Common API security pitfalls

Dr. Philippe De Ryck

https://PragmaticWebSecurity.com

Dr. Philippe De Ryck

- Deep understanding of the web security landscape
- Google Developer Expert (not employed by Google)
- Course curator of the SecAppDev course

(https://secappdev.org)

Pragmatic Web Security

High-quality security training for developers and managers

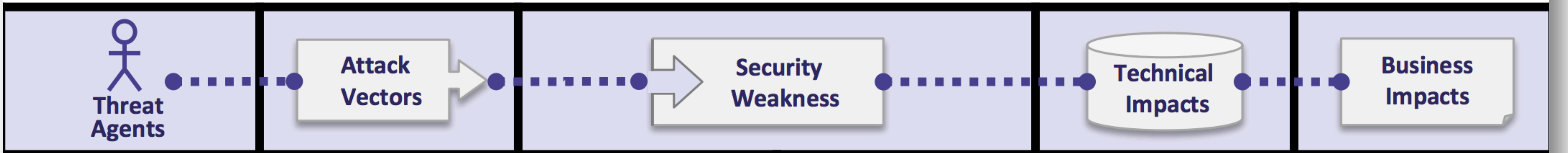
Custom courses covering web security, API security, Angular security, …

Consulting services on security, Oauth 2.0, OpenID Connect, …

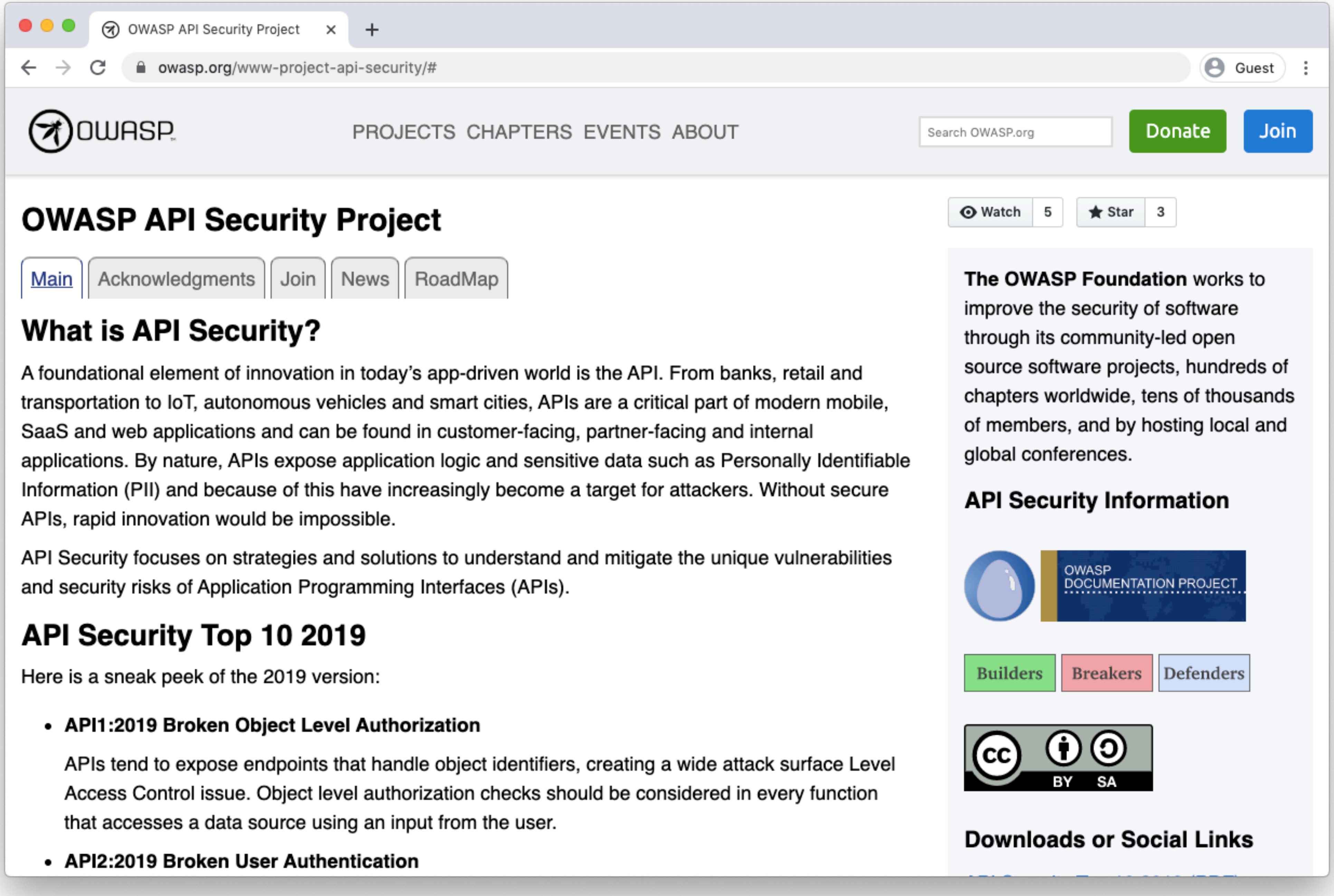
@PhilippeDeRyck

https://PragmaticWebSecurity.com

A10 Underprotected APIs



Application Specific	Exploitability AVERAGE	Prevalence COMMON	Detectability DIFFICULT	Impact MODERATE	Application / Business Specific
Consider anyone with the ability to send requests to your APIs. Client software is easily reversed and communications are easily intercepted, so obscurity is no defense for APIs.	Attackers can reverse engineer APIs by examining client code, or simply monitoring communications. Some API vulnerabilities can be automatically discovered, others only by experts.	Modern web applications and APIs are increasingly composed of rich clients (browser, mobile, desktop) that connect to backend APIs (XML, JSON, RPC, GWT, custom). APIs (microservices, services, endpoints) can be vulnerable to the full range of attacks. Unfortunately, dynamic and sometimes even static tools don't work well on APIs, and they can be difficult to analyze manually, so these vulnerabilities are often undiscovered.		The full range of negative outcomes is possible, including data theft, corruption, and destruction; unauthorized access to the entire application; and complete host takeover.	Consider the impact of an API attack on the business. Does the API access critical data or functions? Many APIs are mission critical, so also consider the impact of denial of service attacks.



OWASP API Security Project
owasp.org/www-project-api-security/#
Guest
OWASP
PROJECTS CHAPTERS EVENTS ABOUT
Search OWASP.org
Donate
Join
Watch 5
Star 3

OWASP API Security Project

Main
Acknowledgments
Join
News
RoadMap

What is API Security?

A foundational element of innovation in today's app-driven world is the API. From banks, retail and transportation to IoT, autonomous vehicles and smart cities, APIs are a critical part of modern mobile, SaaS and web applications and can be found in customer-facing, partner-facing and internal applications. By nature, APIs expose application logic and sensitive data such as Personally Identifiable Information (PII) and because of this have increasingly become a target for attackers. Without secure APIs, rapid innovation would be impossible.

API Security focuses on strategies and solutions to understand and mitigate the unique vulnerabilities and security risks of Application Programming Interfaces (APIs).

API Security Top 10 2019

Here is a sneak peek of the 2019 version:

- **API1:2019 Broken Object Level Authorization**

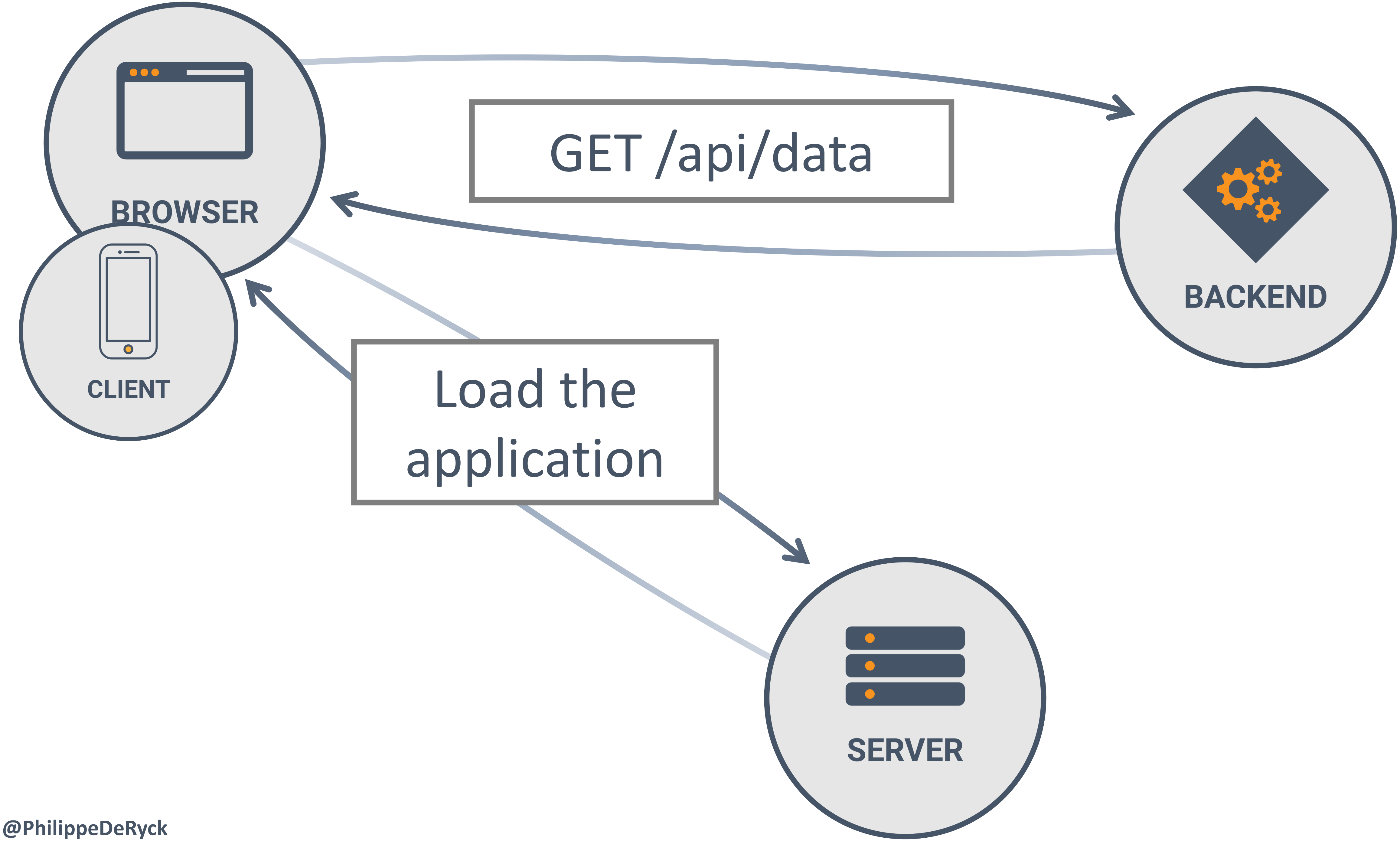
  APIs tend to expose endpoints that handle object identifiers, creating a wide attack surface Level Access Control issue. Object level authorization checks should be considered in every function that accesses a data source using an input from the user.

- **API2:2019 Broken User Authentication**

The OWASP Foundation works to improve the security of software through its community-led open source software projects, hundreds of chapters worldwide, tens of thousands of members, and by hosting local and global conferences.

API Security Information

Downloads or Social Links



BROWSER

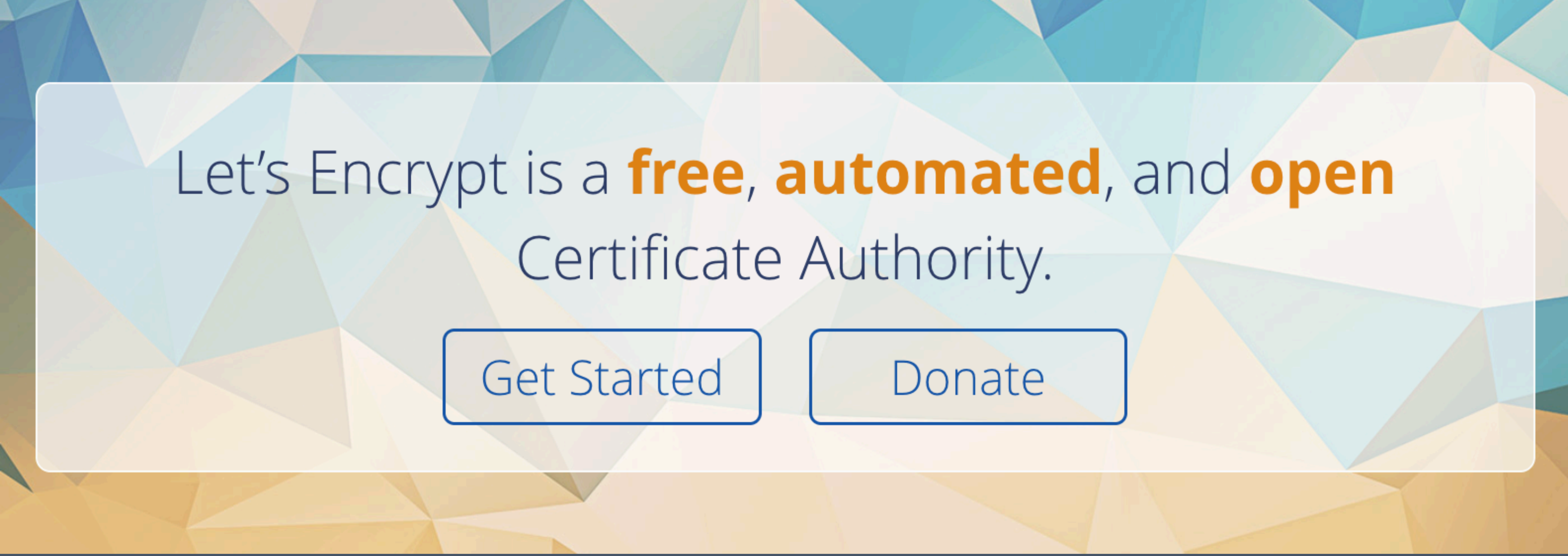
CLIENT

GET /api/data

BACKEND

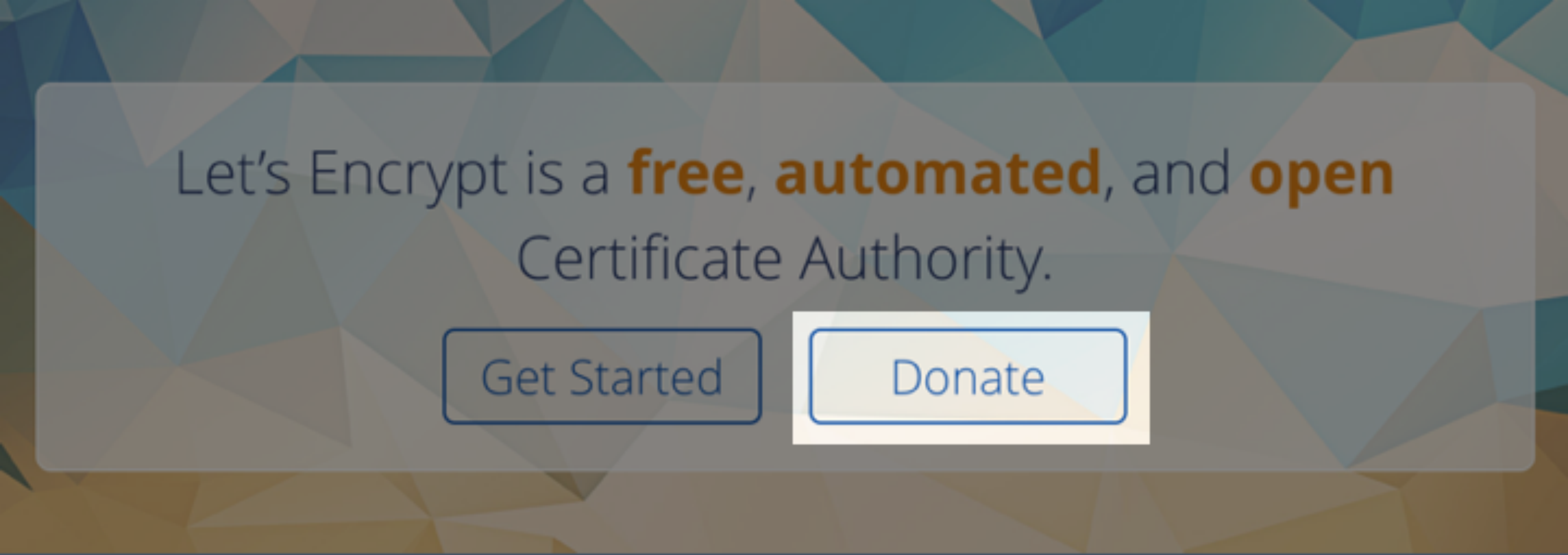
Load the application

SERVER



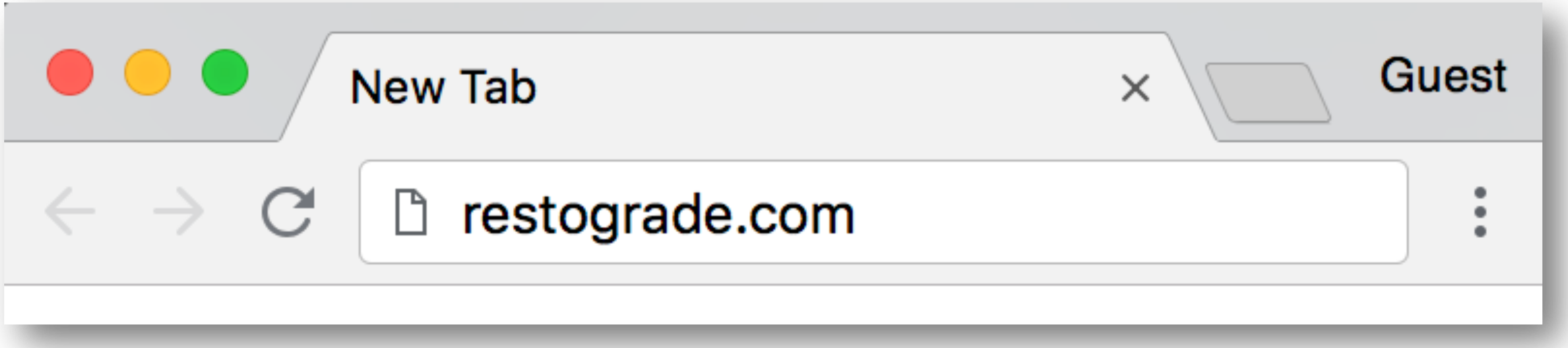
Let's Encrypt is a **free**, **automated**, and **open** Certificate Authority.

> “We do this because we want to create a more secure and privacy-respecting Web.”



Let’s Encrypt is a **free**, **automated**, and **open** Certificate Authority.

