



OWASP

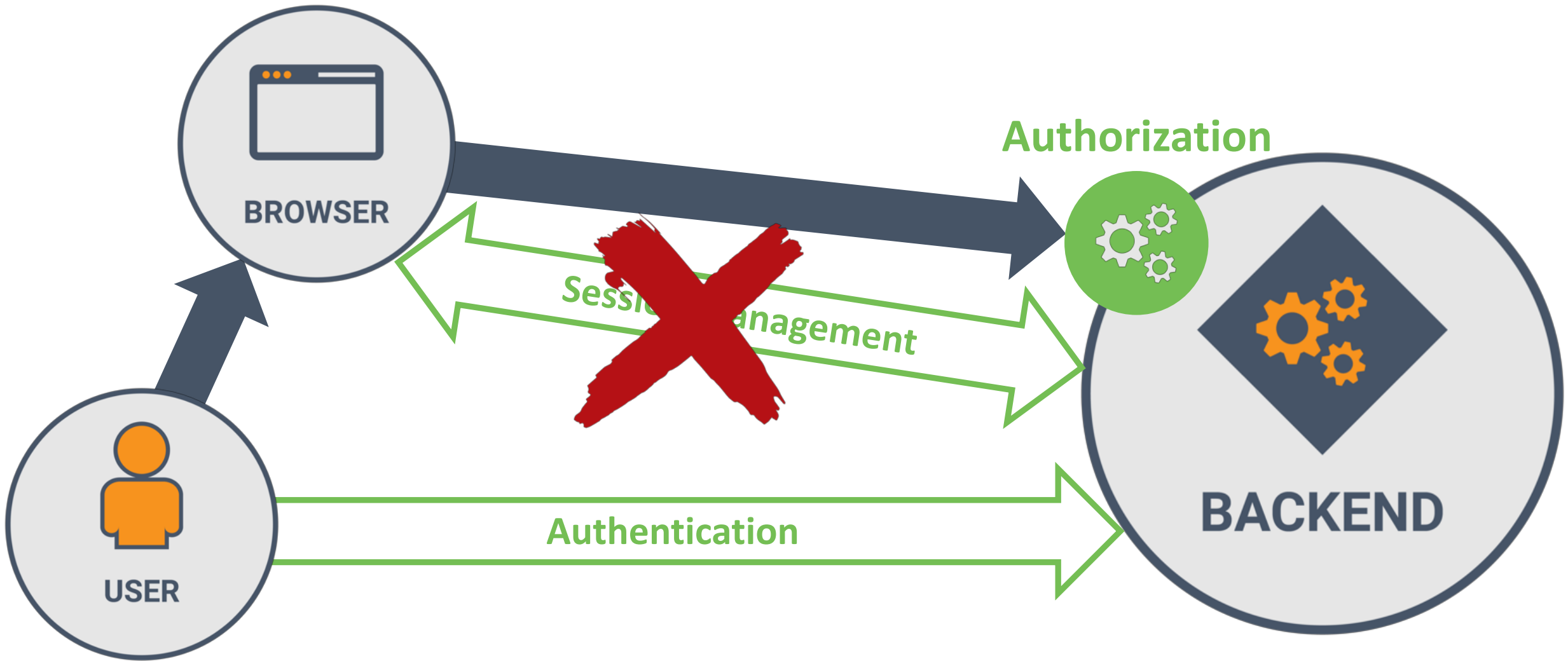
AppSec Conference
Bucharest 2018

COOKIES VS TOKENS

A PARADOXICAL CHOICE

SHOULD YOU EVER USE
COOKIES FOR YOUR API?





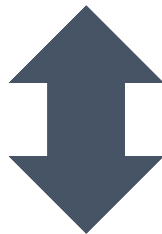
- Founder of **Pragmatic Web Security**
 - In-depth web security training for developers
 - Covering web security, API security & Angular security
- 15+ years of security experience
 - Web security instructor and conference speaker
 - Author of *Primer on client-side web security*
 - Creator of *Web 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

