The reality of building secure APIs

Dr. Philippe De Ryck

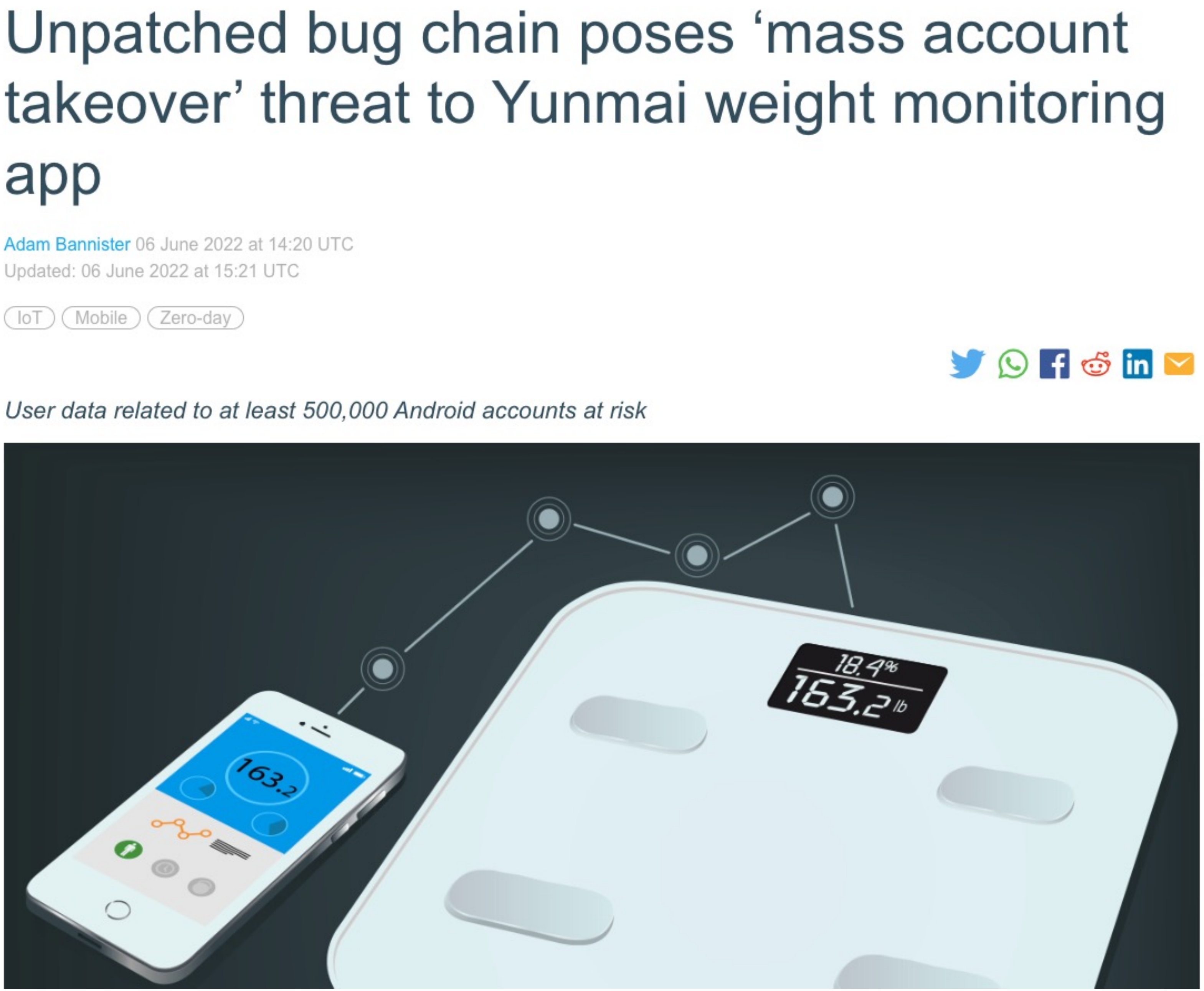
https://PragmaticWebSecurity.com

1	Broken object level authorization
2	Broken authentication
3	Broken object property-level authorization
4	Unrestricted resource consumption
5	Broken function level authorization
6	Unrestricted access to sensitive business flows
7	Server-side request forgery
8	Security misconfiguration
9	Improper inventory management
10	Unsafe consumption of APIs

OWASP

API Security

TOP10



Unpatched bug chain poses ‘mass account takeover’ threat to Yunmai weight monitoring app

Adam Bannister 06 June 2022 at 14:20 UTC
Updated: 06 June 2022 at 15:21 UTC

IoT Mobile Zero-day

User data related to at least 500,000 Android accounts at risk

https://portswigger.net/daily-swig/unpatched-bug-chain-poses-mass-account-takeover-threat-to-yunmai-weight-monitoring-app

Defending against SSRF attacks (with help from our bug bounty program)

// By Po-Ning Tseng • Sep 20, 2022

https://dropbox.tech/security/bug-bounty-program-ssrf-attack

I am *Dr. Philippe De Ryck*

Pragmatic Web Security
Security for developers

Founder of Pragmatic Web Security

Google Developers
Experts

Google Developer Expert

SecAppDev

SecAppDev organizer

I help developers with security

- ✓ Hands-on in-depth security training
- ✓ Advanced online security courses
- ✓ Security advisory services

https://pdr.online

Grab a copy of the slides ...

https://pragmaticwebsecurity.com/talks

/in/PhilippeDeRyck

https://infosec.exchange/@PhilippeDeRyck

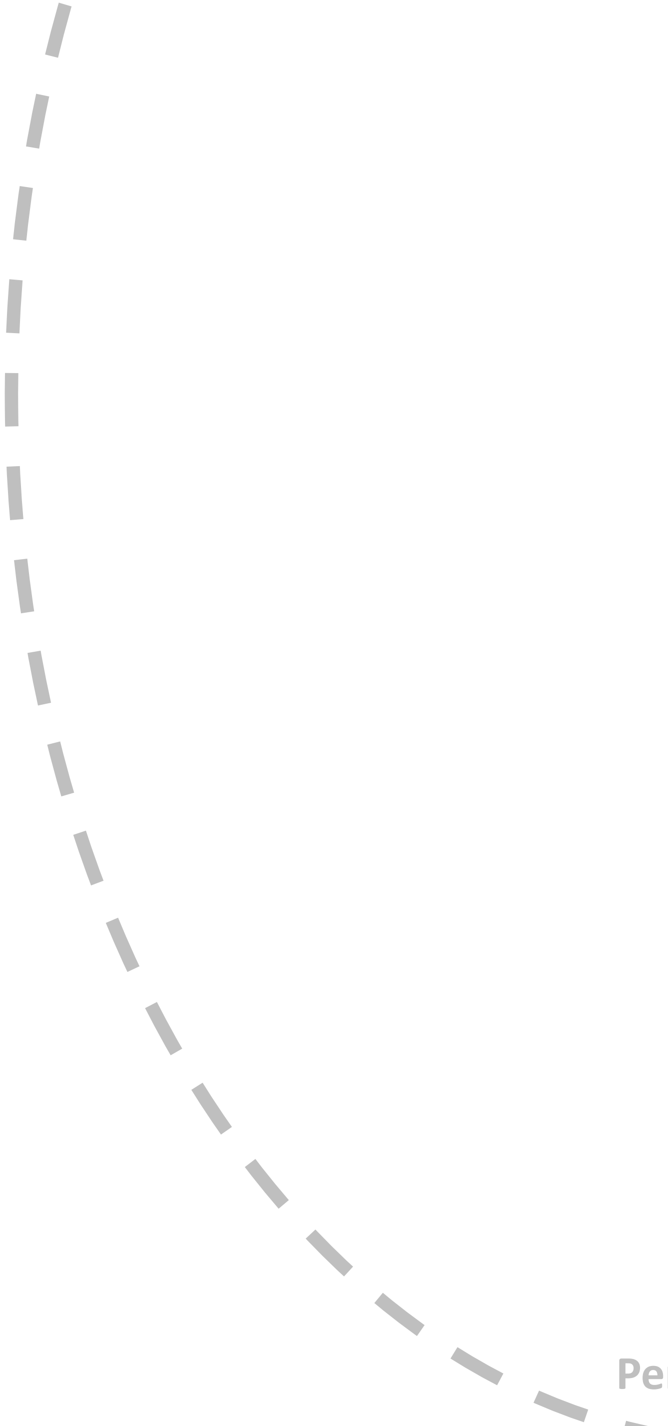
Website icons created by Uniconlabs - Flaticon

1	Broken object level authorization
2	Broken authentication
3	Broken object property-level authorization
4	Unrestricted resource consumption
5	Broken function level authorization
6	Unrestricted access to sensitive business flows
7	Server-side request forgery
8	Security misconfiguration
9	Improper inventory management
10	Unsafe consumption of APIs

OWASP

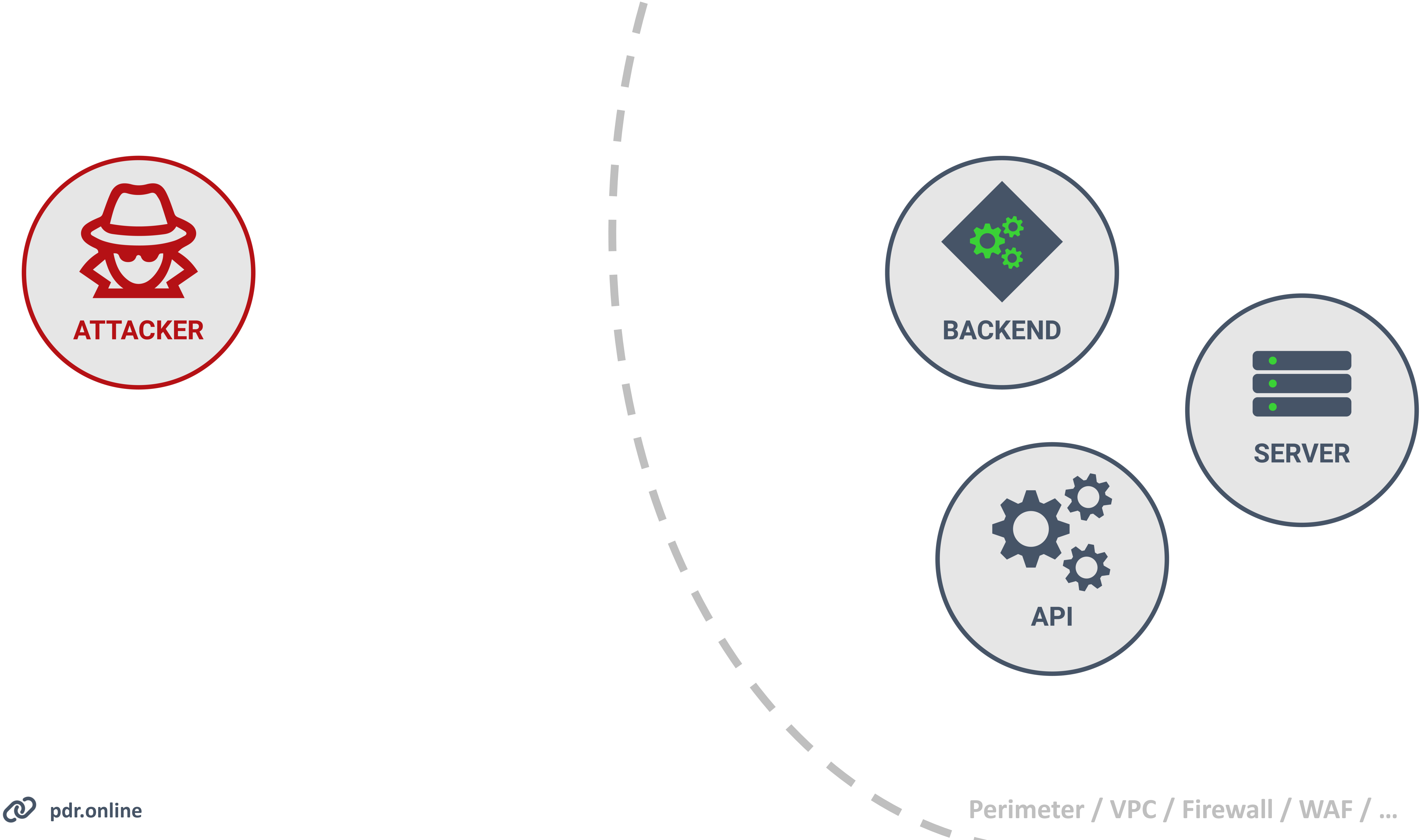
API Security

TOP10



Perimeter / VPC / Firewall / WAF / …

Photo by Robert V. Ruggiero on Unsplash
Photo by Thula Na on Unsplash



ATTACKER

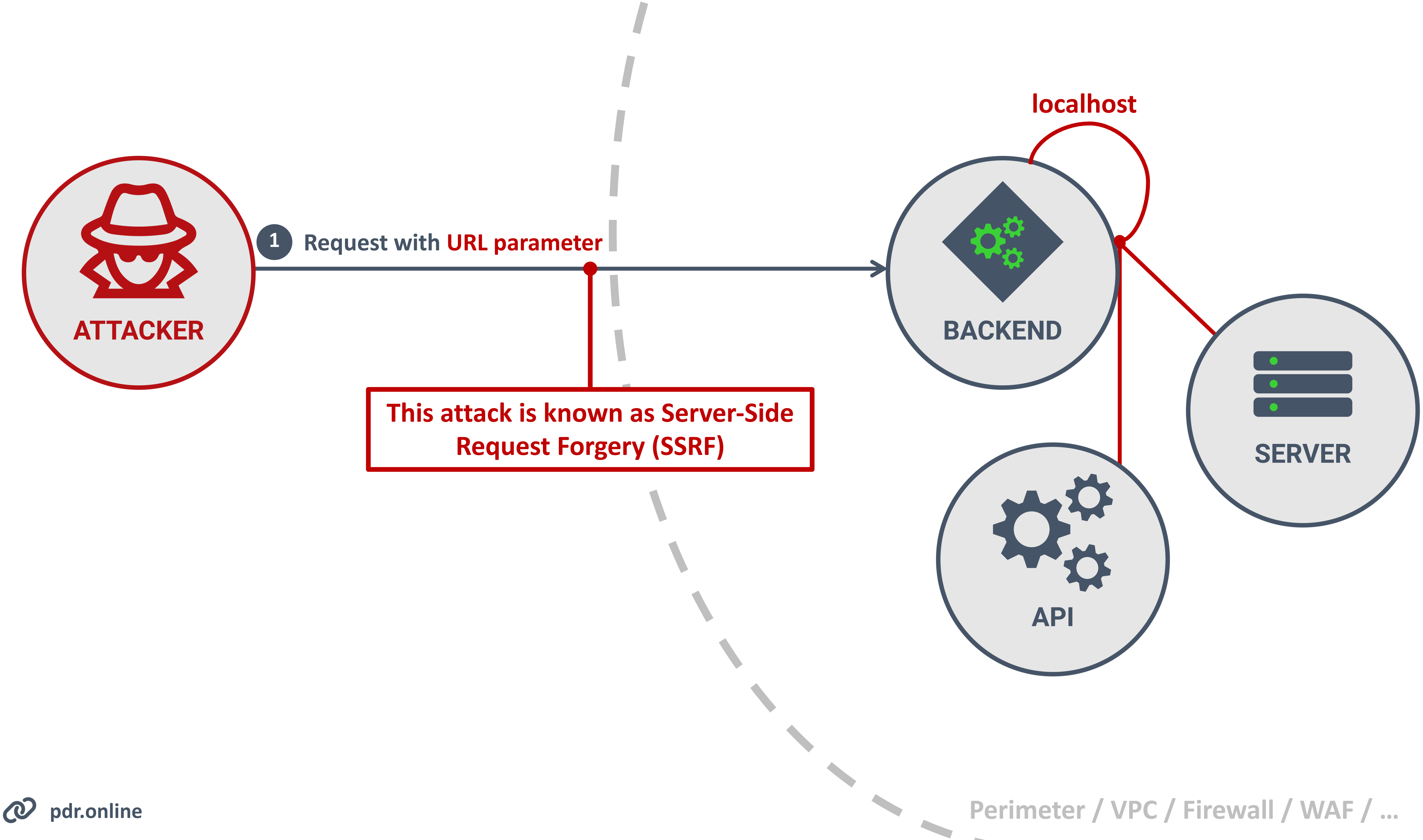
BACKEND

SERVER

API

Perimeter / VPC / Firewall / WAF / ...

Photo by Shalone Cason on Unsplash



ATTACKER

1 Request with URL parameter

This attack is known as Server-Side Request Forgery (SSRF)

localhost

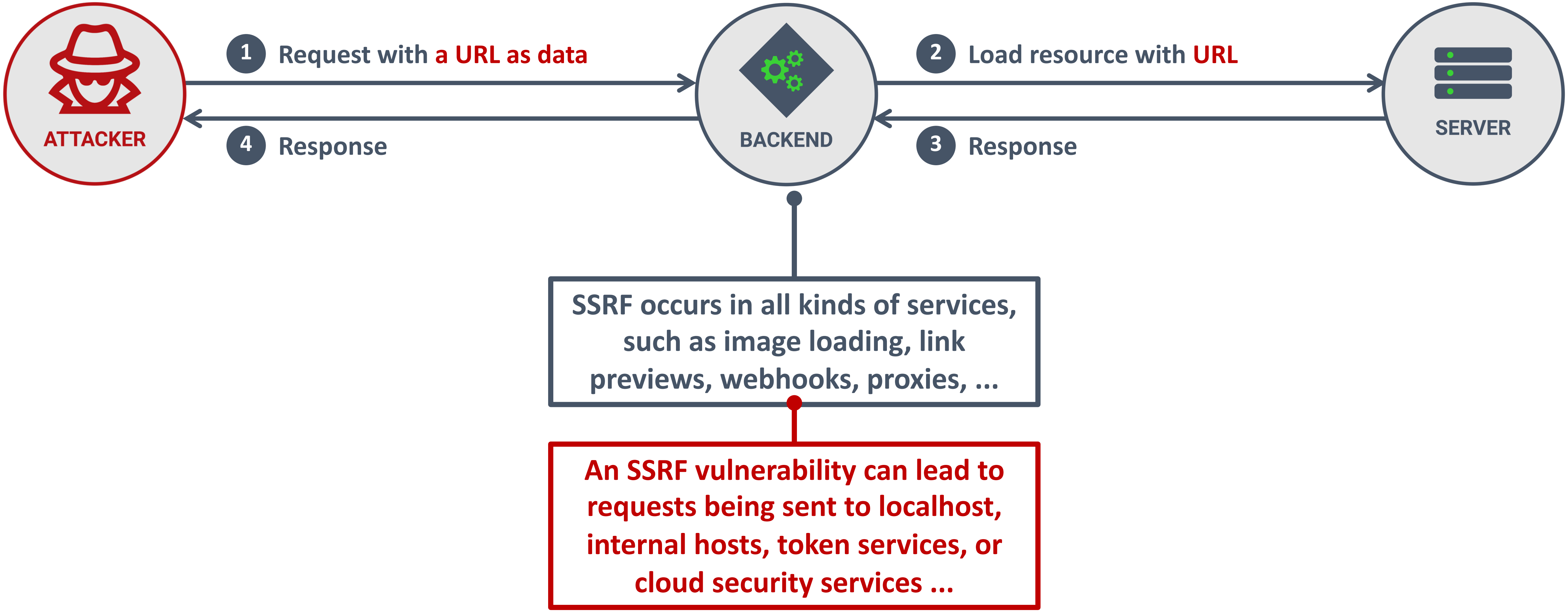
BACKEND

SERVER

API

Perimeter / VPC / Firewall / WAF / ...

How does SSRF happen?



ATTACKER

BACKEND

SERVER

1 Request with **a URL as data**

2 Load resource with **URL**

3 Response

4 Response

SSRF occurs in all kinds of services, such as image loading, link previews, webhooks, proxies, ...