> **“We do this because we want to create a more secure and privacy-respecting Web.”**



restograde.com

```
# NginX config
location / {
  return 301 https://$host$request_uri;
}
```

HTTPS AS A BASELINE REQUIREMENT

- Moving your sites to 100% HTTPS should be a priority
  - HTTPS has become too important to ignore, even for public content
  - A single HTTP step in the chain is already a vulnerability, so 100% HTTPS is a must
  - HTTPS is often depended upon as the baseline for security

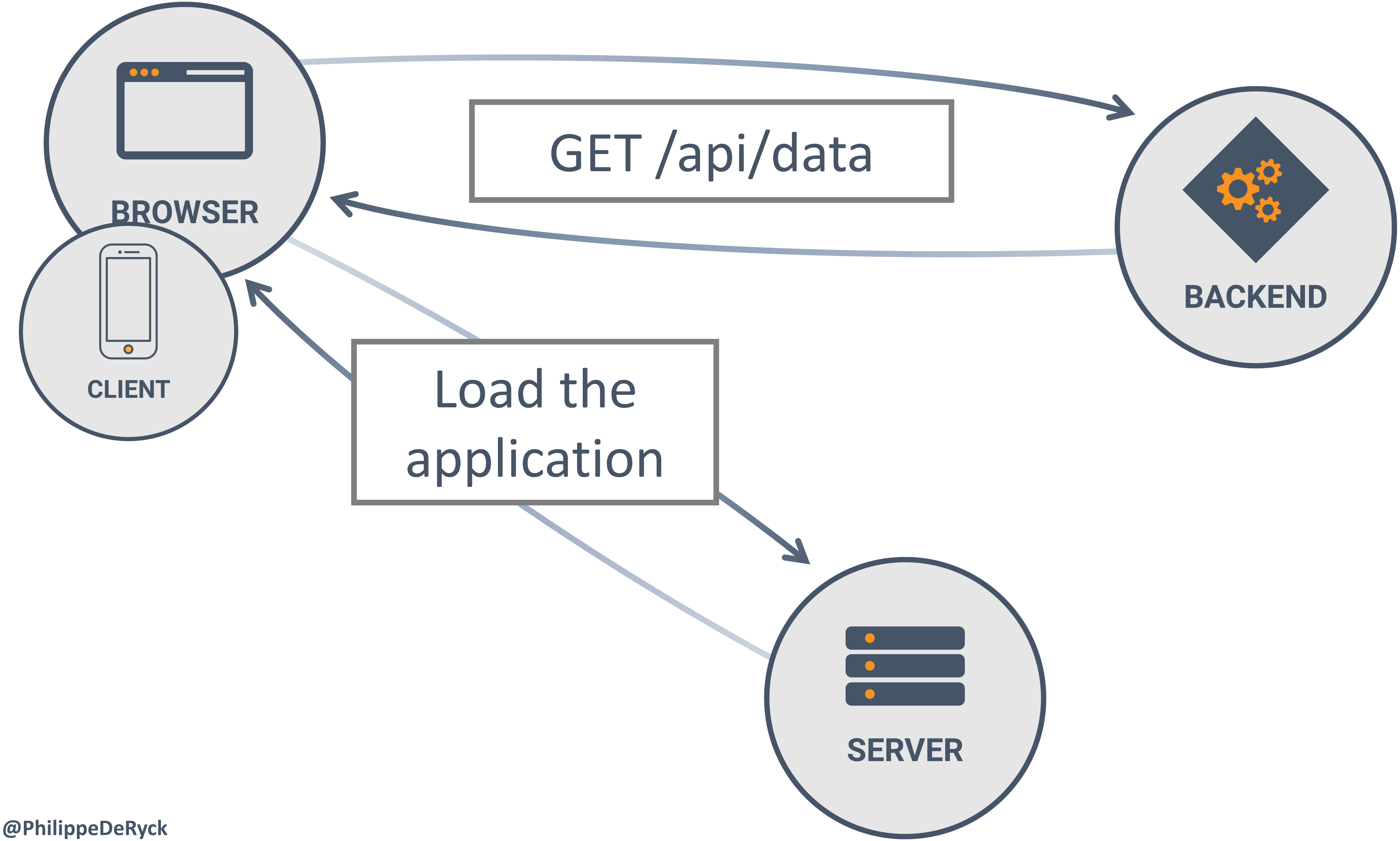
- After the move to HTTPS, redirect HTTP traffic to the HTTPS endpoint
  - Only relevant for endpoints dealing with ***navigational requests from a browser***
  - API-only endpoints should disable HTTP and only need to support HTTPS

- Enable HTTP Strict Transport Security for all HTTPS domains
  - Install a long-lived HSTS policy on as many domains as possible
  - Carefully move to a global HSTS policy with ***includeSubDomains***

Supporting HTTP

APIs are accessed from code, so there is no need to support a redirect from HTTP to HTTPS.

Lock your API further down by enabling HSTS.



BROWSER

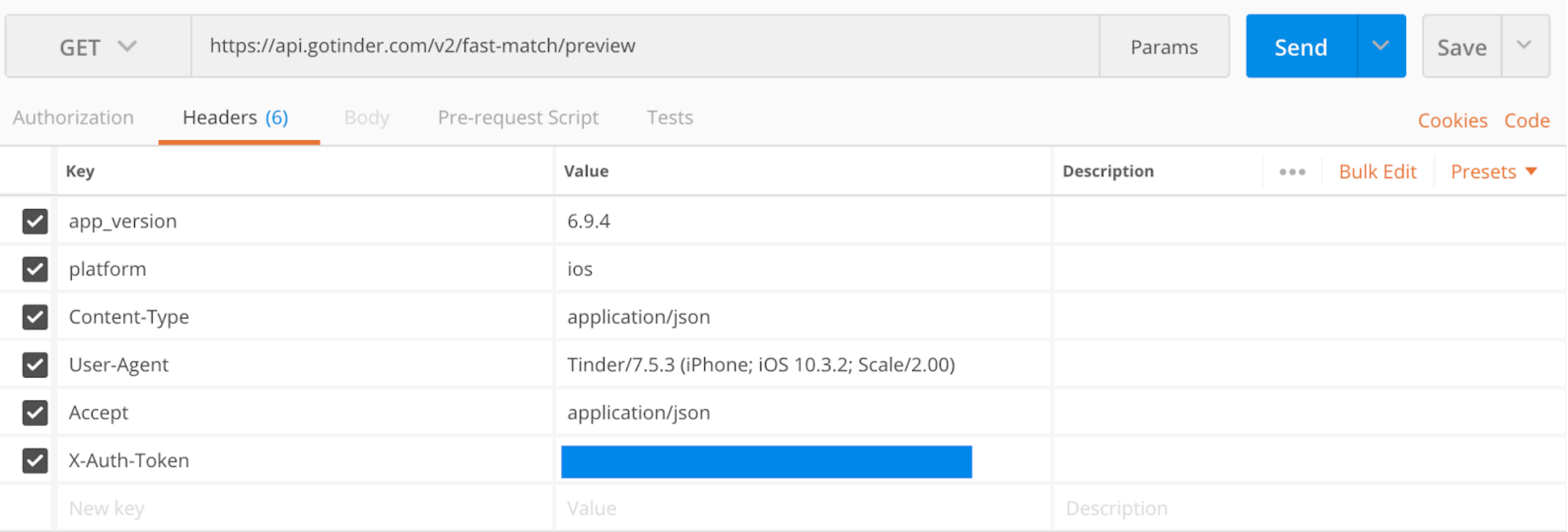
CLIENT

GET /api/data

BACKEND

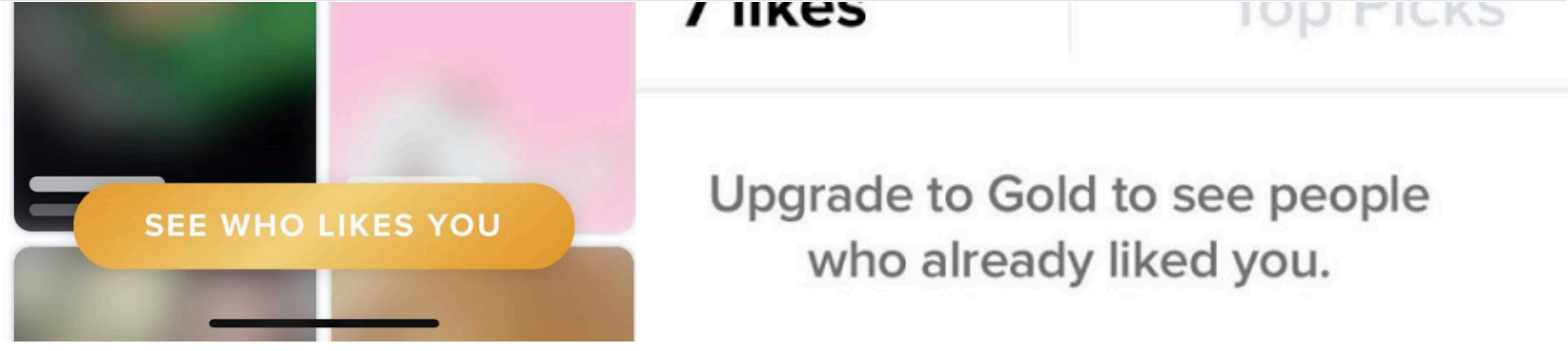
Load the application

SERVER



GET https://api.gotinder.com/v2/fast-match/preview

	Key	Value	Description
☑	app_version	6.9.4	
☑	platform	ios	
☑	Content-Type	application/json	
☑	User-Agent	Tinder/7.5.3 (iPhone; iOS 10.3.2; Scale/2.00)	
☑	Accept	application/json	
☑	X-Auth-Token	[redacted]	
	New key	Value	Description



Upgrade to Gold to see people who already liked you.

Relying on client-side security measures

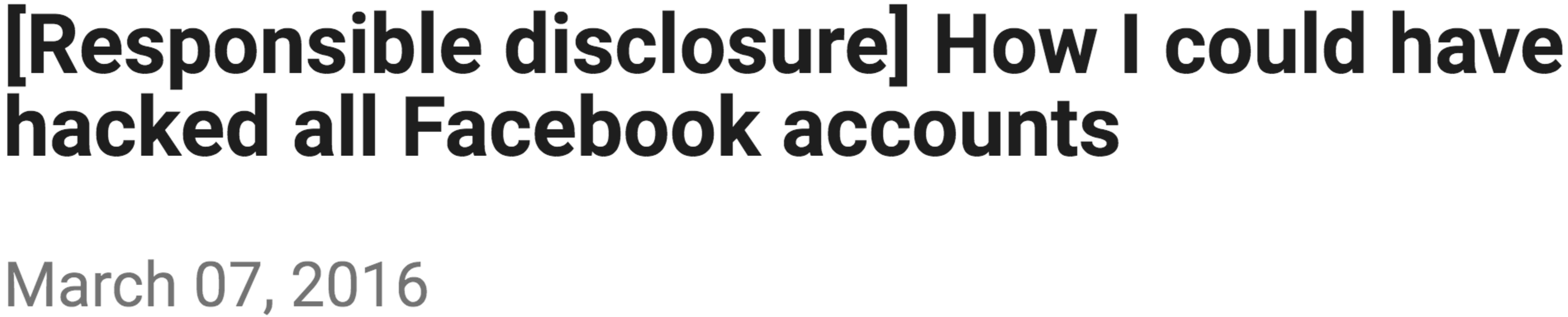
- Client applications run independent of an API
  - Every call to the API is easy to analyze and intercept
  - Attackers can make direct calls to APIs by impersonating the client application

- Common security issues are hiding data or features in the client
  - E.g., blurring images or not showing certain data fields
  - E.g., relying on client-side authorization to shield admin access routes

- Always perform security-relevant filtering and processing on the server-side
  - Ensure that all data leaving the API is properly secured or processed

Over-exposing API data

Never rely on client-side data processing or filtering to hide information. Always assume an attacker has full access to all API endpoints.



[Responsible disclosure] How I could have hacked all Facebook accounts

March 07, 2016

> “on *beta.facebook.com* and *mbasic.beta.facebook.com* rate limiting was missing on forgot password endpoints”

Unlimited access to an API

- Unlimited access to an API can have severe consequences
  - Denial of service is probably the best case scenario
  - Extracting information or brute forcing access codes are a lot worse

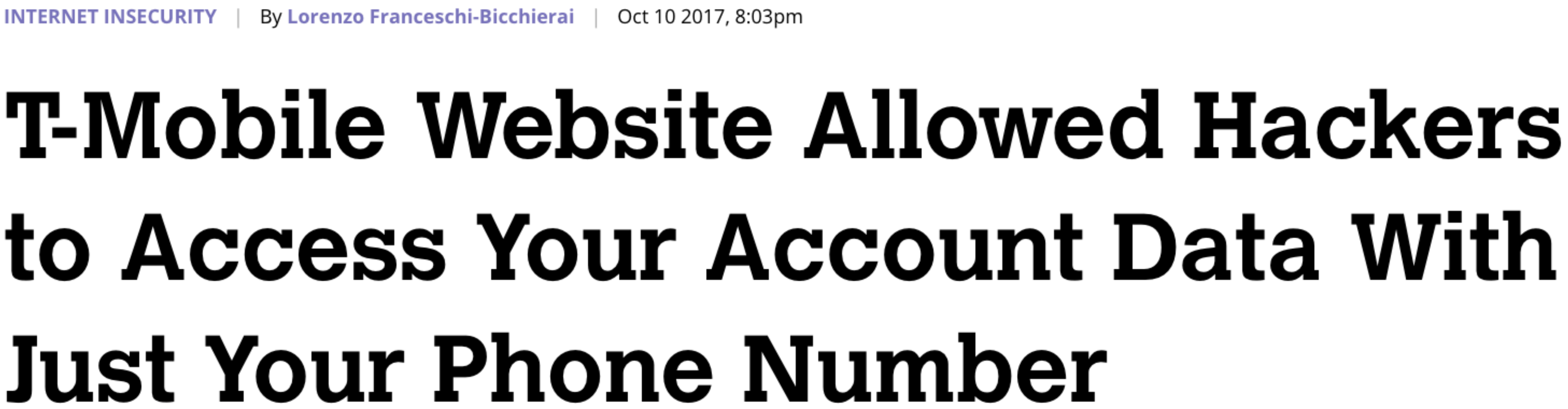
- Various rate-limiting strategies can be used
  - Limiting per connection property (IP address)
  - Limiting per user (account / access token / API key)
  - Limiting per application property (user account / resource type)
  - Limiting based on context (region / type of app)

- Often implemented as a business driver instead of a security feature
  - These limits are quite liberal, so complement with stricter limits in shorter windows

```
HTTP/1.1 429 Too Many Requests
Retry-After: 3600
```

No rate limiting

Rate limiting prevents malicious code from abusing legitimate / illegitimate access to your API



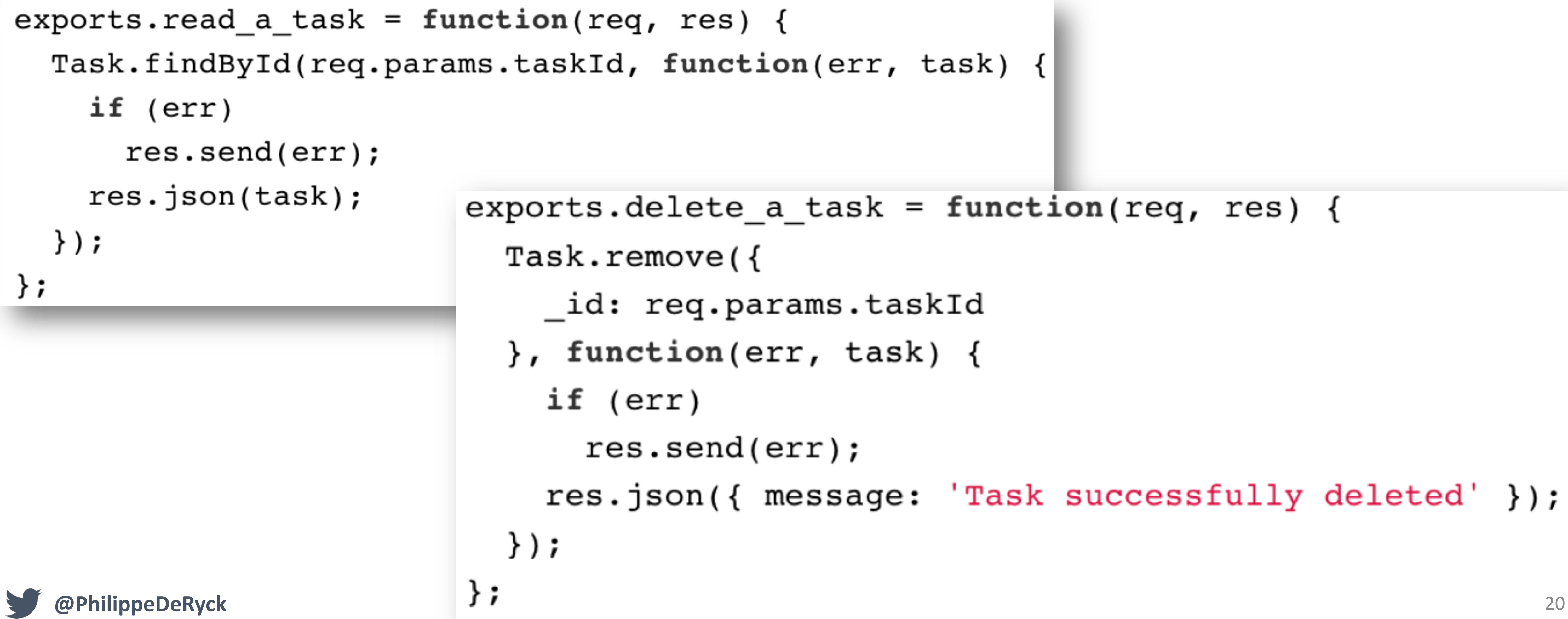
INTERNET INSECURITY | By Lorenzo Franceschi-Bicchierai | Oct 10 2017, 8:03pm

T-Mobile Website Allowed Hackers to Access Your Account Data With Just Your Phone Number

> “he could query for someone else's phone number and the API would simply send back a response containing the other person's data.”

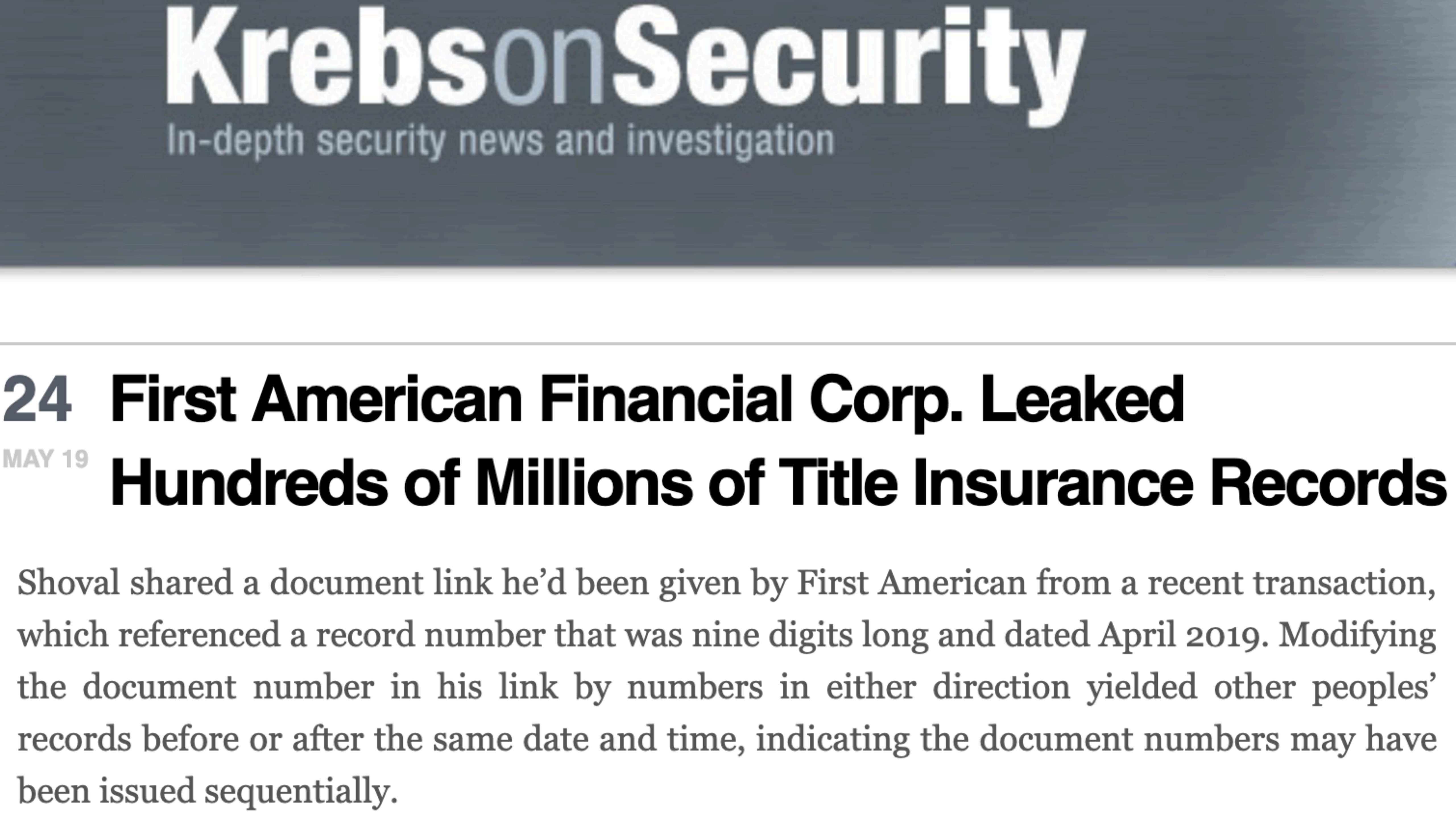
Build Node.js RESTful APIs in 10 Minutes

Published Jan 12, 2017 Last updated Aug 18, 2017



```
exports.read_a_task = function(req, res) {
  Task.findById(req.params.taskId, function(err, task) {
    if (err)
      res.send(err);
    res.json(task);
  });
};
```

```
exports.delete_a_task = function(req, res) {
  Task.remove({
    _id: req.params.taskId
  }, function(err, task) {
    if (err)
      res.send(err);
    res.json({ message: 'Task successfully deleted' });
  });
};
```



KrebsonSecurity

In-depth security news and investigation

24 MAY 19

First American Financial Corp. Leaked Hundreds of Millions of Title Insurance Records

Shoval shared a document link he'd been given by First American from a recent transaction, which referenced a record number that was nine digits long and dated April 2019. Modifying the document number in his link by numbers in either direction yielded other peoples' records before or after the same date and time, indicating the document numbers may have been issued sequentially.