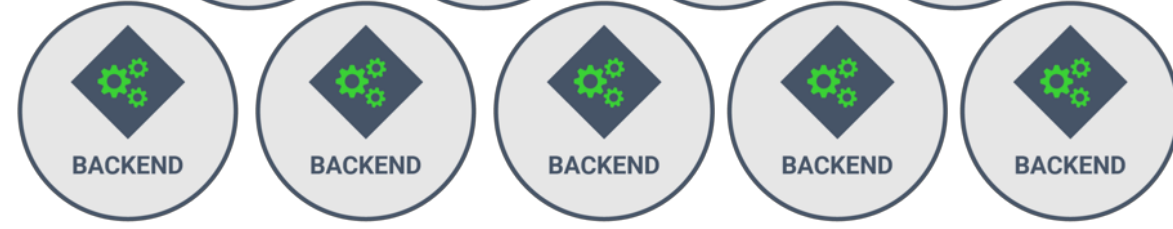
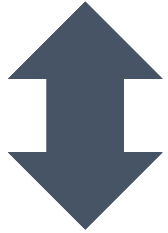
GOOGLE DEVELOPER EXPERT



Works fine with
a stateful ~~REST~~
backend



Works fine with
a stateful ~~REST~~
backend



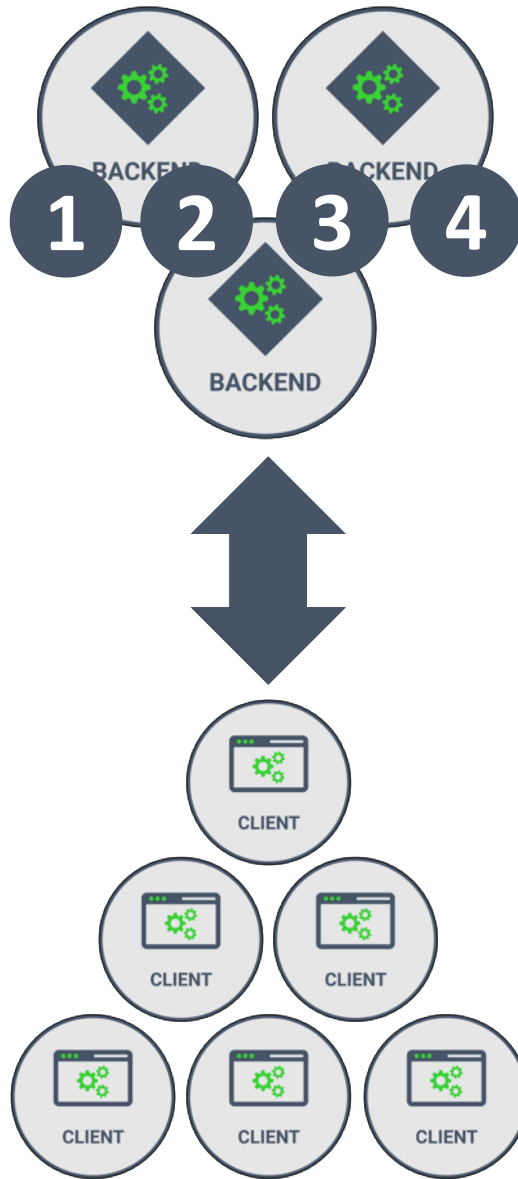
Might benefit
from a stateless
REST backend

