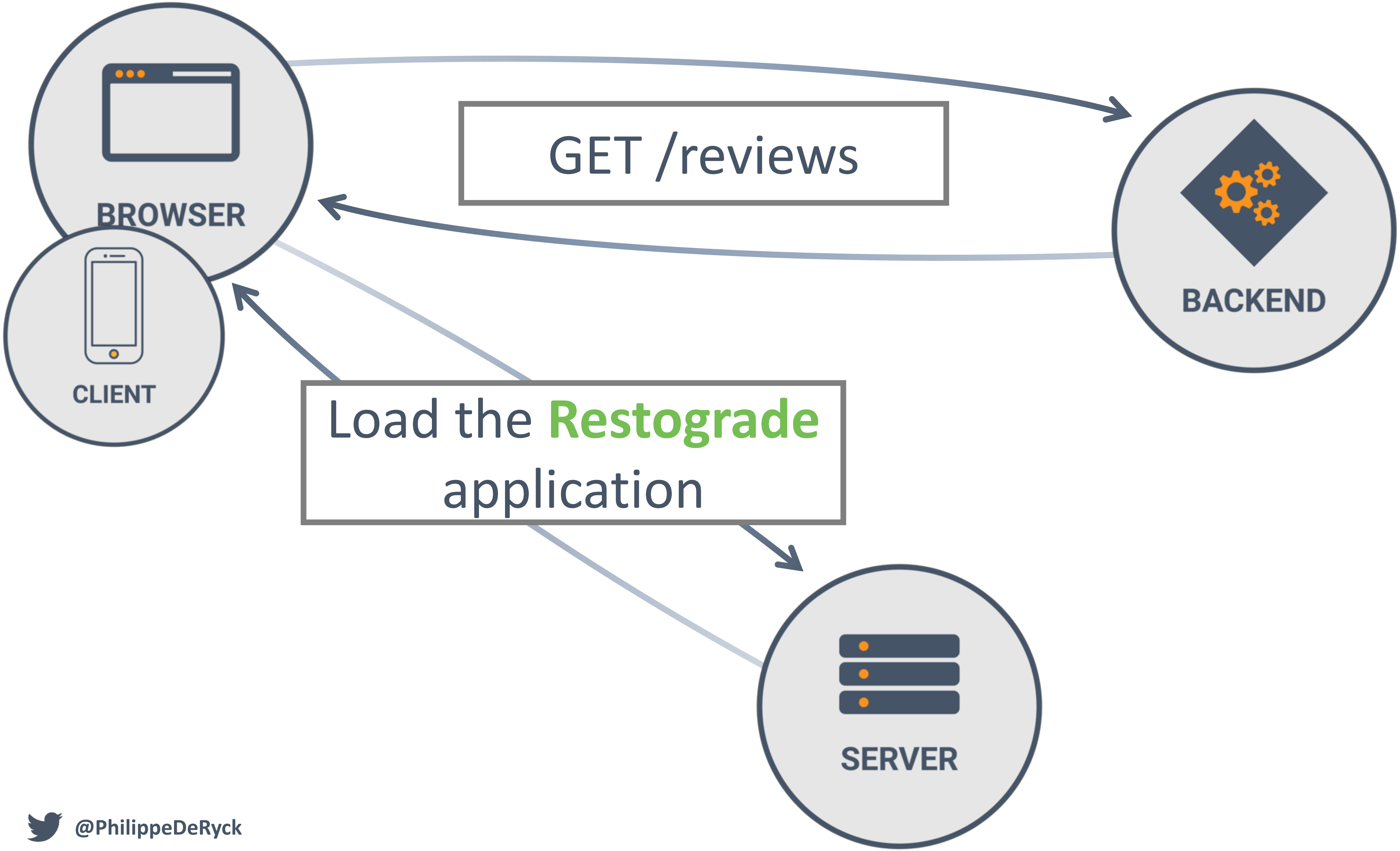
Pragmatic Web Security

Security training for developers

Common API security pitfalls

Philippe De Ryck

@PhilippeDeRyck – philippe@PragmaticWebSecurity.com


BROWSER
CLIENT
GET /reviews
BACKEND
Load the Restograde application
SERVER

A10 Underprotected APIs

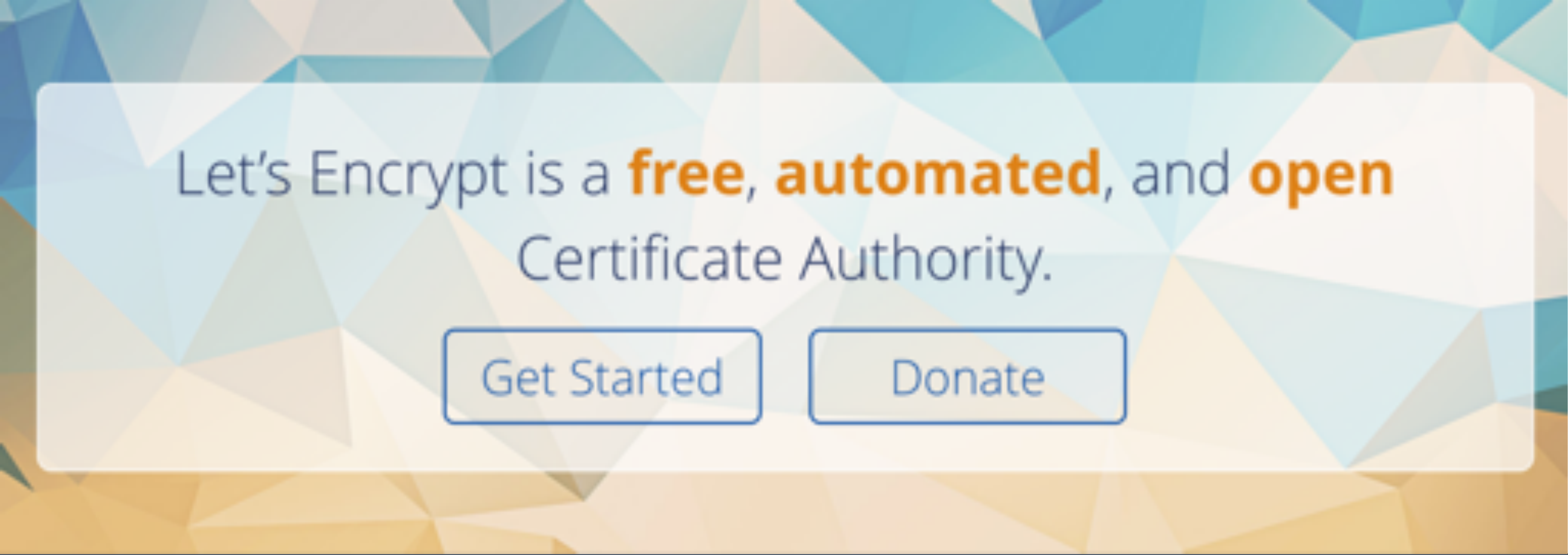
Application Specific	Exploitability AVERAGE	Prevalence COMMON	Detectability DIFFICULT	Impact MODERATE	Application / Business Specific
Consider anyone with the ability to send requests to your APIs. Client software is easily reversed and