INSECURE DIRECT OBJECT REFERENCES

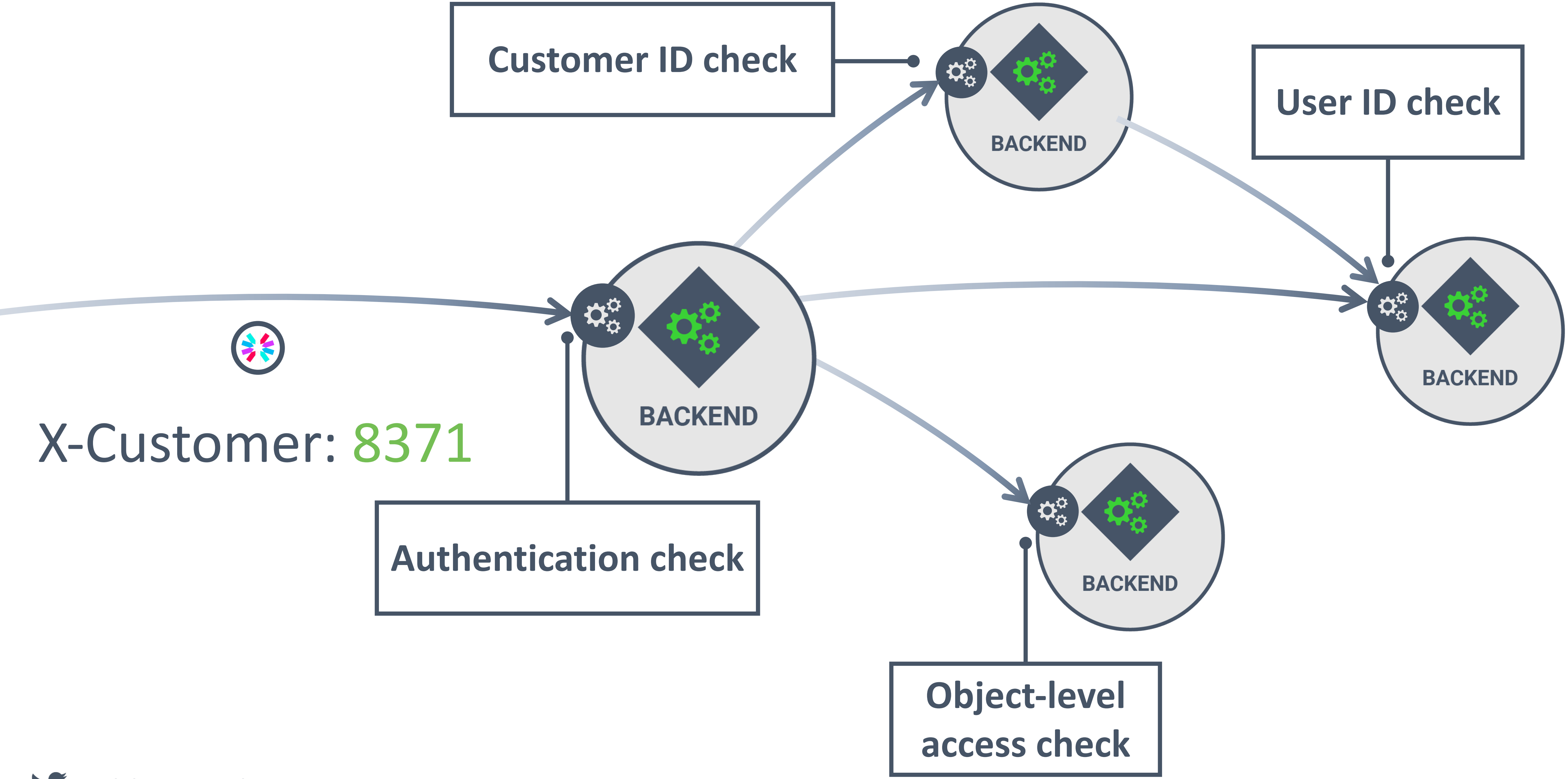
- Predictable identifiers enable the enumeration of resources
  - Dangerous if resources are not shielded by strict authorization checks
  - Many APIs only check authentication status, but not ***which*** user is authenticated

- The only proper mitigation is implementing proper authorization checks
  - E.g. checking if the current user is the owner of the resource

- The use of non-predictable identifiers is a complementary strategy
  - UUIDs are a good example of such an identifier
  - Just be careful about using them as primary keys in the database

Lack of proper authorization

Always complement an initial authentication check with appropriate authorization checks (e.g. ownership of a resource)


Customer ID check

User ID check

X-Customer: 8371

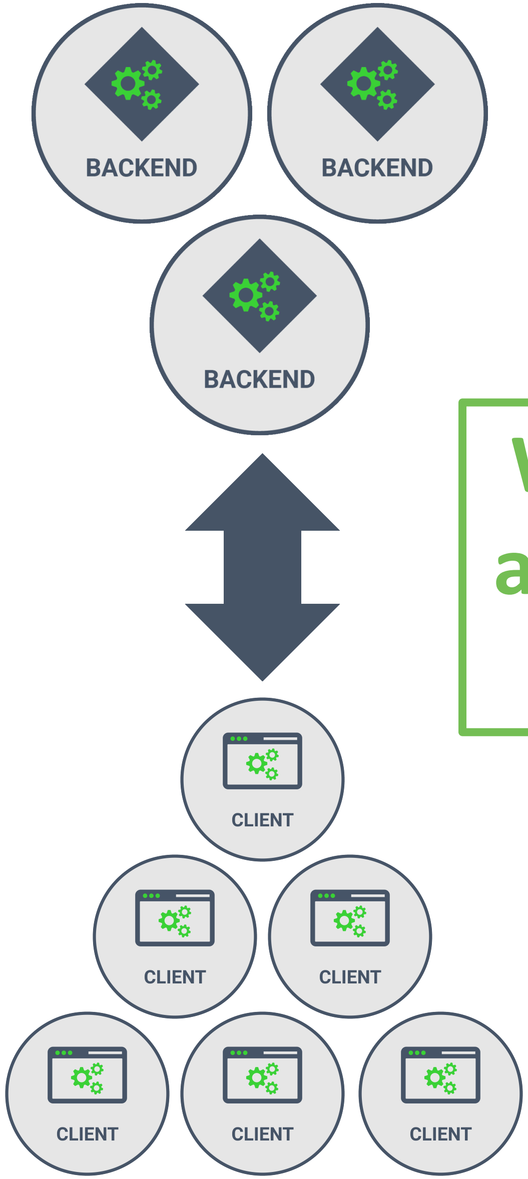
Authentication check

Object-level access check

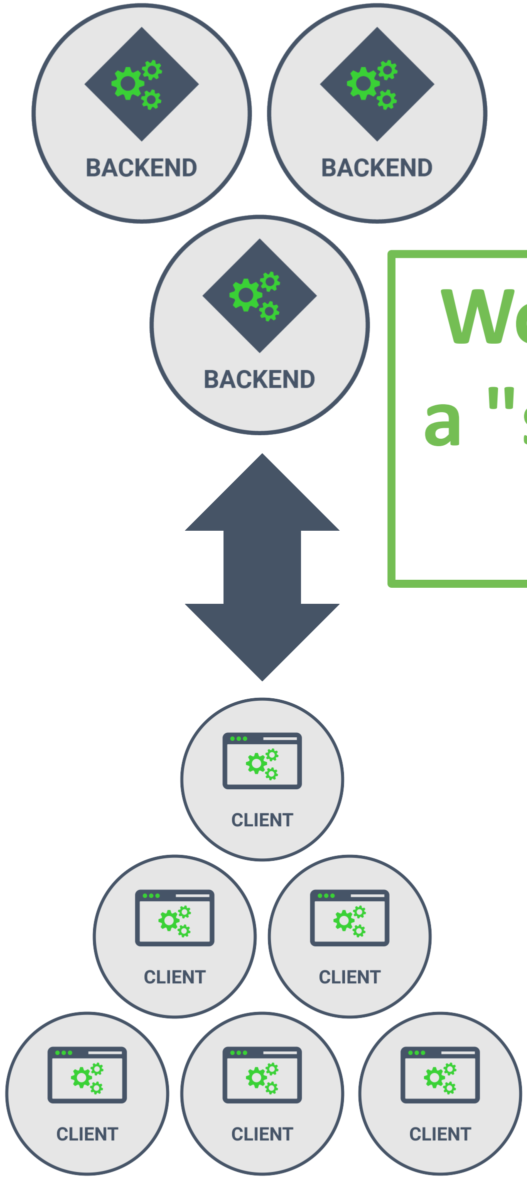
Failure to audit the authorization policy

Use a centralized authorization policy that can be audited in isolation of the application code.

Use code-level authorization checks as a second line of defense

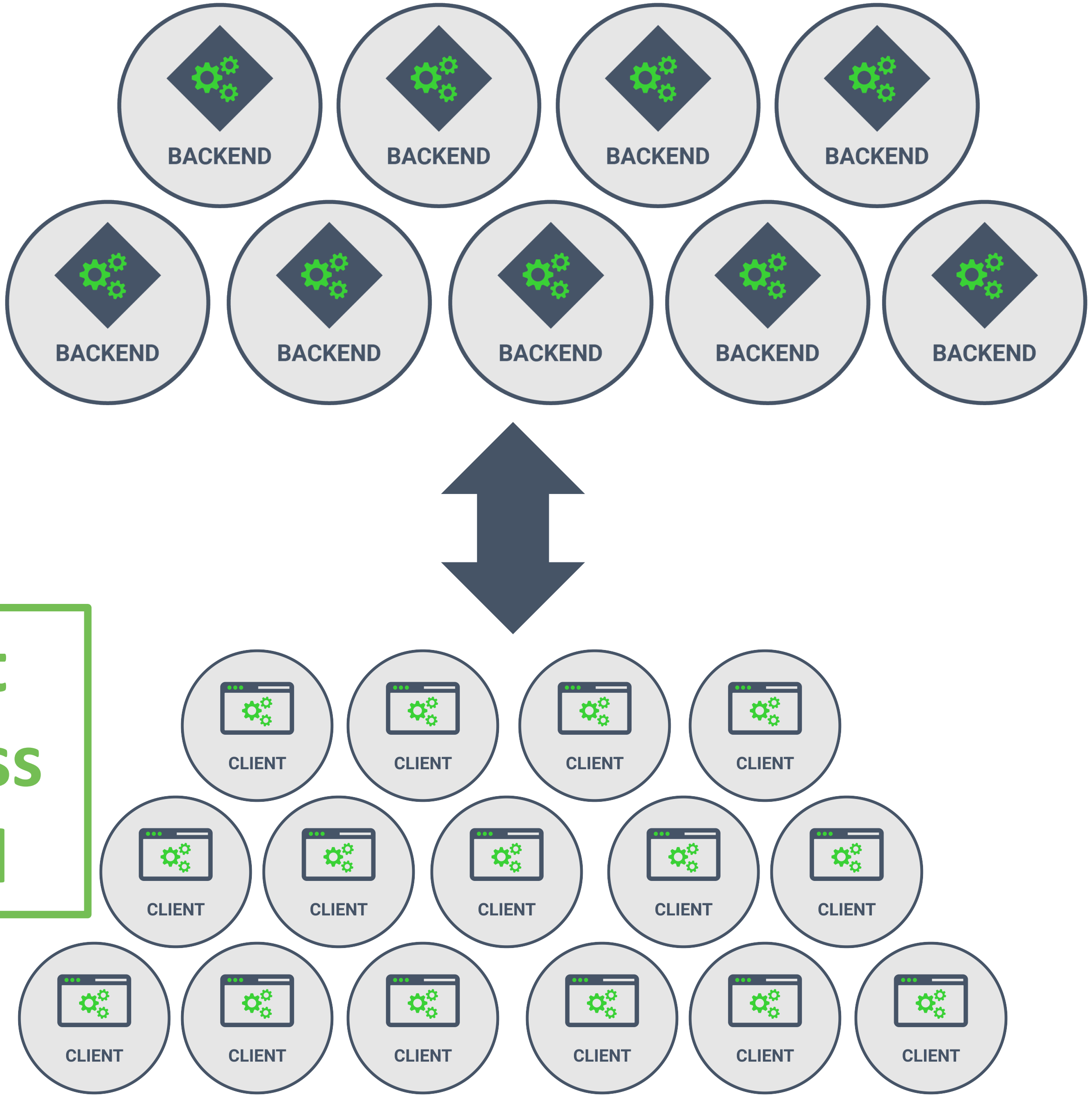


Works fine with a "stateful" ~~REST~~ backend



Works fine with a "stateful" ~~REST~~ backend

Might benefit from a stateless REST backend



The truth is a lot more complicated

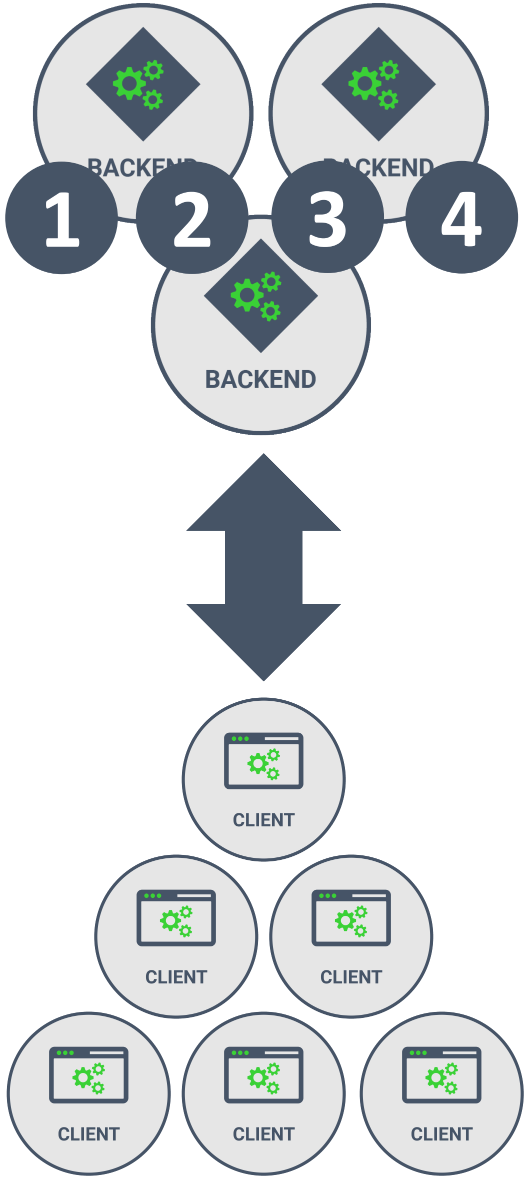
- Pure REST APIs should be stateless
  - The server is stateless, and the client provides all the required information
  - A valid argument for stateless backends is flexible scalability

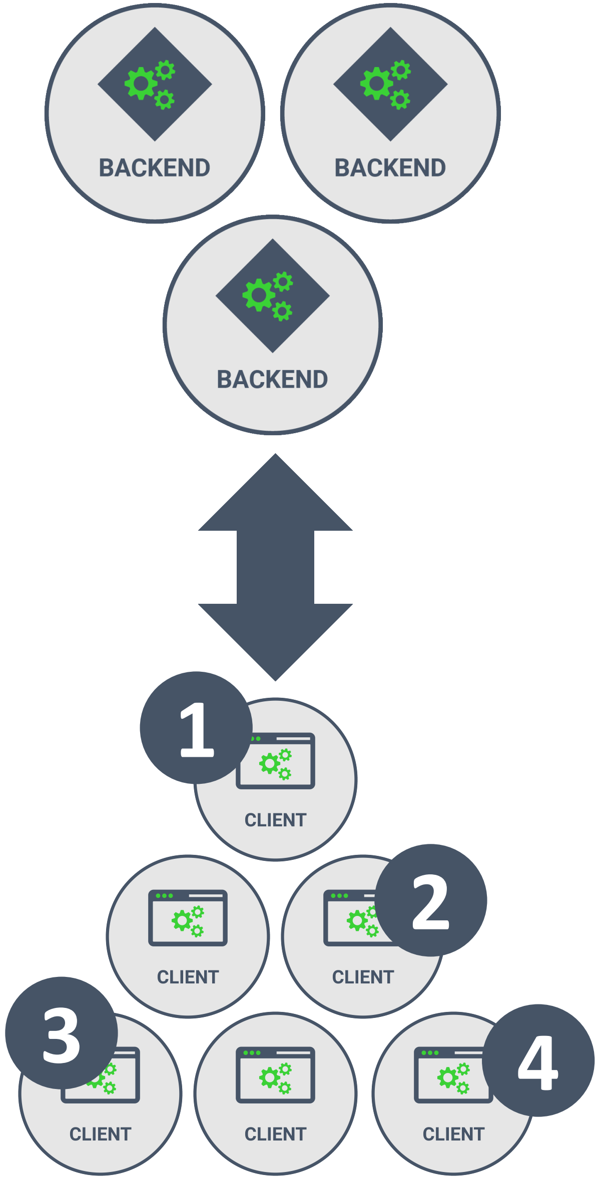
- Purity is rarely a good argument to throw working solutions overboard
  - An API can just as well keep session state on the server
  - Works perfectly well with small to medium-scale applications
  - Makes scalability harder, but not impossible
    - We have been doing this for 20 years with sticky 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

CHANGING SESSIONS FOR NO GOOD REASON

Server-side session data is not compatible with the REST paradigm, but still works well with small to medium-scale applications.