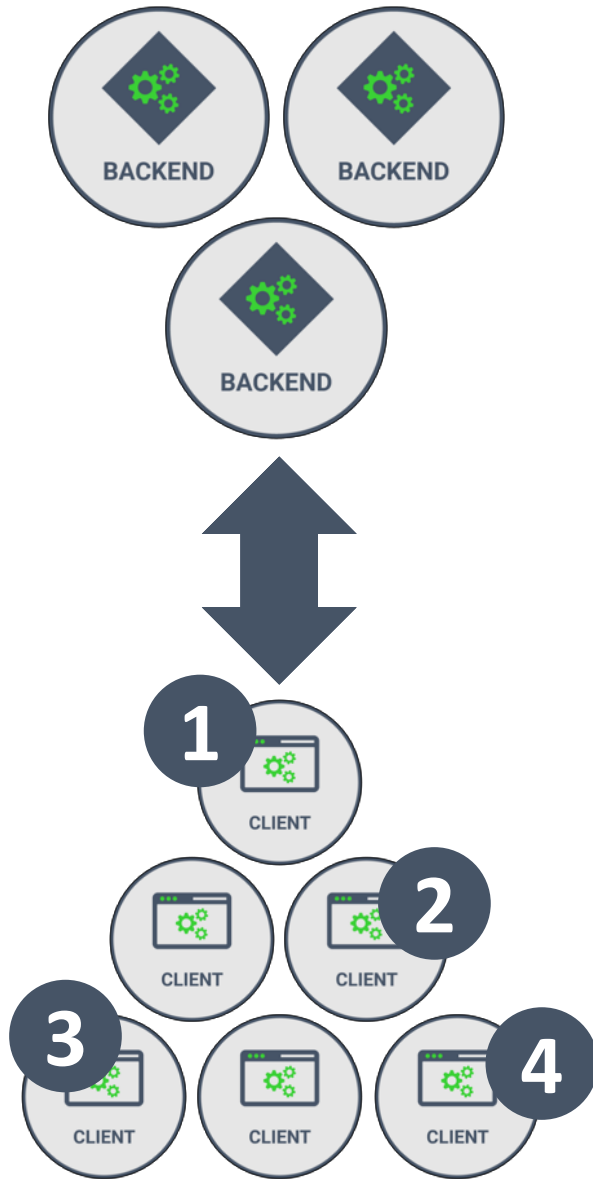


SESSION DATA REPRESENTATION AND LOCALITY



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





Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6ImlwIiwiaWF0IjoiYXZzRUJhcUqXETD2A
```

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

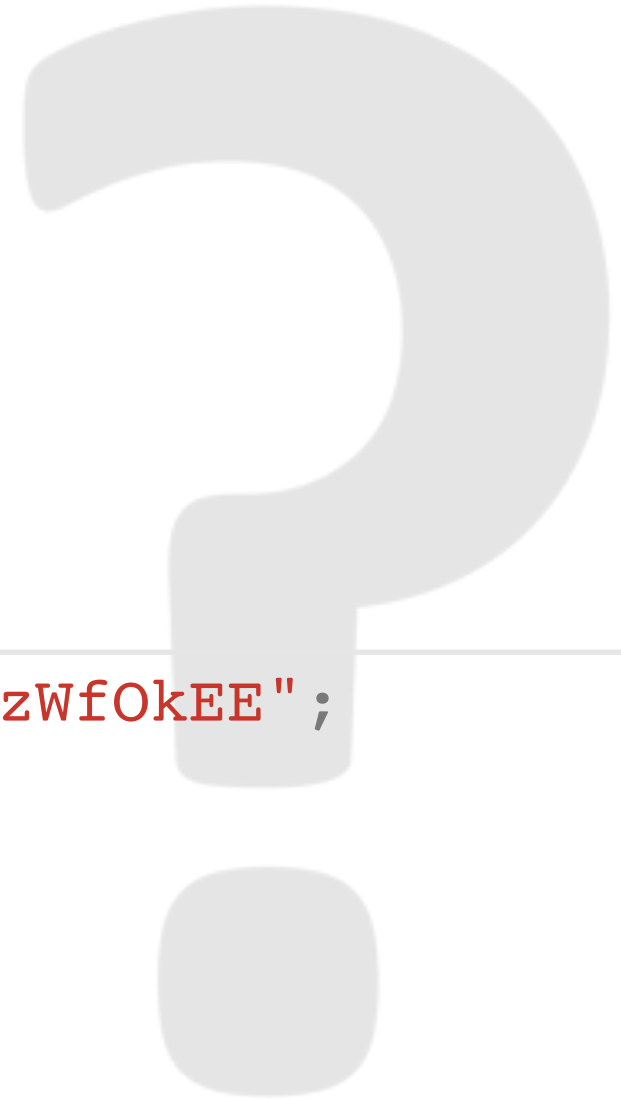
PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "Philippe De Ryck",
  "roles": "user restaurantowner",
  "iat": 1516239822
}
```

VERIFY SIGNATURE


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

CAN YOU SPOT A PROBLEM HERE?




```
1 String token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IWRXb2kEE";  
2 try {  
3     DecodedJWT jwt = JWT.decode(token);  
4 } catch (JWTDecodeException exception){  
5     //Invalid token  
6 }
```