communications are easily intercepted, so obscurity is no defense 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.

- 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

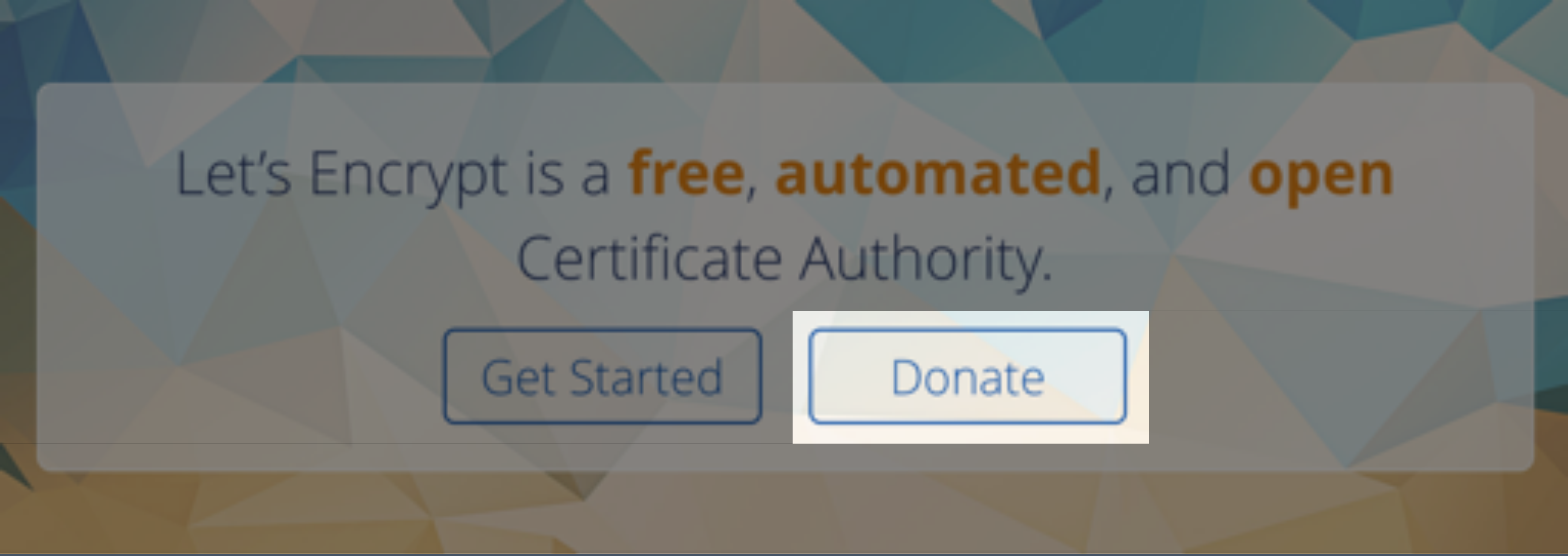
DR. PHILIPPE DE RYCK

PH.D. IN WEB SECURITY

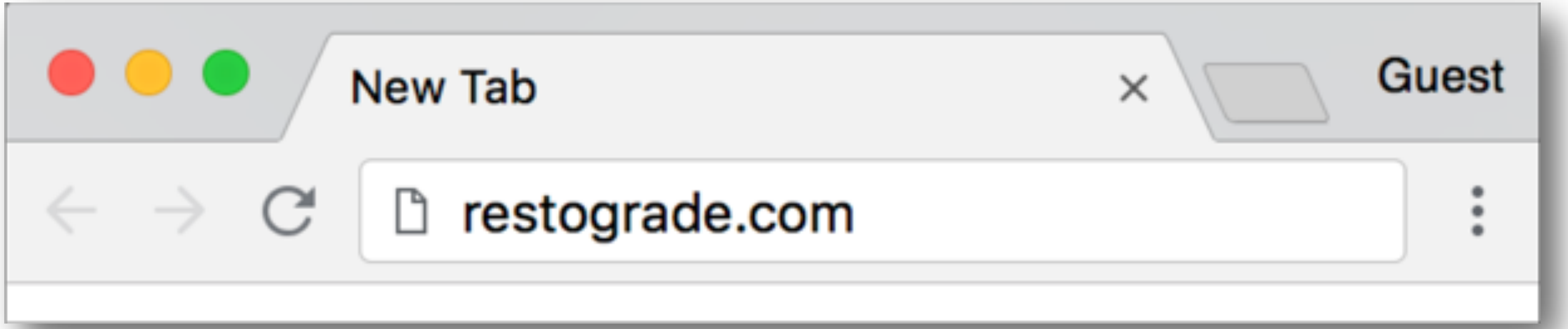
GOOGLE DEVELOPER EXPERT



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



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




```
# 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.

[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);
  });
};
```