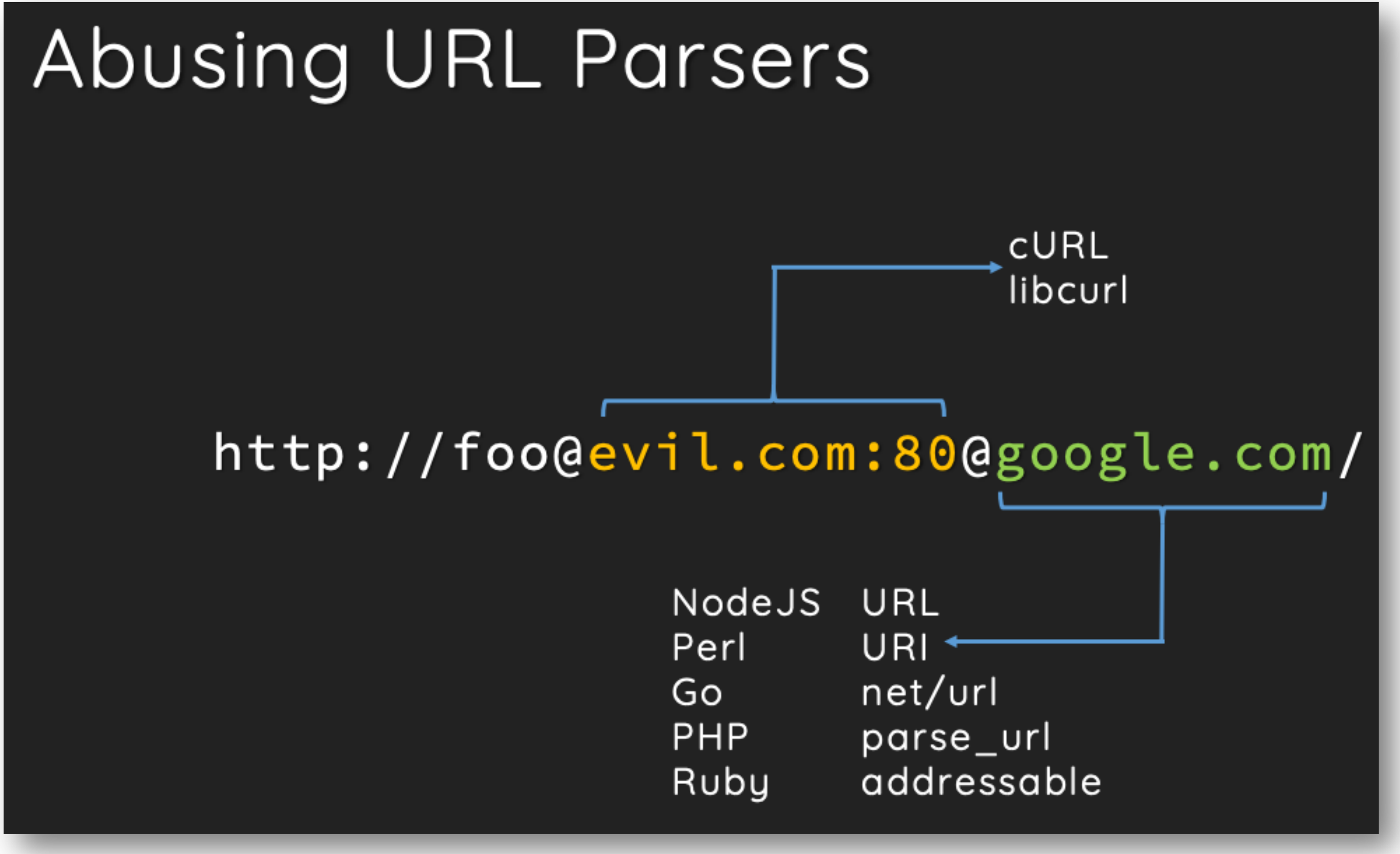
An SSRF vulnerability can lead to requests being sent to localhost, internal hosts, token services, or cloud security services ...

```
https://mycdn.example.com/image.png
```

Valid input

```
https://127.0.0.1:8080
```

Invalid input



Abusing URL Parsers

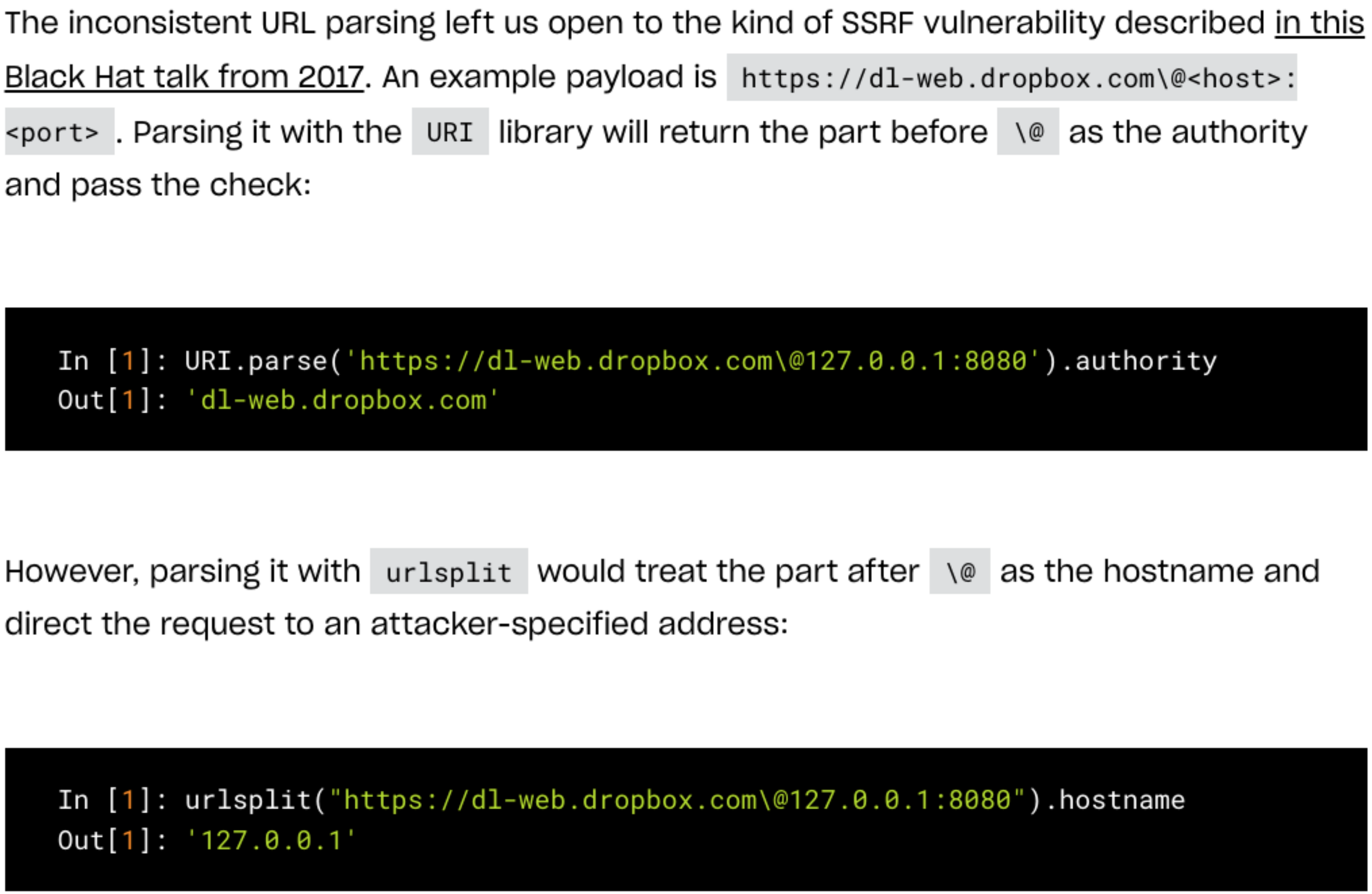
```
http://foo@evil.com:80@google.com/
```

cURL
libcurl

NodeJS	URL
Perl	URI
Go	net/url
PHP	parse_url
Ruby	addressable

https://cheatsheetseries.owasp.org/assets/Server_Side_Request_Forgery_Prevention_Cheat_Sheet_Orange_Tsai_Talk.pdf

SSRF at Dropbox



The inconsistent URL parsing left us open to the kind of SSRF vulnerability described in this Black Hat talk from 2017. An example payload is `https://dl-web.dropbox.com\@<host>:<port>`. Parsing it with the `URI` library will return the part before `\@` as the authority and pass the check:

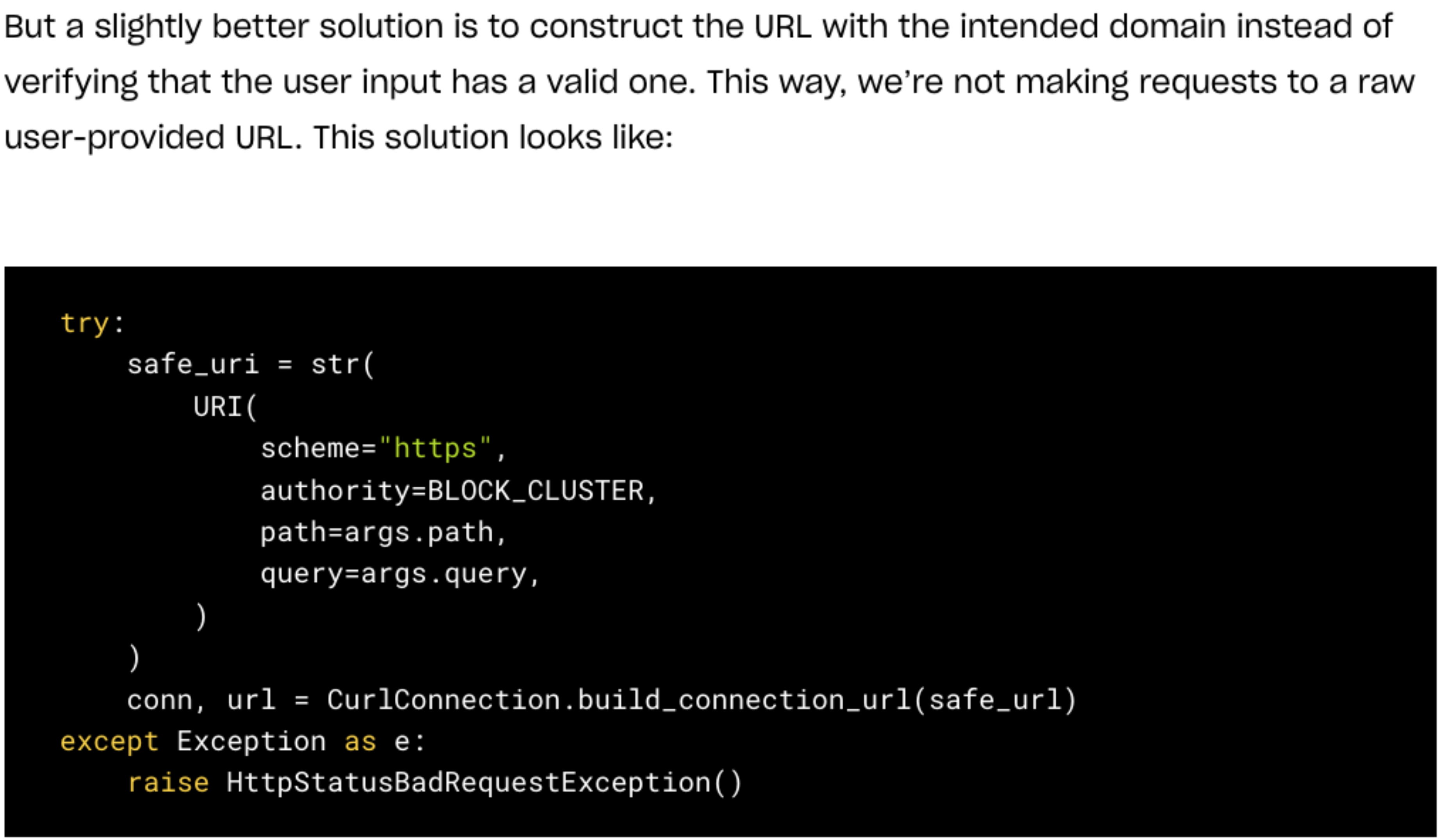
```
In [1]: URI.parse('https://dl-web.dropbox.com\@127.0.0.1:8080').authority
Out[1]: 'dl-web.dropbox.com'
```

However, parsing it with `urlsplit` would treat the part after `\@` as the hostname and direct the request to an attacker-specified address:

```
In [1]: urlsplit("https://dl-web.dropbox.com\@127.0.0.1:8080").hostname
Out[1]: '127.0.0.1'
```

https://dropbox.tech/security/bug-bounty-program-ssrf-attack

Fixing SSRF at Dropbox



But a slightly better solution is to construct the URL with the intended domain instead of verifying that the user input has a valid one. This way, we're not making requests to a raw user-provided URL. This solution looks like:

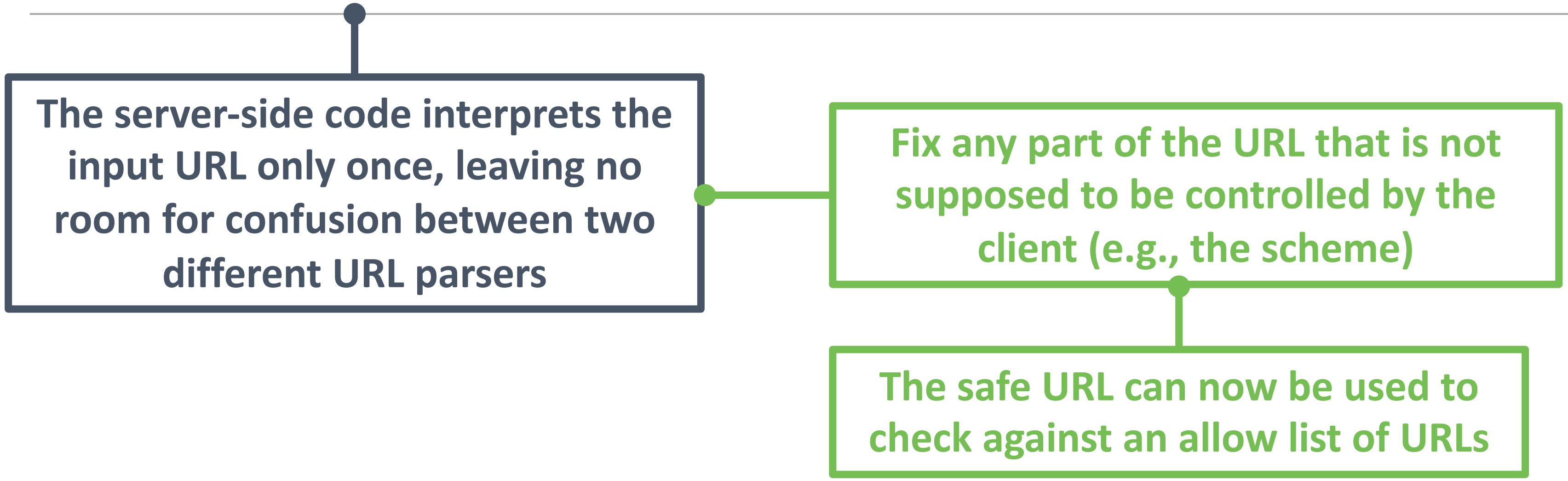
```
try:
    safe_uri = str(
        URI(
            scheme="https",
            authority=BLOCK_CLUSTER,
            path=args.path,
            query=args.query,
        )
    )
    conn, url = CurlConnection.build_connection_url(safe_url)
except Exception as e:
    raise HttpStatusBadRequestException()
```

https://dropbox.tech/security/bug-bounty-program-ssrf-attack

Fixing SSRF by removing ambiguity on the server

Accept a URL as input on the server and immediately transform it into a unambiguous value

```javascript
const input = req.body.url
// Transform the input into a single representation
const u = new URL(input)
// Construct a unambiguous safe URL to use in the application
const safeUrl = new URL(`https://${u.host}${u.pathname}${u.search}`)
```



The server-side code interprets the input URL only once, leaving no room for confusion between two different URL parsers

Fix any part of the URL that is not supposed to be controlled by the client (e.g., the scheme)

The safe URL can now be used to check against an allow list of URLs

FIXING SSRF BY REMOVING AMBIGUITY ON THE CLIENT

The code handling the URL input

```
function saveUrl() {
  let strUrl = document.getElementById("cb").value;
  let url = new URL(strUrl);

  let urlData = {
    "scheme": url.protocol,
    "hostname": url.hostname,
    "port": url.port,
    "path": url.pathname,
    "params": url.search
  }

  // Send this data to the backend for processing
}
```

The client accepts a full URL in the UI, and then parses it in the browser before sending it to the backend

The data received by the API

```
{
  "scheme":"https:",
  "hostname":"restograde.com",
  "port":"",
  "path":"/callback",
  "params":"",
}
```

The server-side code only accepts a decomposed URL, which enables *strict input validation* on each component, which automatically removes all ambiguity

Protect against SSRF

SSRF vulnerabilities often occur when there's ambiguity in matching against allow-lists.

Ensure the data used for server-side requests is unambiguous and trustworthy according to your security policy.

1	Broken object level authorization
2	Broken authentication
3	Broken object property-level authorization
4	Unrestricted resource consumption
5	Broken function level authorization
6	Unrestricted access to sensitive business flows
7	Server-side request forgery
8	Security misconfiguration
9	Improper inventory management
10	Unsafe consumption of APIs

OWASP

API Security

TOP10

MASS ACCOUNT TAKEOVER IN THE YUNMAI SMART SCALE API

https://fortbridge.co.uk/research/mass-account-takeover-yunmai/

Account takeover through ‘forgot password’ functionality.

The victim will get an email with a unique 6 digit code that allows to reset the password.

Breaking authentication

Avoid leaking information

APIs often (unknowingly) leak information that enables attacks such as username enumeration.

Carefully analyze your APIs for explicit and implicit data leakage.

Implement rate limiting

Many endpoints fail to implement rate limiting, which allows attackers to launch brute force attacks. Examples include SMS code prompts, reset tokens, and authentication forms.

Implement rate limiting to minimize the attacker's ability to abuse these endpoints.

Mitigate guessing attacks

Attackers often abuse unsigned values to implement guessing attacks.

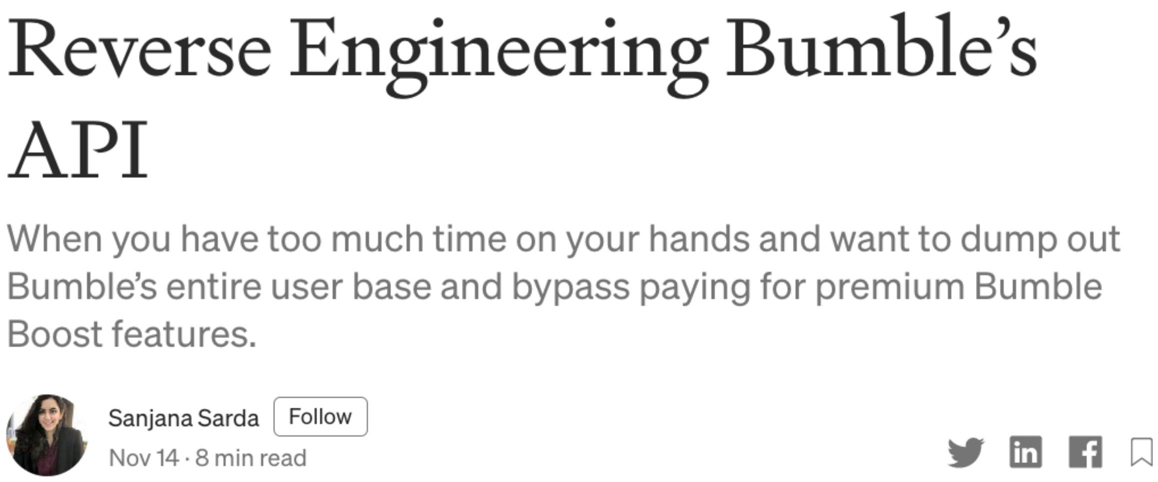
Mitigation techniques against guessing attacks include using long random identifiers (e.g., a UUID) or using signed values that allow the detection of tampering.

1	Broken object level authorization
2	Broken authentication
3	Broken object property-level authorization
4	Unrestricted resource consumption
5	Broken function level authorization
6	Unrestricted access to sensitive business flows
7	Server-side request forgery
8	Security misconfiguration
9	Improper inventory management
10	Unsafe consumption of APIs

OWASP

API Security

TOP10



Reverse Engineering Bumble's API

When you have too much time on your hands and want to dump out Bumble's entire user base and bypass paying for premium Bumble Boost features.