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

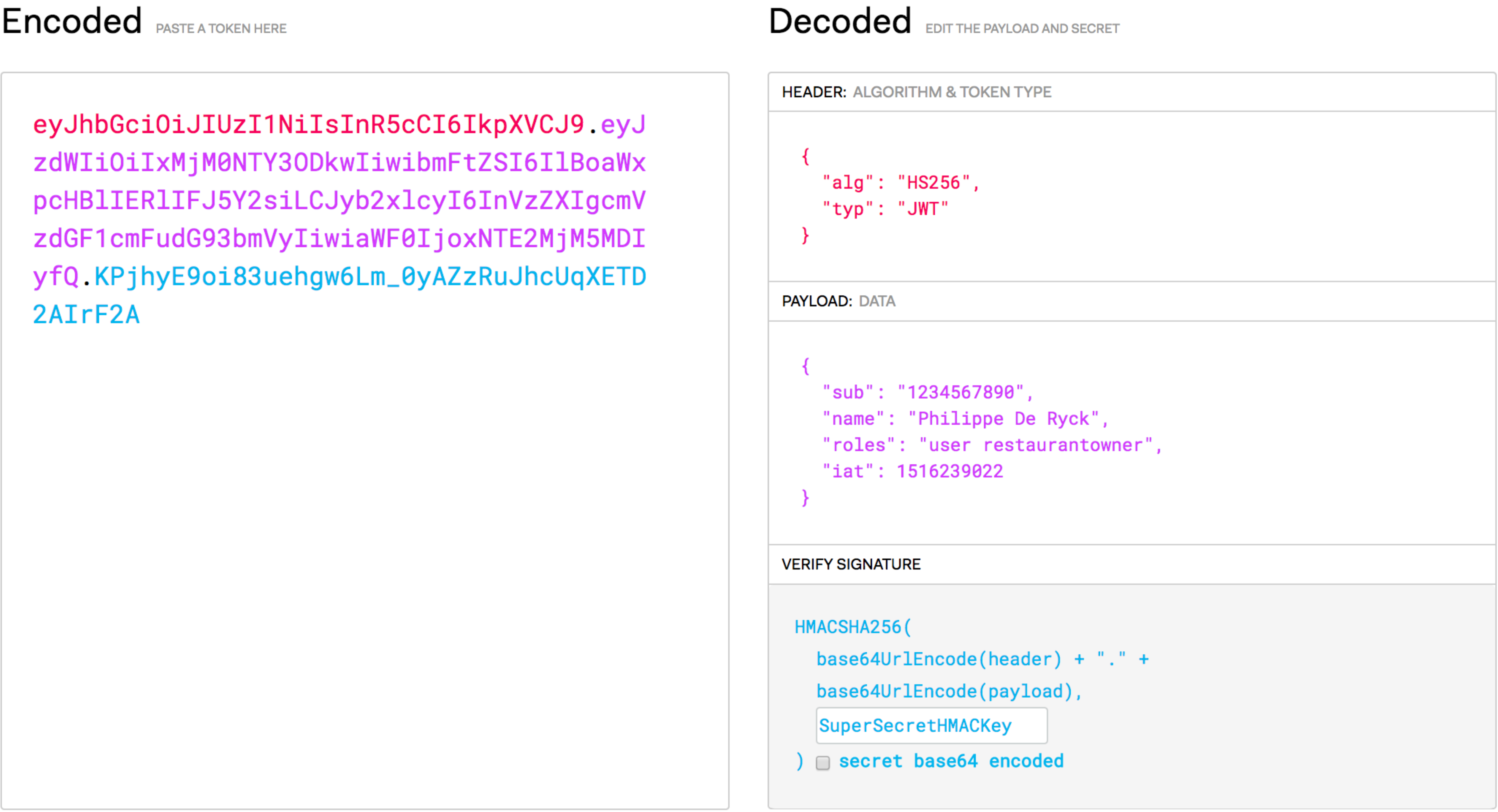

BACKEND
BACKEND
BACKEND
1
CLIENT
CLIENT
2
CLIENT
3
CLIENT
CLIENT
4
CLIENT

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.eyJ
zdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IlBoaWx
pcHBlIERlIFJ5Y2siLCJyb2xlcyI6InVzZXIgcmV
zdGF1cmFudG93bmVyIiwiaWF0IjoxNTE2MjM5MDI
yfQ.KPjhyE9oi83uehgw6Lm_0yAZzRuJhcUqXETD
2AIrF2A

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
```

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

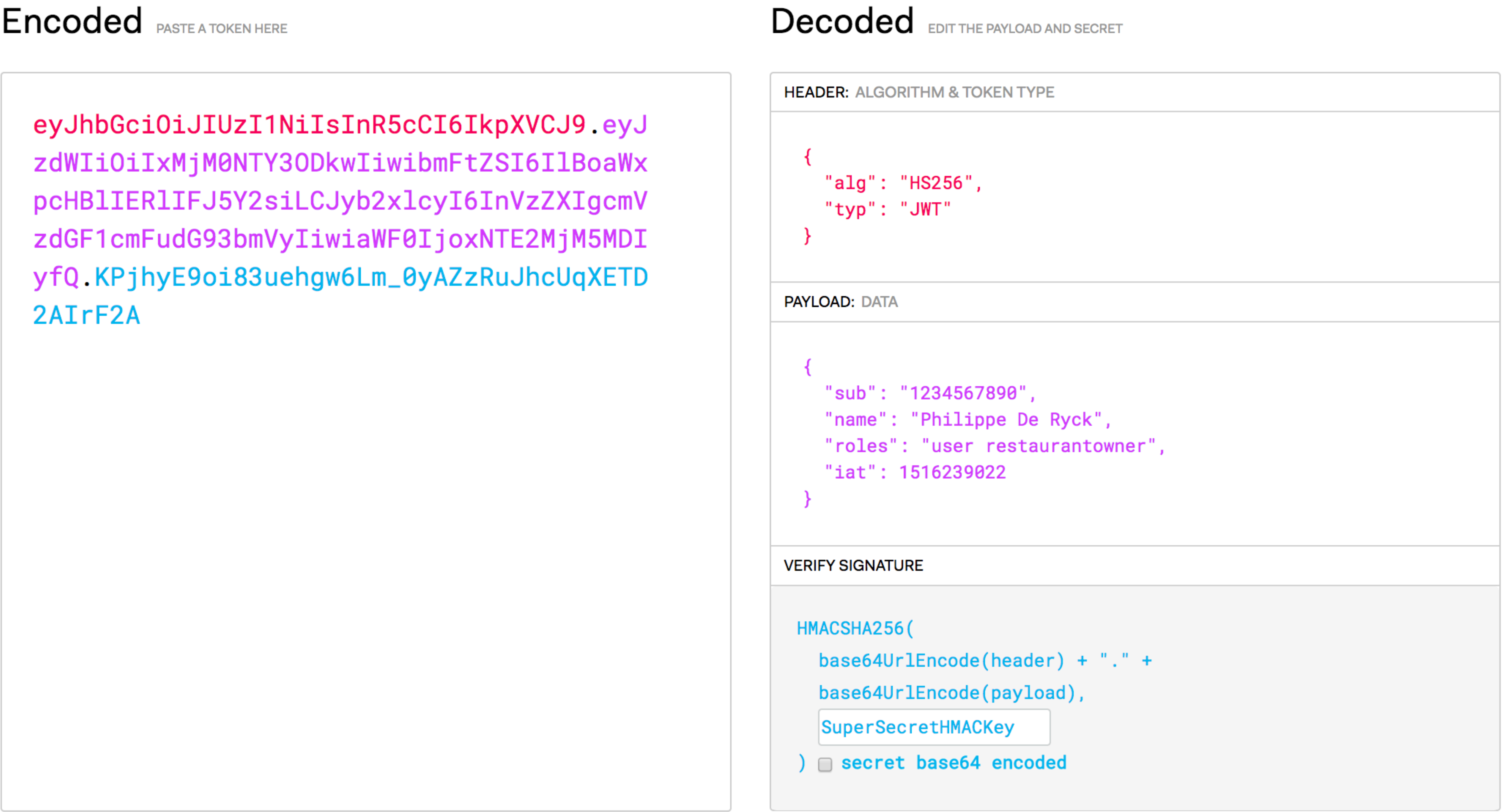
Decoding only

```java
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

Mishandling client-side session data

Client-side session data is easy to read and manipulate. You need to ensure confidentiality and integrity before using any of the session data.



Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
zdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IlBoaWx
pcHBlIERlIFJ5Y2siLCJyb2xlcyI6InVzZXIgcmV
zdGF1cmFudG93bmVyIiwiaWF0IjoxNTE2MjM5MDI
yfQ.KPjhyE9oi83uehgw6Lm_0yAZzRuJhcUqXETD
2AIrF2A
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