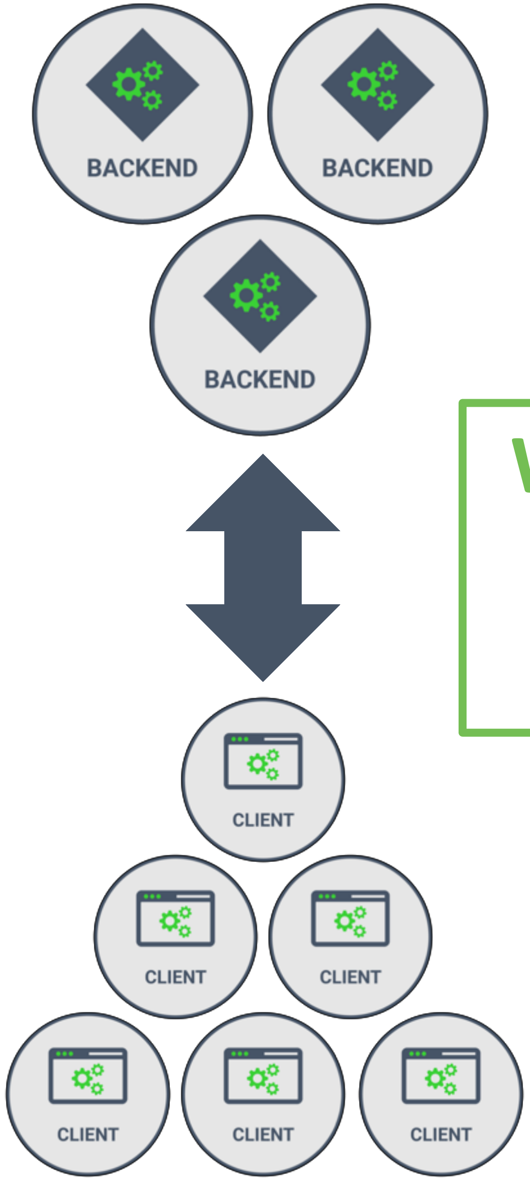
```
exports.delete_a_task = function(req, res) {
  Task.remove({
    _id: req.params.taskId
  }, function(err, task) {
    if (err)
      res.send(err);
    res.json({ message: 'Task successfully deleted' });
  });
};
```

Insecure direct object references

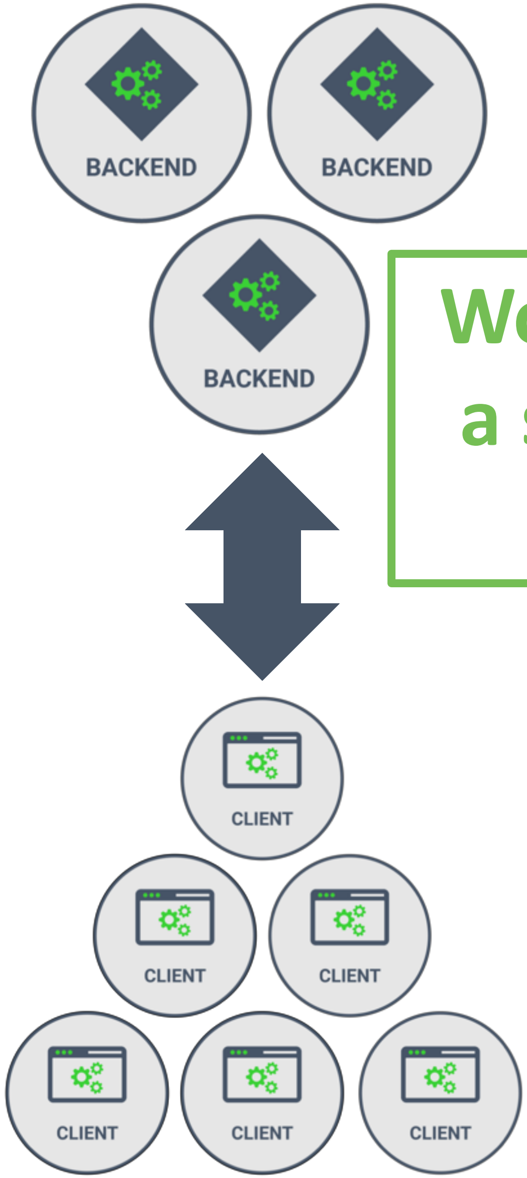
- Predictable identifiers enable the enumeration of resources
  - Dangerous if resources are not shielded by strict authorization checks
  - Many APIs only check authentication status, but not ***which*** user is authenticated
- The only proper mitigation is implementing proper authorization checks
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