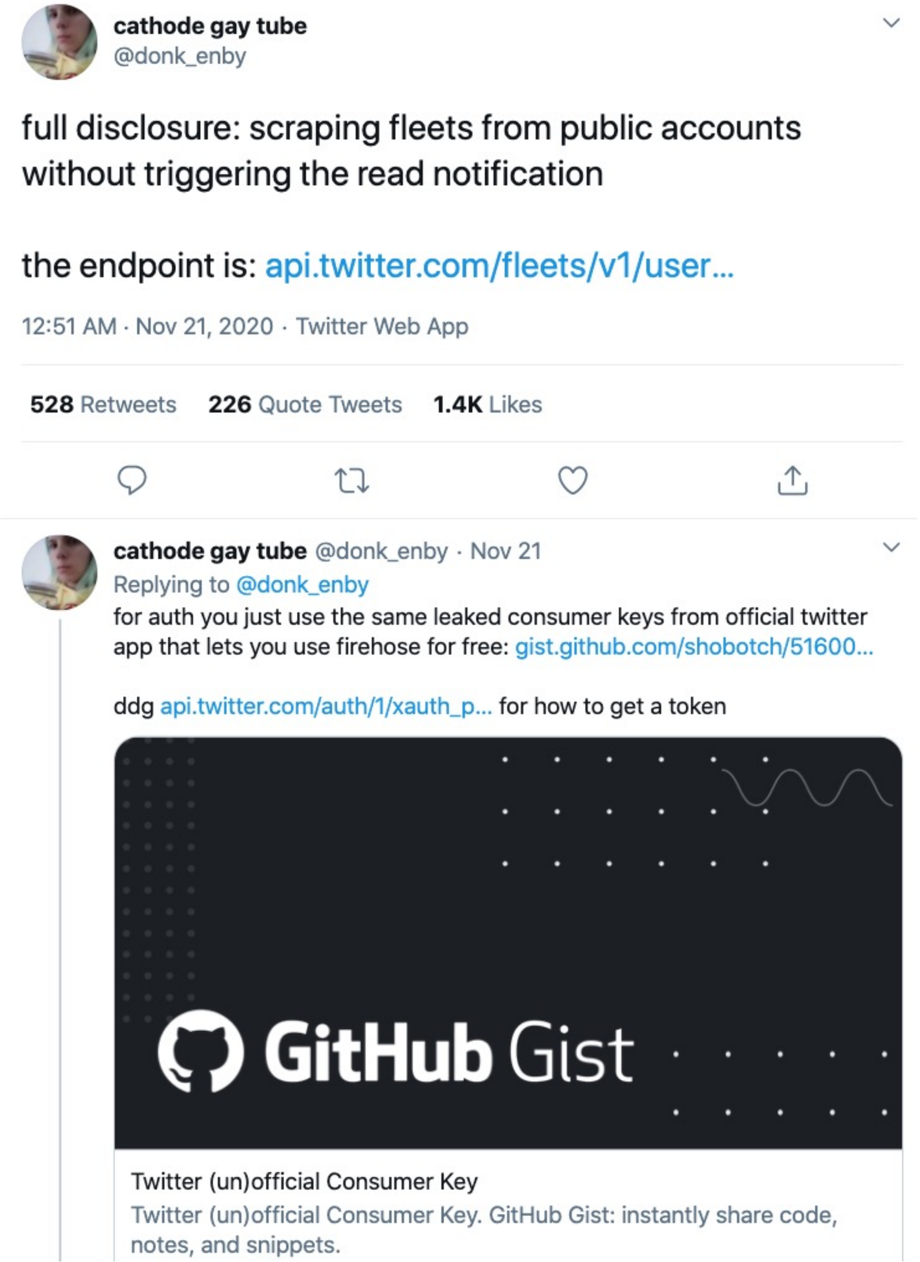
Sanjana Sarda Follow

Nov 14 · 8 min read

> “Our accounts eventually got locked and hidden for more verification requirements. We tested retrieving user data while our account was locked, and it still worked.”

https://blog.securityevaluators.com/
reverse-engineering-bumbles-api-a2a0d39b3a87

It's possible to view deleted fleets via Twitter's API endpoint, to view existing fleets without giving the poster a read notification and you can do both without being logged into Twitter.



cathode gay tube
@donk_enby

full disclosure: scraping fleets from public accounts without triggering the read notification

the endpoint is: api.twitter.com/fleets/v1/user...

12:51 AM · Nov 21, 2020 · Twitter Web App

528 Retweets **226** Quote Tweets **1.4K** Likes

cathode gay tube @donk_enby · Nov 21
Replying to @donk_enby
for auth you just use the same leaked consumer keys from official twitter app that lets you use firehose for free: gist.github.com/shobotch/51600...

ddg api.twitter.com/auth/1/xauth_p... for how to get a token

Twitter (un)official Consumer Key
Twitter (un)official Consumer Key. GitHub Gist: instantly share code, notes, and snippets.

https://twitter.com/donk_enby/status/1329935540049817600

THE CLIENT IS IRRELEVANT FOR SECURITY

The attack surface of an API
consists of all accessible endpoints,
regardless of how and if they are used by the client

Are client-side controls and validation procedures really useless?

The client is irrelevant to enforce security

Strict security controls on the client make your API security controls an effective detection mechanism for malicious behavior

1	Broken object level authorization
2	Broken authentication
3	Broken object property-level authorization
4	Unrestricted resource consumption
5	Broken function level authorization
6	Unrestricted access to sensitive business flows
7	Server-side request forgery
8	Security misconfiguration
9	Improper inventory management
10	Unsafe consumption of APIs

OWASP

API Security

TOP 10

Explicitly checking for conditions that are not allowed is a bad practice

```java
public void editRestaurant(long id, Restaurant restaurant) {
  if(user.hasRole("FOH_MANAGER") || user.hasRole("EXEC_CHEF")) {
    throw new AuthorizationException(":(");
  }

  restaurantData.update(id, restaurant);
};
```

Authorization policies should start from a "deny-by-default" position to avoid bypasses. Missing a role in this list will cause an authorization bypass.

Denying everything except the expected scenario is a best practice

```java
public void editRestaurant(long id, Restaurant restaurant) {
  if(user.hasRole("OWNER") || user.hasRole("GENERAL_MANAGER")) {
    restaurantData.update(id, restaurant);
  }

  throw new AuthorizationException(":(");
};
```

A mistake in a "deny-by-default" policy will cause a functional problem, but never an authorization bypass

Automatically enforcing role-based access control on endpoints in Spring

```java
@PreAuthorize("hasRole('OWNER') or hasRole('GENERAL_MANAGER')")
public void editRestaurant(long id, Restaurant restaurant) {
  restaurantData.update(id, restaurant);
};
```

Role-based access control (RBAC) is very intuitive and widely used, as evidenced by framework-level support for enforcing authorization.

Automatically enforcing role-based access control on endpoints in Spring

```
@PreAuthorize("hasRole('OWNER')
            or hasRole('GENERAL_MANAGER')
            or hasRole('CONTENT_MODERATOR')
            or hasRole('ADMIN')")
public void editRestaurant(long id, Restaurant restaurant) {
  restaurantData.update(id, restaurant);
};
```

RBAC is hard to manage and maintain, and often leads to a problem known as *role explosion*

Permission-based security decouples the code from the authorization policy

```java
@PreAuthorize("hasPermission('EDIT_RESTAURANT')")
public void editRestaurant(long id, Restaurant restaurant) {
  restaurantData.update(id, restaurant);
};
```

Using permissions decouples the authorization policy from the implementation.

Auditing this code becomes straightforward and does not require specific knowledge of the authorization policy.

Roles	Permissions		
	VIEW_RESTAURANT	EDIT_RESTAURANT	DELETE_RESTAURANT
FOH_MANAGER	✓	✗	✗
OWNER	✓	✓	✓
GENERAL_MANAGER	✓	✓	✗
CONTENT_MODERATOR	✓	✓	✗
ADMIN	✓	✓	✓

Hmm, that wasn't so hard?

Permission-based security decouples the code from the authorization policy

```
@PreAuthorize("hasPermission('EDIT_RESTAURANT')")
public void editRestaurant(long id, Restaurant restaurant) {
  restaurantData.update(id, restaurant);
};
```

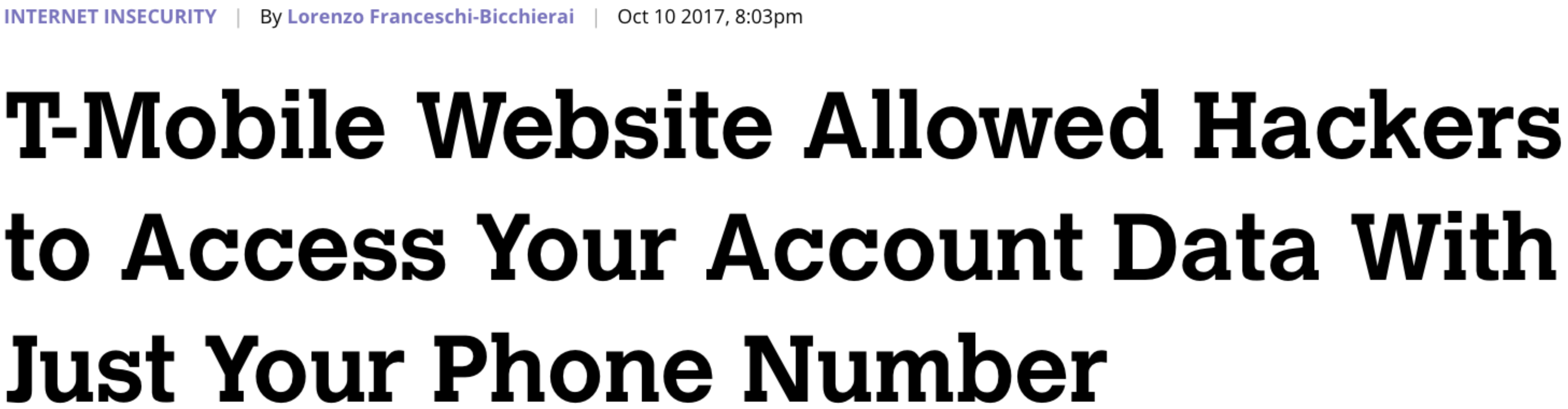
Editing a restaurant

```
PATCH /restaurants/1 HTTP/1.1
{ name: … }
```

Editing a restaurant

```
PATCH /restaurants/2 HTTP/1.1
{ name: … }
```

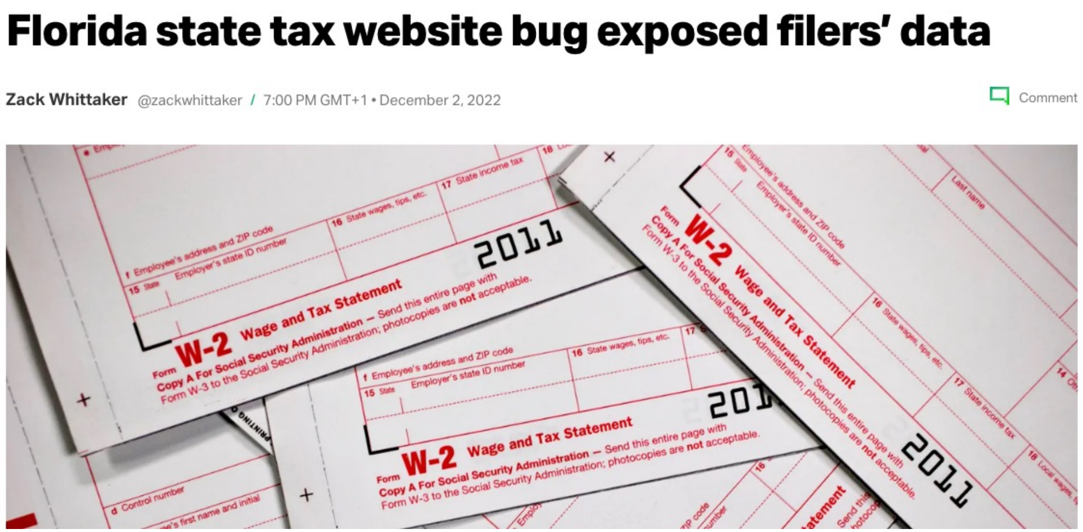
Without object-level access control, a user with the *edit* permission can change any restaurant



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.”



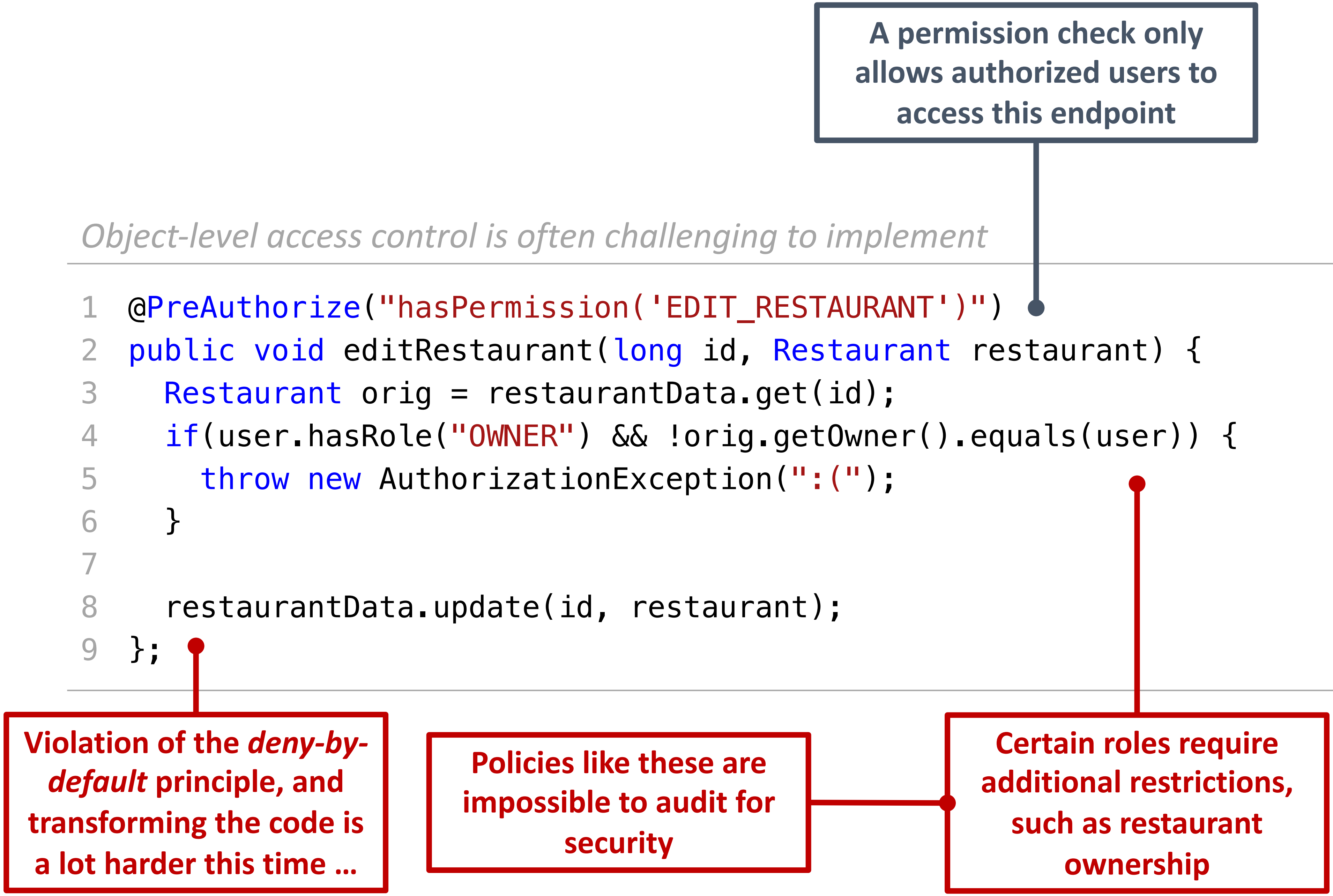
Florida state tax website bug exposed filers' data

Zack Whittaker @zackwhittaker / 7:00 PM GMT+1 • December 2, 2022

Comment

https://techcrunch.com/2022/12/02/florida-tax-bug-data-exposed/

> **the security flaw allowed anyone who was logged in to access, modify and delete the personal data of business owners by modifying the part of the web address that contains the taxpayers' application number.**



A permission check only allows authorized users to access this endpoint

Object-level access control is often challenging to implement

```
@PreAuthorize("hasPermission('EDIT_RESTAURANT')")
public void editRestaurant(long id, Restaurant restaurant) {
  Restaurant orig = restaurantData.get(id);
  if(user.hasRole("OWNER") && !orig.getOwner().equals(user)) {
    throw new AuthorizationException(":(");
  }

  restaurantData.update(id, restaurant);
};
```

Violation of the *deny-by-default* principle, and transforming the code is a lot harder this time …

Policies like these are impossible to audit for security

Certain roles require additional restrictions, such as restaurant ownership

BOLA in practice

A "centralized" policy results in a clear and auditable authorization policy

```
public void editRestaurant(long id, Restaurant restaurant) {
  Restaurant orig = restaurantData.get(id);
  if(policy.canEditRestaurant(user, orig)) {
    restaurantData.update(id, restaurant);
  }
  else {
    throw new AuthorizationException(":(");
  }
}
```

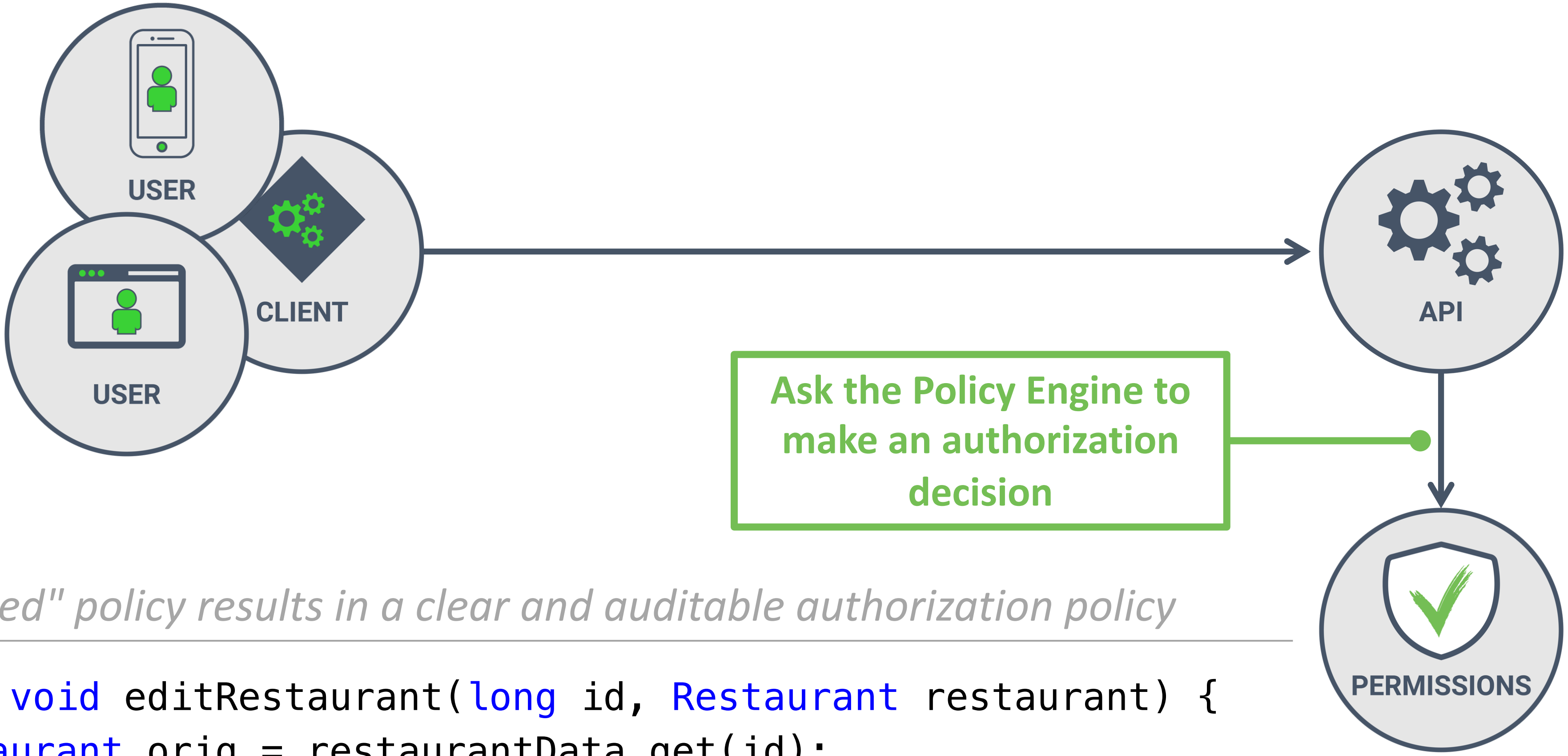
The code is easy to audit, without specific knowledge of the detailed authorization policy

Centralizing the authorization logic encapsulates complexity in a single location

```
public static boolean canEditRestaurant(User u, Restaurant r) {
  // Admins are always allowed
  if(isAdmin(u)) return true;
  // Owners can only update their own restaurant
  if(r.getOwner().equals(u)) return true;

  return false;
}
```

A single place to implement complex authorization logic makes things more manageable and easy to control