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



Decoding only

```
1 String token = "eyJhbGciOiJIUzI1NiIsInR5c...zWfOkEE";
2 try {
3     Algorithm algorithm = Algorithm.HMAC256("secret");
4     JWTVerifier verifier = JWT.require(algorithm)
5         .build(); //Reusable verifier instance
6     DecodedJWT jwt = verifier.verify(token);
7 } catch (JWTVerificationException exception) {
8     //Invalid signature/claims
9 }
```



Signature verification



Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IiBoaWwibG9uZSIsInR5cCI6IkpXVCJ9.eyJzdGF1cmFudG93bmVyaWwifQ.KPjhyE9oi83uehgw6Lm_0yAZzRuJhcUqXETD2AIRF2A
```

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

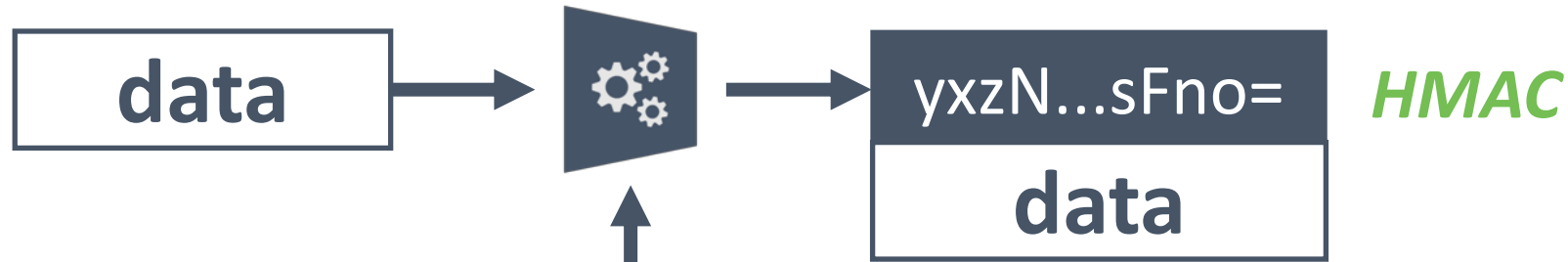
```
{
  "sub": "1234567890",
  "name": "Philippe De Ryck",
  "roles": "user restaurantowner",
  "iat": 1516239822
}
```

VERIFY SIGNATURE

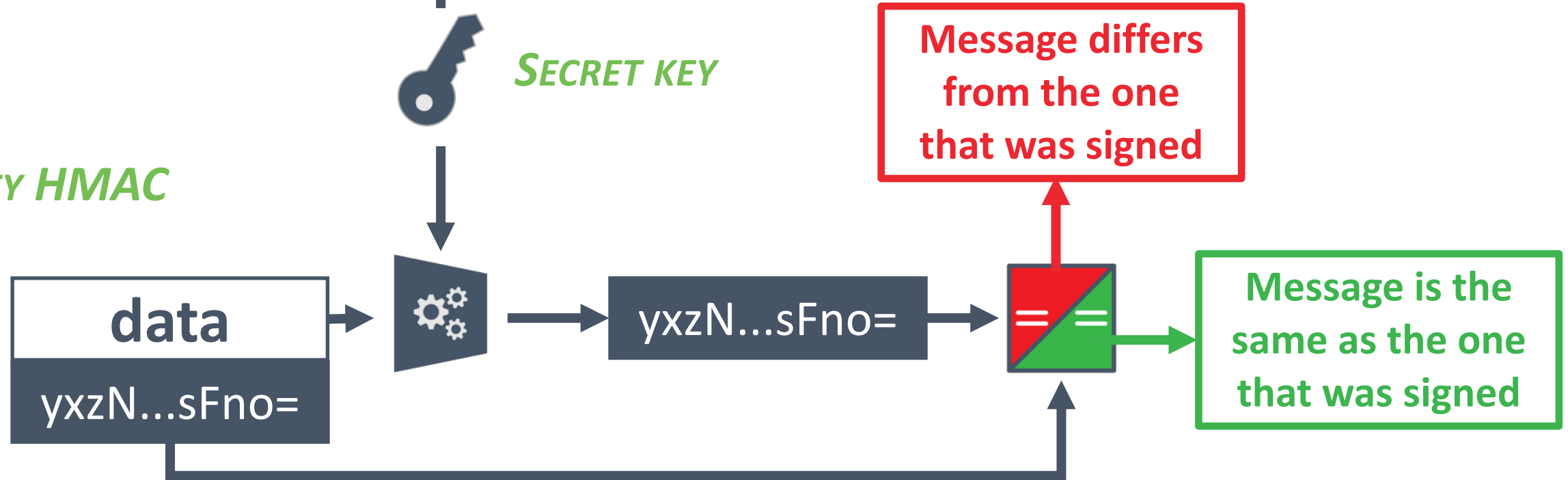
```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  SuperSecretHMACKey
