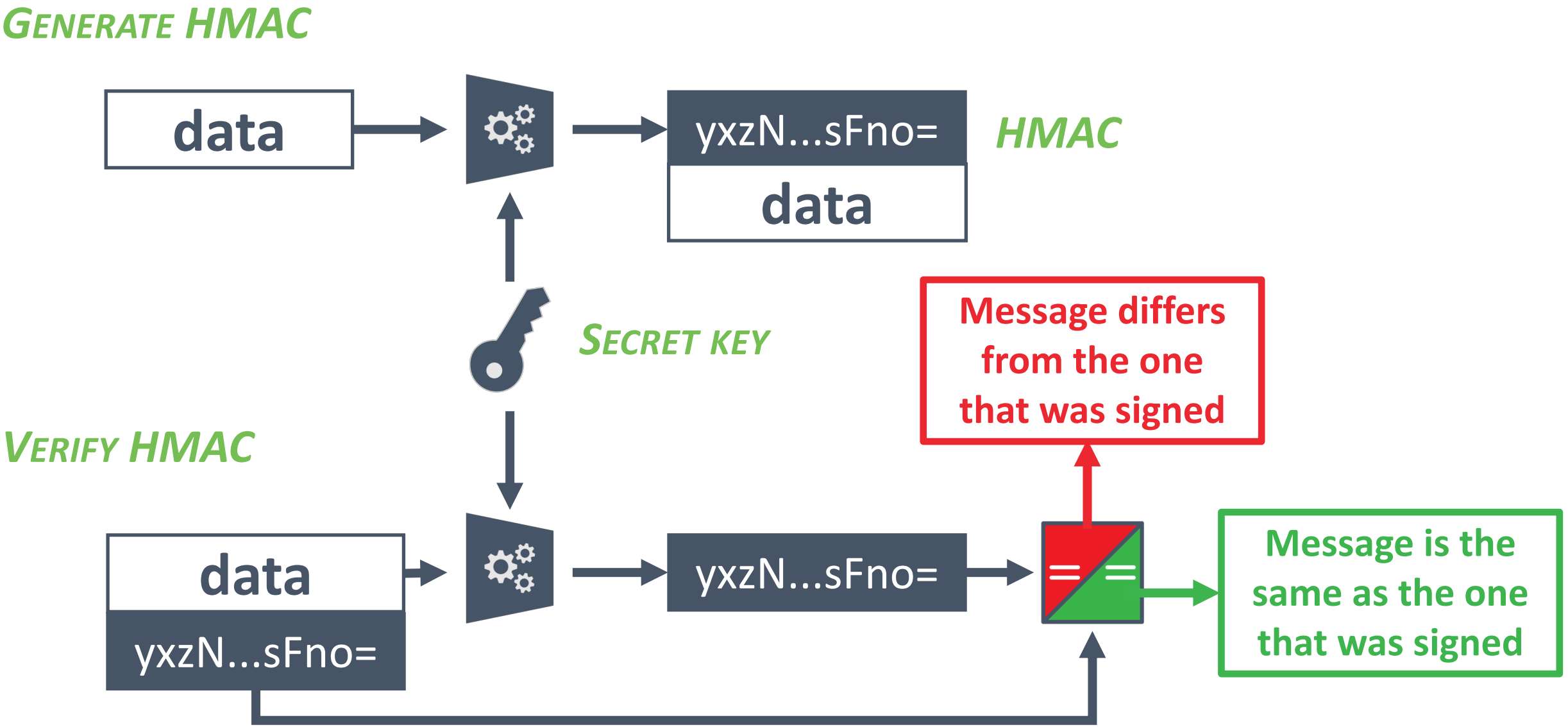

PAYLOAD: DATA


```
{
  "sub": "1234567890",
  "name": "Philippe De Ryck",
  "roles": "user restaurantowner",
  "iat": 1516239022
}
```