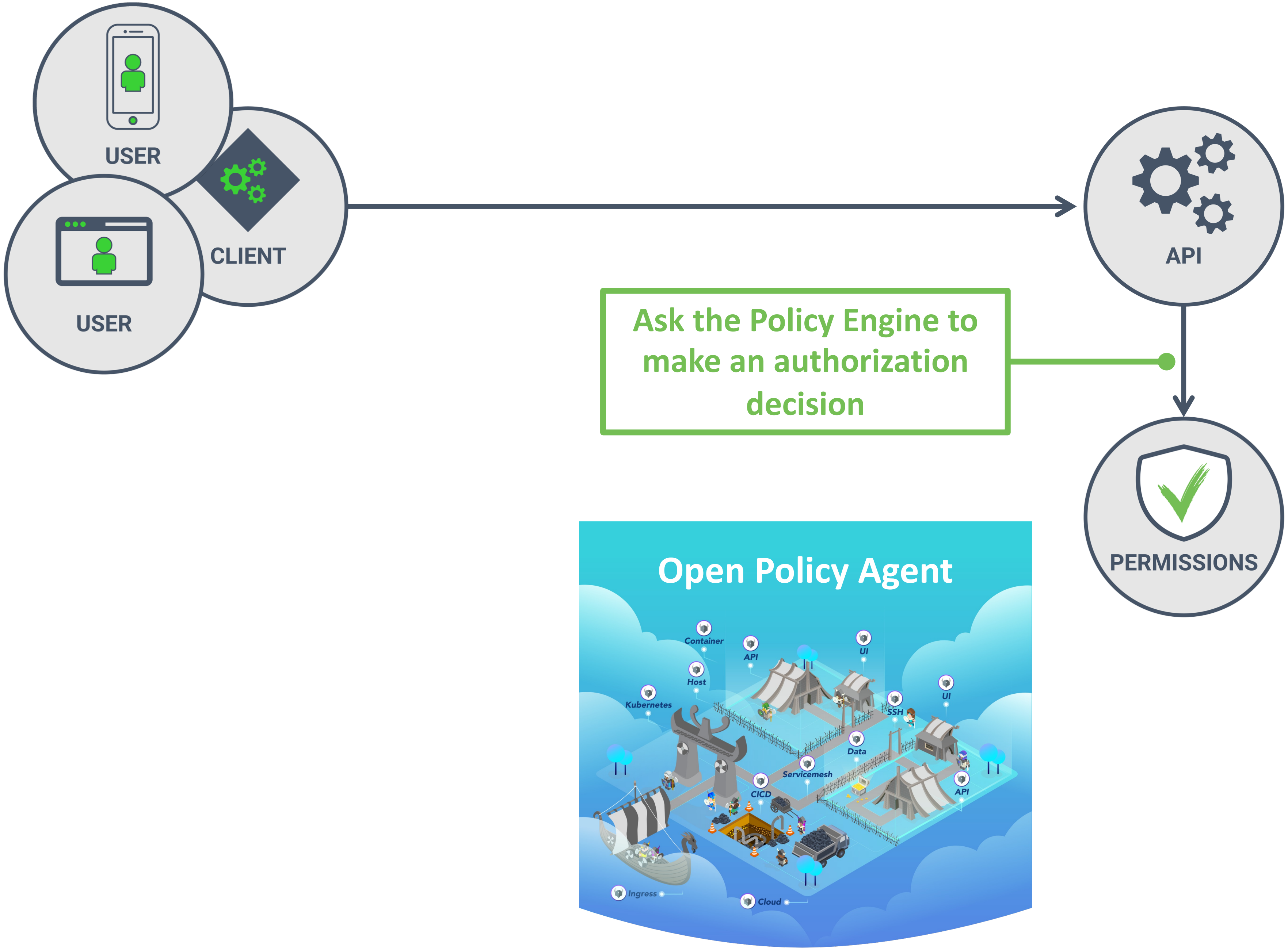
Note the deny-by-default approach



Ask the Policy Engine to make an authorization decision

A "centralized" policy results in a clear and auditable authorization policy

```java
public void editRestaurant(long id, Restaurant restaurant) {
  Restaurant orig = restaurantData.get(id);
  if(policy.canEditRestaurant(user, orig)) {
    restaurantData.update(id, restaurant);
  }
  else {
    throw new AuthorizationException(":(");
  }
}
```



Ask the Policy Engine to make an authorization decision

https://www.openpolicyagent.org/

Centralize complex authorization logic

Complex authorization logic should not be scattered throughout the code, but is best defined in a clear and understandable authorization policy

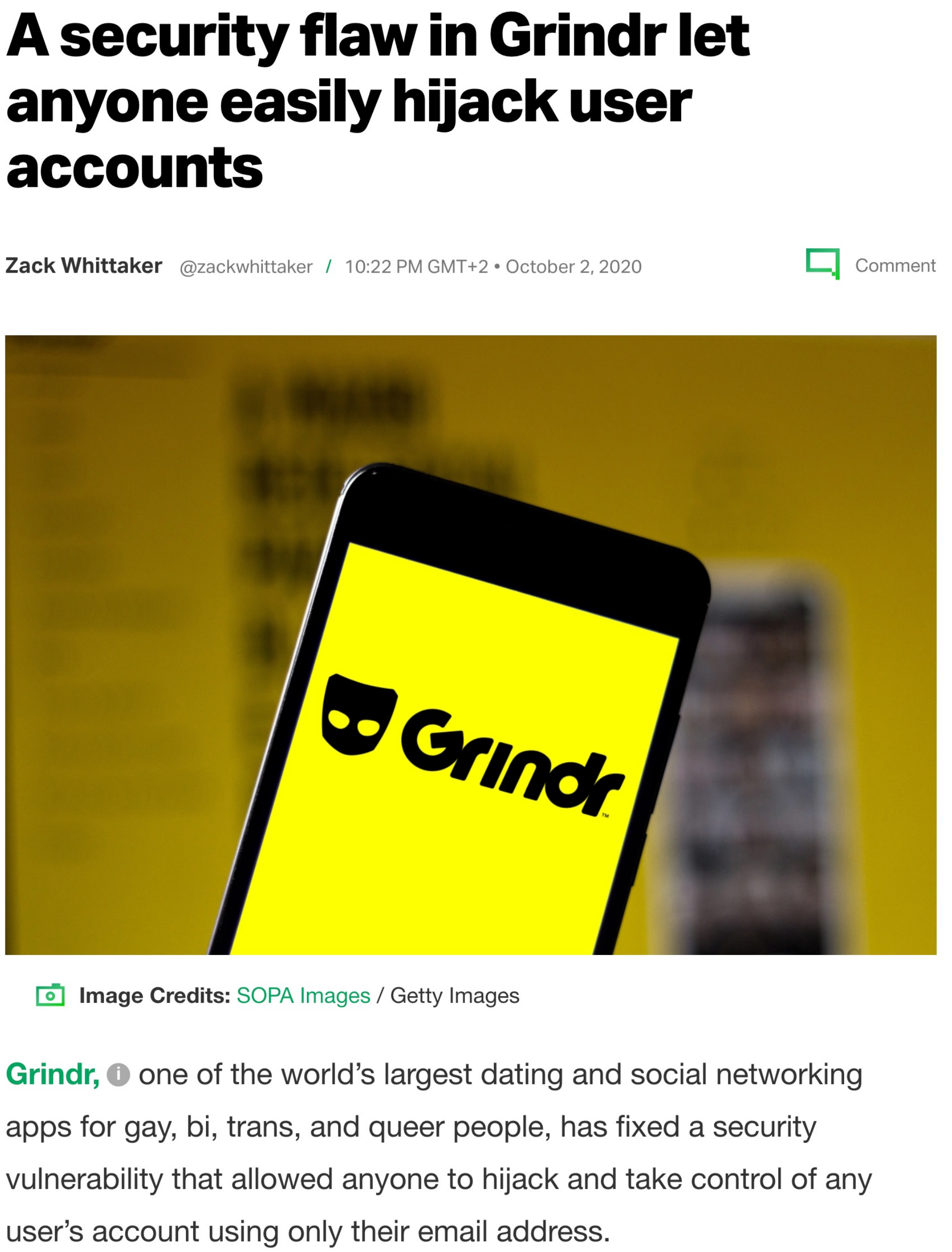
Empower auditability

Simplify the auditing of your authorization policy by making authorization logic explicit, even when endpoints have no specific authorization requirements.

1. Broken object level authorization
2. Broken authentication
3. Broken object property-level authorization
4. Unrestricted resource consumption
5. Broken function level authorization
6. Unrestricted access to sensitive business flows
7. Server-side request forgery
8. Security misconfiguration
9. Improper inventory management
10. Unsafe consumption of APIs

OWASP

API Security TOP10



A security flaw in Grindr let anyone easily hijack user accounts

Zack Whittaker @zackwhittaker / 10:22 PM GMT+2 • October 2, 2020

Image Credits: SOPA Images / Getty Images

Grindr, one of the world's largest dating and social networking apps for gay, bi, trans, and queer people, has fixed a security vulnerability that allowed anyone to hijack and take control of any user's account using only their email address.

https://techcrunch.com/2020/10/02/grindr-account-hijack-flaw/

> To reset a password, Grindr sends the user an email with a clickable link containing an account password reset token.
>
> Grindr's password reset page was leaking password reset tokens to the browser.

The Java Spring endpoint returning users

```java
@RequestMapping(path = "/online/users", method = GET, produces = "application/json")
public ResponseEntity<Object> getOnlineUsers() {
  List<User> users = UserService.getOnlineUsers();
  return new ResponseEntity<Object>(users, HttpStatus.OK);
}
```

The User data class

```java
public class User {
  private String id, name, address;
  …
  public String getName() {
    return name;
  }

  public String getAddress() {
    return address;
  }
}
```

Data fields are automatically translated to JSON, even when they are not supposed to be exposed

The Java Spring endpoint returning users

```
@RequestMapping(path = "/online/users", method = GET, produces = "application/json")
public ResponseEntity<Object> getOnlineUsers() {
  List<User> users = UserService.getOnlineUsers();
  return new ResponseEntity<Object>(users, HttpStatus.OK);
}
```

The User data class

```
public class User {
  private String id, name, address;
  …
  public String getName() {
    return name;
  }

  @JsonIgnore
  public String getAddress() {
    return address;
  }
}
```

Annotations can be used to avoid including sensitive fields in JSON responses, but this approach is impossible at scale and violates the "deny-by-default" best practice

The Java Spring endpoint returning users

```
@RequestMapping(path = "/online/users", method = GET, produces = "application/json")
public ResponseEntity<Object> getOnlineUsers() {
  List<User> users = UserService.getOnlineUsers();
  return new ResponseEntity<Object>(users.stream().map(PublicUserInfo::new), HttpStatus.OK);
}
```

The PublicUserInfo DTO class

```
public class PublicUserInfo {
  private String id, name;

  public PublicUserInfo(User user) {
    this.setId(user.getId());
    this.setName(user.getName());
  }
  …
  public String getName() {
    return name;
  }
}
```

The DTO class only defines fields that are supposed to be exposed.

A User object is never directly exposed to the client.

Avoid sensitive data exposure

Avoid directly returning internal application data, as this often results in the exposure of sensitive data.

Use strict schemas or DTOs in combination with a well-defined OpenAPI specification of your API.

If an API automatically exposes data, does it also automatically accept data?

The Java Spring endpoint returning users

```
@RequestMapping(path = "/user/{id}", method = PATCH, consumes = "application/json")
public void updateUser(String id, @RequestBody User user) {
  UserService.updateUser(id, user);
}
```

Updates the DB with new field values for the user with the given ID

The User data class

```
public class User {
  private String id, name, role;
  …
  public void setName(String name) {
    this.name = name;
  }

  public String setRole(String role) {
    this. role = role;
  }
}
```

A legitimate request payload to update the user's name

```
{
  "name": "Dr. Phil"
}
```

A malicious request payload to update restricted fields

```
{
  "name": ”Philippe becomes admin",
  "role": "admin"
}
```

The Java Spring endpoint returning users

```java
@RequestMapping(path = "/user/{id}", method = PATCH, consumes = "application/json")
public void updateUser(String id, @RequestBody User user) {
  UserService.updateUser(id, user);
}
```

The User data class

```java
public class User {
  private String id, name, role;
  …
  public void setName(String name) {
    this.name = name;
  }

  @JsonProperty(access = Access.READ_ONLY)
  public String setRole(String role) {
    this. role = role;
  }
}
```

Annotations can be used to avoid populating sensitive fields with JSON data, **but this approach is impossible at scale and violates the "deny-by-default" best practice**

The Java Spring endpoint returning users

```java
@RequestMapping(path = "/user/{id}", method = PATCH, consumes = "application/json")
public void updateUser(String id, @RequestBody UpdateUserInfo user) {
  UserService.updateUser(id, user);
}
```

The UpdateUserInfo DTO class

```java
public class UpdateUserInfo {
  private name;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

The DTO class only defines fields that are supposed to be populated.

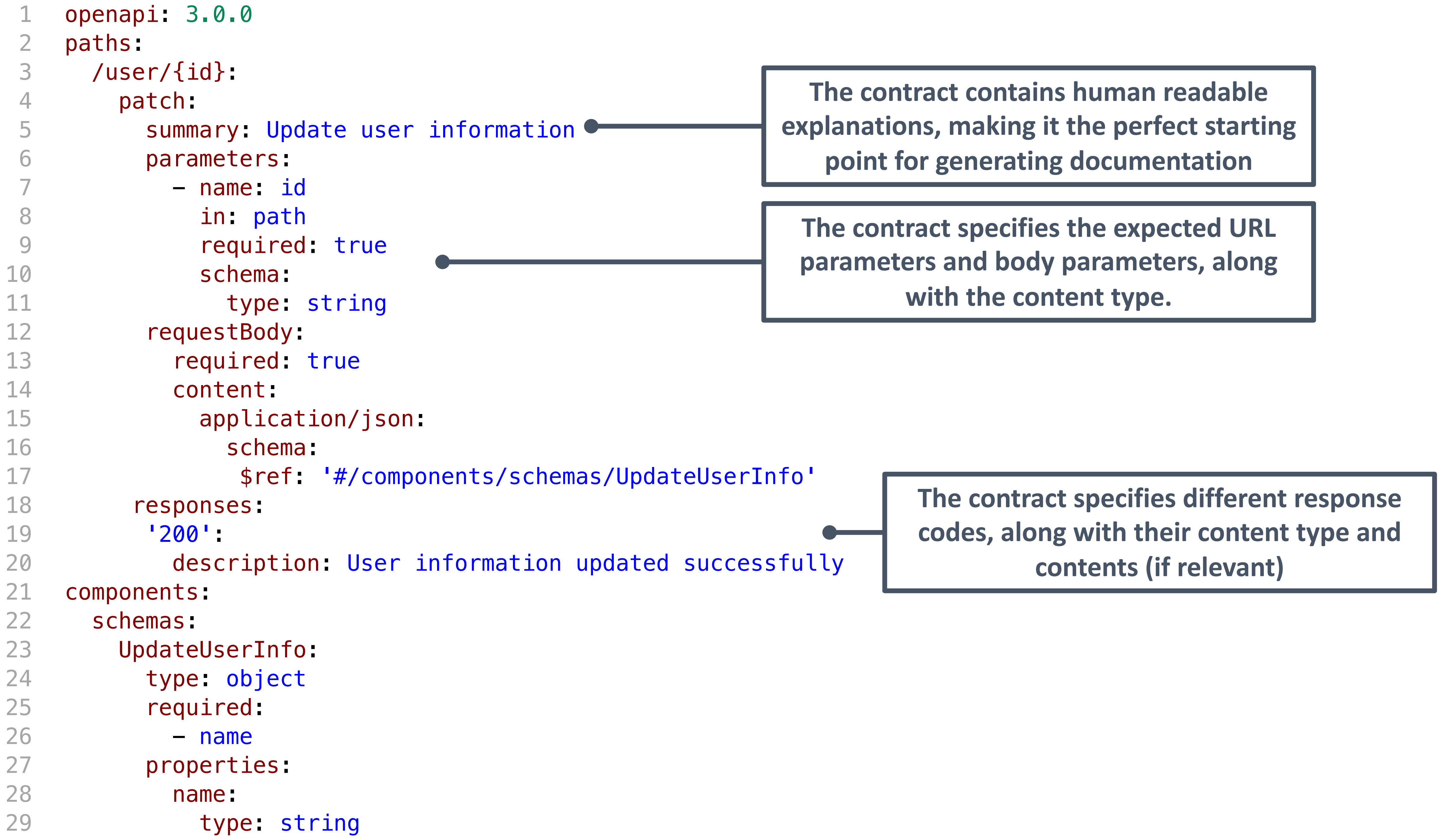
A User object is never directly accepted as input from the client.

Verify your APIs for Mass Assignment

Avoid directly transforming incoming data into model objects, as this often results in the accidental writing of internal values.

Use strict schemas or DTOs in combination with a well-defined OpenAPI specification of your API.

How can we address BOPLA issues?

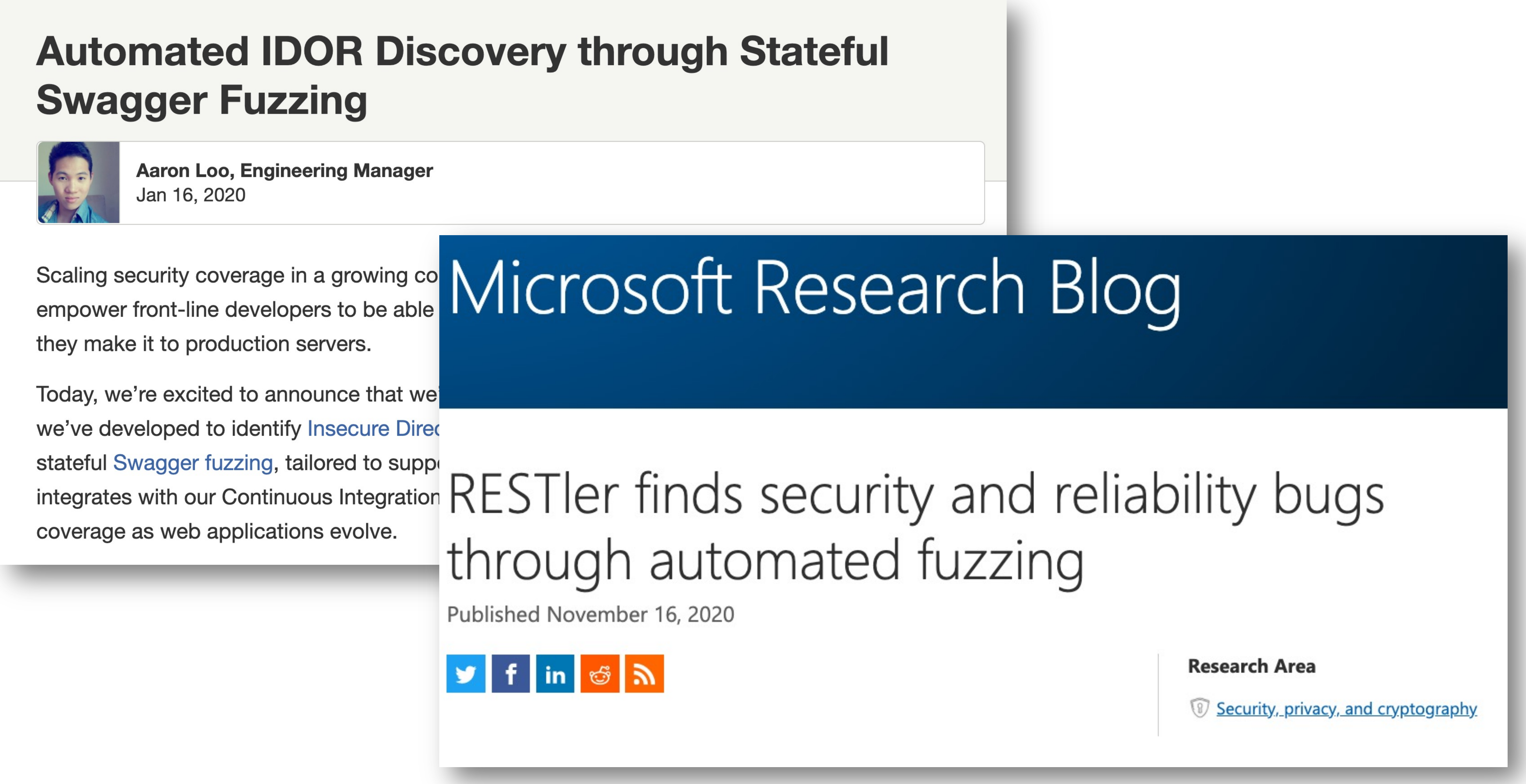


