COOKIES VS TOKENS
A PARADOXICAL CHOICE
SHOULD YOU EVER USE COOKIES FOR YOUR API?
Authentication 

Authorization 

Session Management
• 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
Works fine with a stateful REST backend
Works fine with a stateful REST backend

Might benefit from a stateless REST backend
How will you represent session data?
Do you keep the data on the client or the server?
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IlBoaWxpcHBIERlIFJ5YW55Iiwib2x1cyI6ImVzZXIgcmVzcGFjZSIsImlhdCI6MTY1NzIyMjExMCwiZXhwIjoxNjMxMjIzNjI2fQ.KPjhyE9oi8uehw6Lm_0yAzRuJhcUqXETD2AIrF2A

```

```
CAN YOU SPOT A PROBLEM HERE?

```java
String token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9." + ...;
try {
    DecodedJWT jwt = JWT.decode(token);
} catch (JWTDecodeException exception) {
    //Invalid token
}
```
String token = "eyJhbGciOiJIUzI1NiIsInR5c...zWfOkEE";

try {
    DecodedJWT jwt = JWT.decode(token);
} catch (JWTDecodeException exception) {
    //Invalid token
}

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
}
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IiIlBoaWxpCHBlIERlIFJ5Y2siLCJyb2x1cyI6InVzZXIgcmVzdGFtcmFudG93bmVyIiwiaWF0IjoxNTExMDIyMDI2fQ.KPjhyE9oi8uehg6Lm_0yAZzRuJhcUqXETD2AIRF2A

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

```
{
  "sub": "1234567890",
  "name": "Philippe De Ryck",
  "roles": "user restaurantowner",
  "iat": 1518239822
}
```

```
HMACSHA256(
    base64UrlEncode(header) + "," + 
    base64UrlEncode(payload),
    SuperSecretHMACKey
) secret base64 encoded
```
HMAC-BASED JWT SIGNATURES

GENERATE HMAC

```
data
```

```
yxzN...sFno= data
```

```
SECRET KEY
```

VERIFY HMAC

```
data
```

```
yxzN...sFno=
```

```
Message differs from the one that was signed
```

```
Message is the same as the one that was signed
```

@PhilippeDeRyck
**ASymmetric JWT SIGNatures**

**Generate Signature**

- **payload** → C171...dfb → **signature**

**Verify Signature**

- **C171...dfb** → **payload**

- **C171...dfb** → **signature**

- **C171...dfb**, **payload**: Message differs from the one that was signed

- **C171...dfb**, **signature**: Message is the same as the one that was signed
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

@PhilippeDeRyck
## Comparing Client-Side Storage Mechanisms

<table>
<thead>
<tr>
<th>Local Storage</th>
<th>Session Storage</th>
<th>In-Memory</th>
<th>Cookies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available to the entire origin</td>
<td>Available to the window and children</td>
<td>Available to running code only</td>
<td>Can be fully hidden from JavaScript</td>
</tr>
<tr>
<td>Survives a page reload</td>
<td>Survives a page reload</td>
<td>Does not survive a page reload</td>
<td>Survives a page reload</td>
</tr>
<tr>
<td>Cannot be shielded from malicious code</td>
<td>Can be a bit shielded from malicious code</td>
<td>Can be shielded from malicious code</td>
<td>Can be shielded from malicious code</td>
</tr>
<tr>
<td>Application code required for handling</td>
<td>Application code required for handling</td>
<td>Application code required for handling</td>
<td>Application code not required for handling</td>
</tr>
</tbody>
</table>
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`
Legitimate requests within the application

Forced requests

Load unrelated page

Restograde context

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

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.
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?
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, ...)?
<table>
<thead>
<tr>
<th><strong>Cookies</strong></th>
<th><strong>Authorization Header</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Can contain identifiers &amp; session objects</td>
<td>Can contain identifiers &amp; session objects</td>
</tr>
<tr>
<td>Only works well with a single domain</td>
<td>Freedom to include headers to any domain</td>
</tr>
<tr>
<td>Automatically handled by the browser</td>
<td>Requires custom code to get, store and send session data</td>
</tr>
<tr>
<td>Always present, including on DOM resources</td>
<td>Only present on XHR calls, unless you add it through a ServiceWorker</td>
</tr>
</tbody>
</table>
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.”
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, ...