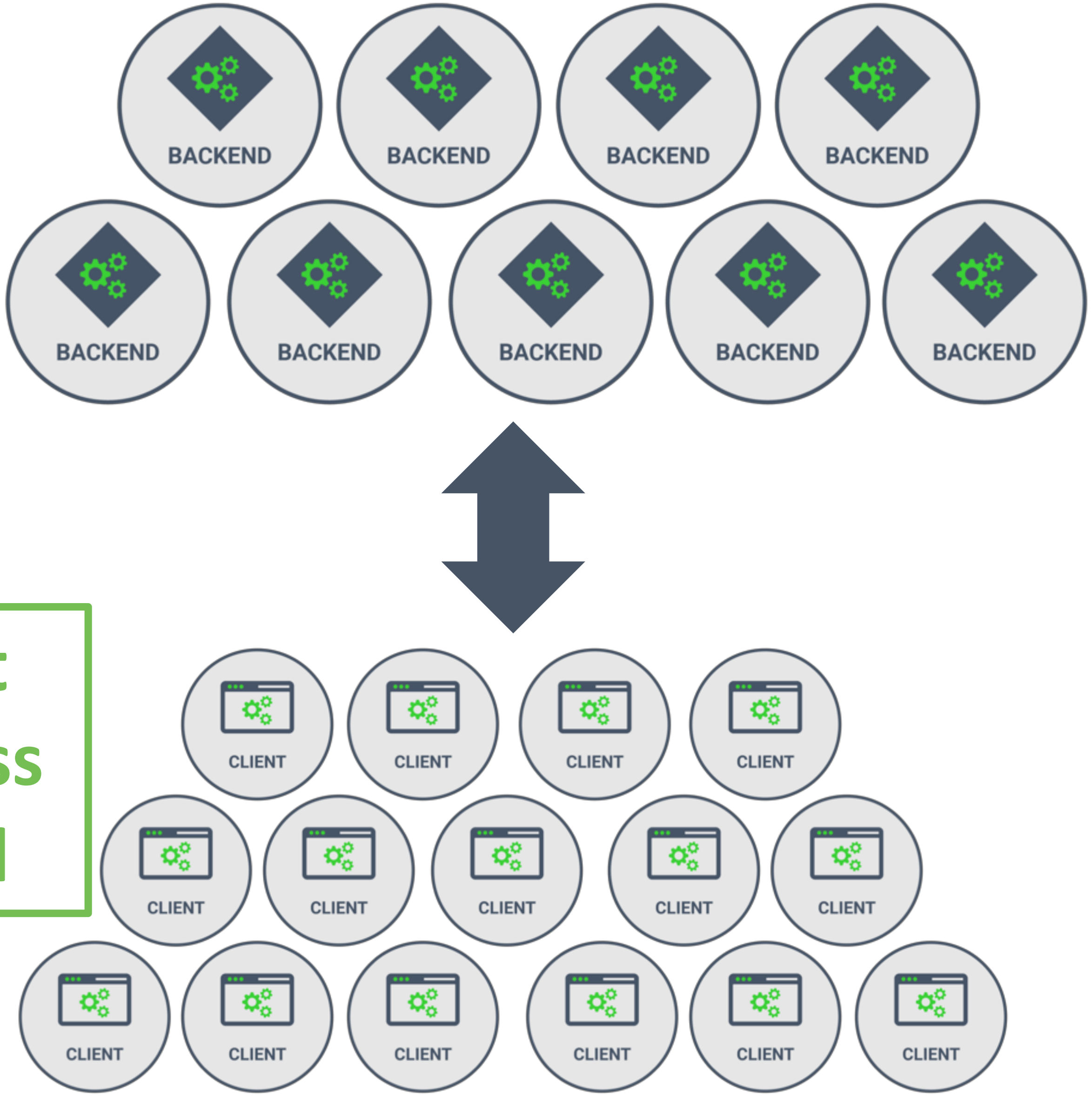


Works fine with a stateful ~~REST~~ backend



Works fine with a stateful ~~REST~~ backend

Might benefit from a stateless REST backend

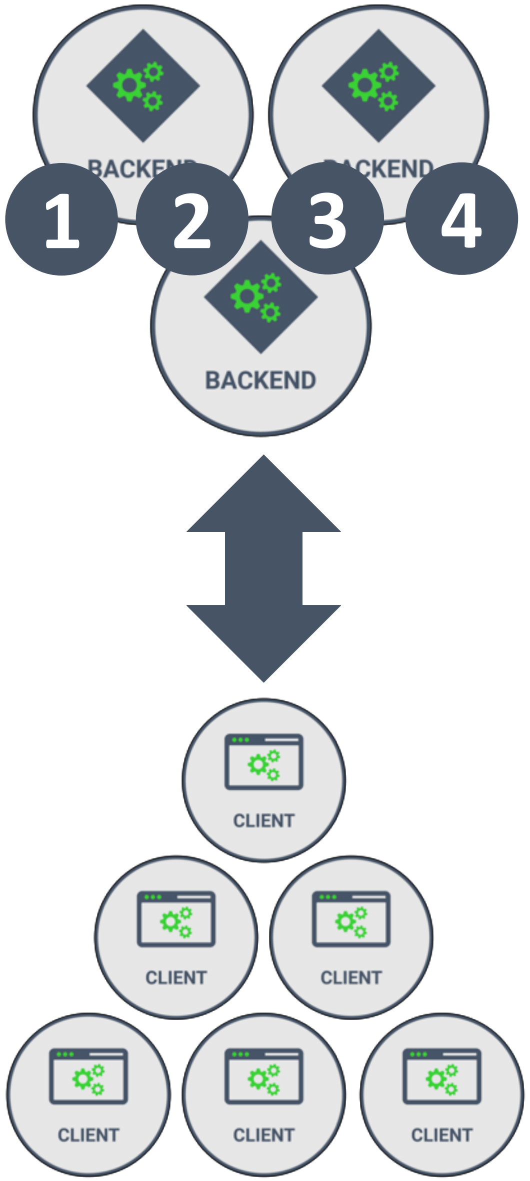


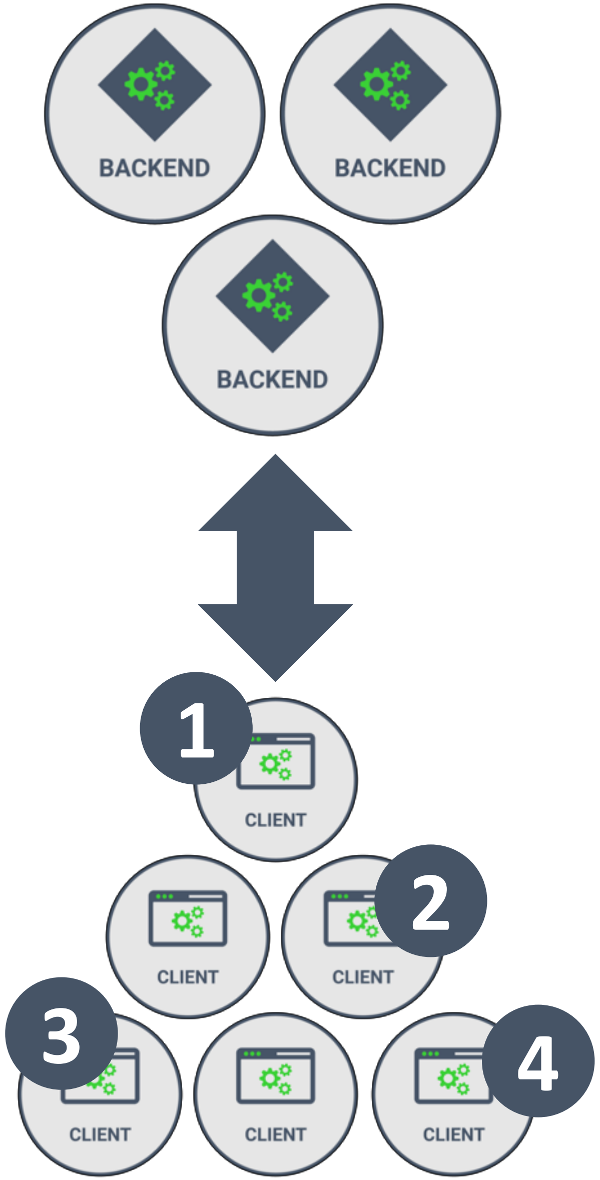