```
openapi: 3.0.0
paths:
  /user/{id}:
    patch:
      summary: Update user information
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: string
      requestBody:
        required: true
        content:
          application/json:
            schema:
             $ref: '#/components/schemas/UpdateUserInfo'
     responses:
      '200':
        description: User information updated successfully
components:
  schemas:
    UpdateUserInfo:
      type: object
      required:
        - name
      properties:
        name:
          type: string
```

The contract contains human readable explanations, making it the perfect starting point for generating documentation

The contract specifies the expected URL parameters and body parameters, along with the content type.

The contract specifies different response codes, along with their content type and contents (if relevant)



Automated IDOR Discovery through Stateful Swagger Fuzzing

Aaron Loo, Engineering Manager
Jan 16, 2020

Scaling security coverage in a growing co
empower front-line developers to be able
they make it to production servers.

Today, we’re excited to announce that we
we’ve developed to identify Insecure Direc
stateful Swagger fuzzing, tailored to supp
integrates with our Continuous Integration
coverage as web applications evolve.

Microsoft Research Blog

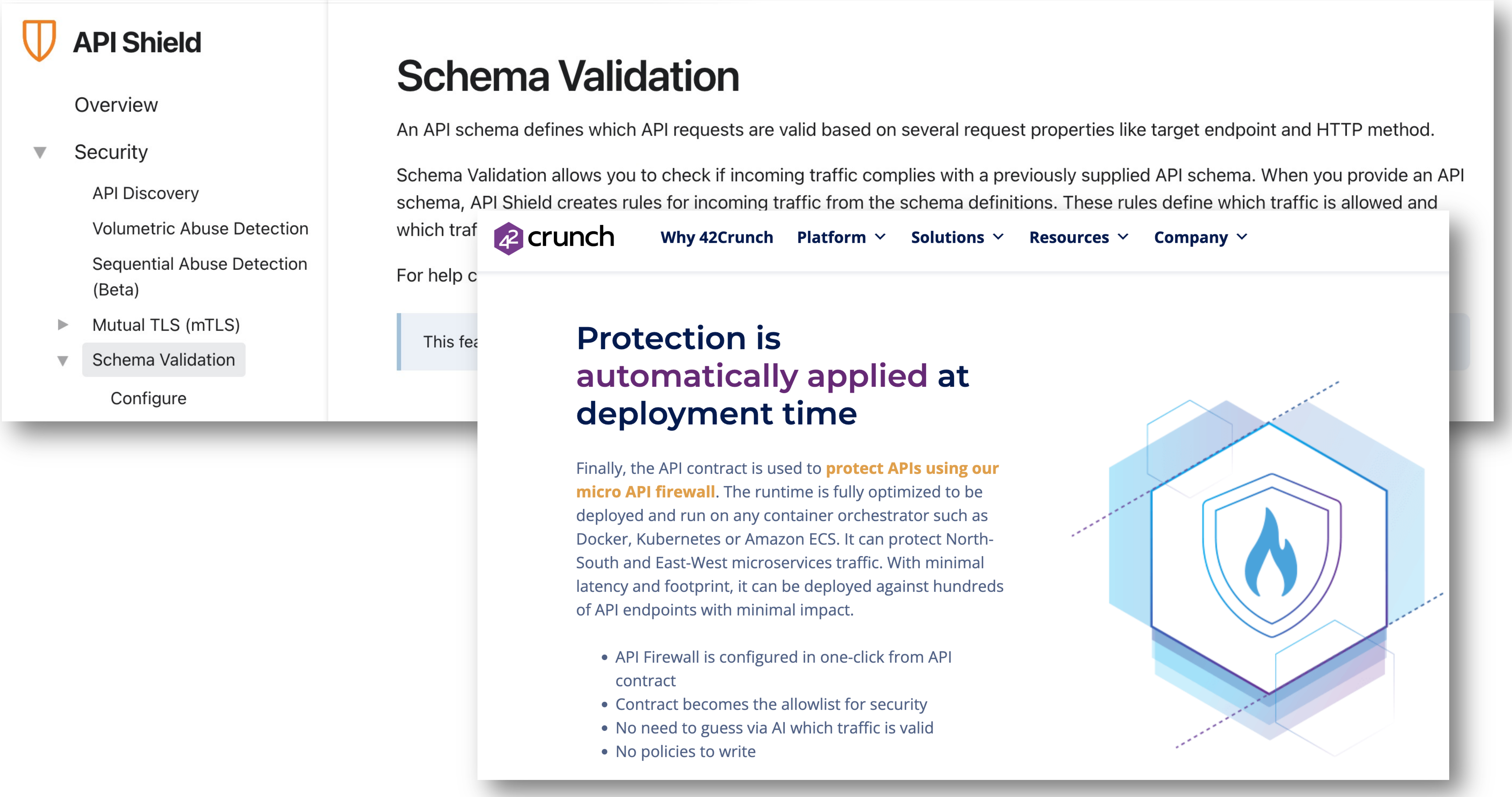
RESTler finds security and reliability bugs through automated fuzzing

Published November 16, 2020

Research Area

Security, privacy, and cryptography

https://engineeringblog.yelp.com/2020/01/automated-idor-discovery-through-stateful-swagger-fuzzing.html
https://www.microsoft.com/en-us/research/blog/restler-finds-security-and-reliability-bugs-through-automated-fuzzing/



Schema Validation

An API schema defines which API requests are valid based on several request properties like target endpoint and HTTP method.

Schema Validation allows you to check if incoming traffic complies with a previously supplied API schema. When you provide an API schema, API Shield creates rules for incoming traffic from the schema definitions. These rules define which traffic is allowed and which traf

For help c

This fea

Protection is automatically applied at deployment time

Finally, the API contract is used to **protect APIs using our micro API firewall**. The runtime is fully optimized to be deployed and run on any container orchestrator such as Docker, Kubernetes or Amazon ECS. It can protect North-South and East-West microservices traffic. With minimal latency and footprint, it can be deployed against hundreds of API endpoints with minimal impact.

- API Firewall is configured in one-click from API contract
- Contract becomes the allowlist for security
- No need to guess via AI which traffic is valid
- No policies to write

https://42crunch.com/api-security-platform/
https://developers.cloudflare.com/api-shield/security/schema-validation//

Use Swagger/OpenAPI definitions for security

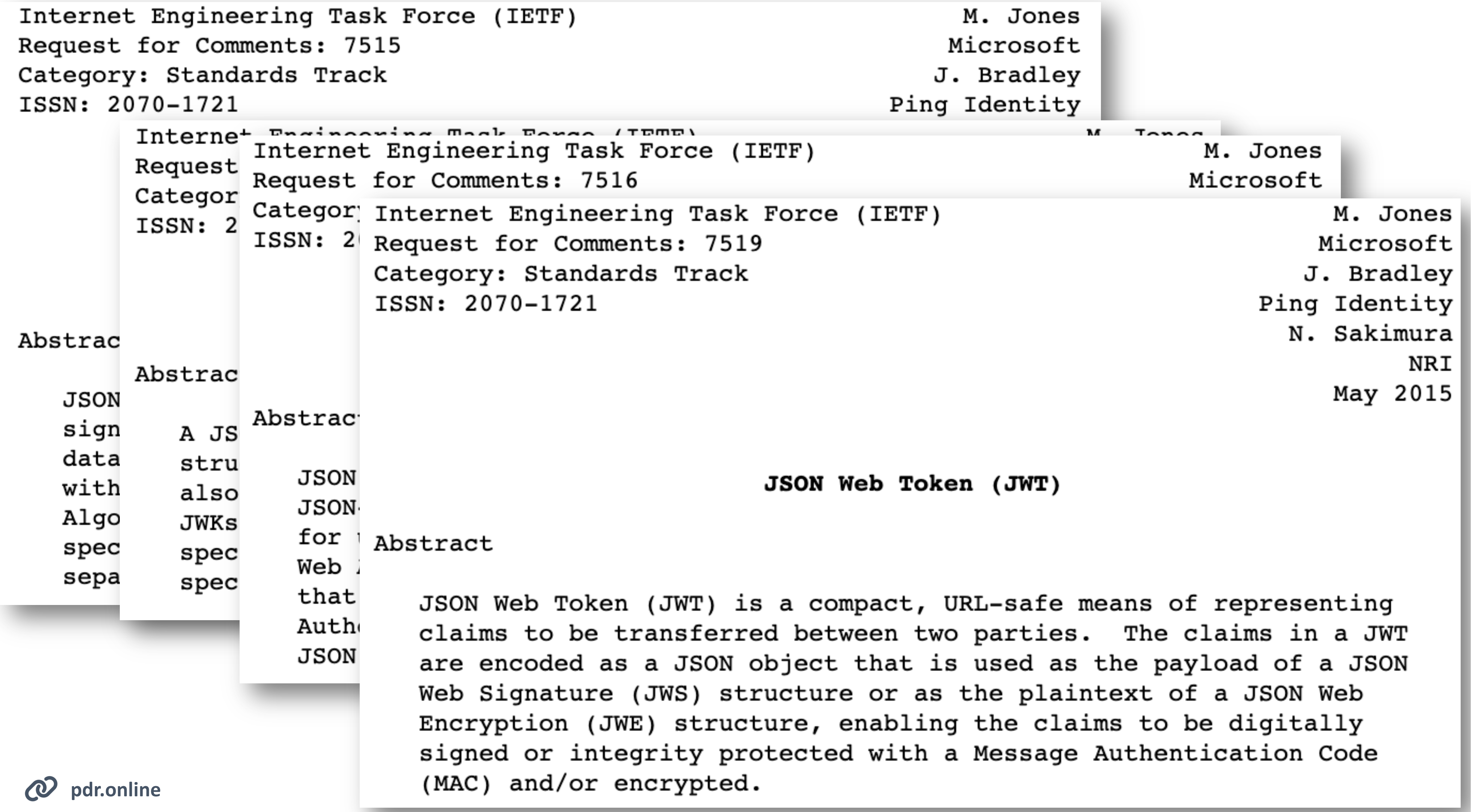
Write Swagger/OpenAPI definitions to specify the behavior of your API. Security tools consume such definitions for automatic detection and protection.

1	Broken object level authorization
2	Broken authentication
3	Broken object property-level authorization
4	Unrestricted resource consumption
5	Broken function level authorization
6	Unrestricted access to sensitive business flows
7	Server-side request forgery
8	Security misconfiguration
9	Improper inventory management
10	Unsafe consumption of APIs

OWASP

API Security

TOP 10



Internet Engineering Task Force (IETF) M. Jones
Request for Comments: 7515 Microsoft
Category: Standards Track J. Bradley
ISSN: 2070-1721 Ping Identity

Abstrac

JSON
sign
data
with
Algo
spec
sepa

Internet Engineering Task Force (IETF) M. Jones
Request
Categor
ISSN: 2

Abstrac

A JS
stru
also
JWKs
spec
spec

Internet Engineering Task Force (IETF) M. Jones
Request for Comments: 7516 Microsoft
Categor
ISSN: 2

Abstrac

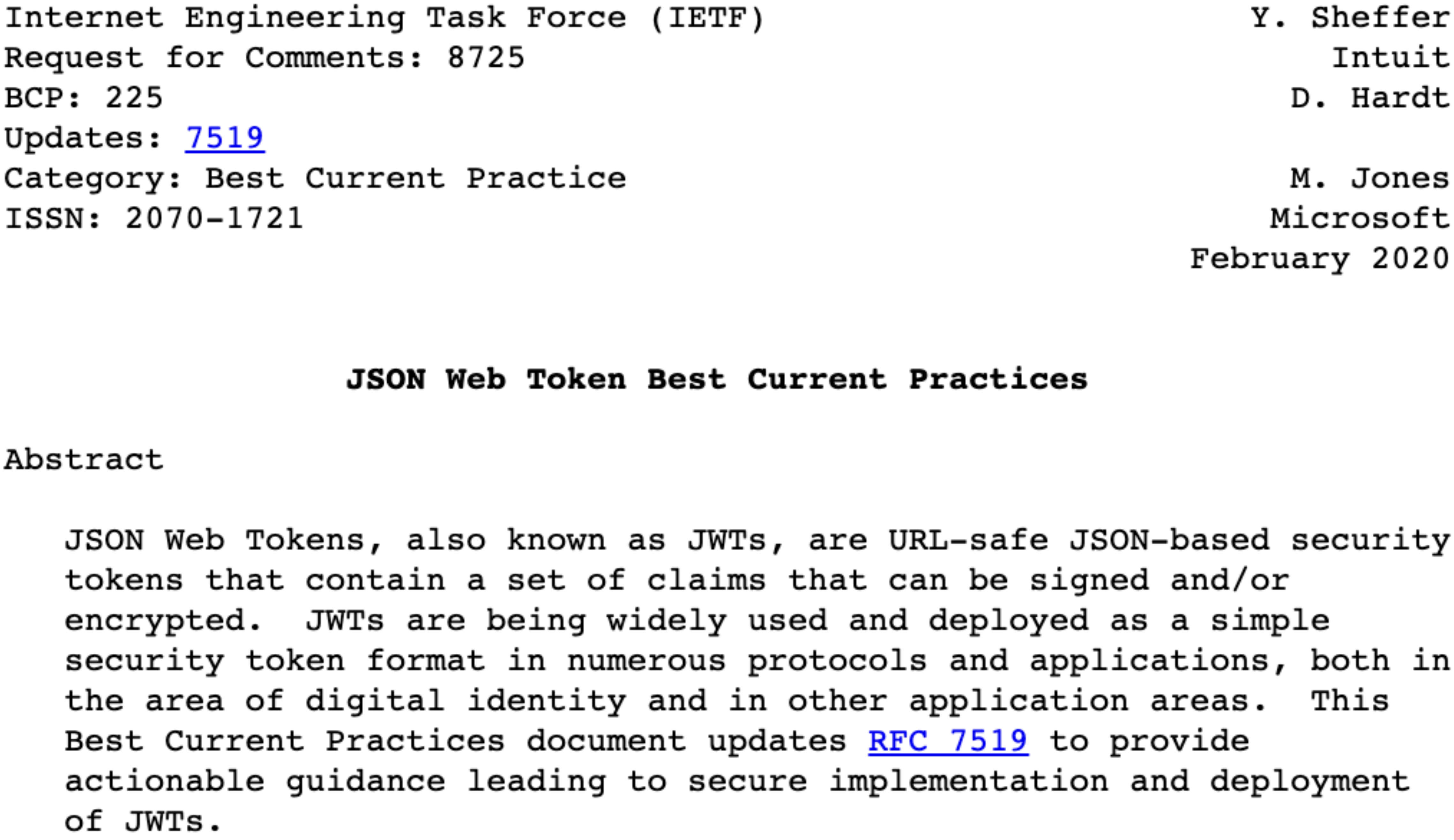
JSON
JSON-
for
Web
that
Auth
JSON

Internet Engineering Task Force (IETF) M. Jones
Request for Comments: 7519 Microsoft
Category: Standards Track J. Bradley
ISSN: 2070-1721 Ping Identity
N. Sakimura
NRI
May 2015

JSON Web Token (JWT)

Abstract

JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted.



Internet Engineering Task Force (IETF)
Request for Comments: 8725
BCP: 225
Updates: 7519
Category: Best Current Practice
ISSN: 2070-1721

Y. Sheffer
Intuit
D. Hardt

M. Jones
Microsoft

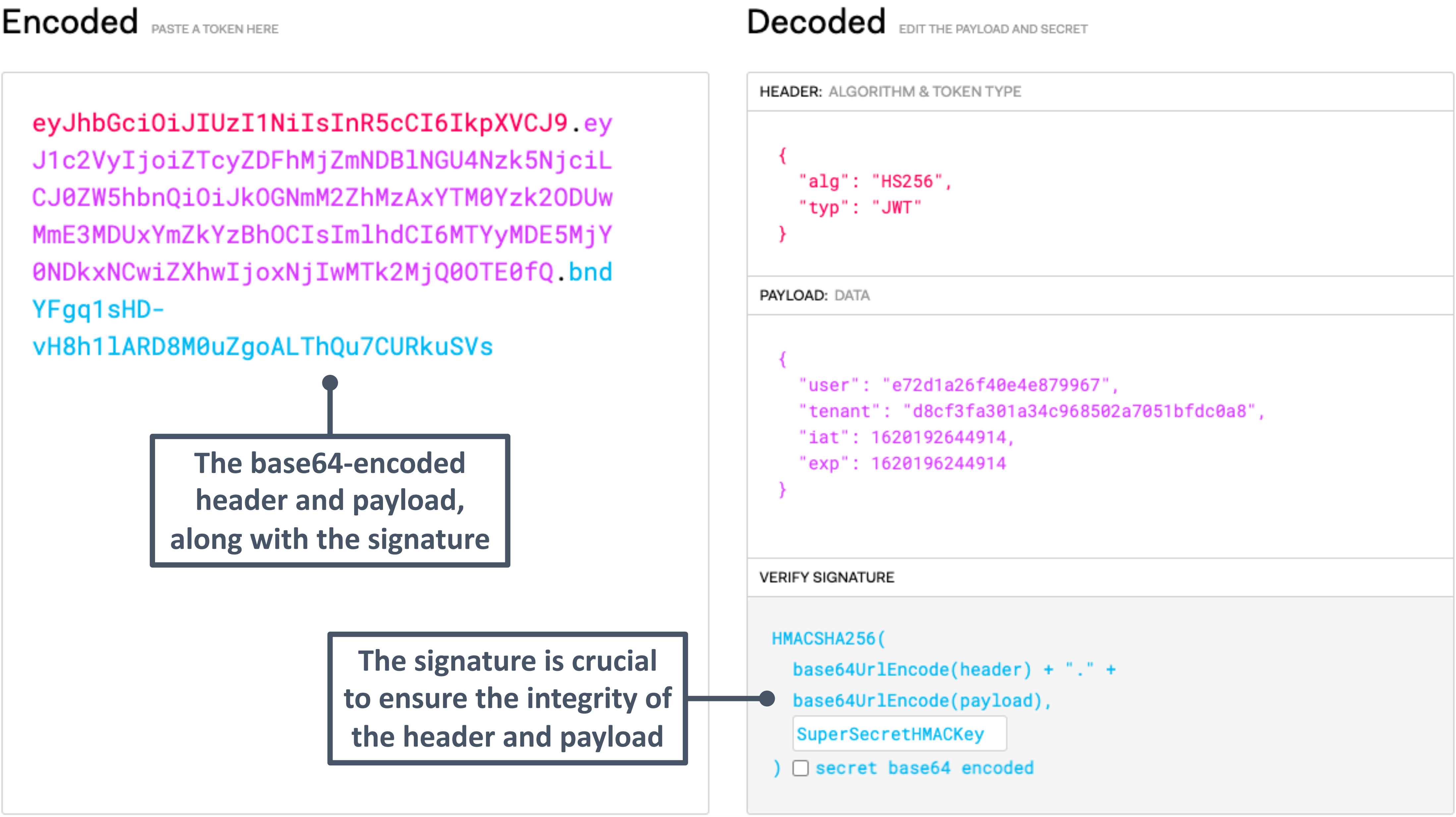
February 2020

JSON Web Token Best Current Practices

Abstract

JSON Web Tokens, also known as JWTs, are URL-safe JSON-based security tokens that contain a set of claims that can be signed and/or encrypted. JWTs are being widely used and deployed as a simple security token format in numerous protocols and applications, both in the area of digital identity and in other application areas. This Best Current Practices document updates RFC 7519 to provide actionable guidance leading to secure implementation and deployment of JWTs.

https://tools.ietf.org/html/rfc8725



Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
J1c2VyIjoiZTcyZDFhMjZmNDBlNGU4Nzk5NjciL
CJ0ZW5hbnQiOiJkOGNmM2ZhMzAxYTM0Yzk2ODUw
MmE3MDUxYmZkYzBhOCIsImlhdCI6MTYyMDE5MjY
0NDkxNCwiZXhwIjoxNjIwMTk2MjQ0OTE0fQ.bnd
YFgq1sHD-
vH8h1lARD8M0uZgoALThQu7CURkuSVs
```

The base64-encoded header and payload, along with the signature

The signature is crucial to ensure the integrity of the header and payload

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

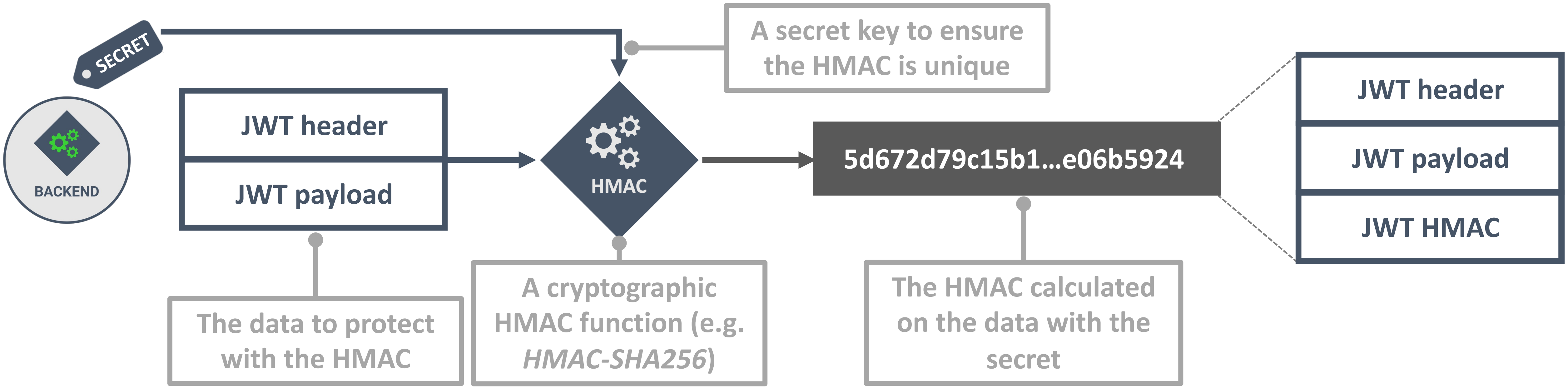
```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "user": "e72d1a26f40e4e879967",
  "tenant": "d8cf3fa301a34c968502a7051bfdc0a8",
  "iat": 1620192644914,
  "exp": 1620196244914
}
```