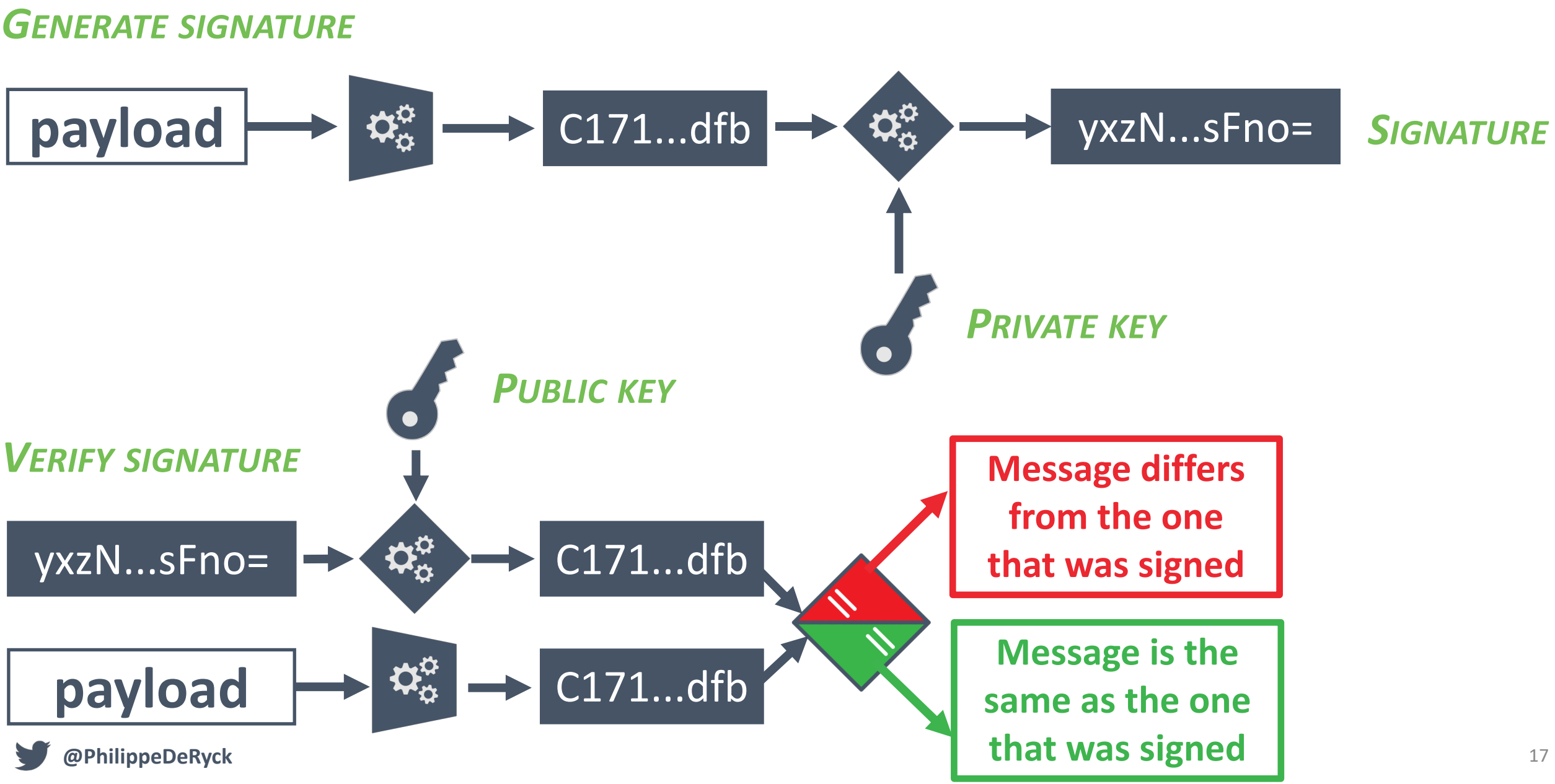

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  SuperSecretHMACKey
) ☐ secret base64 encoded
```

HMAC-BASED JWT SIGNATURES



Asymmetric JWT signatures



JWT Signatures

- JWTs support both symmetric and asymmetric signatures
  - Symmetric signatures are HMACs that depend on a shared secret key
  - Asymmetric are digital signatures that depend on a public/private key pair

- Symmetric signatures are useful to use within a single trust zone
  - Backend service storing claims in a JWT for use within the application
  - Symmetric signatures are not the right choice when other (internal) services are involved
    - ***Never ever share your secret signing key!***

- Asymmetric signatures are useful in distributed scenarios
  - SSO or OAuth 2.0 scenarios using JWTs to transfer claims to other services
  - Everyone with the public key can verify the signature
  - Used in OpenID Connect (e.g., social login scenarios)

Session data storage

Where do you store your session data in the browser?

Your API-Centric Web App Is Probably Not Safe Against XSS and CSRF

Most of the developments I've participated in recently follow the "single-page application based on a public API with authentication" architecture. Using Angular.js or React.js, and based on a RESTful API, these applications move most of the complexity to the client side.

"The browser offers a storage that can't be read by JavaScript: HttpOnly cookies. It's a good way to identify a requester without risking XSS attacks."

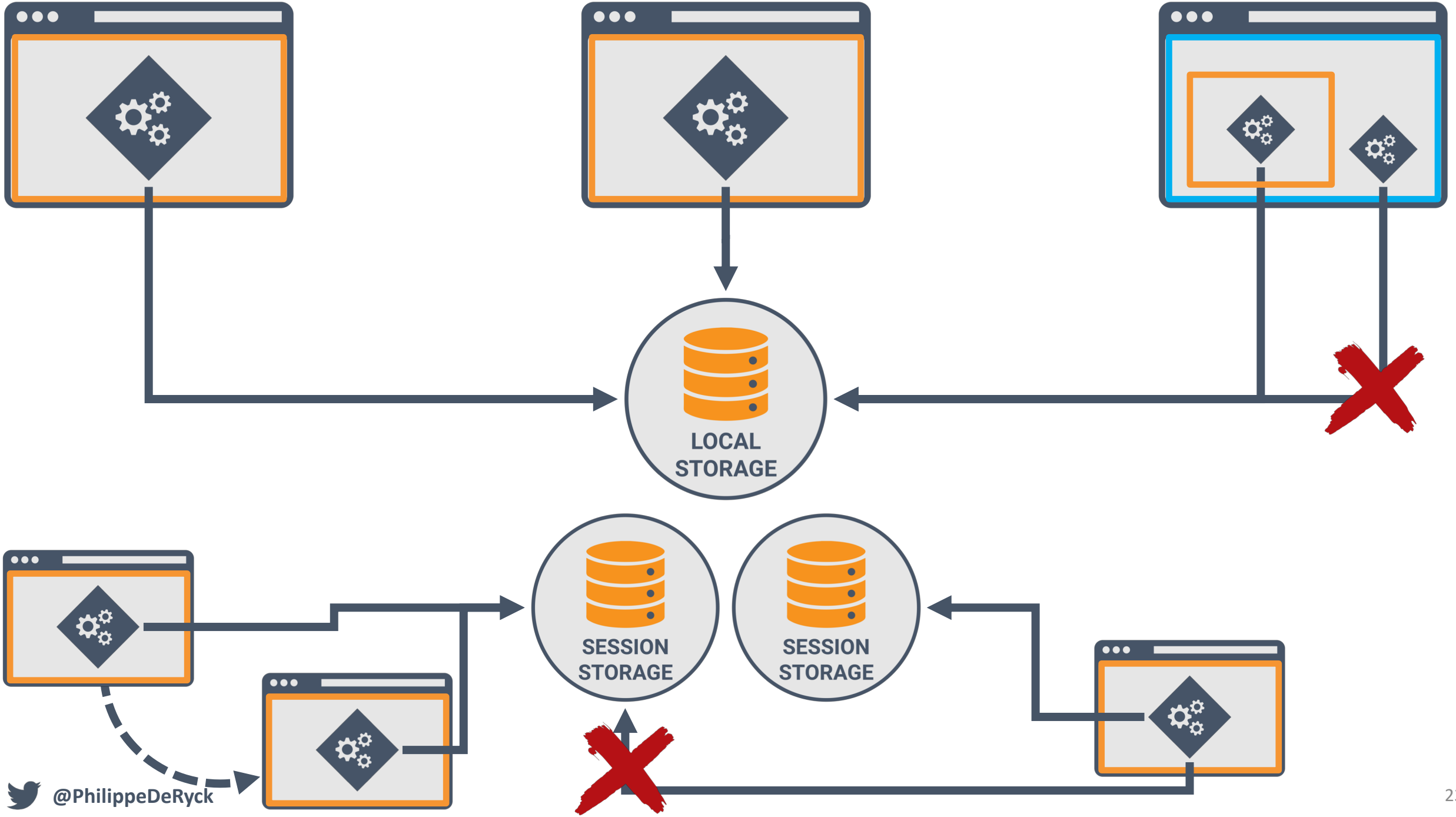
HttpOnly cookies

The deal with HttpOnly

- The ***HttpOnly*** flag resolves a consequence of an XSS attack