)  secret base64 encoded
```

HMAC-BASED JWT SIGNATURES

GENERATE HMAC

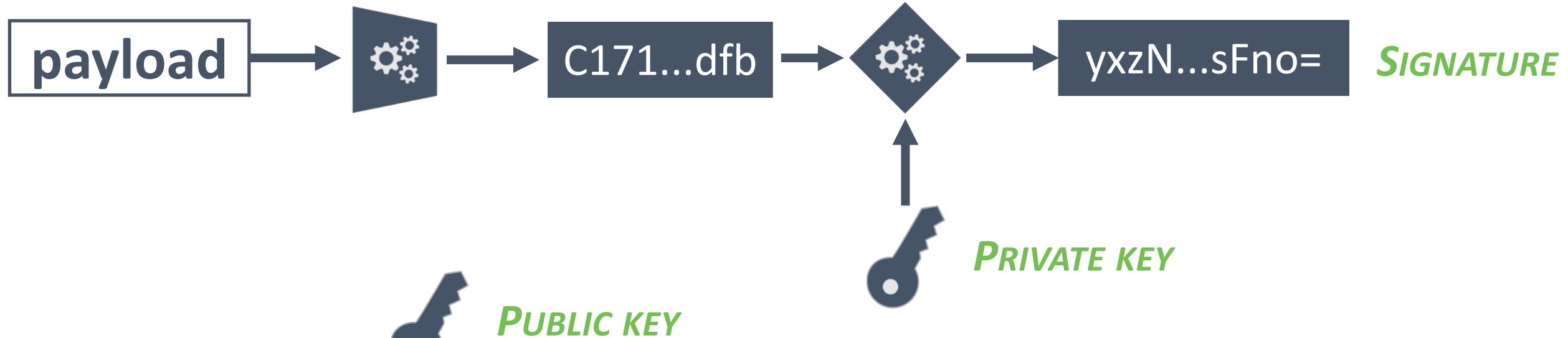


VERIFY HMAC

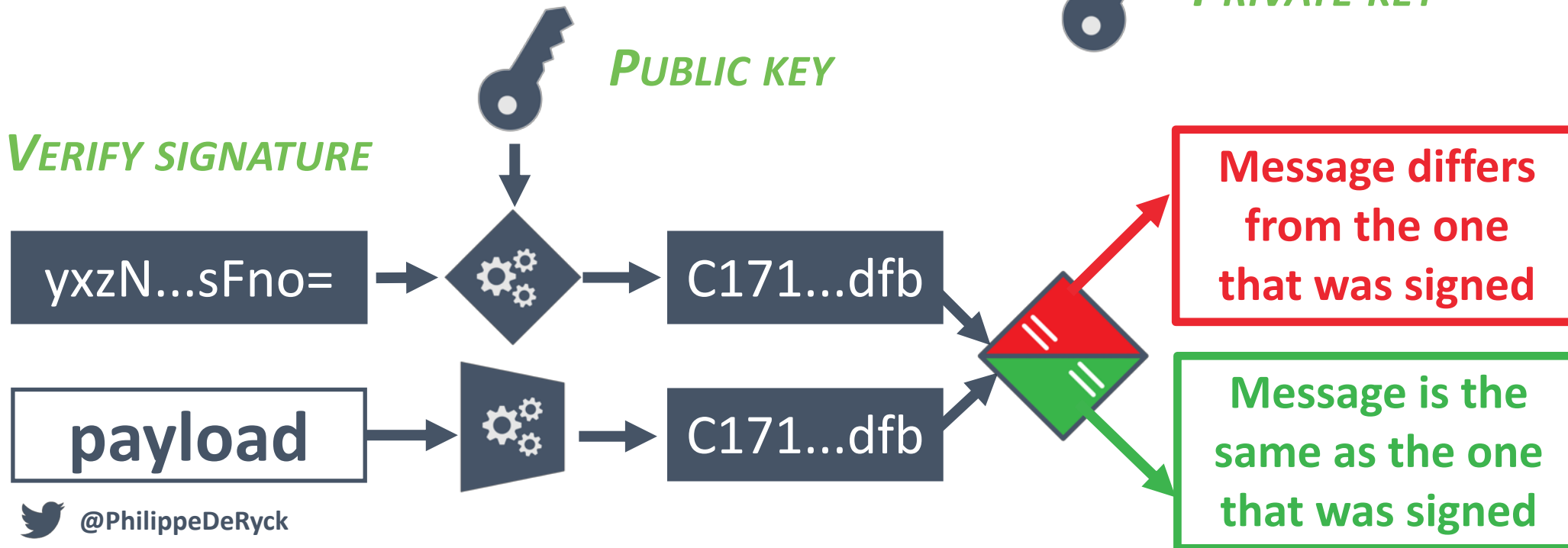


ASYMMETRIC JWT SIGNATURES

GENERATE SIGNATURE



VERIFY SIGNATURE



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.”

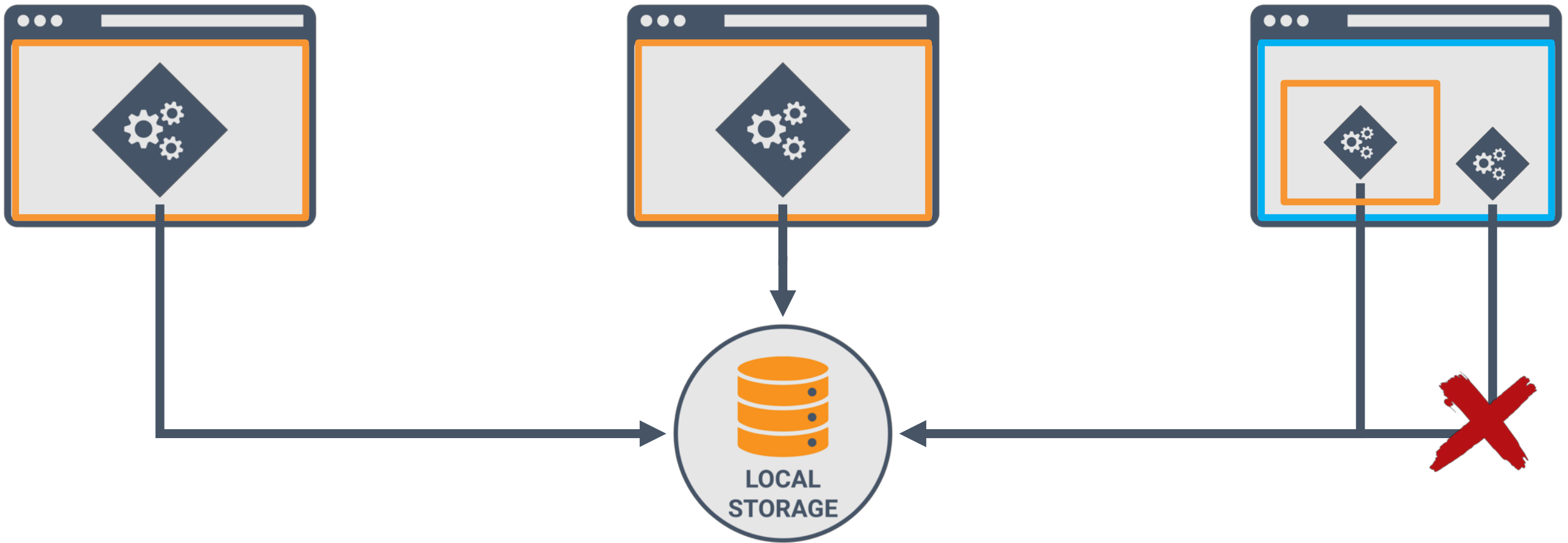


HttpOnly cookies

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





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**