VERIFY SIGNATURE

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

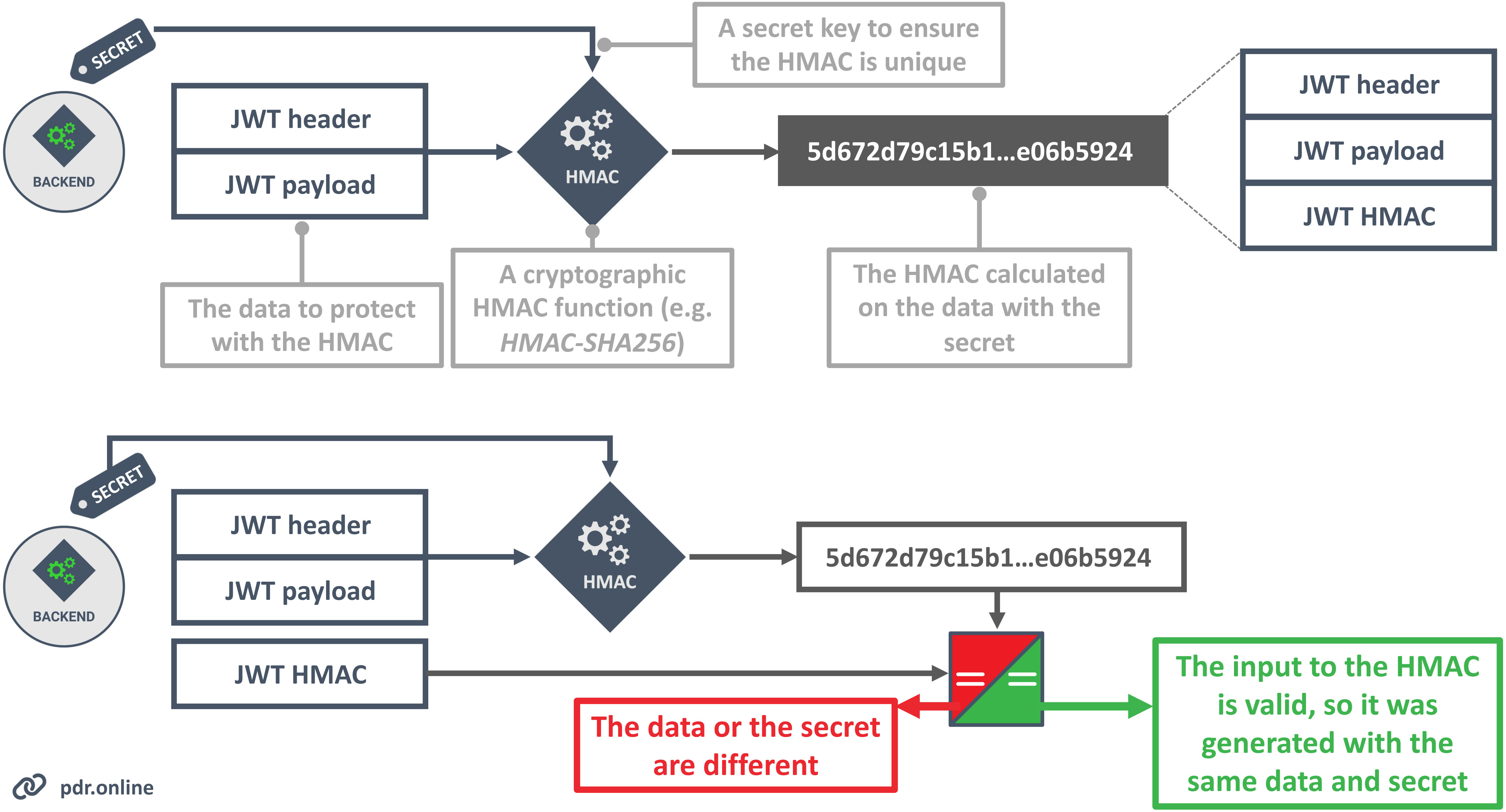


A secret key to ensure the HMAC is unique

The data to protect with the HMAC

A cryptographic HMAC function (e.g. *HMAC-SHA256*)

The HMAC calculated on the data with the secret



A secret key to ensure the HMAC is unique

The data to protect with the HMAC

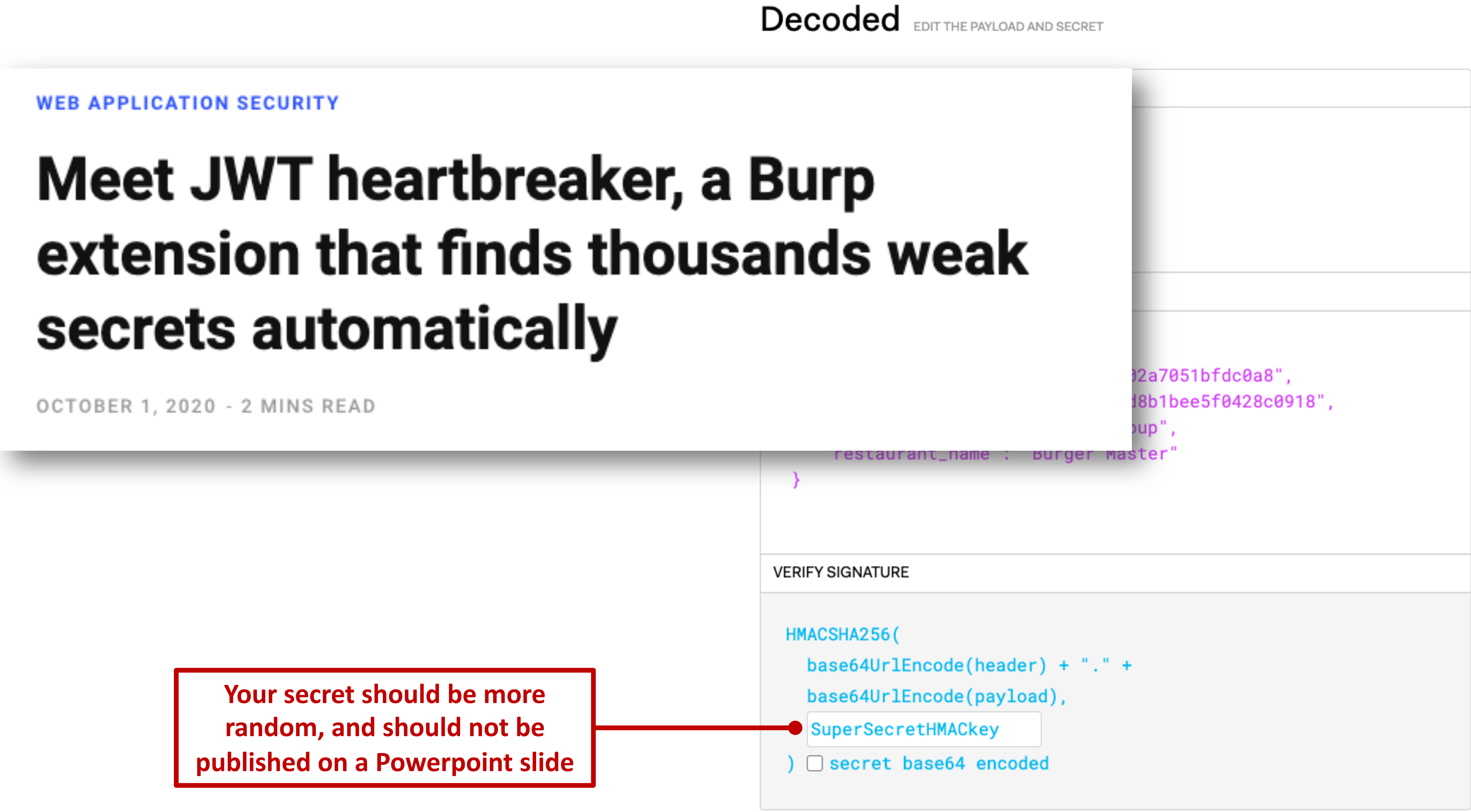
A cryptographic HMAC function (e.g. *HMAC-SHA256*)

The HMAC calculated on the data with the secret

The data or the secret are different

The input to the HMAC is valid, so it was generated with the same data and secret

pdr.online



Decoded EDIT THE PAYLOAD AND SECRET

WEB APPLICATION SECURITY

Meet JWT heartbreaker, a Burp extension that finds thousands weak secrets automatically

OCTOBER 1, 2020 - 2 MINS READ

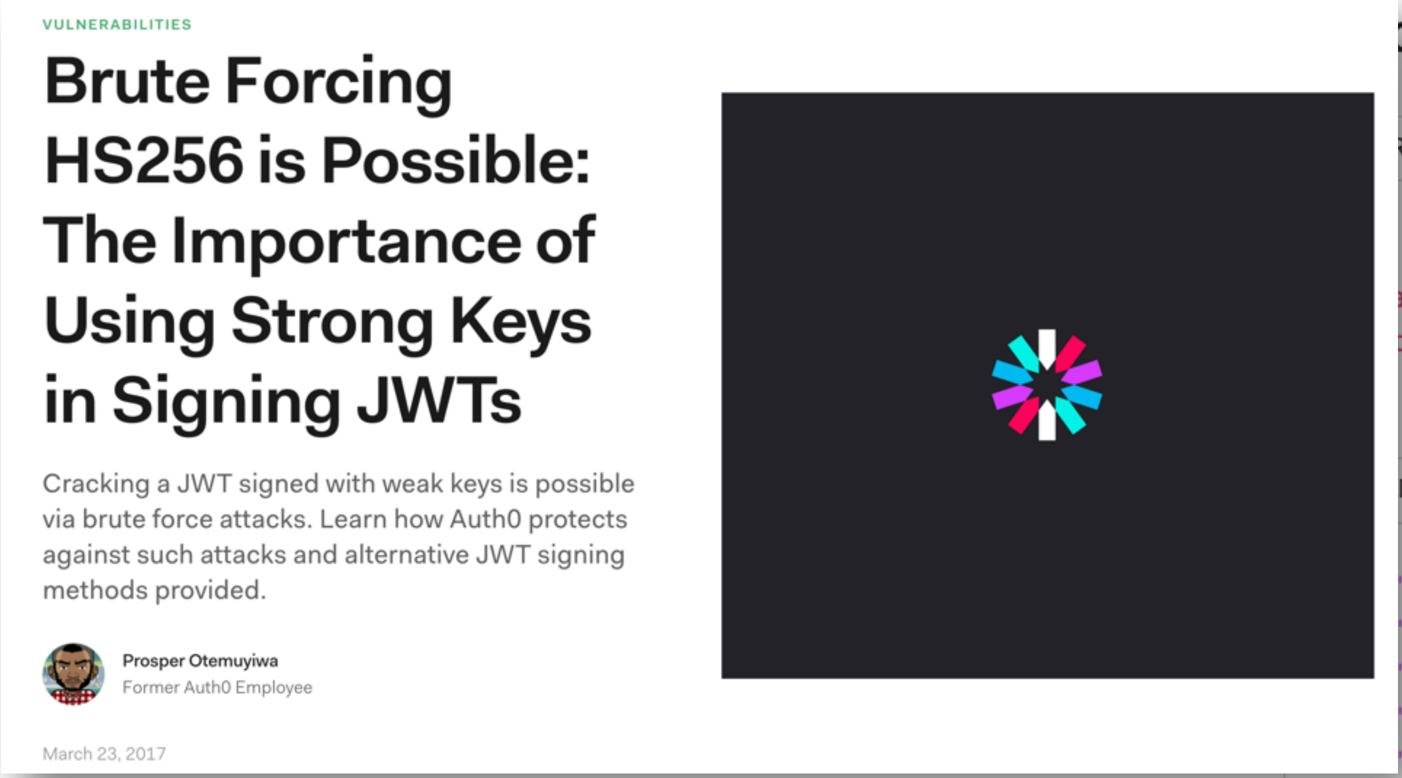
```
                              2a7051bfdc0a8",
                              d8b1bee5f0428c0918",
                              oup",
    restaurant_name : Burger Master"
  }
```

VERIFY SIGNATURE

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

Your secret should be more random, and should not be published on a Powerpoint slide

https://lab.wallarm.com/meet-jwt-heartbreaker-a-burp-extension-that-finds-thousands-weak-secrets-automatically/



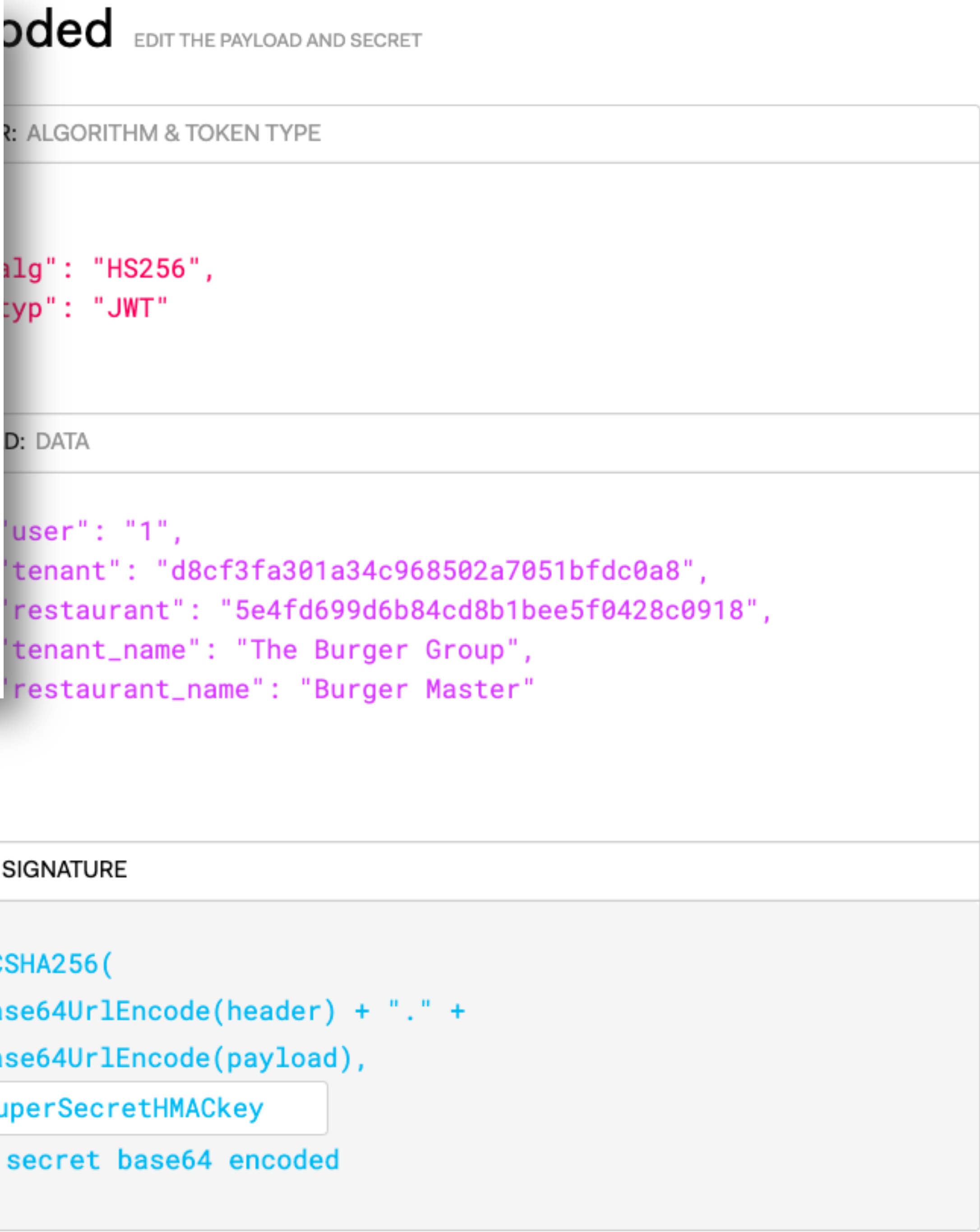
VULNERABILITIES

Brute Forcing HS256 is Possible: The Importance of Using Strong Keys in Signing JWTs

Cracking a JWT signed with weak keys is possible via brute force attacks. Learn how Auth0 protects against such attacks and alternative JWT signing methods provided.

Prosper Otemuyiwa
Former Auth0 Employee

March 23, 2017



oded EDIT THE PAYLOAD AND SECRET

R: ALGORITHM & TOKEN TYPE

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

D: DATA

```
'user": "1",
'tenant": "d8cf3fa301a34c968502a7051bfdc0a8",
'restaurant": "5e4fd699d6b84cd8b1bee5f0428c0918",
'tenant_name": "The Burger Group",
'restaurant_name": "Burger Master"
}
```

VERIFY SIGNATURE

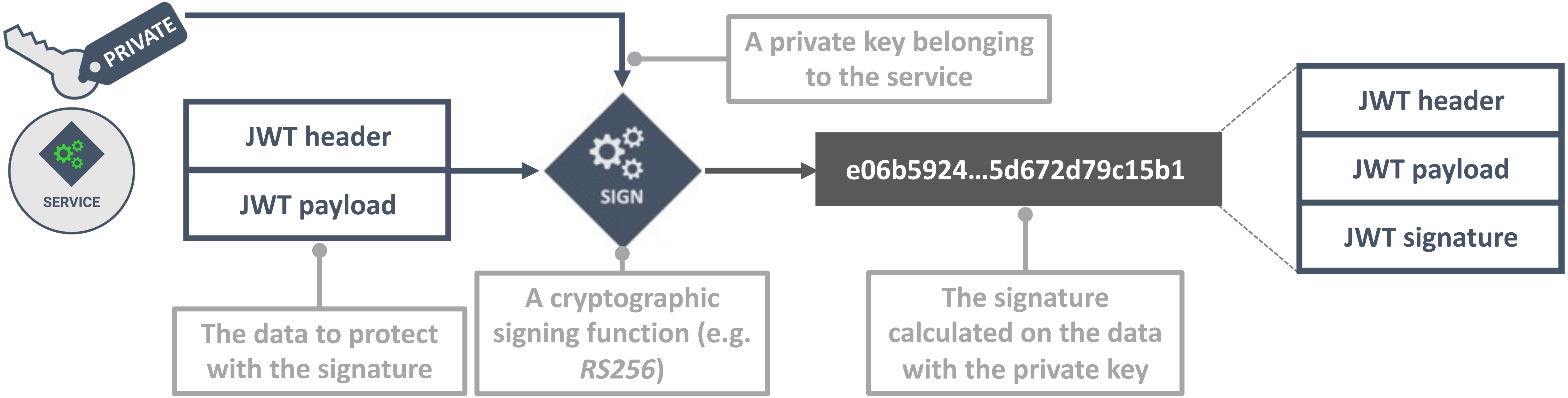
```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  SuperSecretHMACkey
) ☐ secret base64 encoded
```

A key of the same size as the hash output (for instance, 256 bits for "HS256") or larger MUST be used with this algorithm.