THE TRUTH IS A LOT MORE COMPLICATED

- 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

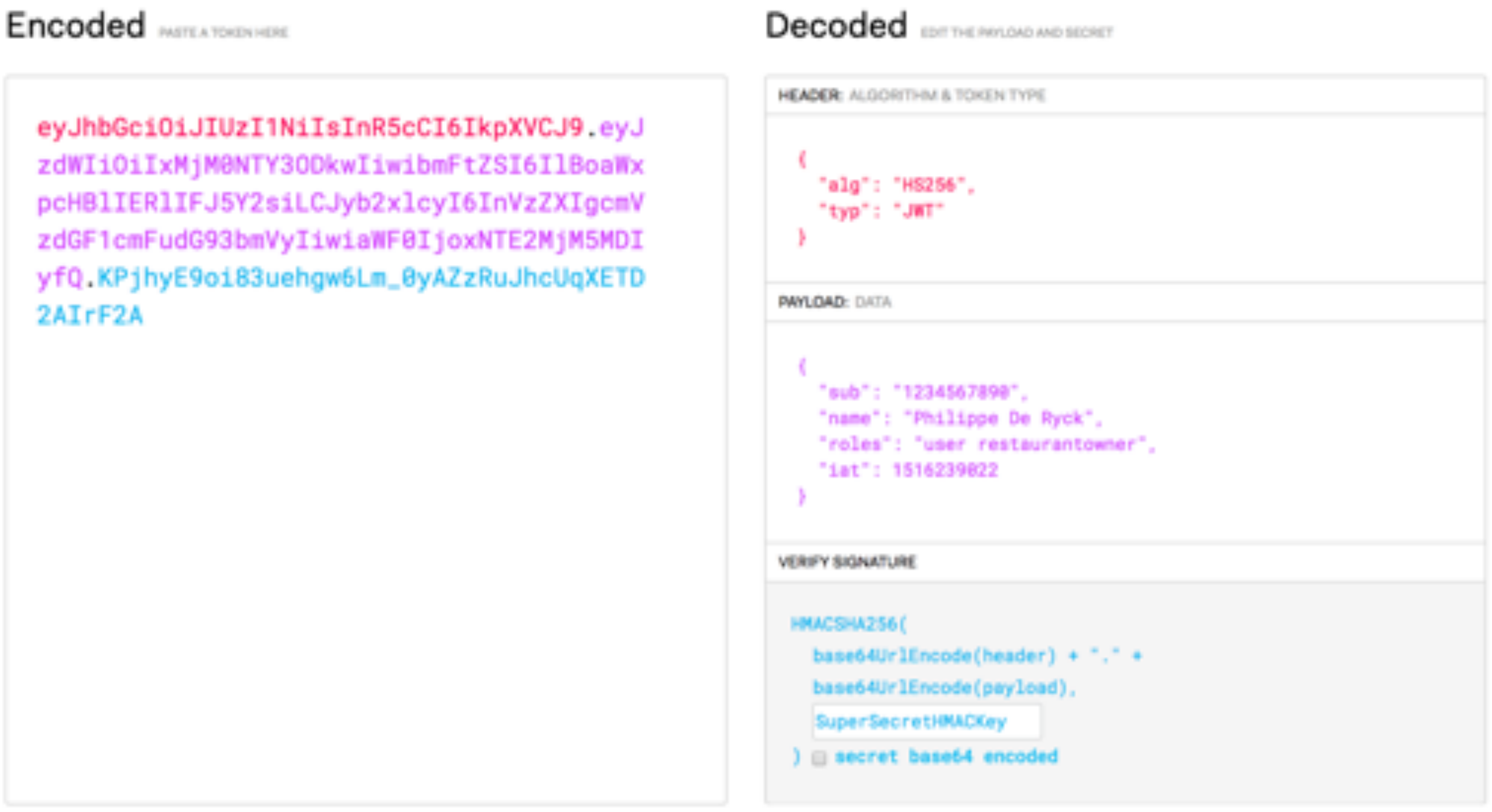
Server-side session data is not compatible with the REST paradigm, but still works well with small to medium-scale applications.