  - Stealing the session identifier becomes a lot harder
  - **But you still have an XSS vulnerability in your application**
    - XSS allows the attacker to execute arbitrary code
    - That code can trigger authenticated requests, modify the DOM, ...

- ***HttpOnly*** is still recommended, because it raises the bar
  - XSS attacks become a little bit harder to execute and to persist
  - XSS attacks from subdomains become less powerful (with domain-based cookies)

- In Chrome, ***HttpOnly*** prevents cookies from entering the rendering process
  - Useful to reduce the impact of CPU-based ***Spectre*** and ***Meltdown*** attacks


LOCAL
STORAGE
SESSION
STORAGE
SESSION
STORAGE

Comparing client-side storage mechanisms

localStorage	sessionStorage	In-memory	Cookies
Available to the entire origin	Available to the window and children	Available to running code only	Can be fully hidden from JavaScript
Survives a page reload	Survives a page reload	Does not survive a page reload	Survives a page reload
Cannot be shielded from malicious code	Can be a bit shielded from malicious code	Can be shielded from malicious code	Can be shielded from malicious code
Application code required for handling	Application code required for handling	Application code required for handling	Application code not required for handling

Session data transport

How will you send session data to the server?

Cookie: **ID=42**

Authorization: **Bearer 42**

Cookie: **JWT=eyJhbGci…**

Authorization: **Bearer eyJhbGci…**

WHICH OF THESE IS THE BEST PRACTICE FOR ISOLATED APPLICATIONS?

A. **Session=...; Secure; HttpOnly**

B. **__Secure-Session=...; Secure; HttpOnly**