Your secret should be more random, and should not be published on a Powerpoint slide

https://auth0.com/blog/brute-forcing-hs256-is-possible-the-importance-of-using-strong-keys-to-sign-jwts/

HMACs are not recommended for security-sensitive JWTs

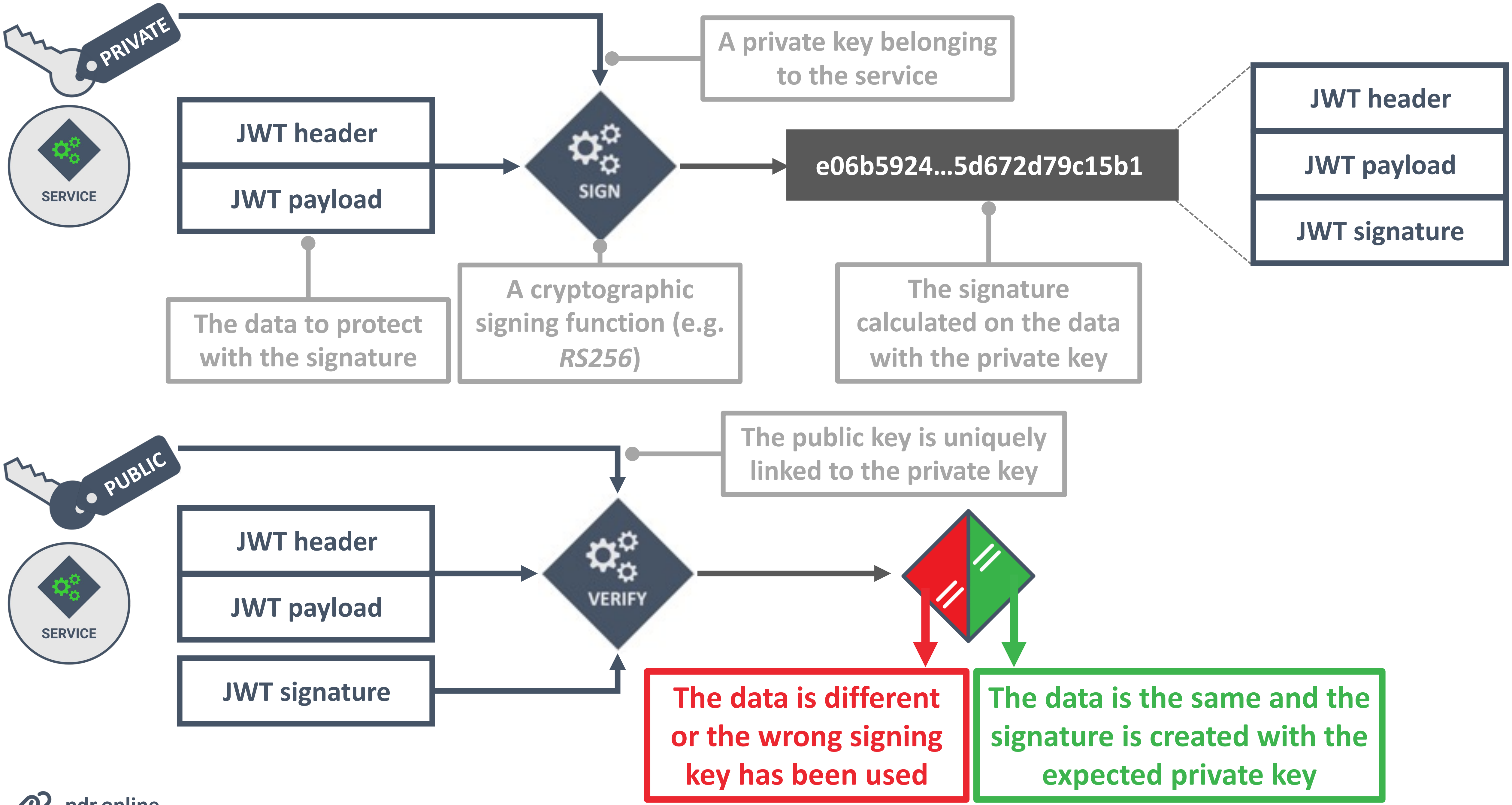


A private key belonging to the service

The data to protect with the signature

A cryptographic signing function (e.g. *RS256*)

The signature calculated on the data with the private key



PRIVATE

SERVICE

JWT header

JWT payload

SIGN

e06b5924…5d672d79c15b1

JWT header

JWT payload

JWT signature

A private key belonging to the service

The data to protect with the signature

A cryptographic signing function (e.g. *RS256*)

The signature calculated on the data with the private key

PUBLIC

SERVICE

JWT header

JWT payload

JWT signature

VERIFY

The public key is uniquely linked to the private key

The data is different or the wrong signing key has been used

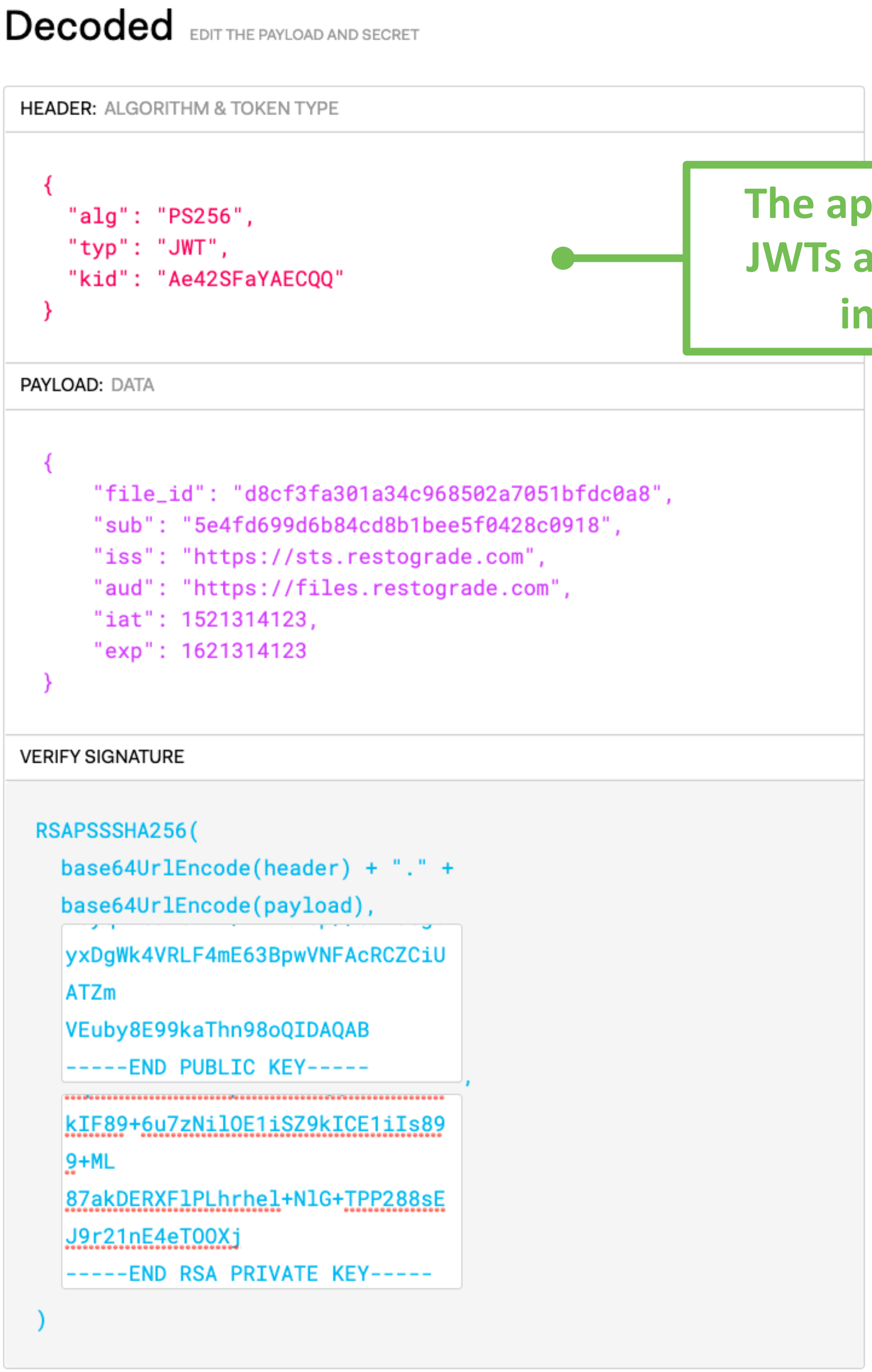
The data is the same and the signature is created with the expected private key

Use the right 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.

```
alg: none
```



Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "PS256",
  "typ": "JWT",
  "kid": "Ae42SFaYAECQQ"
}
```

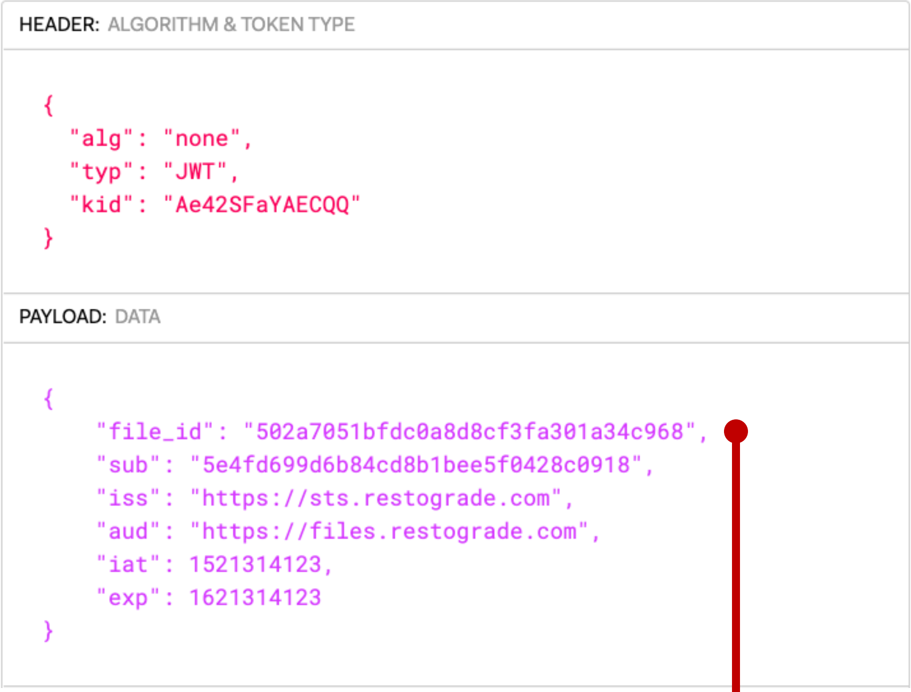
PAYLOAD: DATA

```
{
    "file_id": "d8cf3fa301a34c968502a7051bfdc0a8",
    "sub": "5e4fd699d6b84cd8b1bee5f0428c0918",
    "iss": "https://sts.restograde.com",
    "aud": "https://files.restograde.com",
    "iat": 1521314123,
    "exp": 1621314123
}
```

VERIFY SIGNATURE

```
RSAPSSSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  yxDgWk4VRLF4mE63BpwVNFAcRCZCiU
  ATZm
  VEuby8E99kaThn98oQIDAQAB
  -----END PUBLIC KEY-----
  ,
  kIF89+6u7zNil0E1iSZ9kICE1iIs89
  9+ML
  87akDERXFlPLhrhel+N1G+TPP288sE
  J9r21nE4eTOOXj
  -----END RSA PRIVATE KEY-----
)
```

The application uses signed JWTs and rejects JWTs with invalid signatures



HEADER: ALGORITHM & TOKEN TYPE

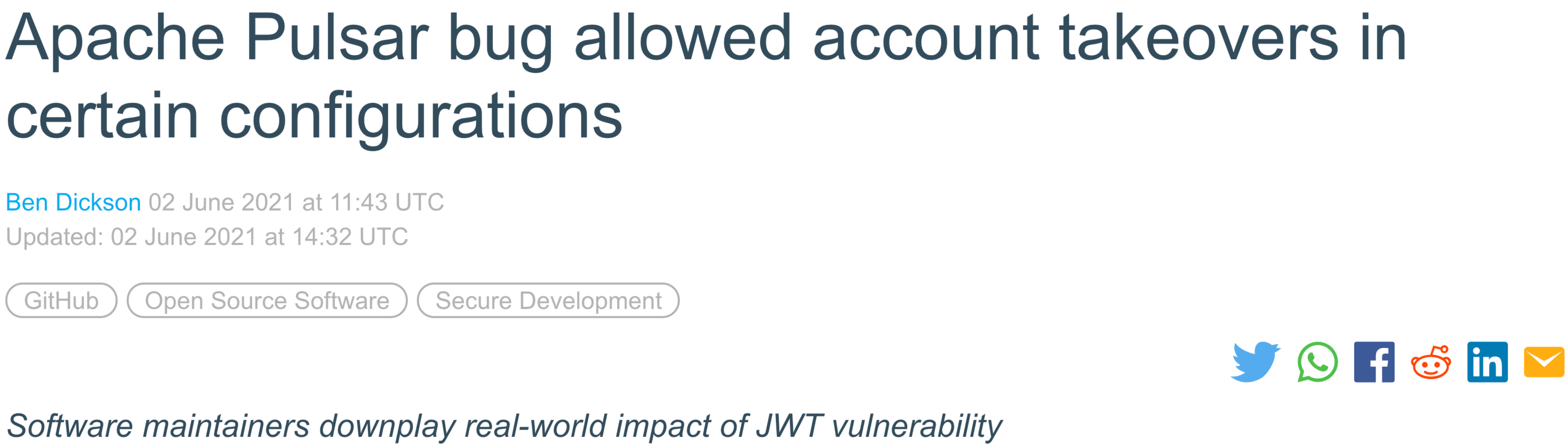
```
{
  "alg": "none",
  "typ": "JWT",
  "kid": "Ae42SFaYAECQQ"
}
```

PAYLOAD: DATA

```
{
    "file_id": "502a7051bfdc0a8d8cf3fa301a34c968",
    "sub": "5e4fd699d6b84cd8b1bee5f0428c0918",
    "iss": "https://sts.restograde.com",
    "aud": "https://files.restograde.com",
    "iat": 1521314123,
    "exp": 1621314123
}
```

By using *none* as the signature, the attacker can create a JWT that is not signed

An unsigned JWT can hold arbitrary data, giving access to arbitrary files on this system



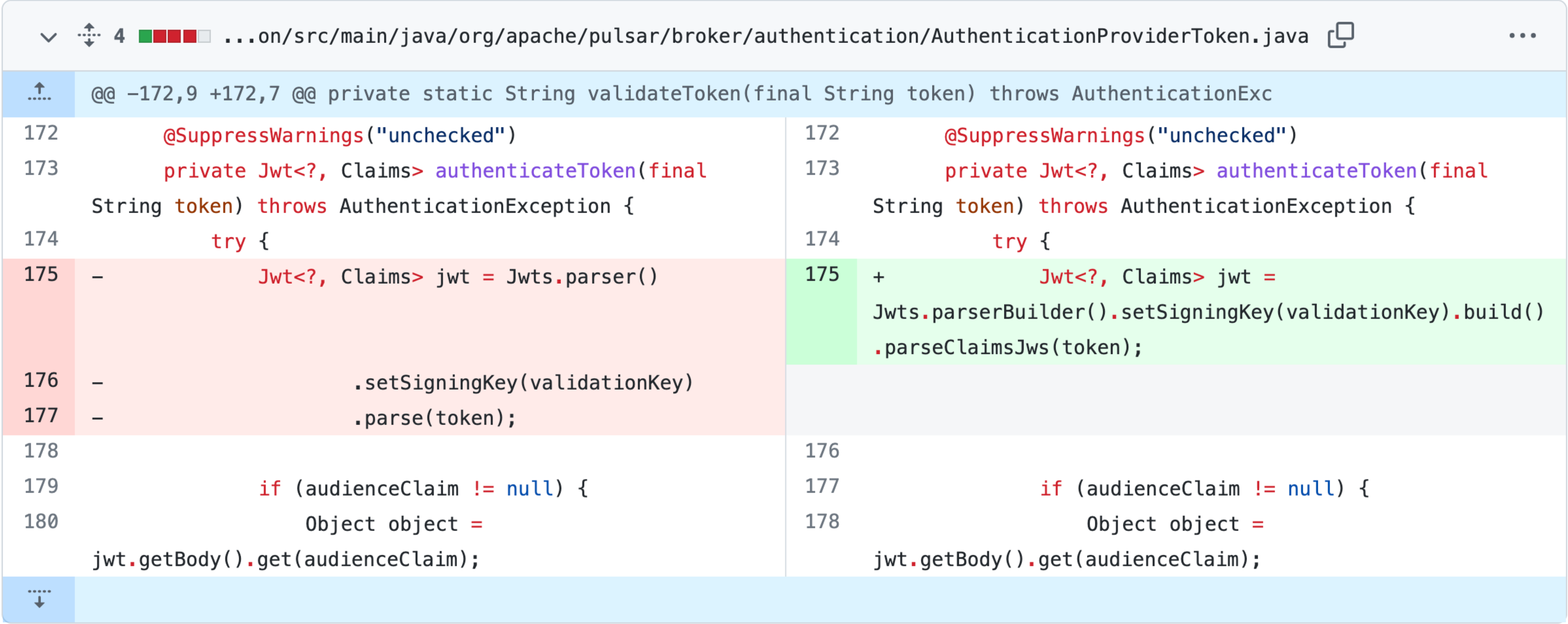
Apache Pulsar bug allowed account takeovers in certain configurations

Ben Dickson 02 June 2021 at 11:43 UTC
Updated: 02 June 2021 at 14:32 UTC

GitHub Open Source Software Secure Development

Software maintainers downplay real-world impact of JWT vulnerability

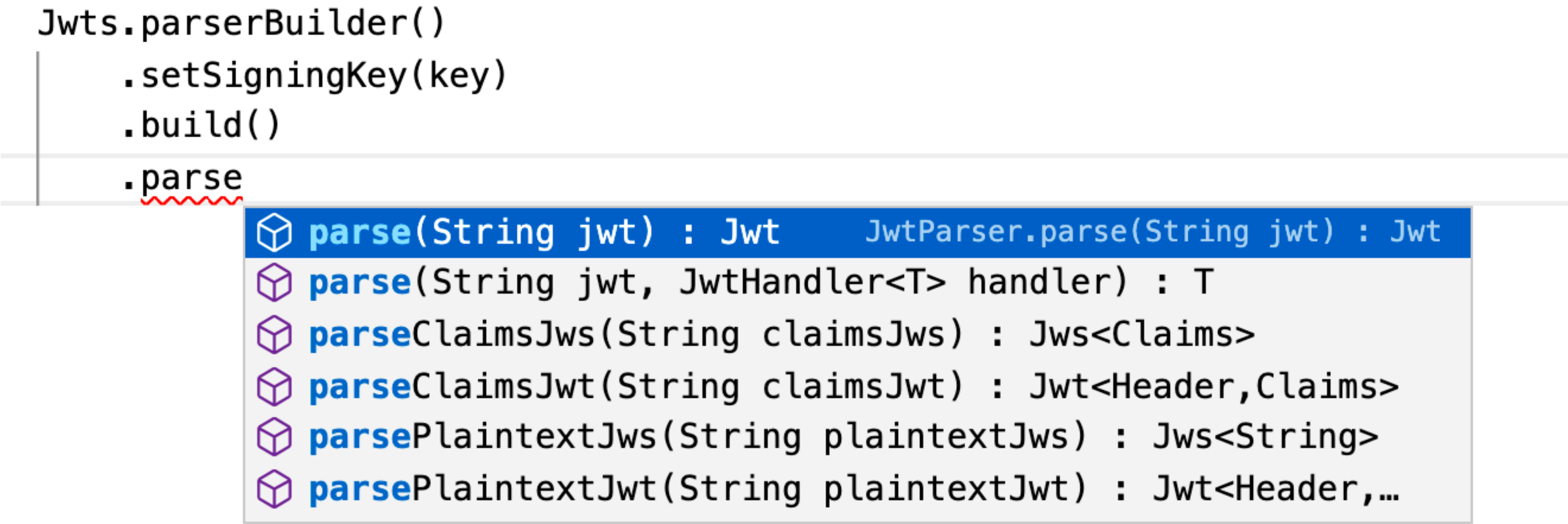
https://portswigger.net/daily-swig/apache-pulsar-bug-allowed-account-takeovers-in-certain-configurations



4 🟩🟥🟥🟥⬜ ...on/src/main/java/org/apache/pulsar/broker/authentication/AuthenticationProviderToken.java

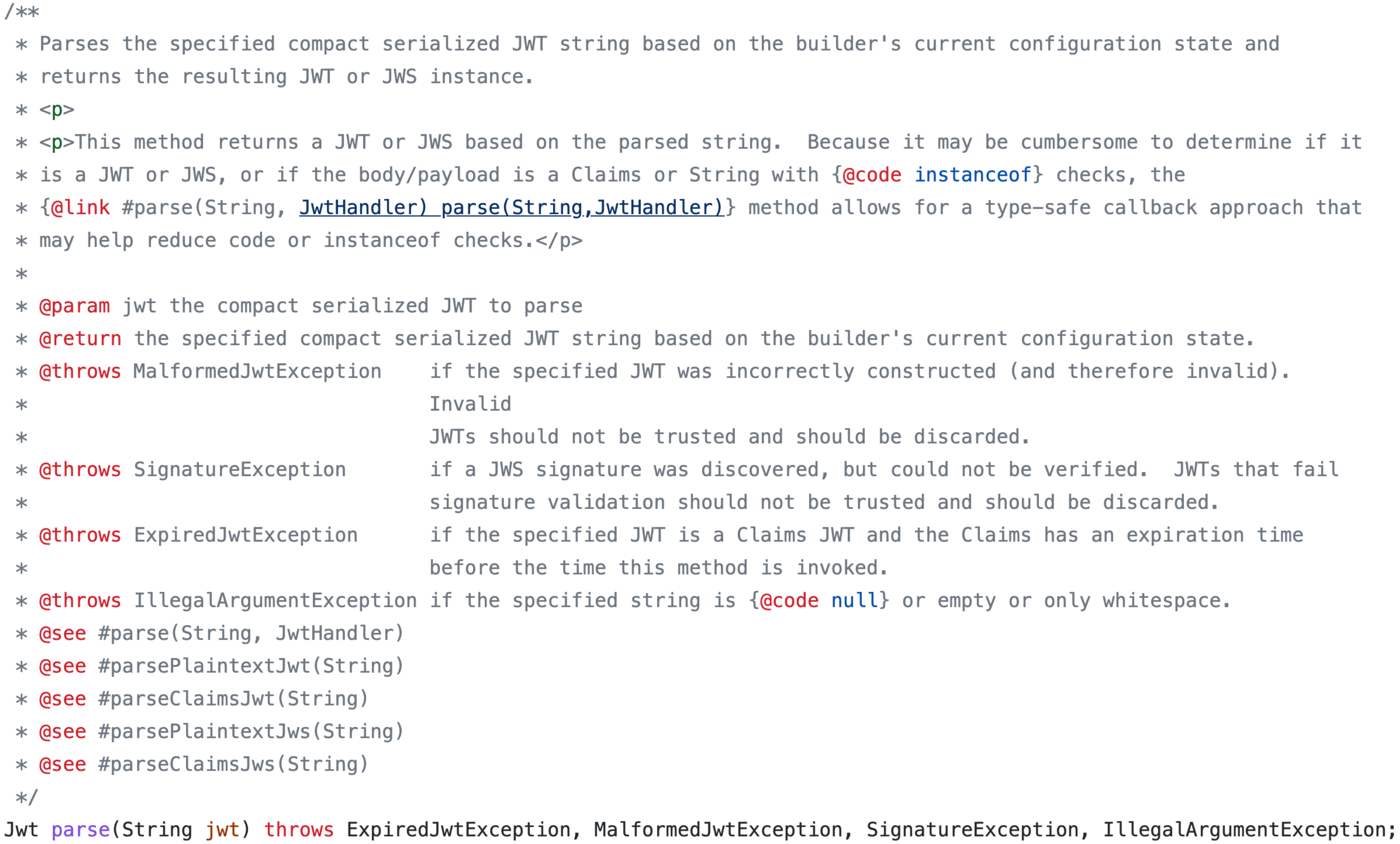
```
@@ -172,9 +172,7 @@ private static String validateToken(final String token) throws AuthenticationExc
        @SuppressWarnings("unchecked")
        private Jwt<?, Claims> authenticateToken(final String token) throws AuthenticationException {
            try {
-               Jwt<?, Claims> jwt = Jwts.parser()
-                       .setSigningKey(validationKey)
-                       .parse(token);
+               Jwt<?, Claims> jwt = Jwts.parserBuilder().setSigningKey(validationKey).build().parseClaimsJws(token);

                if (audienceClaim != null) {
                    Object object = jwt.getBody().get(audienceClaim);
```

https://github.com/apache/pulsar/pull/9172/commits/94247dac93542bbcb45fb7104f7204363aad7441