BACKEND
1
2
3
4
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.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
}
```

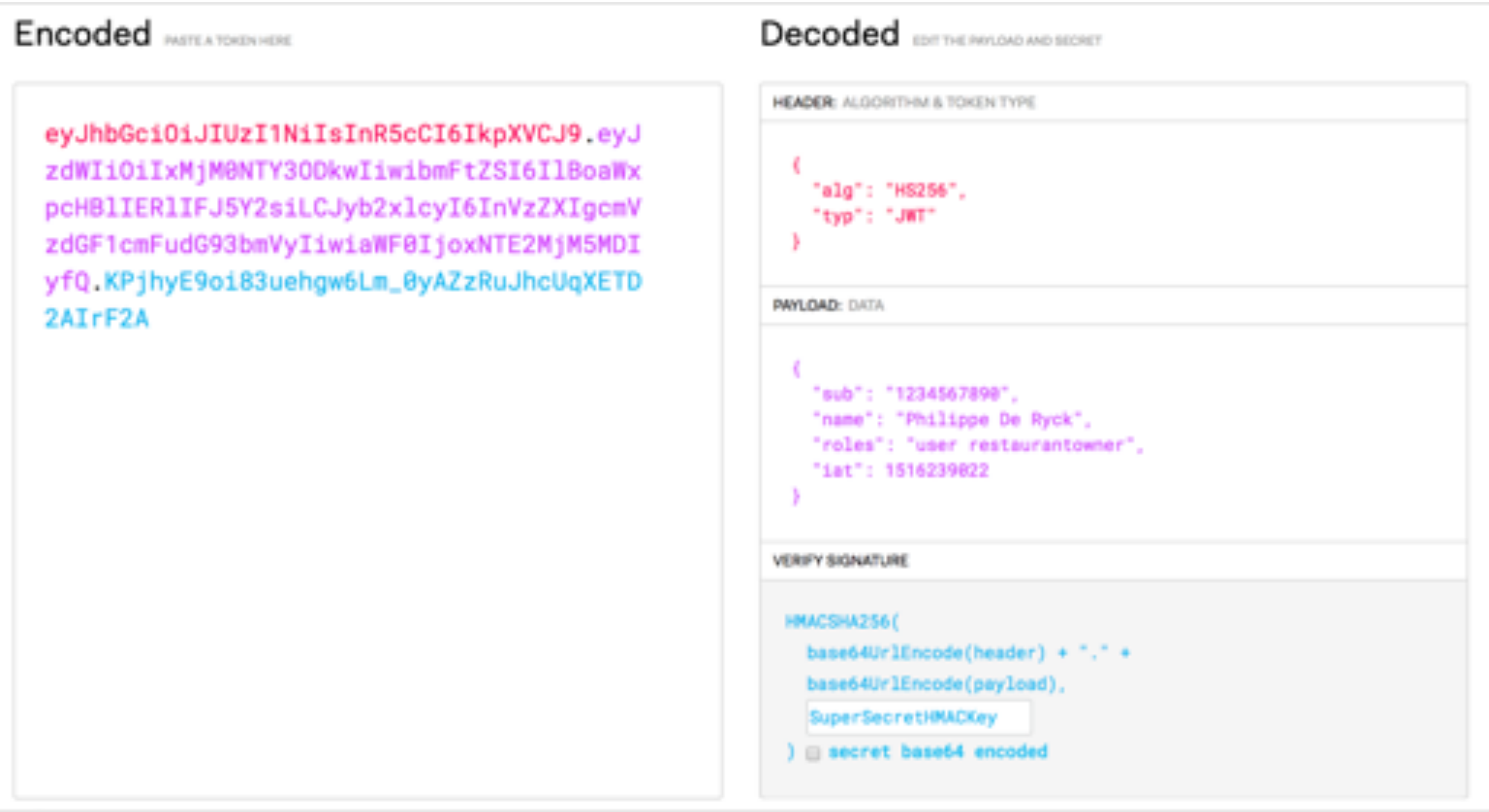
Decoding only