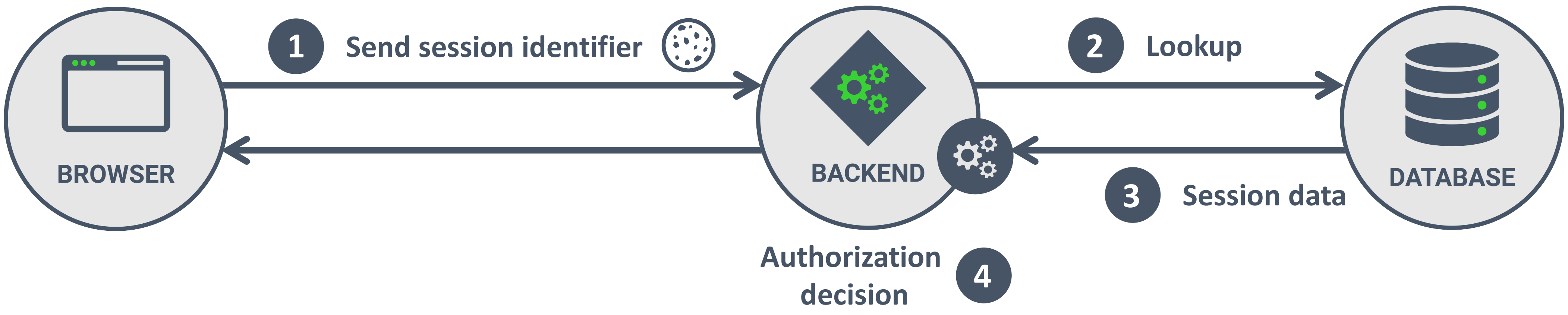
```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

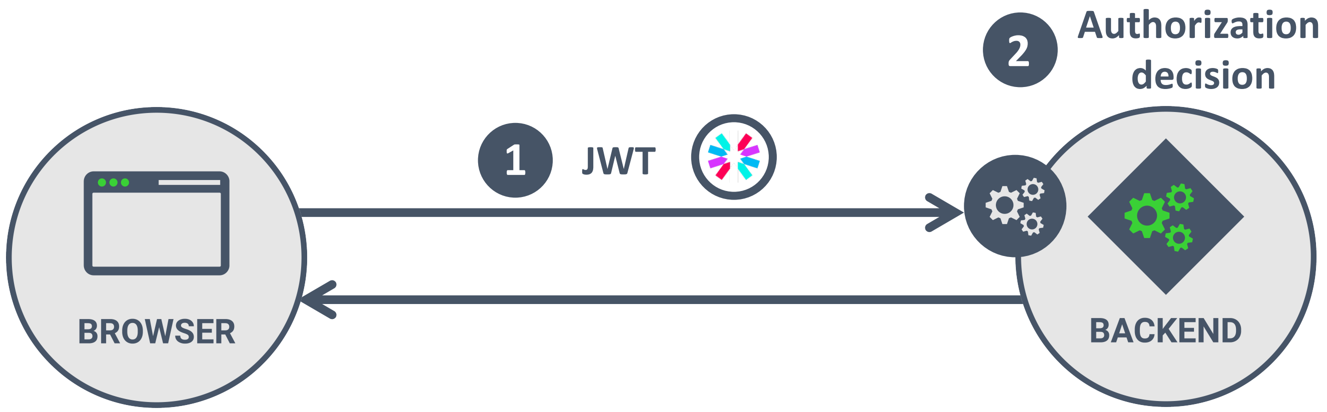
PAYLOAD: DATA

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

VERIFY SIGNATURE

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


BROWSER
1 Send session identifier
BACKEND
2 Lookup
DATABASE
3 Session data
Authorization decision 4

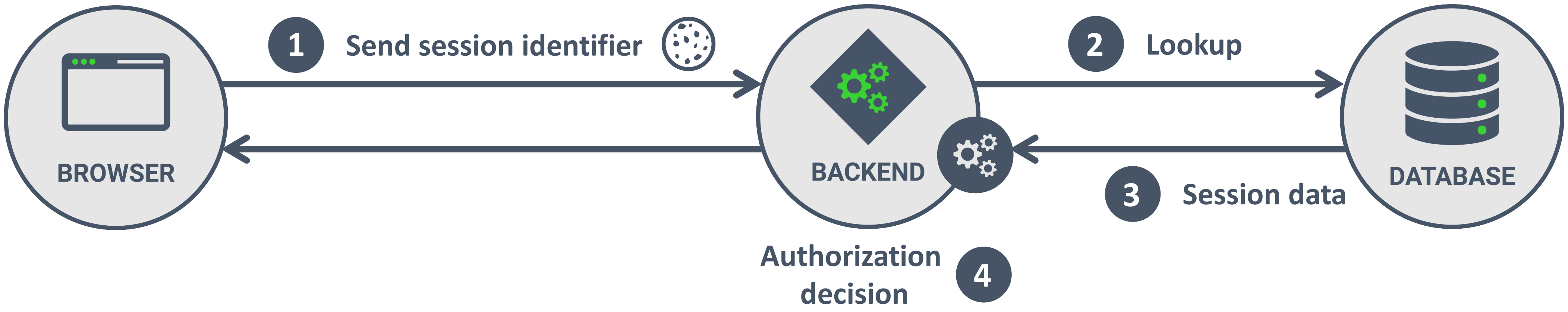

BROWSER
1 JWT
2 Authorization decision
BACKEND

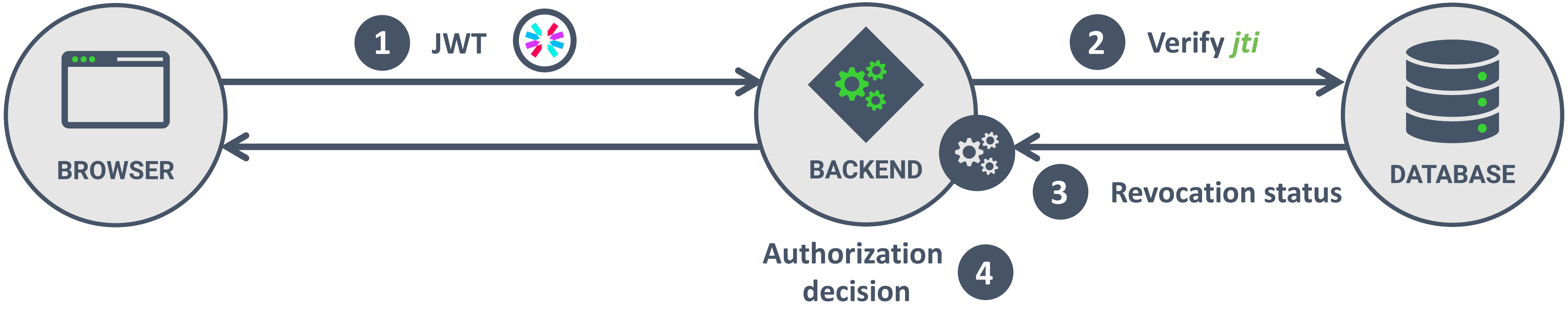
JWT REVOCATION

- A common revocation pattern uses the JWTs unique identifier
  - Keeping a list of invalid identifiers enables the backend to reject revoked JWTs

- Revoking a specific token for a specific device is challenging
  - The backend needs to keep a list of all issued ***jti*** claims
  - These identifiers need to be correlated to users and devices

- Verifying incoming JWTs against a revocation list requires explicit action
  - Depends on a centralized list of invalid identifiers
  - Check needs to happen on each incoming request
  - Adds a form of state to an otherwise stateless backend


BROWSER
1 Send session identifier
BACKEND
2 Lookup
DATABASE
3 Session data
Authorization decision 4

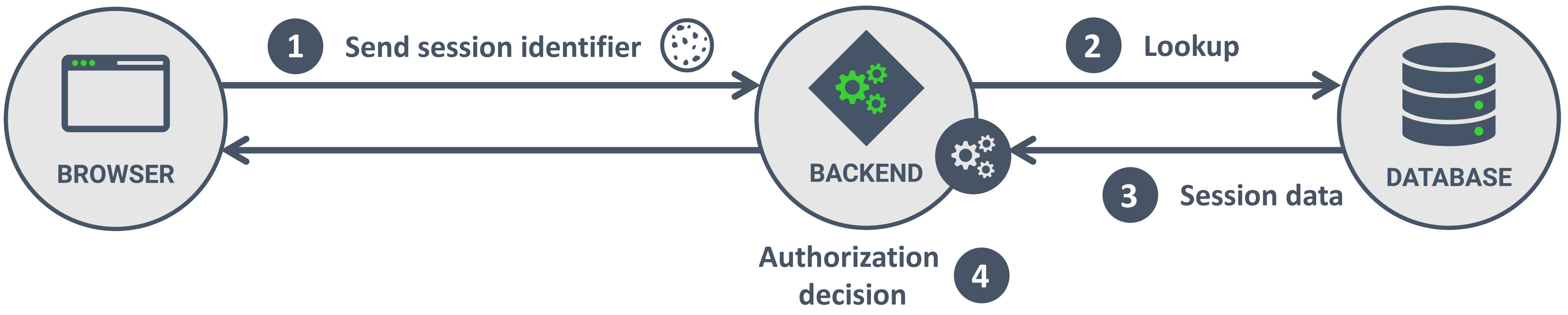

BROWSER
1 JWT
BACKEND
2 Verify *jti*
DATABASE
3 Revocation status
Authorization decision 4

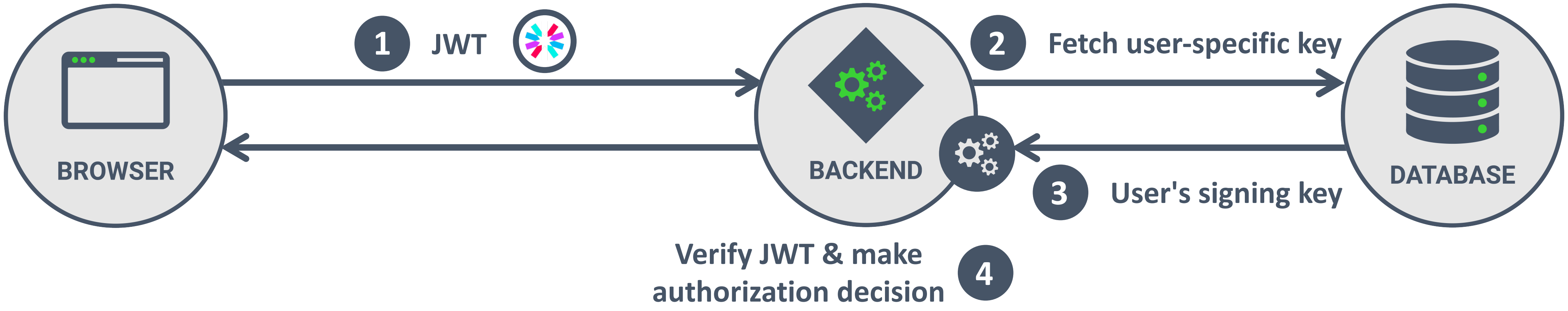
JWT REVOCATION USING KEY ROTATION

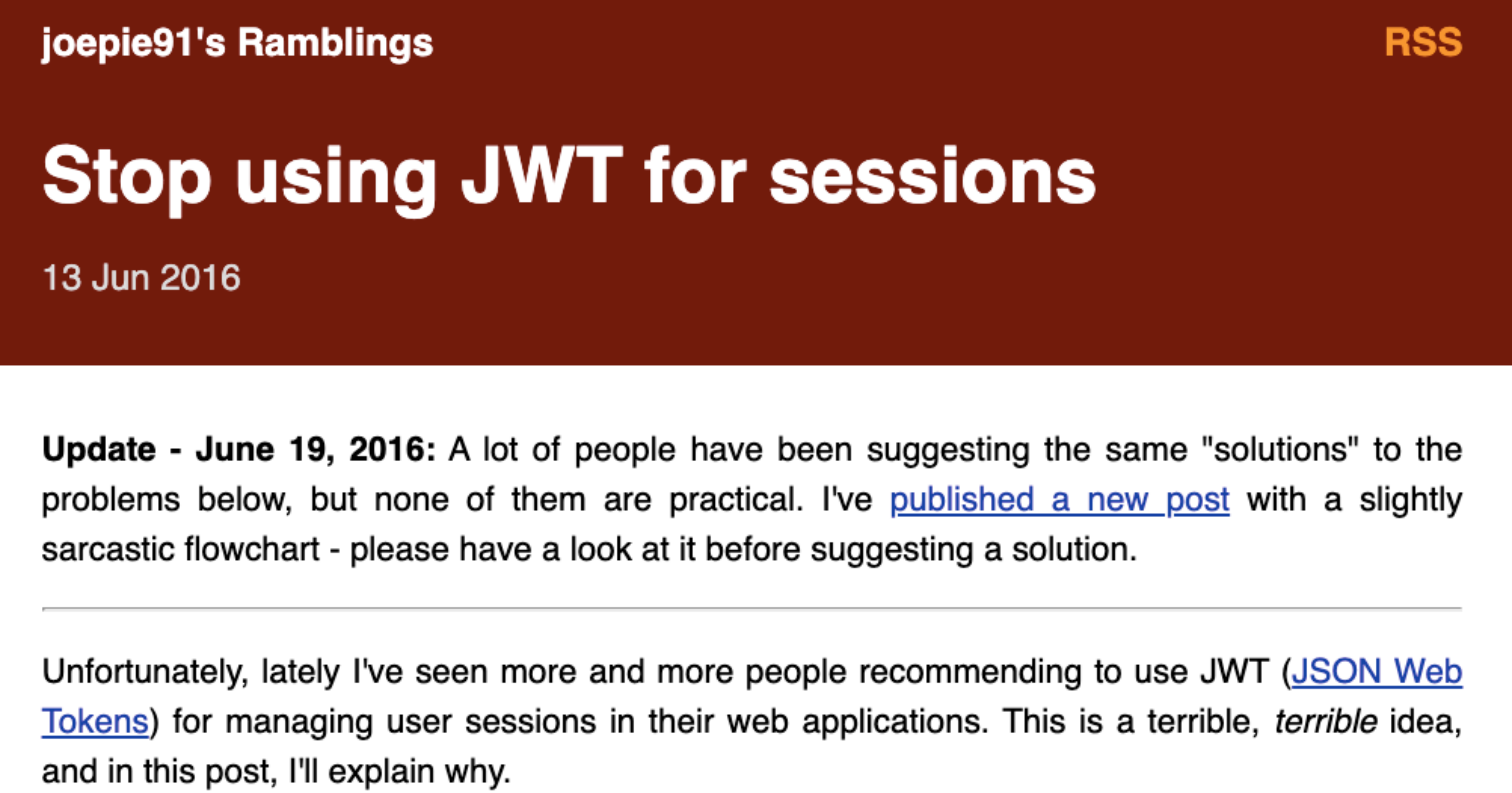
- Forcing a change in signing key turns every existing JWT signature invalid
  - Previously issued tokens will no longer be accepted, resembling revocation
  - Keys can be rotated globally, or on a per-user basis

- Global key rotation is only useful for emergency incident response
  - Rotating an application-wide signing key causes ***all*** JWTs to become invalid
  - Doing this impacts every device of every user of the application

- Using per-user keys enables more granular rotation of keys
  - By changing a single user's signing key, all tokens of that user can be revoked
  - Impact remains limited to that single user, making this option seem viable


1 Send session identifier
2 Lookup
3 Session data
4 Authorization decision
BROWSER
BACKEND
DATABASE


1 JWT
2 Fetch user-specific key
3 User's signing key
4 Verify JWT & make authorization decision
BROWSER
BACKEND
DATABASE



joepie91's Ramblings

RSS

Stop using JWT for sessions

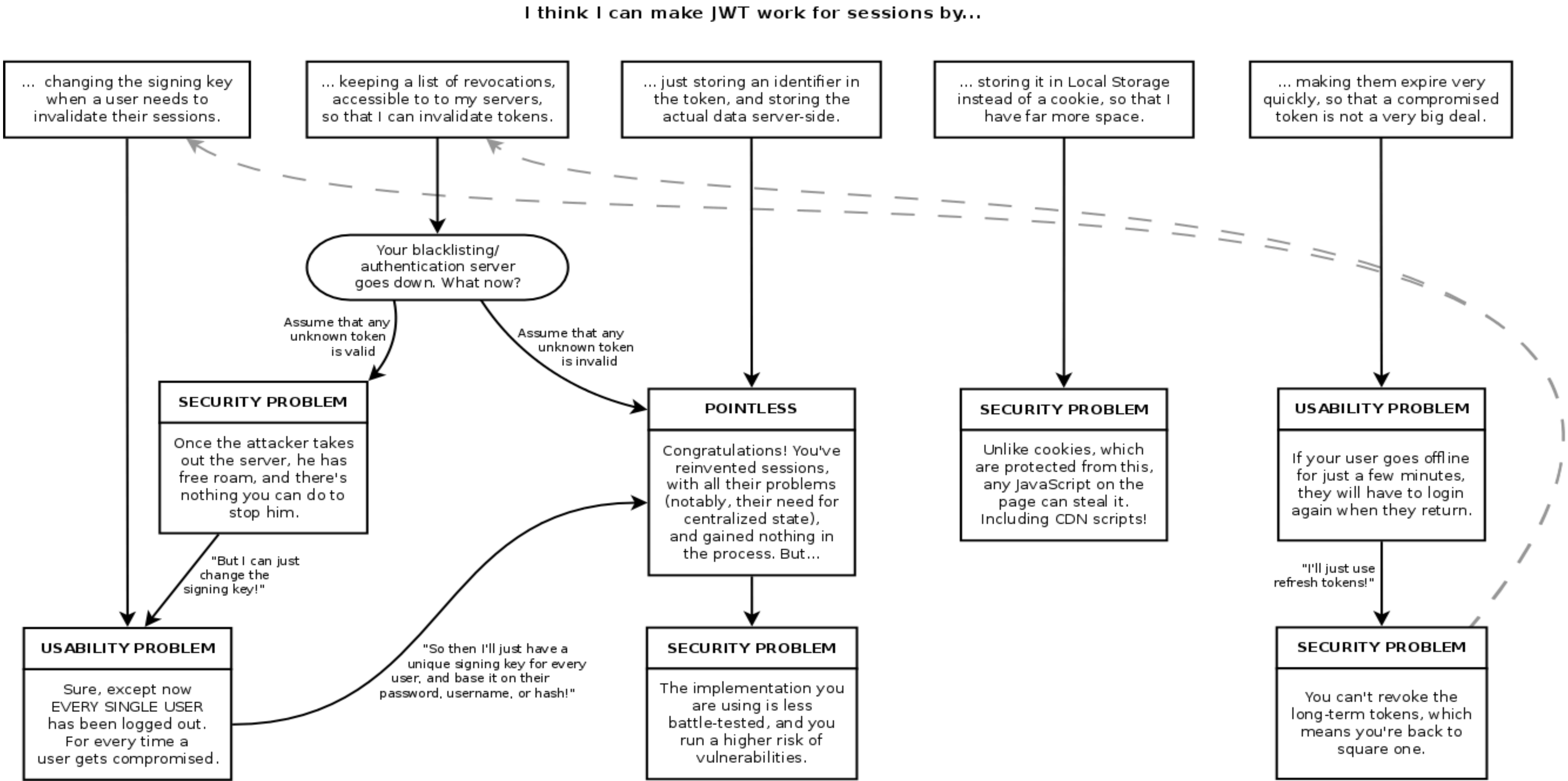
13 Jun 2016

Update - June 19, 2016: A lot of people have been suggesting the same "solutions" to the problems below, but none of them are practical. I've published a new post with a slightly sarcastic flowchart - please have a look at it before suggesting a solution.

Unfortunately, lately I've seen more and more people recommending to use JWT (JSON Web Tokens) for managing user sessions in their web applications. This is a terrible, *terrible* idea, and in this post, I'll explain why.

Stop using JWT for sessions, part 2

A handy dandy (and slightly sarcastic) flowchart about why your "solution" doesn't work



I think I can make JWT work for sessions by...

... changing the signing key when a user needs to invalidate their sessions.

... keeping a list of revocations, accessible to to my servers, so that I can invalidate tokens.

... just storing an identifier in the token, and storing the actual data server-side.

... storing it in Local Storage instead of a cookie, so that I have far more space.

... making them expire very quickly, so that a compromised token is not a very big deal.

Your blacklisting/ authentication server goes down. What now?

Assume that any unknown token is valid

Assume that any unknown token is invalid

SECURITY PROBLEM

Once the attacker takes out the server, he has free roam, and there's nothing you can do to stop him.

"But I can just change the signing key!"

USABILITY PROBLEM

Sure, except now EVERY SINGLE USER has been logged out. For every time a user gets compromised.

"So then I'll just have a unique signing key for every user, and base it on their password, username, or hash!"

POINTLESS

Congratulations! You've reinvented sessions, with all their problems (notably, their need for centralized state), and gained nothing in the process. But...

SECURITY PROBLEM

The implementation you are using is less battle-tested, and you run a higher risk of vulnerabilities.

SECURITY PROBLEM

Unlike cookies, which are protected from this, any JavaScript on the page can steal it. Including CDN scripts!

USABILITY PROBLEM

If your user goes offline for just a few minutes, they will have to login again when they return.

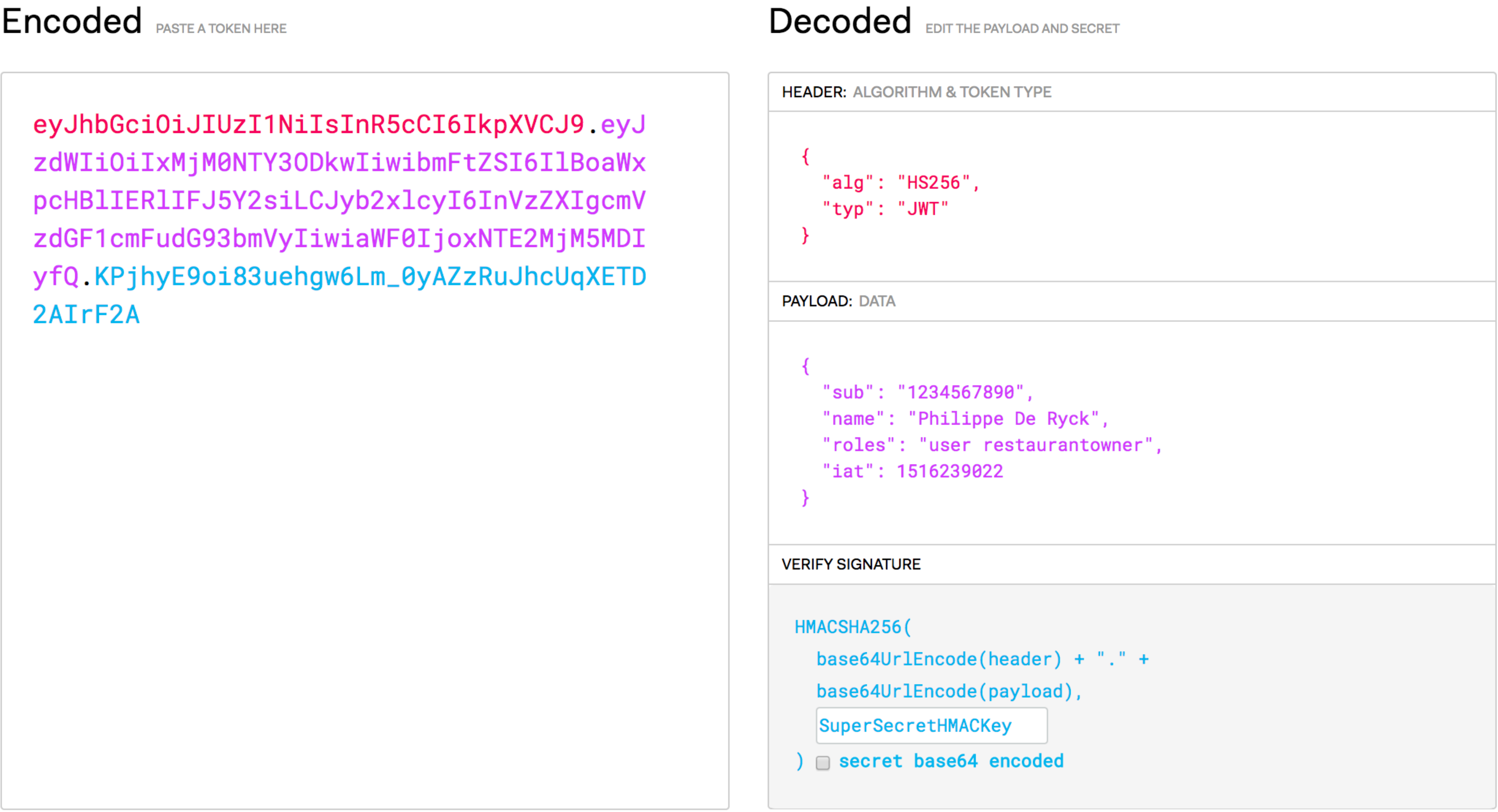
"I'll just use refresh tokens!"

SECURITY PROBLEM

You can't revoke the long-term tokens, which means you're back to square one.

Mistaking JWTs for sessions

JWTs are a way to represent claims, nothing more. Using them for authorization data requires an elaborate support system, such as OAuth 2.0



Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
zdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IlBoaWx
pcHBlIERlIFJ5Y2siLCJyb2xlcyI6InVzZXIgcmV
zdGF1cmFudG93bmVyIiwiaWF0IjoxNTE2MjM5MDI
yfQ.KPjhyE9oi83uehgw6Lm_0yAZzRuJhcUqXETD
2AIrF2A
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

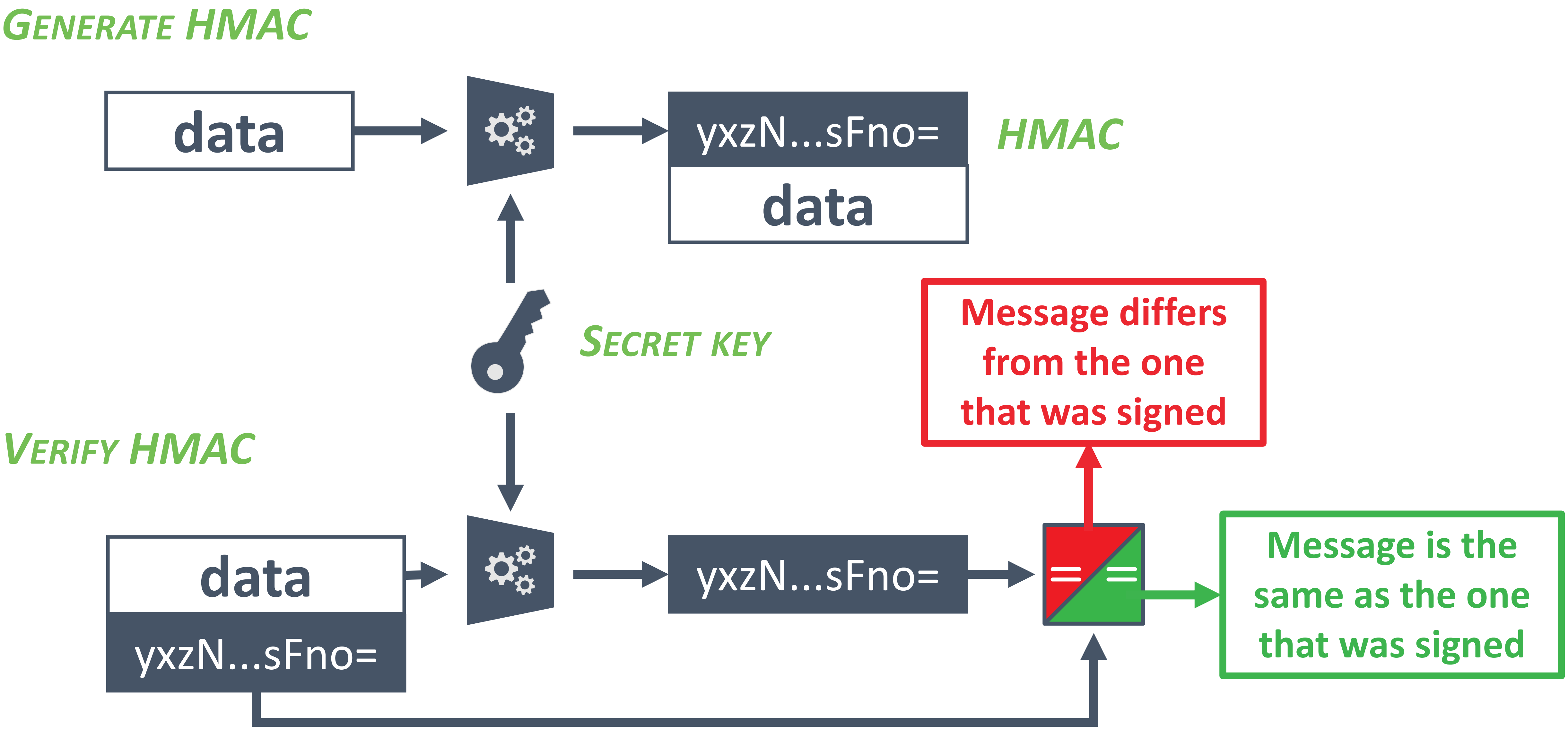
PAYLOAD: DATA

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

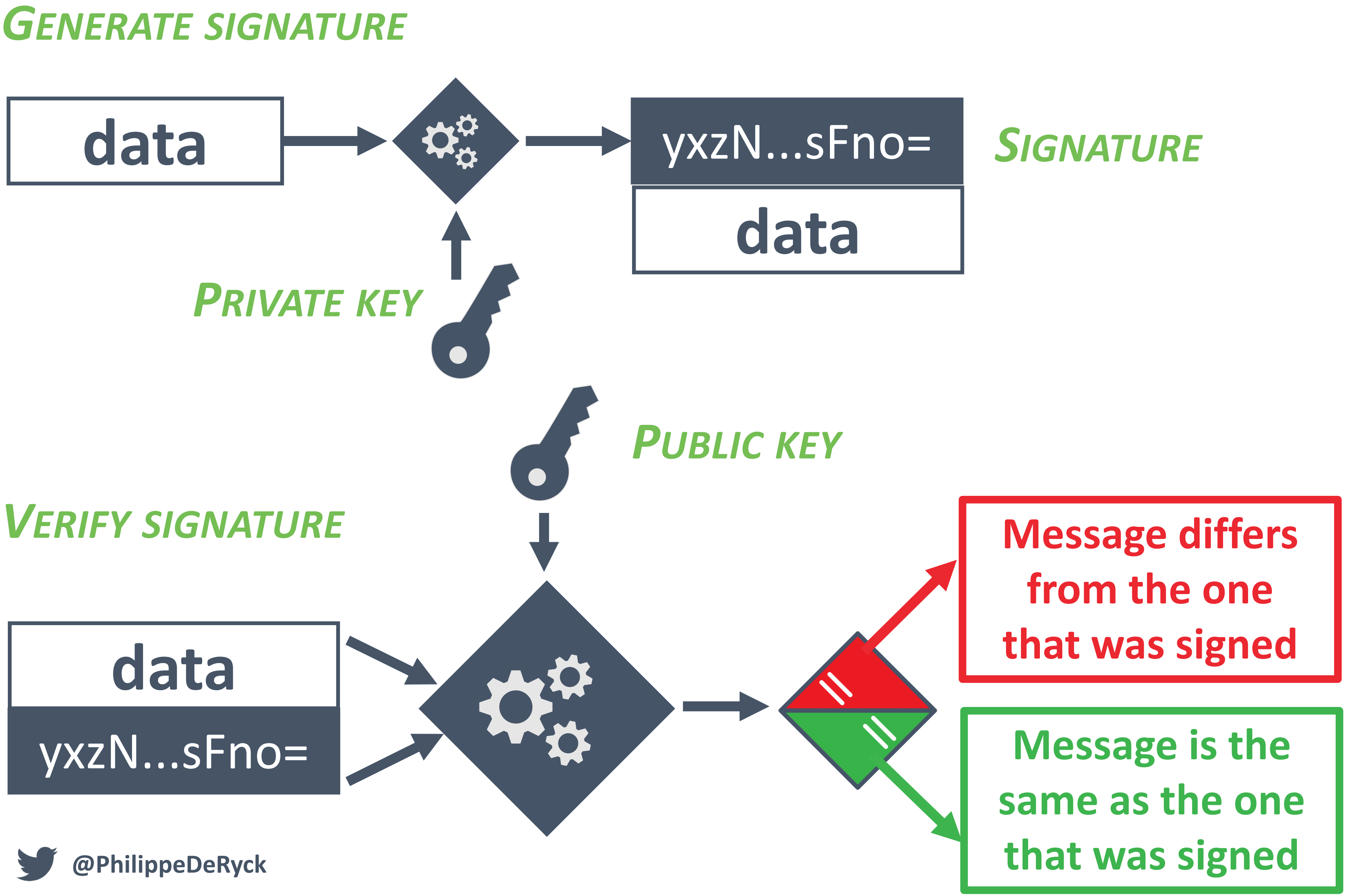
VERIFY SIGNATURE

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

HMAC-based JWT signatures


Generate HMAC
data
yxzN...sFno=
data
HMAC
Secret key
Verify HMAC
data
yxzN...sFno=
yxzN...sFno=
=
=
Message differs from the one that was signed
Message is the same as the one that was signed

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

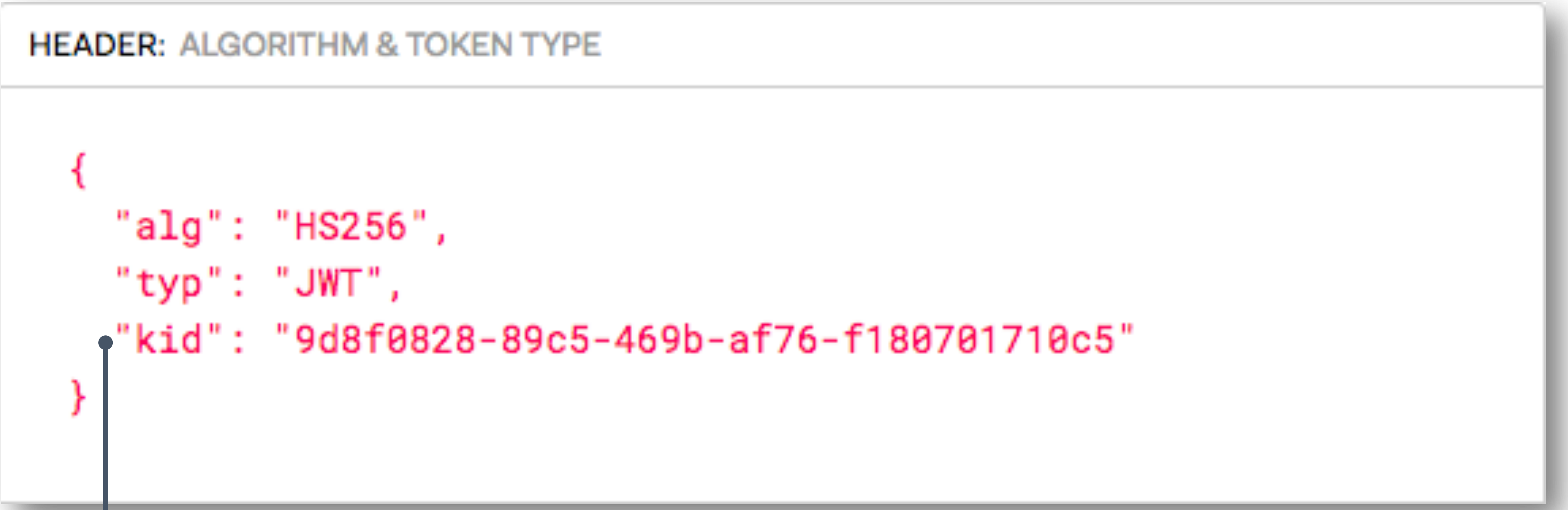
- Symmetric signatures are useful to use within a single trust zone
  - Backend service storing claims in a JWT for use within the application
  - Symmetric signatures are not the right choice when other (internal) services are involved
    - ***Never ever share your secret signing key!***

- 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)

MISUSING THE JWT SIGNATURE SCHEME

Shared secrets for verifying JWT tokens are for use within the boundaries of the application.

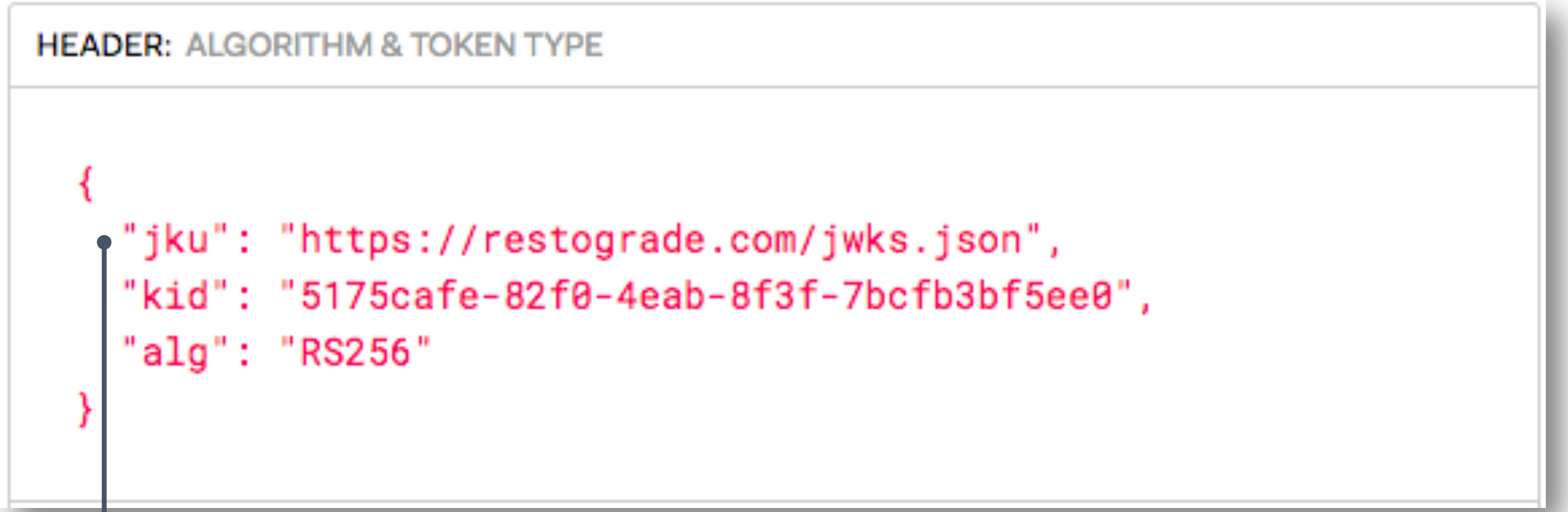
Most scenarios should use a public/private key pair.



HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT",
  "kid": "9d8f0828-89c5-469b-af76-f180701710c5"
}
```