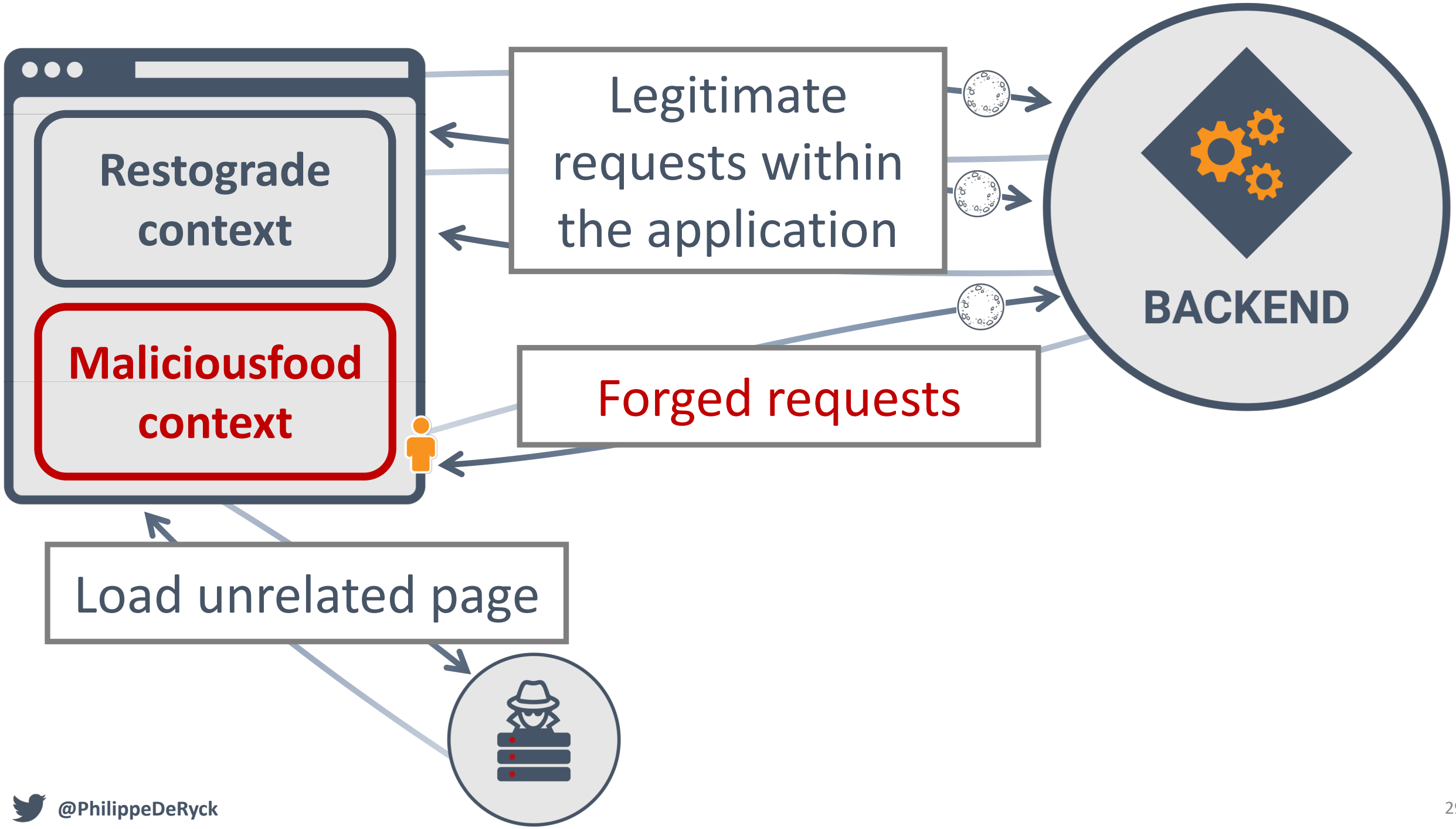
C. **__Host-Session=...; Secure; HttpOnly**

D. **__Host-Session=...; Secure; HttpOnly; SameSite**

E. **__Host-Session=...; Secure; HttpOnly; SameSite; LockOrigin**

Recommendations for secure cookies

- Since everything runs over HTTPS, cookies can be locked down
  - Set the ***Secure*** flag on all cookies
  - Add the _ _ ***Secure-*** prefix to all cookies
- Most cookies do not need to be accessed from JavaScript
  - Set the ***HttpOnly*** flag on those cookies
- Most cookies are set and used by one application only
  - Do not set the ***Domain*** attribute on cookies
  - Replace the _ _ ***Secure-*** prefix with the _ _ ***Host-*** prefix


Restograde context
Maliciousfood context
Legitimate requests within the application
Forged requests
BACKEND
Load unrelated page

Defending against CSRF attacks

- To defend against CSRF, the application must identify forged requests
  - By design, there is no way to identify if a request came from a malicious context
  - The ***Referer*** header may help, but is not always present
- Common CSRF defenses add a secret token to legitimate requests
  - Only legitimate contexts have the token
  - Attackers can still make requests with cookies, but not with the secret token
- Recently, additional client-side security mechanisms have been introduced
  - The ***Origin*** header tells the server where a request is coming from
  - The ***SameSite*** cookie flag prevents the use of cookies on forged requests

Overview of CSRF defenses

- Hidden form tokens (synchronizer tokens)
  - Requires server-side storage of CSRF tokens, which may be resource-intensive
- Double submit cookies (transparent tokens)
  - Stateless CSRF defense mechanism
  - Extremely compatible with client-side JavaScript applications (e.g. AngularJS)
- Checking the origin header
  - Useful when other context information is missing
  - Plays an important role when accessing APIs with Cross-Origin Resource Sharing (CORS)
  - Practical defense during the setup of a WebSocket connection
- SameSite cookies
  - Addresses the root of the problem, but browser support is still limited

In a common anti-XSRF technique, the application server sends a randomly generated authentication token in a cookie. The client code reads the cookie and adds a custom request header with the token in all subsequent requests. The server compares the received cookie value to the request header value and rejects the request if the values are missing or don't match.

This technique is effective because all browsers implement the *same origin policy*. Only code from the website on which cookies are set can read the cookies from that site and set custom headers on requests to that site. That means only your application can read this cookie token and set the custom header. The malicious code on `evil.com` can't.

Angular's `HttpClient` has built-in support for the client-side half of this technique. Read about it more in the HttpClient guide.

For information about CSRF at the Open Web Application Security Project (OWASP), see Cross-Site Request Forgery (CSRF) and Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet. The Stanford University paper Robust Defenses for Cross-Site Request Forgery is a rich source of detail.

“Angular's HttpClient has built-in support for [the double submit cookie pattern]”