```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

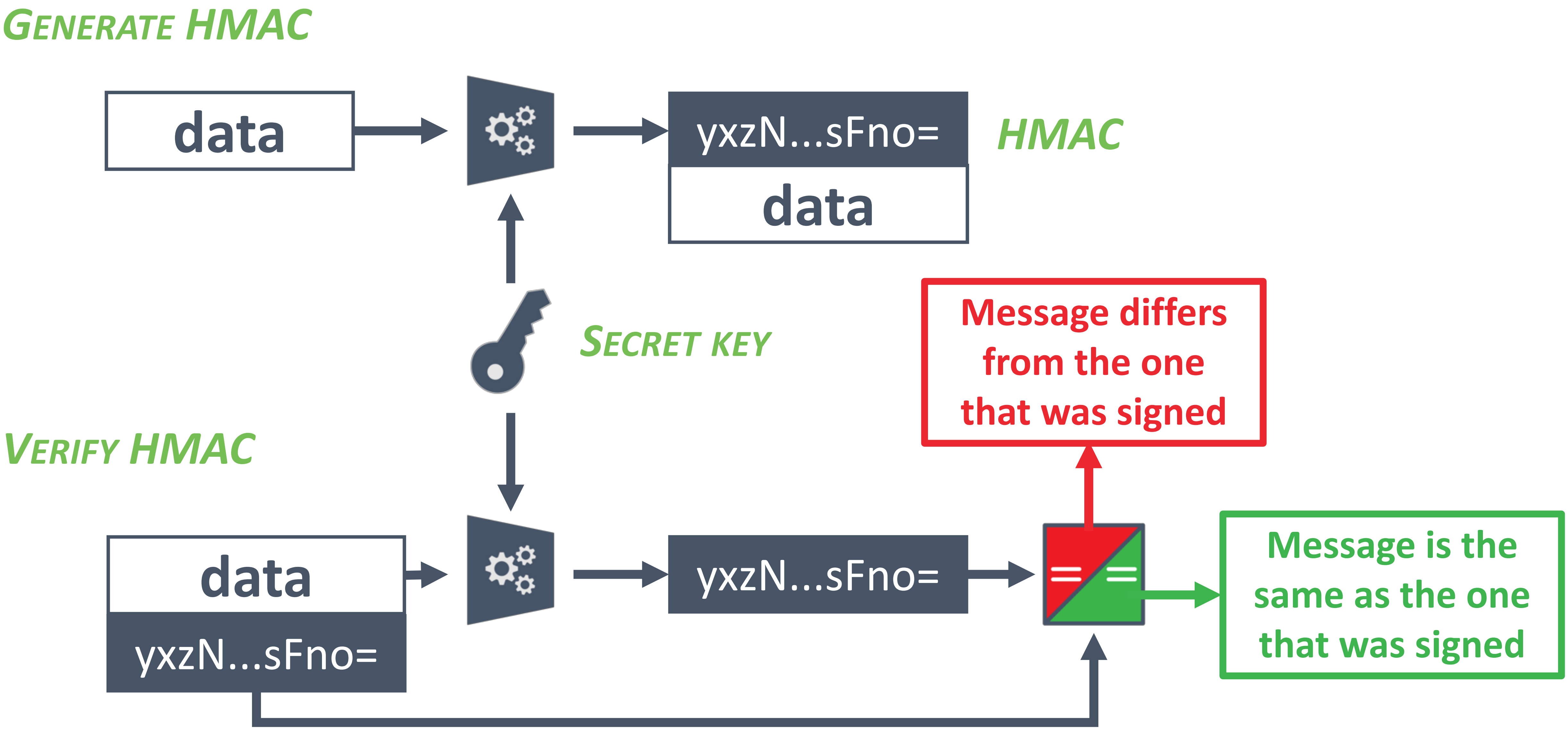
Mishandling client-side session data

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

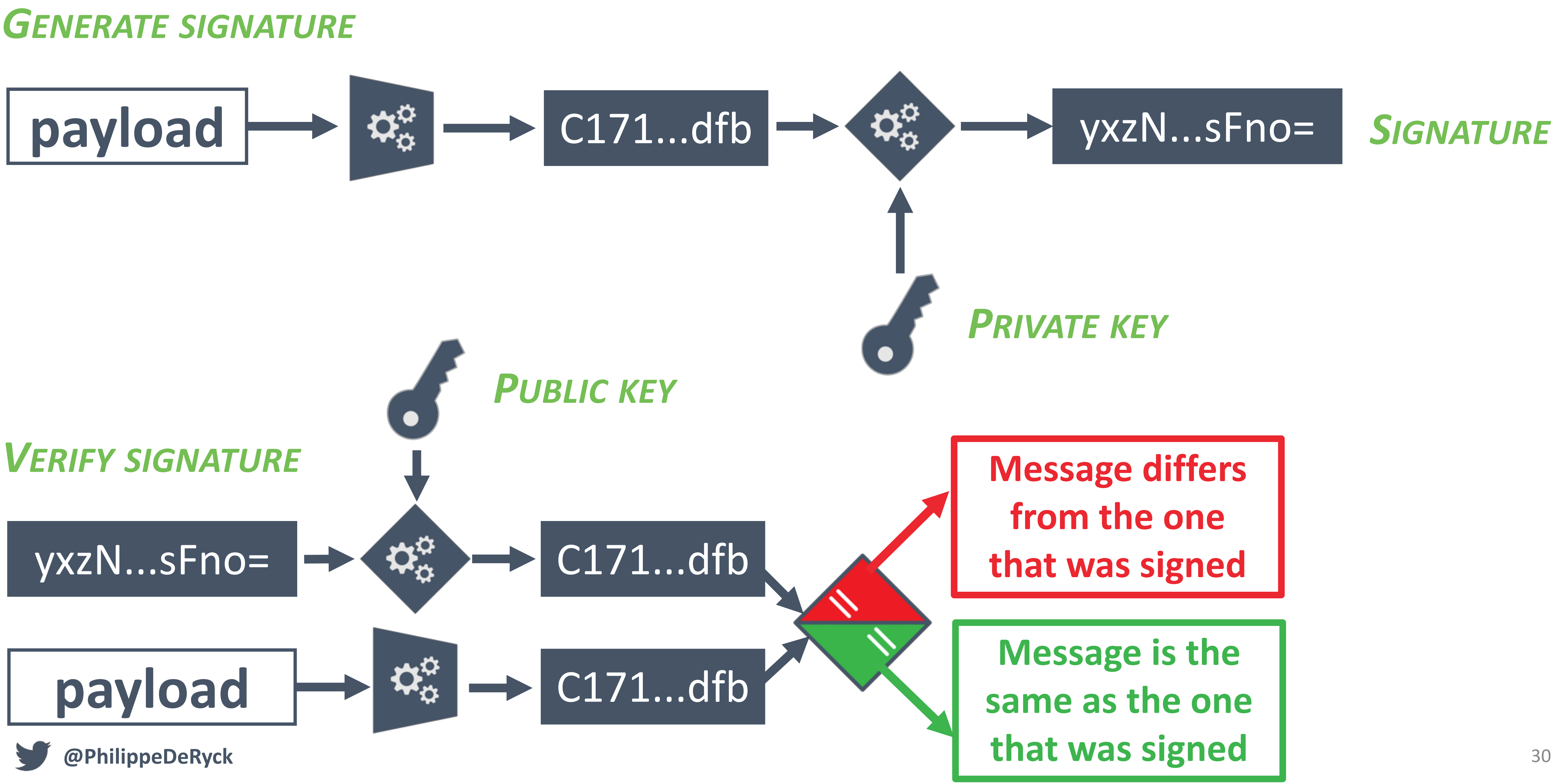


Encoded
PASTE A TOKEN HERE
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
zdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IlBoaWx
pcHBlIERlIFJ5Y2siLCJyb2xlcyI6InVzZXIgcmV
zdGF1cmFudG93bmVyIiwiaWF0IjoxNTE2MjM5MDI
yfQ.KPjhyE9oi83uehgw6Lm_0yAZzRuJhcUqXETD
2AIrF2A
Decoded
EDIT THE PAYLOAD AND SECRET
HEADER: ALGORITHM & TOKEN 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

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)

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"
}
```

```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();