Identify a key known by the receiver



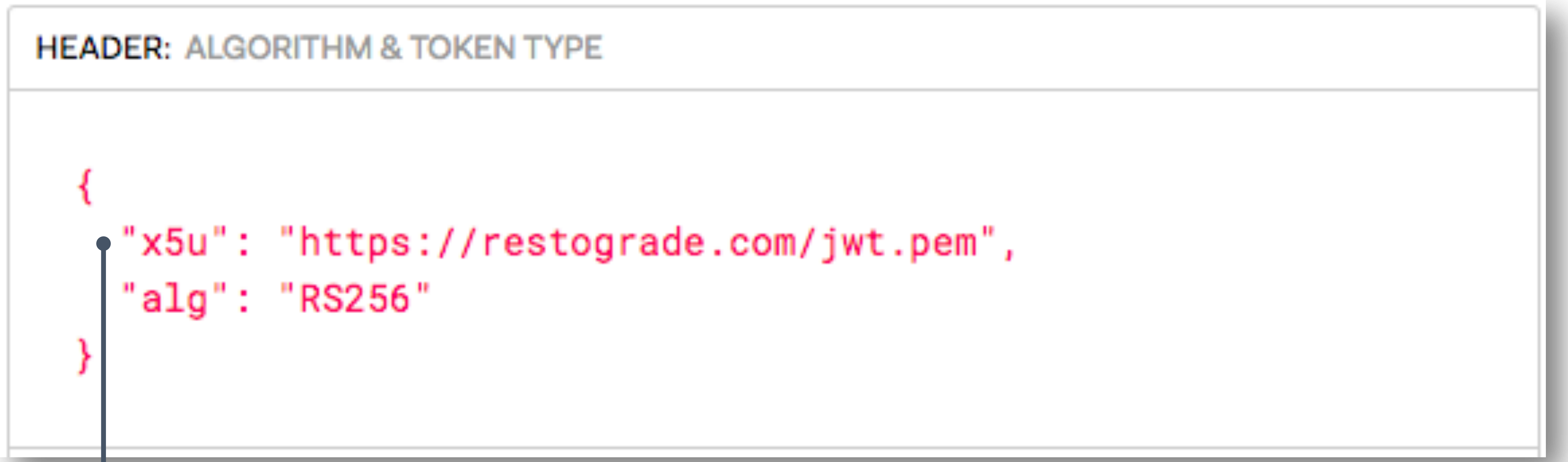
```
{
  "jku": "https://restograde.com/jwks.json",
  "kid": "5175cafe-82f0-4eab-8f3f-7bcfb3bf5ee0",
  "alg": "RS256"
}
```

Provide a URL containing a set of keys

```java
// Library: com.nimbusds.nimbus-jose-jwt
JWSHeader header = new JWSHeader.Builder(JWSAlgorithm.RS256)
   .jwkURL(new URI("https://restograde.com/jwks.json"))
   .keyID(keyID)
   .build();

JWTClaimsSet claimsSet = new JWTClaimsSet.Builder()
   .issueTime(new Date())
   .issuer(”https://restograde.com")
   .claim("username", "philippe")
   .build();

JWSSigner signer = new RSASSASigner(privateKey);
SignedJWT jwt = new SignedJWT(header, claimsSet);
jwt.sign(signer);
result = jwt.serialize();
```



```
{
  "x5u": "https://restograde.com/jwt.pem",
  "alg": "RS256"
}
```

Provide a X.509 certificate with a key

KEY IDENTIFICATION IN JWTS

- Asymmetric algorithms use a key pair
  - The private key is used to generate a signature and is kept secret
  - The public key is used to verify a signature and can be publicly known

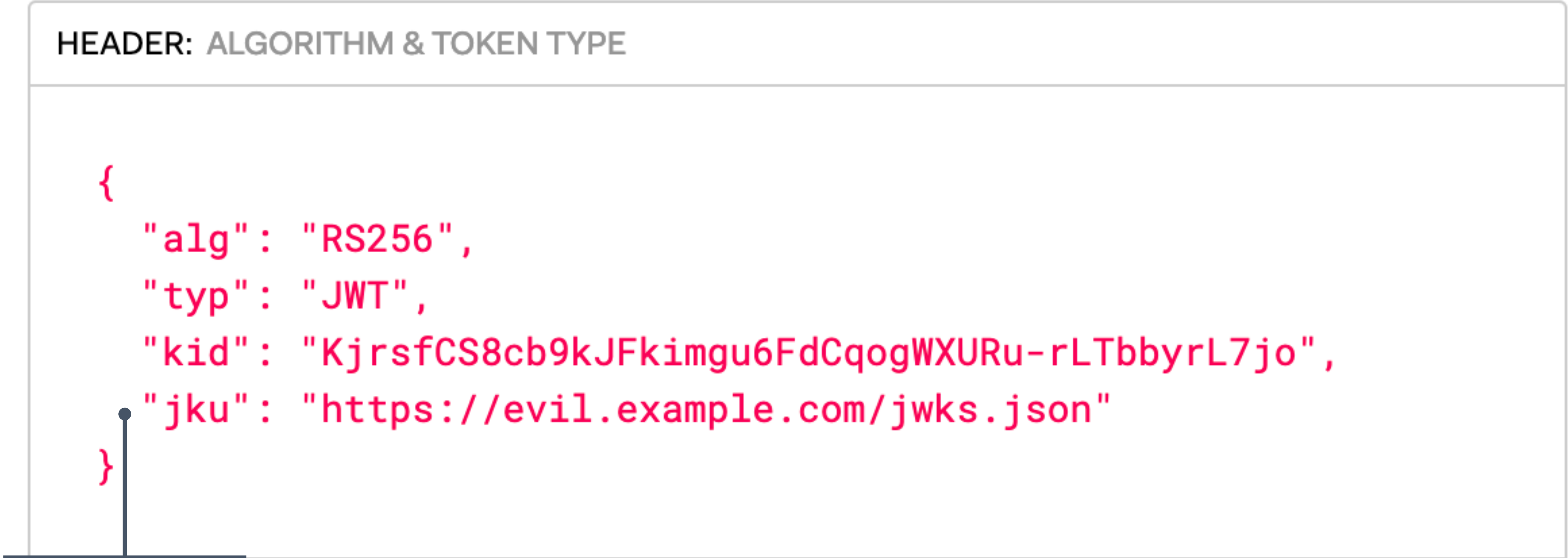
- Simple approach uses the ***kid*** parameter to identify the public key
  - The parameter could include a fingerprint of the public key
  - Of course, this still requires the receiver to obtain the public key one way or another

- But the public key is public, so it can also be included as part of the JWT token
  - The specification supports this through various parameters
  - The set of parameters are ***jku***, ***jwk***, ***kid***, ***x5u***, and ***x5c***

```java
// Library: com.nimbusds.nimbus-jose-jwt
JWSHeader header = new JWSHeader.Builder(JWSAlgorithm.RS256)
   .jwkURL(new URI("https://restograde.com/jwks.json"))
   .keyID(keyID)
   .build();

JWTClaimsSet claimsSet = new JWTClaimsSet.Builder()
   .issueTime(new Date())
   .issuer("restograde.com")
   .claim("username", "philippe")
   .build();

JWSSigner signer = new RSASSASigner(privateKey);
SignedJWT jwt = new SignedJWT(header, claimsSet);
jwt.sign(signer);
result = jwt.serialize();
```



HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "KjrsfCS8cb9kJFkimgu6FdCqogWXURu-rLTbbyrL7jo",
  "jku": "https://evil.example.com/jwks.json"
}
```

🔥

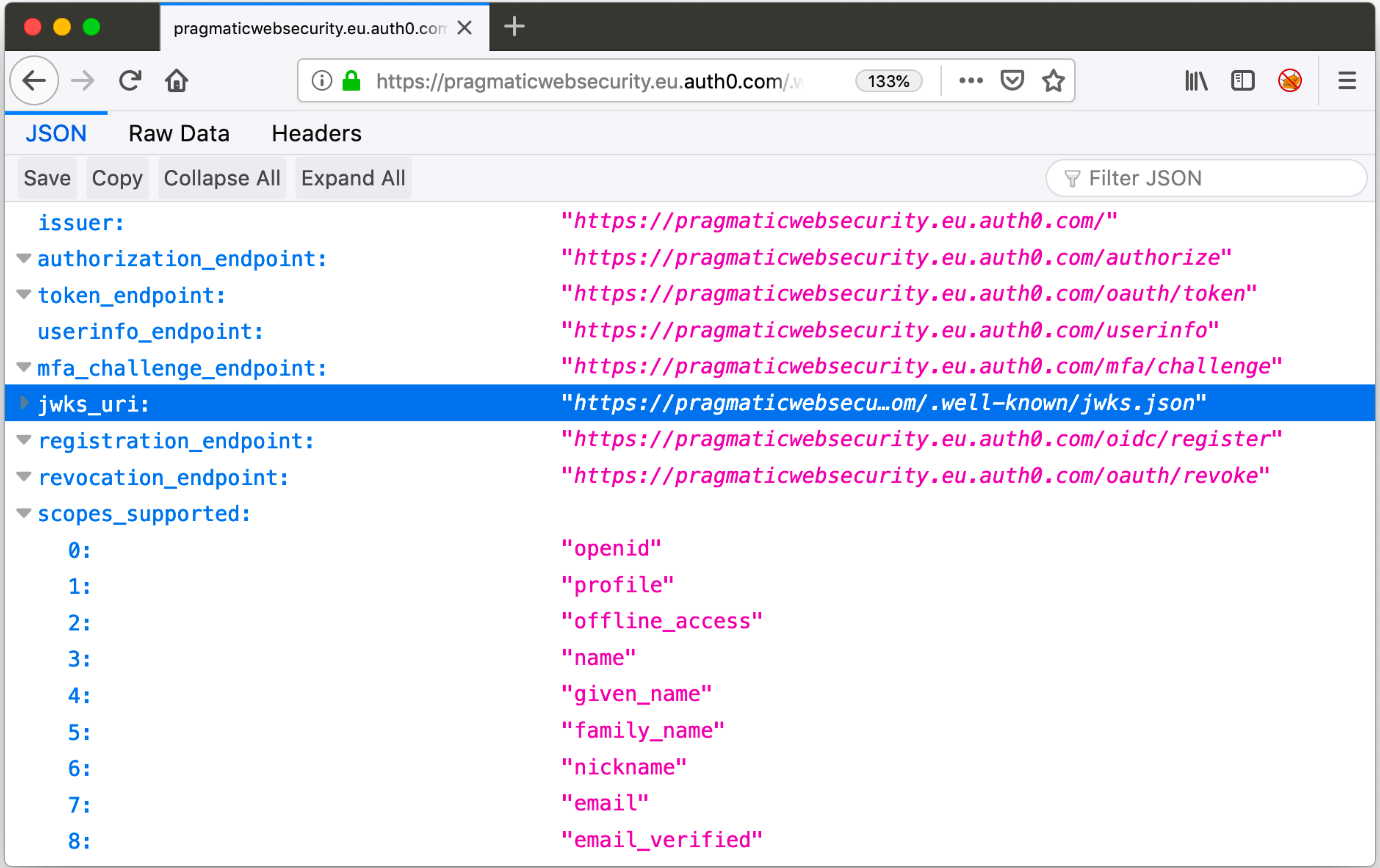
TRUSTING THE KEY

- Trusting the key which is embedded in the JWT is a difficult problem
  - Your application should restrict which keys it accepts
  - The attacker can always provide a signed JWT containing a valid key

- Approving specific keys
  - The application can identify a set of valid keys using their fingerprints
  - Dynamic whitelisting can be done using backchannel requests to load keys
    - Only load keys from trusted sources

- Limiting valid sources of the keys
  - Dynamic JWK URLs can be whitelisted per valid domain (and path if possible)
  - Certificate-based keys should be checked for a valid ***Common Name*** in the certificate

.well-known/openid-configuration


pragmaticwebsecurity.eu.auth0.com
https://pragmaticwebsecurity.eu.auth0.com/.w
133%
JSON
Raw Data
Headers
Save
Copy
Collapse All
Expand All
Filter JSON