```
'request': function (config) {
    config.headers = config.headers || {};
    if ($localStorage.token) {
        config.headers.Authorization = 'Bearer ' + $localStorage.token;
    }
    return config;
},
```

```
@Injectable()
export class TokenInterceptor implements HttpInterceptor {

  constructor(public auth: AuthService) {}

  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    request = request.clone({
      setHeaders: {
        Authorization: `Bearer ${this.auth.getToken()}`
      }
    });

    return next.handle(request);
  }
}
```

CAN YOU SPOT THE SECURITY ISSUE HERE?

```
@Injectable()
export class TokenInterceptor implements HttpInterceptor {

  constructor(public auth: AuthService) {}

  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    request = request.clone({
      setHeaders: {
        Authorization: `Bearer ${this.auth.getToken()}`
      }
    });

    return next.handle(request);
  }
}
```

```
import { JwtModule } from '@auth0/angular-jwt';
import { HttpClientModule } from '@angular/common/http';

export function tokenGetter() {
  return localStorage.getItem('access_token');
}

@NgModule({
  bootstrap: [AppComponent],
  imports: [
    // ...
    HttpClientModule,
    JwtModule.forRoot({
      config: {
        tokenGetter: tokenGetter,
        whitelistedDomains: ['localhost:3001'],
        blacklistedRoutes: ['localhost:3001/auth/']
      }
    })
  ]
})
export class AppModule {}
```

SECURITY CONSIDERATIONS WITH CUSTOM TRANSPORT MECHANISMS

- Implementing a custom transport mechanism has security implications
  - All of a sudden, developers need to implement code to attach session data to requests
  - Angular interceptors look simple enough, but are often insecure

- Interceptors are applied to ***every*** outgoing request
  - The moment you send a request to a third-party API, the interceptor adds session data
  - This would leak session data to a third party, allowing them to take over the session
  - Instead, the interceptor should only attach data to whitelisted origins

- Good libraries support whitelisting out of the box
  - The ***@auth0/angular-jwt*** library is popular to use JWT with the ***Authorization*** header
  - Allows you to decode and extract the JWT information
  - Supports adding tokens based on a whitelist of origins

How to authorize the loading of DOM resources (img, script, ...)?

(DIS)ADVANTAGES OF THE *AUTHORIZATION* HEADER

- The ***Authorization*** header offers a lot of flexibility
  - Custom control over where and how to add session data in the header
  - Not tied to a specific domain, so easy to support APIs on different domains
    - Cookies are tied to a domain, so are hard to use in such a context
  - No more dealing with cookie security flags and ***Cross-Site Request Forgery (CSRF)***
  - The downside here is that you need to make sure your code is secure

- The ***Authorization*** header is not handled by the browser in any way
  - DOM resources being loaded will not carry any session information
    - Loading scripts, images, stylesheets through HTML elements
  - CORS requests with credentials will carry cookies, but not an ***Authorization*** header
    - Calling third-party APIs requires the application to explicitly obtain session information

COOKIES

Can contain identifiers & session objects

Only works well with a single domain

Automatically handled by the browser

Always present, including on DOM resources

AUTHORIZATION HEADER

Can contain identifiers & session objects

Freedom to include headers to any domain

Requires custom code to get, store and send session data

Only present on XHR calls, unless you add it through a ServiceWorker

Should you ever use cookies for your API?

Sure, why not?

Your API-Centric Web App Is Probably Not Safe Against XSS and CSRF

Most of the developments I've participated in recently follow the "single-page application based on a public API with authentication" architecture. Using Angular.js or React.js, and based on a RESTful API, these applications move most of the complexity to the client side.

"The browser offers a storage that can't be read by JavaScript: HttpOnly cookies. It's a good way to identify a requester without risking XSS attacks."

joepie91's Ramblings RSS

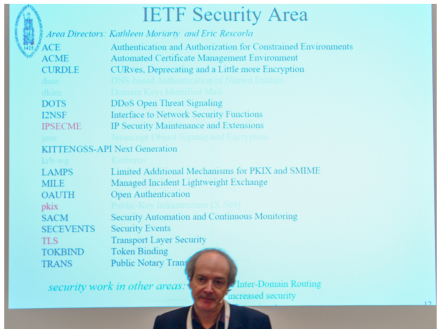
Stop using JWT for sessions

13 Jun 2016

> Unfortunately, lately I've seen more and more people recommending to use JWT for managing user sessions in their web applications. This is a terrible, terrible idea

SecAppDev 2019

February 18 - 22, Leuven, Belgium



1-day workshops

Building secure web & web service applications
Jim Manico

Whiteboard hacking (aka hands-on Threat Modeling)
Sebastien Deleersnyder

Securing Kubernetes the hard way
Jimmy Mesta

5-day dual-track program

Crypto, AppSec Processes, web security, access control, mobile security, ...

Pragmatic Web Security

Security training for developers

/in/PhilippeDeRyck

@PhilippeDeRyck

philippe@pragmaticwebsecurity.com