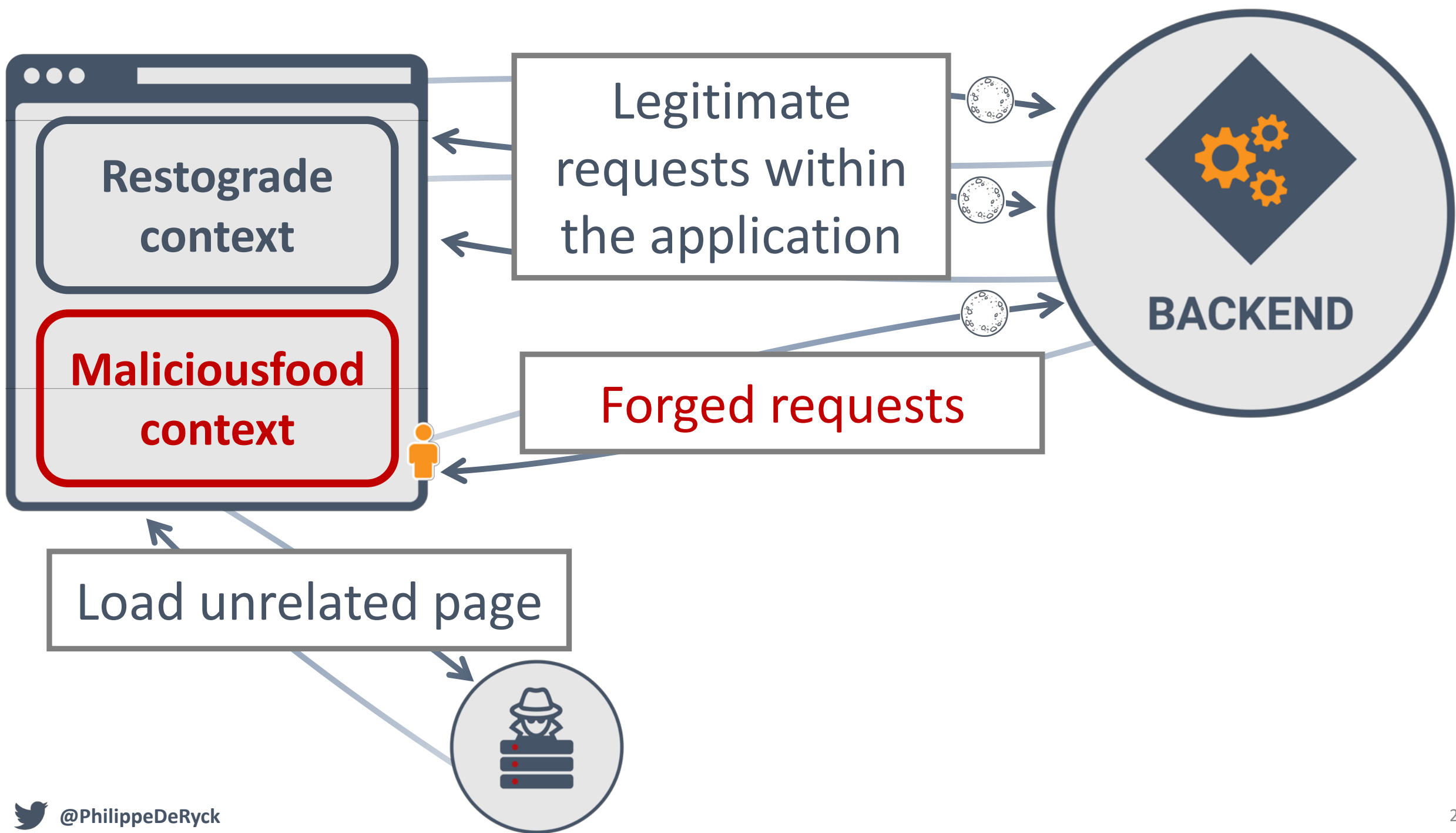
Authorization: **Bearer 42**

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`



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 {}
```



HOW TO AUTHORIZE THE
LOADING OF DOM RESOURCES
(IMG, SCRIPT, ...)?



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 ”



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](https://www.linkedin.com/in/PhilippeDeRyck)



[@PhilippeDeRyck](https://twitter.com/PhilippeDeRyck)

philippe@pragmaticwebsecurity.com