```
issuer:                     "https://pragmaticwebsecurity.eu.auth0.com/"
▼ authorization_endpoint:   "https://pragmaticwebsecurity.eu.auth0.com/authorize"
▼ token_endpoint:           "https://pragmaticwebsecurity.eu.auth0.com/oauth/token"
userinfo_endpoint:          "https://pragmaticwebsecurity.eu.auth0.com/userinfo"
▼ mfa_challenge_endpoint:   "https://pragmaticwebsecurity.eu.auth0.com/mfa/challenge"
▶ jwks_uri:                 "https://pragmaticwebsecu…om/.well-known/jwks.json"
▼ registration_endpoint:    "https://pragmaticwebsecurity.eu.auth0.com/oidc/register"
▼ revocation_endpoint:      "https://pragmaticwebsecurity.eu.auth0.com/oauth/revoke"
▼ scopes_supported:
    0:                      "openid"
    1:                      "profile"
    2:                      "offline_access"
    3:                      "name"
    4:                      "given_name"
    5:                      "family_name"
    6:                      "nickname"
    7:                      "email"
    8:                      "email_verified"
```

```
String domain = ”pragmaticwebsecurity.eu.auth0.com";

// Get the proper key material
DecodedJWT insecureJwt = JWT.decode(identityToken);
String kid = insecureJwt.getKeyId();
Jwk jwk = getProvider(domain).get(kid);

// Verify the signature on the token
Algorithm algorithm = Algorithm.RSA256((RSAPublicKey)
                                       jwk.getPublicKey(), null);
JWTVerifier verifier = JWT.require(algorithm)
.withAudience(clientId)
    .withIssuer(issuer)
    .withClaim("nonce", session.getAttribute("oidc.nonce").toString())
    .build();
DecodedJWT jwt = verifier.verify(identityToken);

logger.info("Successfully verified identity token");
logger.debug(identityToken);
```

Lack of proper JWT key management

Cryptographic keys used for encryption and signatures need to be frequently rotated.

Your API should be prepared to handle key rotation.

Cookie: **ID=42**

Authorization: **Bearer 42**

Cookie: **JWT=eyJhbGci…**

Authorization: **Bearer eyJhbGci…**

```
<img src="https://.../philippe/1.png">

new WebSocket("wss://.../socket");
```

Cookies	Authorization header
Can contain identifiers & session objects	Can contain identifiers & session objects
Only works well with a single domain	Freedom to include headers to any domain
Automatically handled by the browser	Requires custom code to get, store and send session data
Always present, including on DOM resources and WebSockets	Only present on XHR calls, unless you circumvent this with a ServiceWorker

(DIS)ADVANTAGES OF THE *AUTHORIZATION* HEADER

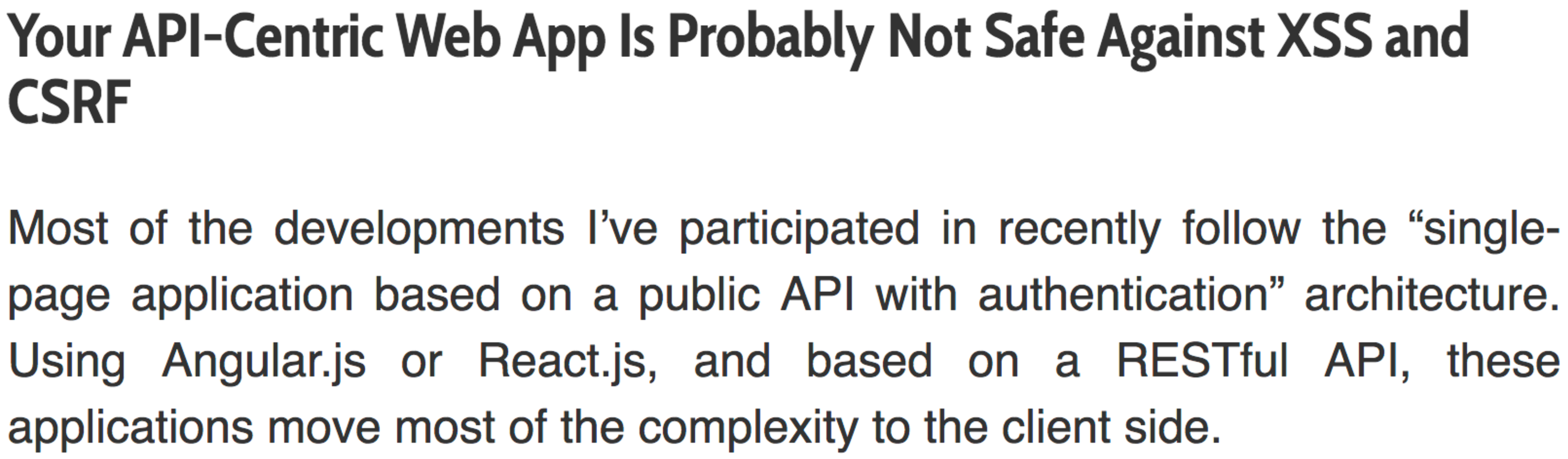
- The ***Authorization*** header offers a lot of flexibility
  - Custom control over where and how to add session data in the header
  - Not tied to a specific domain, so easy to support APIs on different domains
    - Cookies are tied to a domain, so are hard to use in such a context
  - No more dealing 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

Underestimating the impact of session transport

Cookies are often frowned upon in an API world, and custom headers are preferred.

Both have vastly different security properties, so make sure you understand them fully.

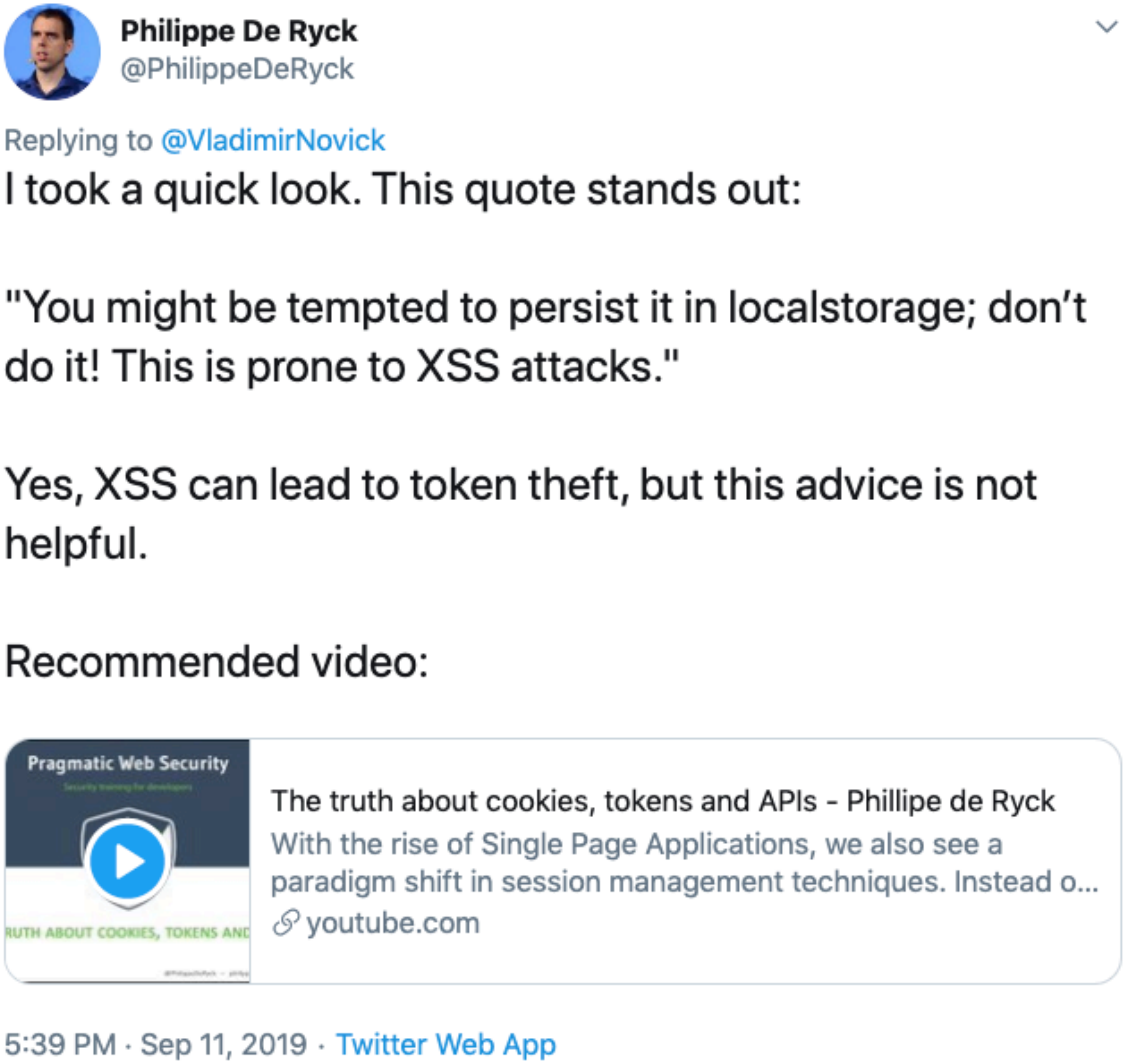


Your API-Centric Web App Is Probably Not Safe Against XSS and CSRF

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



Philippe De Ryck
@PhilippeDeRyck

Replying to @VladimirNovick

I took a quick look. This quote stands out:

"You might be tempted to persist it in localstorage; don't do it! This is prone to XSS attacks."

Yes, XSS can lead to token theft, but this advice is not helpful.

Recommended video:

The truth about cookies, tokens and APIs - Phillipe de Ryck
With the rise of Single Page Applications, we also see a paradigm shift in session management techniques. Instead o...
youtube.com

5:39 PM · Sep 11, 2019 · Twitter Web App

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

APPLY DEFENSE-IN-DEPTH AGAINST XSS

- The primary defense is secure coding to avoid XSS in the first place
  - History has shown us that XSS is still extremely common
  - Additional security techniques might help reduce the attack surface or attack impact

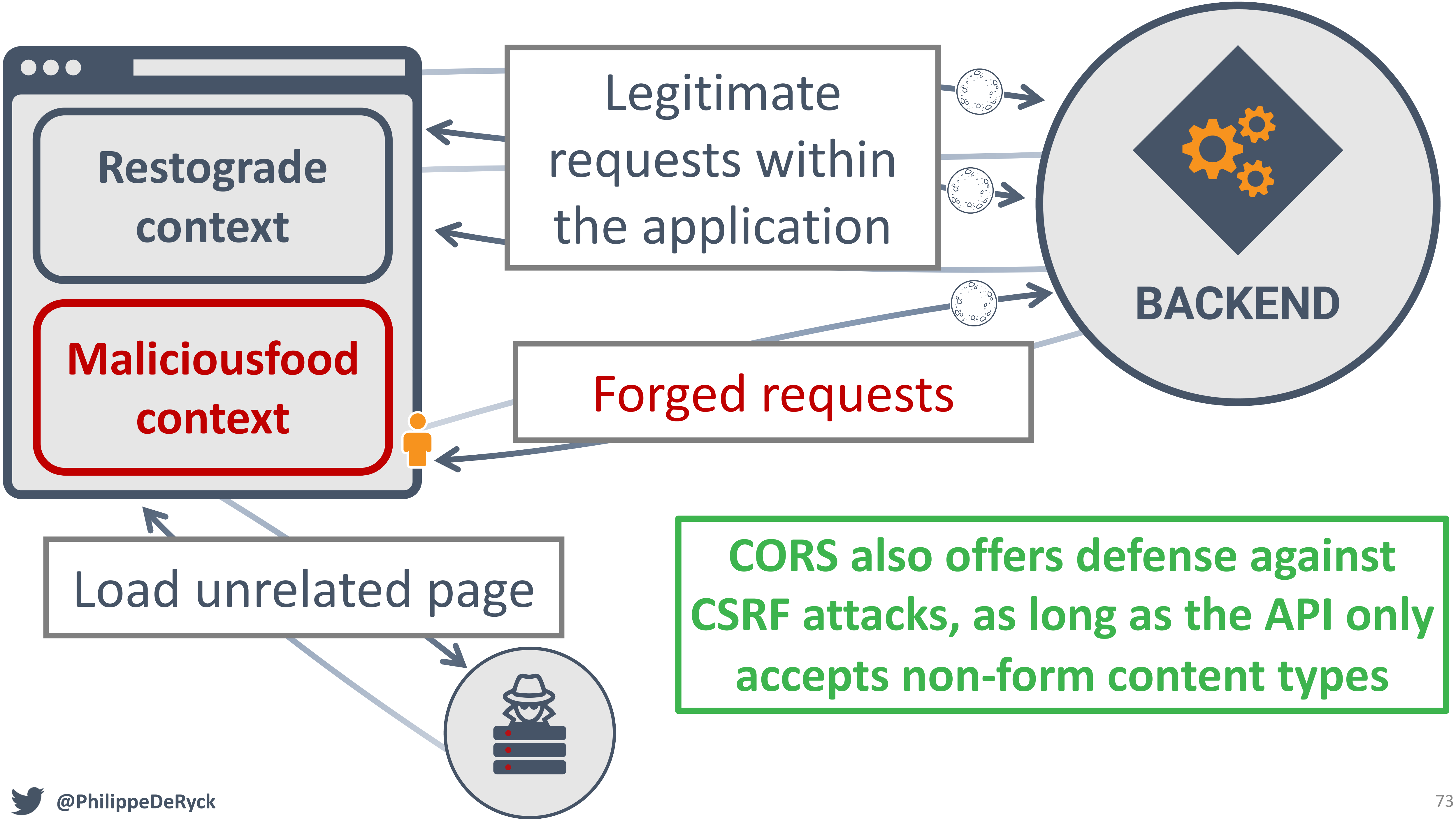
- Content Security Policy gives you control about what is loaded in a context
  - CSP can block the execution of injection script code
  - CSP is also useful to prevent the loading of potentially untrusted content

- The HTML5 sandbox brings behavioral control over an execution context
  - With a sandbox, content can be isolated in its own private origin
  - The sandbox also allows to enforce a set of behavioral restrictions

Underestimating the impact of XSS

Stealing data from localStorage is only a single consequence of XSS.

XSS means game over. You lost.



Restograde context

Maliciousfood context

Legitimate requests within the application

BACKEND

Forged requests

Load unrelated page

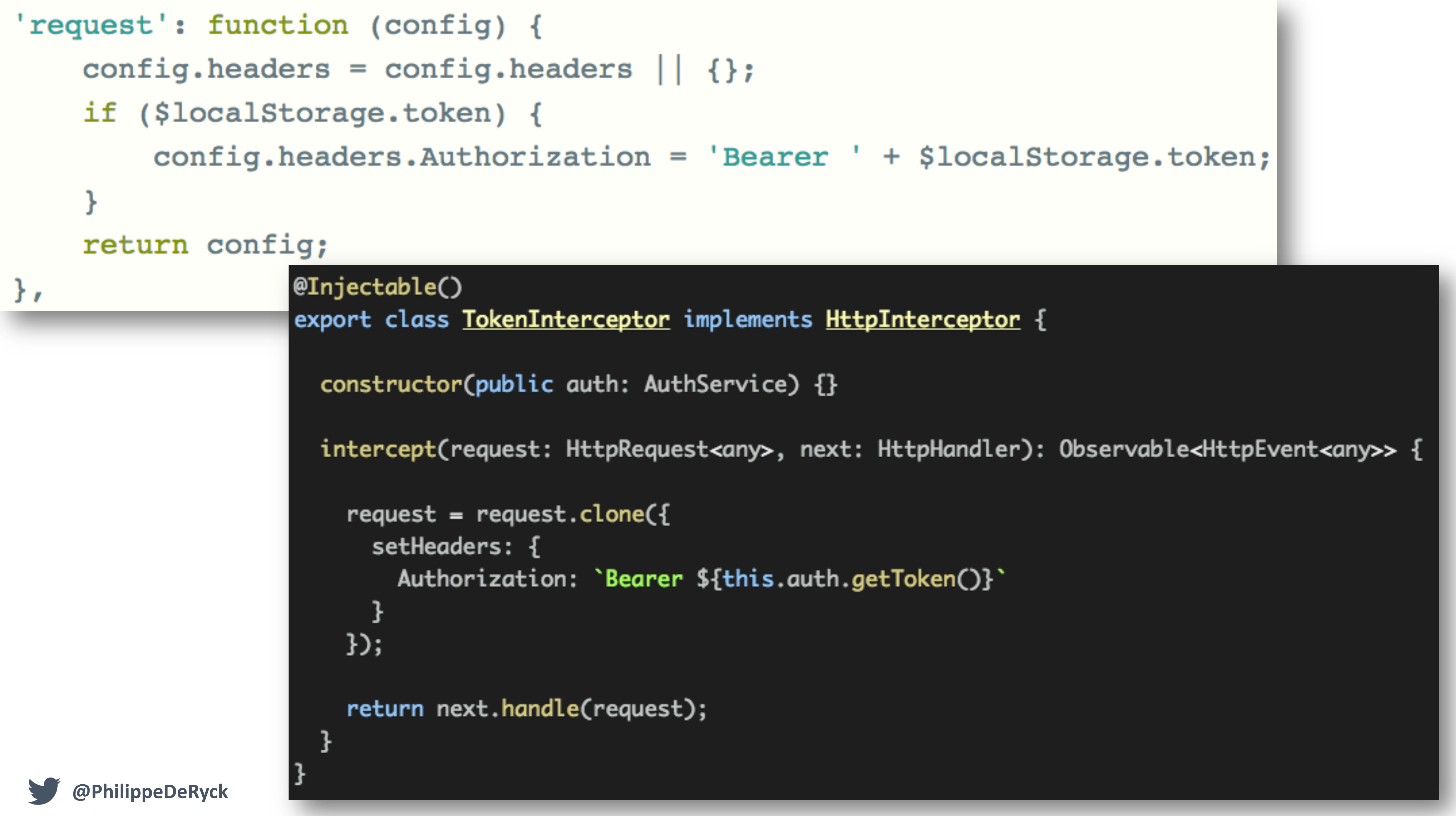
CORS also offers defense against CSRF attacks, as long as the API only accepts non-form content types

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



```
'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);
  }
}
```

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

Regardless of the session storage mechanism, **XSS means game over**

Using cookies requires the use of CSRF protection, or force the use of CORS preflights

Using the Authorization header requires explicitly approving expected destinations

Confusion about CSRF

Cookie-based mechanisms require explicit CSRF defenses. Authorization-header based mechanism require a secure implementation.

application/json

```
OPTIONS /api/reviews/1
Origin: https://maliciousfood.com
Access-Control-Request-Method: PUT
```

THE RELATION BETWEEN CORS AND CSRF

- Before CORS, ”non-simple” requests could be same-origin
  - A server expecting a DELETE would rely on the browser refusing cross-origin DELETEs
  - But with CORS, this security assumption changes

- Simply denying access to the response of such requests is not enough
  - If the request triggered a state-changing action on the server, it is too late
  - Therefore, CORS needs to ask for approval before sending such a request

- CORS asks for approval with a preflight OPTIONS request
  - The request tells the server what the browser wants to do
  - The server needs to respond with the proper CORS headers to authorize the request

Failing to enforce a strict CORS policy

Cross-origin API requests are only fully protected by CORS if they cannot be forged with HTML elements.

Force the use of preflight requests by not accepting form-based content types.