```

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***

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...**

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	Only present on XHR calls, unless you add it through 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.**

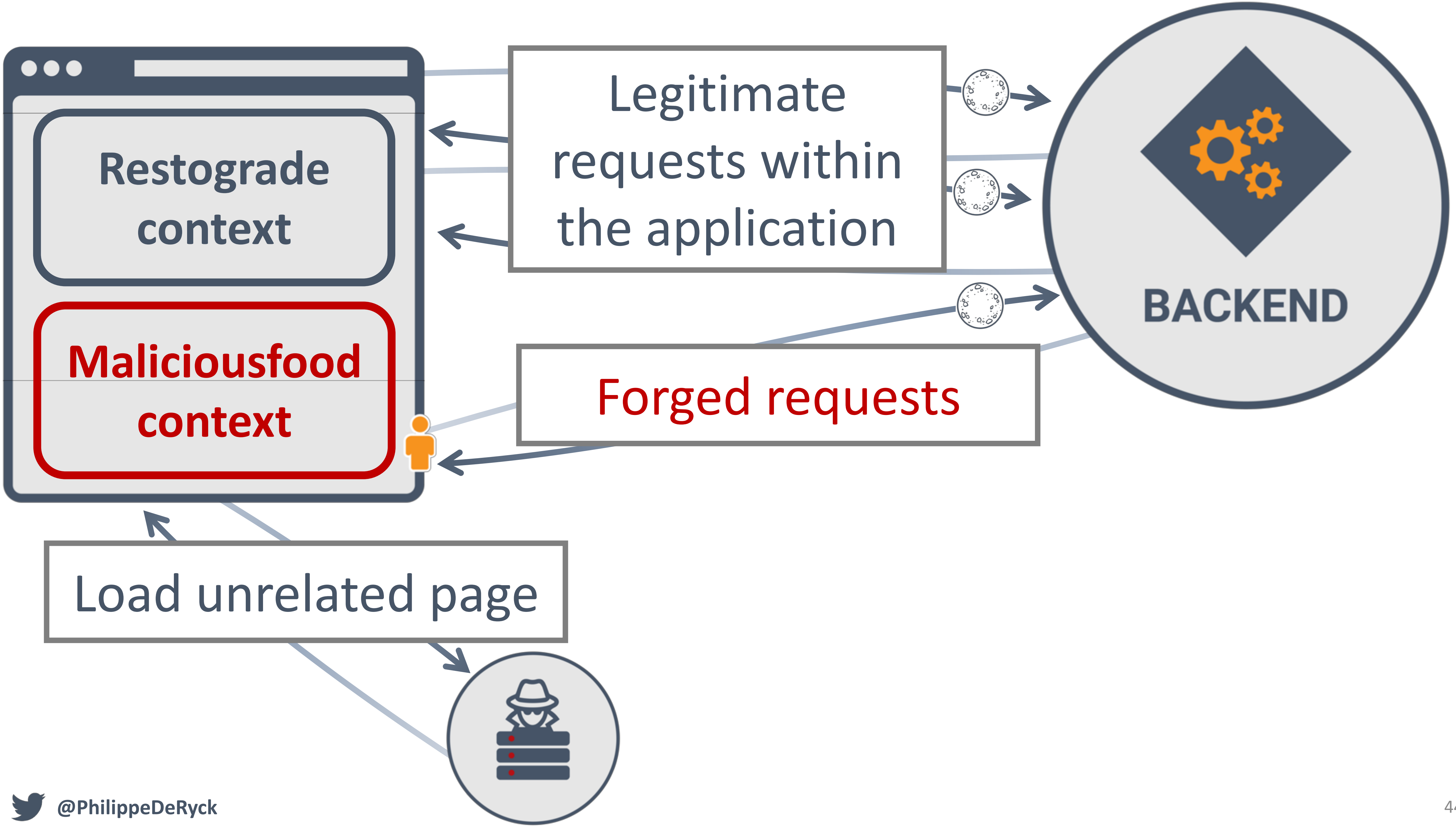
HttpOnly cookies

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


Restograde context
Maliciousfood context
Legitimate requests within the application
BACKEND
Forged requests
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

```
'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 proper destination whitelisting

CONFUSION ABOUT THE IMPACT OF XSS AND 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?

…

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