```
Origin: https://restograde.com
```

```
if(origin.startsWith("https://restograde.com"))

if(origin.endsWith("restograde.com"))

if(origin.contains("restograde.com"))
```

```
Origin: https://restograde.com.maliciousfood.com
```

```
Origin: https://maliciousrestograde.com
```

Mismatching origins

- Matching the value of the ***Origin*** header against a whitelist is crucial
  - The outcome of this matching will directly influence the authorization decision
  - Real-world CORS implementations have trouble implementing matching correctly

- Always perform matching against the full origin
  - Partial matching can be bypassed by registering crafted domains
  - Failing to include the domain allows bypass attacks using HTTP pages

- Do not allow **null** as a valid origin
  - The value ***null*** is used as the canonicalization of an untrusted context
  - Whitelisting ***null*** is worse than using a wildcard, since null allows the use of credentials
  - Whitelisting ***null*** means the endpoint accepts authenticated requests from anywhere

```
SetEnvIf Origin "http(s)?://.*$" ACO=$0
Header add Access-Control-Allow-Origin %{ACO}e env=ACO
Header set Access-Control-Allow-Headers "Range"
Header set Access-Control-Allow-Credentials "true"
```

```
SetEnvIf Origin "http(s)?://.*$" ACO=$0
Header add Access-Control-Allow-Origin %{ACO}e env=ACO
Header set Access-Control-Allow-Headers "Range"
Header set Access-Control-Allow-Credentials "true"
```

Failure to configure or implement CORS

CORS policies heavily depend on checking the value of the Origin header.

Enforce strict whitelisting, and verify your implementation against common mistakes.

/users/1’%20OR%20’1’=‘1

INPUT VALIDATION SHOULD BE A FIRST LINE OF DEFENSE

- Input validation is useful to reject obvious malicious data
  - Helps prevent against DoS attacks by rejecting unreasonably large inputs
  - Helps prevent against injection attacks by rejecting crafted payloads

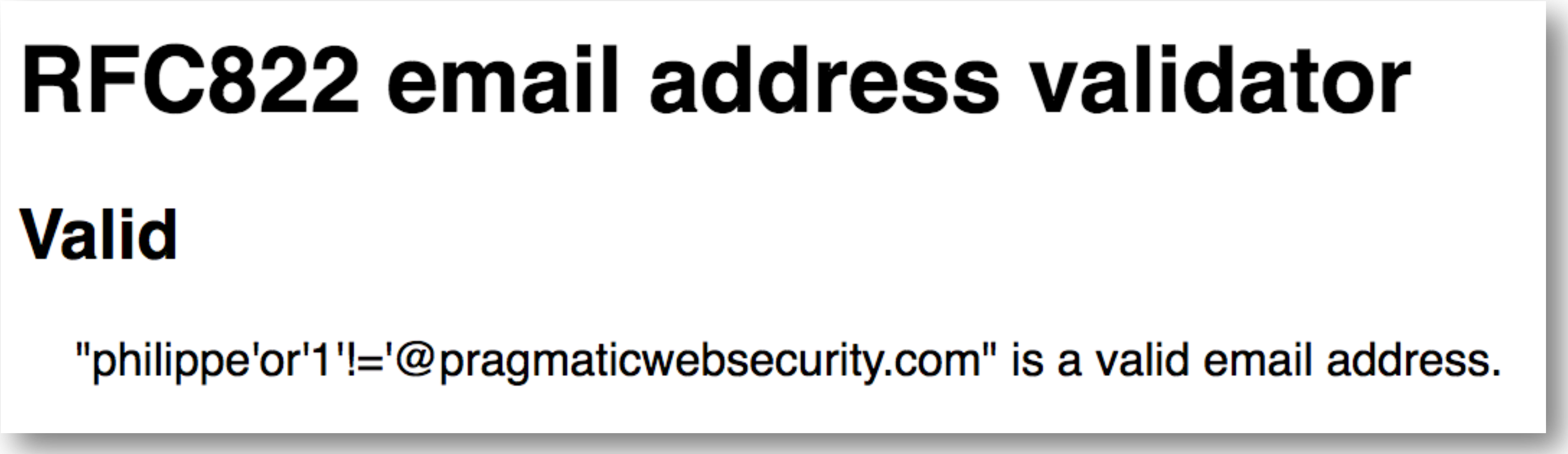
- Rules of thumb of input validation
  - Enforce sensible length limits on inputs
    - E.g., 5MB of text is probably not a valid password
  - Enforce strict content types on provided data
    - E.g., an API expecting JSON data should not accept anything else, even if it looks like JSON
  - Enforce strict data type checking on inputs
    - Numbers are numbers, and SQL code as input should result in an error
  - When unsure about the input, better to be too lax than too strict
    - Being too strict breaks functionality, and input validation is only a first line of defense

Lack of input validation

A lack of input validation is the enabler for various other attacks.

Ensure that input validation is as strict as possible without triggering false positives

```
philippe'or'1'!='@pragmaticwebsecurity.com
```



RFC822 email address validator

Valid

"philippe'or'1'!='@pragmaticwebsecurity.com" is a valid email address.

INPUT VALIDATION FAILS AS A PRIMARY DEFENSE

- Once data is complex enough, input validation will not prevent attacks
  - Determining the validity of complex data at input time is virtually impossible
  - Complex validation procedures often suffer from bypass attacks
  - Overly strict validation procedures will break legitimate functionality

- Many attacks can only be stopped when output is generated
  - At output time, the context determines how data may be considered dangerous
    - Examples are XSS, SQL injection, command injection, ...
  - At input time, it is not possible to anticipate all potential output locations
    - As a consequence, it is not possible to use input validation as a primary defense

Relying on input validation as a primary defense

Even though input validation is a good first line of defense, it will fail as the only defense.

Do not rely on input validation alone.

What happens when 💩 goes wrong?

FAILURE TO COMPARTMENTALIZE

Many APIs combine sensitive features (e.g. Authentication) and application logic (e.g. data access) into a single service. Compartmentalization helps limit the impact of a vulnerability.

Question everything

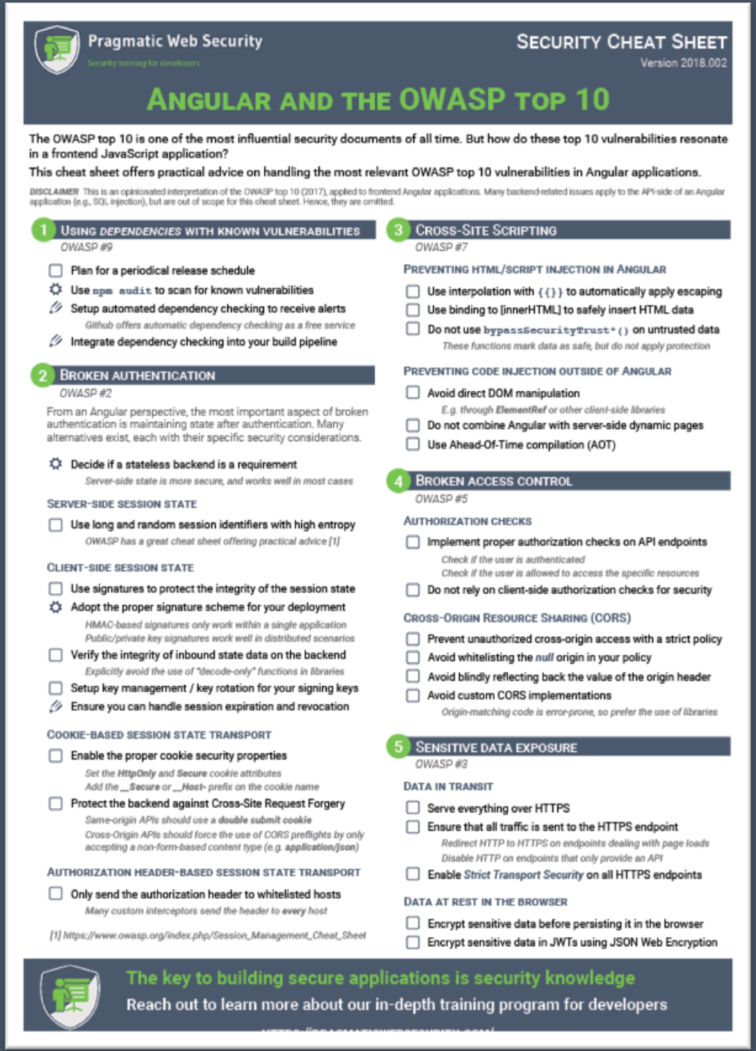
How is this different from what we used to do?

Do we really understand what we’re doing?

Have we validated the integrity and format of that data?

…

FREE SECURITY CHEAT SHEETS FOR MODERN APPLICATIONS



Pragmatic Web Security
Security training for developers

SECURITY CHEAT SHEET
Version 2018.002

ANGULAR AND THE OWASP TOP 10

The OWASP top 10 is one of the most influential security documents of all time. But how do these top 10 vulnerabilities resonate in a frontend JavaScript application?

This cheat sheet offers practical advice on handling the most relevant OWASP top 10 vulnerabilities in Angular applications.

DISCLAIMER This is an opinionated interpretation of the OWASP top 10 (2017), applied to frontend Angular applications. Many backend-related issues apply to the API-side of an Angular application (e.g., SQL injection), but are out of scope for this cheat sheet. Hence, they are omitted.

1 USING *DEPENDENCIES* WITH KNOWN VULNERABILITIES

OWASP #9

□ Plan for a periodical release schedule
⚙ Use `npm audit` to scan for known vulnerabilities
✎ Setup automated dependency checking to receive alerts
Github offers automatic dependency checking as a free service
✎ Integrate dependency checking into your build pipeline

2 BROKEN AUTHENTICATION

OWASP #2

From an Angular perspective, the most important aspect of broken authentication is maintaining state after authentication. Many alternatives exist, each with their specific security considerations.

⚙ Decide if a stateless backend is a requirement
Server-side state is more secure, and works well in most cases

SERVER-SIDE SESSION STATE

□ Use long and random session identifiers with high entropy
OWASP has a great cheat sheet offering practical advice [1]

CLIENT-SIDE SESSION STATE

□ Use signatures to protect the integrity of the session state
⚙ Adopt the proper signature scheme for your deployment
HMAC-based signatures only work within a single application
Public/private key signatures work well in distributed scenarios
□ Verify the integrity of inbound state data on the backend
Explicitly avoid the use of "decode-only" functions in libraries
□ Setup key management / key rotation for your signing keys
✎ Ensure you can handle session expiration and revocation

COOKIE-BASED SESSION STATE TRANSPORT

□ Enable the proper cookie security properties
*Set the **HttpOnly** and **Secure** cookie attributes*
*Add the **__Secure** or **__Host-** prefix on the cookie name*
□ Protect the backend against Cross-Site Request Forgery
*Same-origin APIs should use a **double submit cookie***
*Cross-Origin APIs should force the use of CORS preflights by only accepting a non-form-based content type (e.g. **application/json**)*

AUTHORIZATION HEADER-BASED SESSION STATE TRANSPORT

□ Only send the authorization header to whitelisted hosts
*Many custom interceptors send the header to **every** host*

[1] https://www.owasp.org/index.php/Session_Management_Cheat_Sheet

3 CROSS-SITE SCRIPTING

OWASP #7

PREVENTING HTML/SCRIPT INJECTION IN ANGULAR

□ Use interpolation with `{{}}` to automatically apply escaping
□ Use binding to [innerHTML] to safely insert HTML data
□ Do not use `bypassSecurityTrust*()` on untrusted data
These functions mark data as safe, but do not apply protection

PREVENTING CODE INJECTION OUTSIDE OF ANGULAR

□ Avoid direct DOM manipulation
*E.g. through **ElementRef** or other client-side libraries*
□ Do not combine Angular with server-side dynamic pages
□ Use Ahead-Of-Time compilation (AOT)

4 BROKEN ACCESS CONTROL

OWASP #5

AUTHORIZATION CHECKS

□ Implement proper authorization checks on API endpoints
Check if the user is authenticated
Check if the user is allowed to access the specific resources
□ Do not rely on client-side authorization checks for security

CROSS-ORIGIN RESOURCE SHARING (CORS)

□ Prevent unauthorized cross-origin access with a strict policy
□ Avoid whitelisting the *null* origin in your policy
□ Avoid blindly reflecting back the value of the origin header
□ Avoid custom CORS implementations
Origin-matching code is error-prone, so prefer the use of libraries

5 SENSITIVE DATA EXPOSURE

OWASP #3

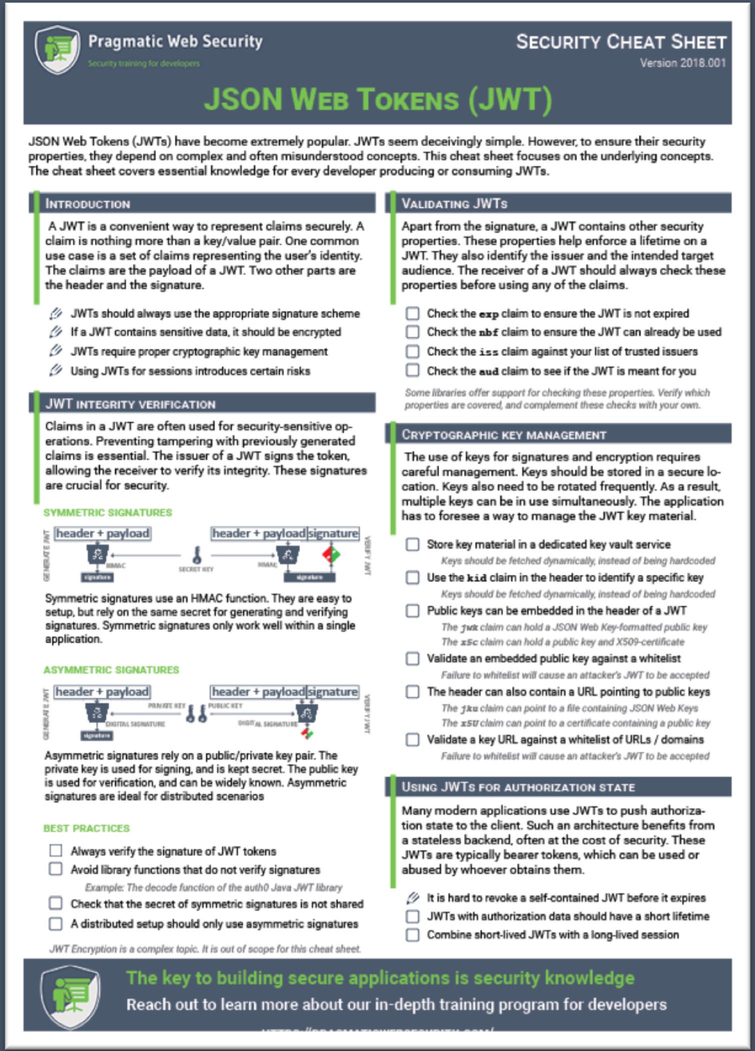
DATA IN TRANSIT

□ Serve everything over HTTPS
□ Ensure that all traffic is sent to the HTTPS endpoint
Redirect HTTP to HTTPS on endpoints dealing with page loads
Disable HTTP on endpoints that only provide an API
□ Enable *Strict Transport Security* on all HTTPS endpoints

DATA AT REST IN THE BROWSER

□ Encrypt sensitive data before persisting it in the browser
□ Encrypt sensitive data in JWTs using JSON Web Encryption

The key to building secure applications is security knowledge
Reach out to learn more about our in-depth training program for developers



Pragmatic Web Security
Security training for developers

SECURITY CHEAT SHEET
Version 2018.001

JSON WEB TOKENS (JWT)

JSON Web Tokens (JWTs) have become extremely popular. JWTs seem deceivingly simple. However, to ensure their security properties, they depend on complex and often misunderstood concepts. This cheat sheet focuses on the underlying concepts. The cheat sheet covers essential knowledge for every developer producing or consuming JWTs.

INTRODUCTION

A JWT is a convenient way to represent claims securely. A claim is nothing more than a key/value pair. One common use case is a set of claims representing the user's identity. The claims are the payload of a JWT. Two other parts are the header and the signature.

✎ JWTs should always use the appropriate signature scheme
✎ If a JWT contains sensitive data, it should be encrypted
✎ JWTs require proper cryptographic key management
✎ Using JWTs for sessions introduces certain risks

JWT INTEGRITY VERIFICATION

Claims in a JWT are often used for security-sensitive operations. Preventing tampering with previously generated claims is essential. The issuer of a JWT signs the token, allowing the receiver to verify its integrity. These signatures are crucial for security.

SYMMETRIC SIGNATURES

Symmetric signatures use an HMAC function. They are easy to setup, but rely on the same secret for generating and verifying signatures. Symmetric signatures only work well within a single application.

ASYMMETRIC SIGNATURES

Asymmetric signatures rely on a public/private key pair. The private key is used for signing, and is kept secret. The public key is used for verification, and can be widely known. Asymmetric signatures are ideal for distributed scenarios

BEST PRACTICES

□ Always verify the signature of JWT tokens
□ Avoid library functions that do not verify signatures
Example: The decode function of the auth0 Java JWT library
□ Check that the secret of symmetric signatures is not shared
□ A distributed setup should only use asymmetric signatures

JWT Encryption is a complex topic. It is out of scope for this cheat sheet.

VALIDATING JWTS

Apart from the signature, a JWT contains other security properties. These properties help enforce a lifetime on a JWT. They also identify the issuer and the intended target audience. The receiver of a JWT should always check these properties before using any of the claims.

□ Check the `exp` claim to ensure the JWT is not expired
□ Check the `nbf` claim to ensure the JWT can already be used
□ Check the `iss` claim against your list of trusted issuers
□ Check the `aud` claim to see if the JWT is meant for you

Some libraries offer support for checking these properties. Verify which properties are covered, and complement these checks with your own.

CRYPTOGRAPHIC KEY MANAGEMENT

The use of keys for signatures and encryption requires careful management. Keys should be stored in a secure location. Keys also need to be rotated frequently. As a result, multiple keys can be in use simultaneously. The application has to foresee a way to manage the JWT key material.

□ Store key material in a dedicated key vault service
Keys should be fetched dynamically, instead of being hardcoded
□ Use the `kid` claim in the header to identify a specific key
Keys should be fetched dynamically, instead of being hardcoded
□ Public keys can be embedded in the header of a JWT
The `jwk` claim can hold a JSON Web Key-formatted public key
The `x5c` claim can hold a public key and X509-certificate
□ Validate an embedded public key against a whitelist
Failure to whitelist will cause an attacker's JWT to be accepted
□ The header can also contain a URL pointing to public keys
The `jku` claim can point to a file containing JSON Web Keys
The `x5u` claim can point to a certificate containing a public key
□ Validate a key URL against a whitelist of URLs / domains
Failure to whitelist will cause an attacker's JWT to be accepted

USING JWTS FOR AUTHORIZATION STATE

Many modern applications use JWTs to push authorization state to the client. Such an architecture benefits from a stateless backend, often at the cost of security. These JWTs are typically bearer tokens, which can be used or abused by whoever obtains them.

✎ It is hard to revoke a self-contained JWT before it expires
□ JWTs with authorization data should have a short lifetime
□ Combine short-lived JWTs with a long-lived session

The key to building secure applications is security knowledge
Reach out to learn more about our in-depth training program for developers

https://cheatsheets.pragmaticwebsecurity.com/

SecAppDev

March 9th – 13th, 2020
Leuven, Belgium

A week-long course on Secure Application Development

Taught by experts from around the world

A scholarship program offering financial support and mentoring

A yearly initiative from the SecAppDev.org non-profit, since 2005

THANK YOU!

Follow me on Twitter to stay up to date on web security best practices

@PhilippeDeRyck