```
Jwts.parserBuilder()
    .setSigningKey(key)
    .build()
    .parse
```

```
parse(String jwt) : Jwt          JwtParser.parse(String jwt) : Jwt
parse(String jwt, JwtHandler<T> handler) : T
parseClaimsJws(String claimsJws) : Jws<Claims>
parseClaimsJwt(String claimsJwt) : Jwt<Header,Claims>
parsePlaintextJws(String plaintextJws) : Jws<String>
parsePlaintextJwt(String plaintextJwt) : Jwt<Header,…
```

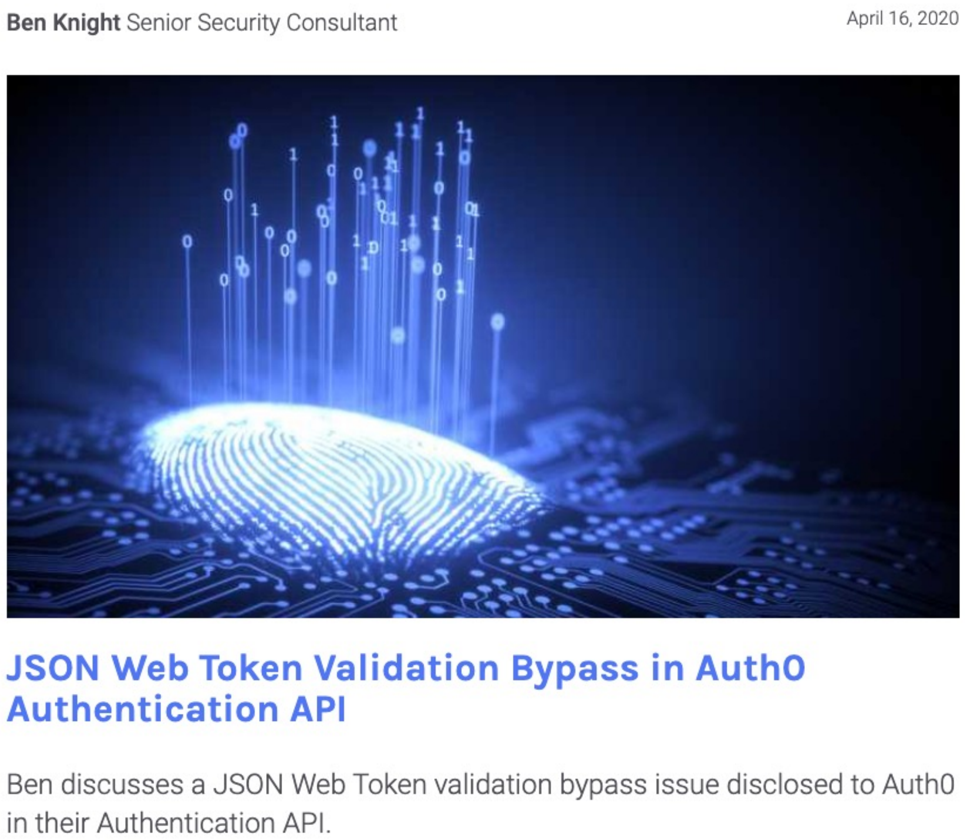


```java
/**
 * Parses the specified compact serialized JWT string based on the builder's current configuration state and
 * returns the resulting JWT or JWS instance.
 * <p>
 * <p>This method returns a JWT or JWS based on the parsed string.  Because it may be cumbersome to determine if it
 * is a JWT or JWS, or if the body/payload is a Claims or String with {@code instanceof} checks, the
 * {@link #parse(String, JwtHandler) parse(String,JwtHandler)} method allows for a type-safe callback approach that
 * may help reduce code or instanceof checks.</p>
 *
 * @param jwt the compact serialized JWT to parse
 * @return the specified compact serialized JWT string based on the builder's current configuration state.
 * @throws MalformedJwtException    if the specified JWT was incorrectly constructed (and therefore invalid).
 *                                  Invalid
 *                                  JWTs should not be trusted and should be discarded.
 * @throws SignatureException       if a JWS signature was discovered, but could not be verified.  JWTs that fail
 *                                  signature validation should not be trusted and should be discarded.
 * @throws ExpiredJwtException      if the specified JWT is a Claims JWT and the Claims has an expiration time
 *                                  before the time this method is invoked.
 * @throws IllegalArgumentException if the specified string is {@code null} or empty or only whitespace.
 * @see #parse(String, JwtHandler)
 * @see #parsePlaintextJwt(String)
 * @see #parseClaimsJwt(String)
 * @see #parsePlaintextJws(String)
 * @see #parseClaimsJws(String)
 */
Jwt parse(String jwt) throws ExpiredJwtException, MalformedJwtException, SignatureException, IllegalArgumentException;
```

https://github.com/jwtk/jjwt/blob/master/api/src/main/java/io/jsonwebtoken/JwtParser.java

Exploiting JWT vulnerabilities

> The Authentication API prevented the use of "alg: none" with a case sensitive filter. This means that simply capitalising any letter ("alg: nonE"), allowed tokens to be forged.



Ben Knight Senior Security Consultant

April 16, 2020

JSON Web Token Validation Bypass in Auth0 Authentication API

Ben discusses a JSON Web Token validation bypass issue disclosed to Auth0 in their Authentication API.

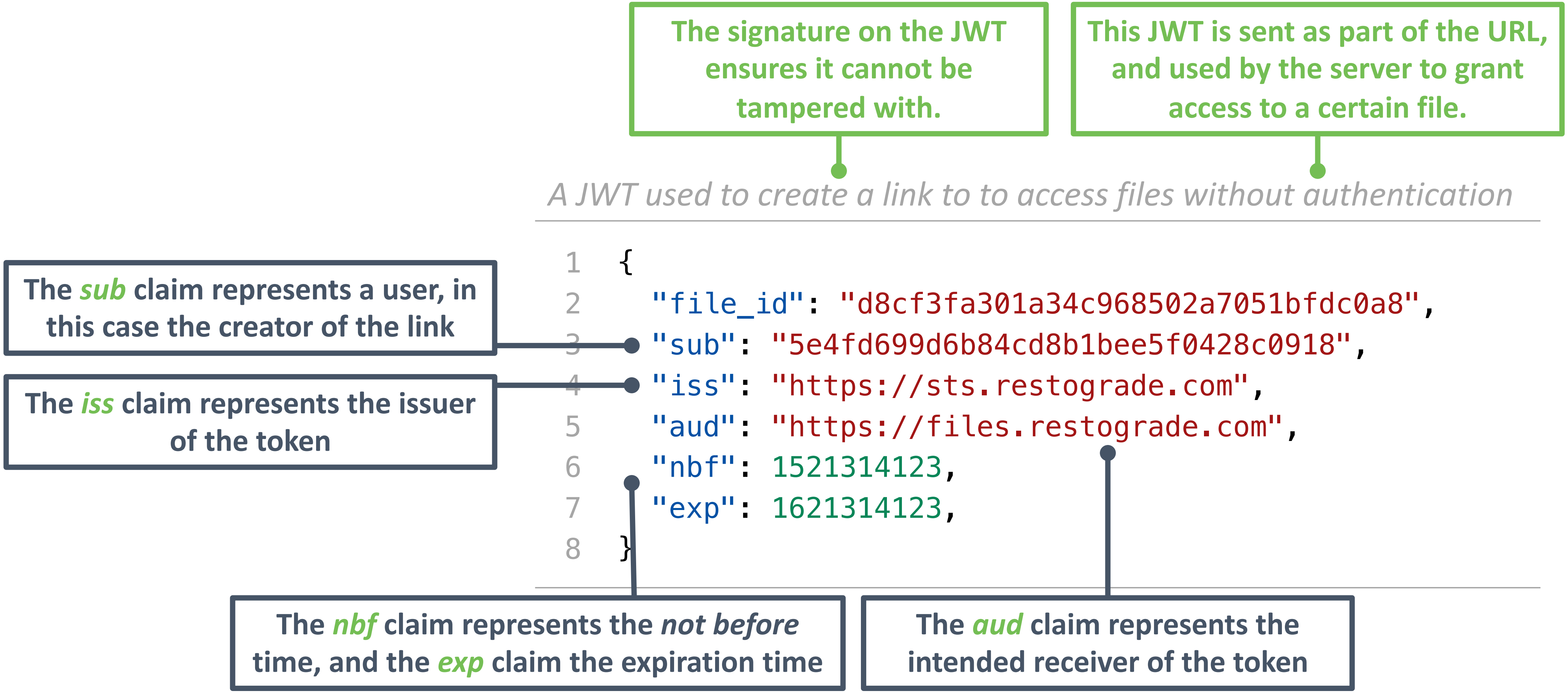
https://insomniasec.com/blog/auth0-jwt-validation-bypass

An *alg:none* token is actively malicious, and should be detected and logged as a security incident

USE WELL-DESIGNED AND UP-TO-DATE JWT LIBRARIES

Avoid using custom JWT validation code.

Rely on well-designed libraries that handle JWTs safely.



The signature on the JWT ensures it cannot be tampered with.

This JWT is sent as part of the URL, and used by the server to grant access to a certain file.

A JWT used to create a link to to access files without authentication

```
{
  "file_id": "d8cf3fa301a34c968502a7051bfdc0a8",
  "sub": "5e4fd699d6b84cd8b1bee5f0428c0918",
  "iss": "https://sts.restograde.com",
  "aud": "https://files.restograde.com",
  "nbf": 1521314123,
  "exp": 1621314123,
}
```

The ***sub*** claim represents a user, in this case the creator of the link

The ***iss*** claim represents the issuer of the token

The ***nbf*** claim represents the *not before* time, and the ***exp*** claim the expiration time

The ***aud*** claim represents the intended receiver of the token

pdr.online

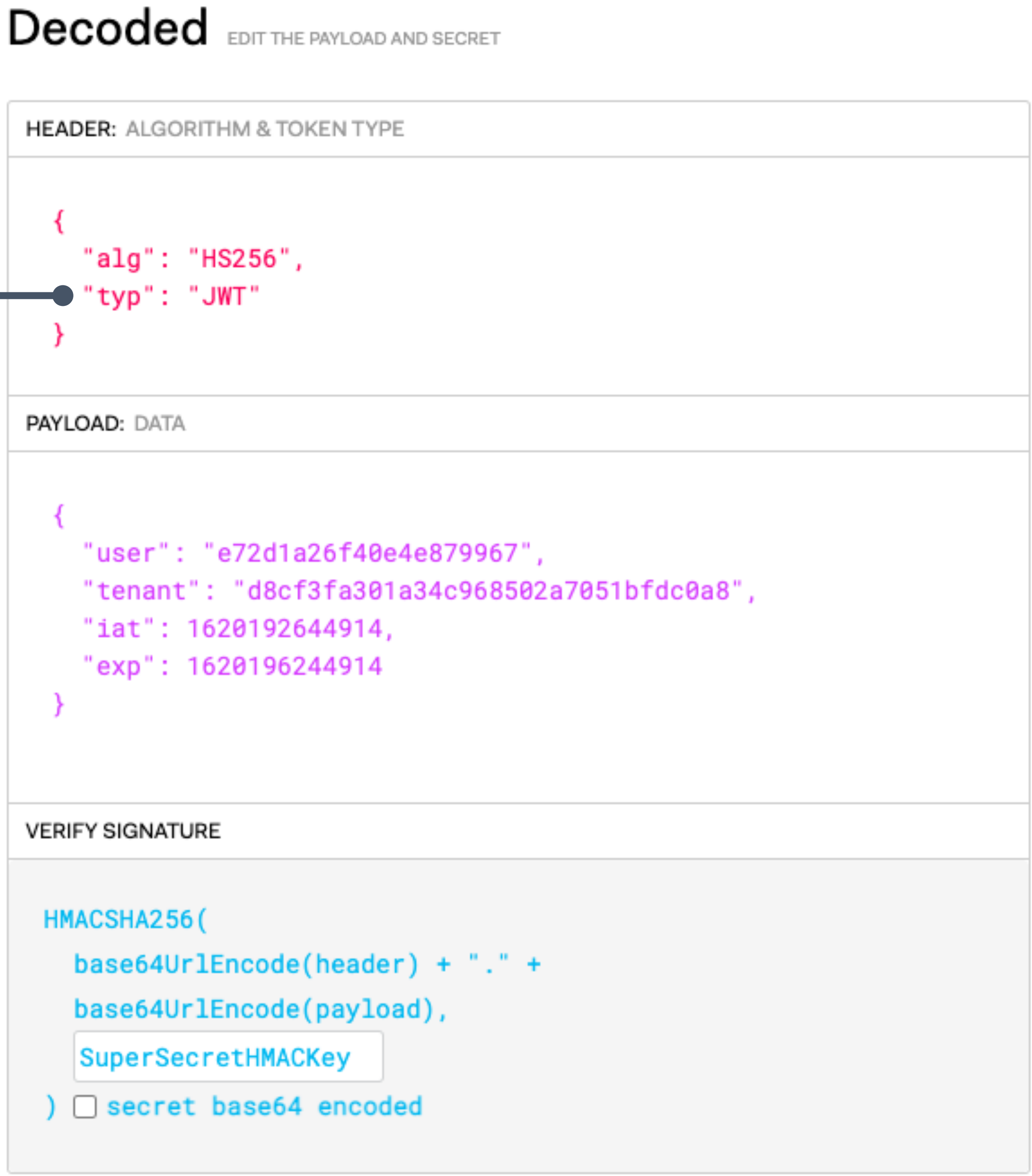
Claims in a JWT

- JWT tokens support reserved claims to hold token metadata
  - All reserved claims are optional, but it is highly recommended to use them when needed
  - The backend is responsible for checking these claims
  - Verify if your library enforces this and make sure this is handled correctly

- Checks that need to be done by the backend
  - The ***iss*** claim should match an expected issuer of JWT tokens
  - The ***aud*** claim indicates the intended target audience, which should match the backend
  - The ***sub*** claim represents a *subject*, useful for authorization decisions
  - The ***exp*** claim indicates the expiration date, which should be in the future
  - The ***nbf*** claim indicates the *not before* date, which should be in the past
  - The ***iat*** claim indicates the *issued at* date, which is mainly informative

- Apart from these claims, JWTs can also hold arbitrary claims

How should you use this JWT?



Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

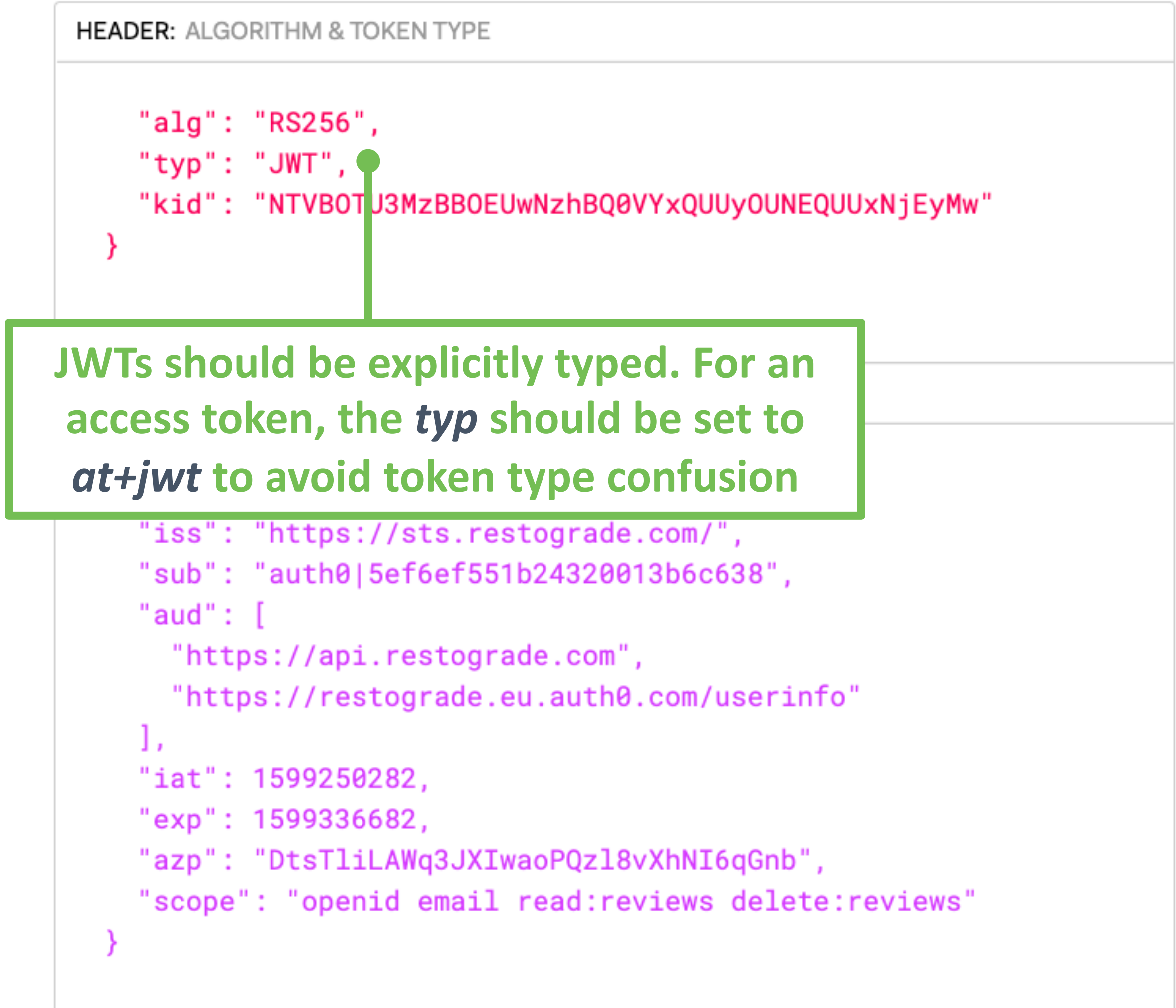
```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "user": "e72d1a26f40e4e879967",
  "tenant": "d8cf3fa301a34c968502a7051bfdc0a8",
  "iat": 1620192644914,
  "exp": 1620196244914
}
```

VERIFY SIGNATURE

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

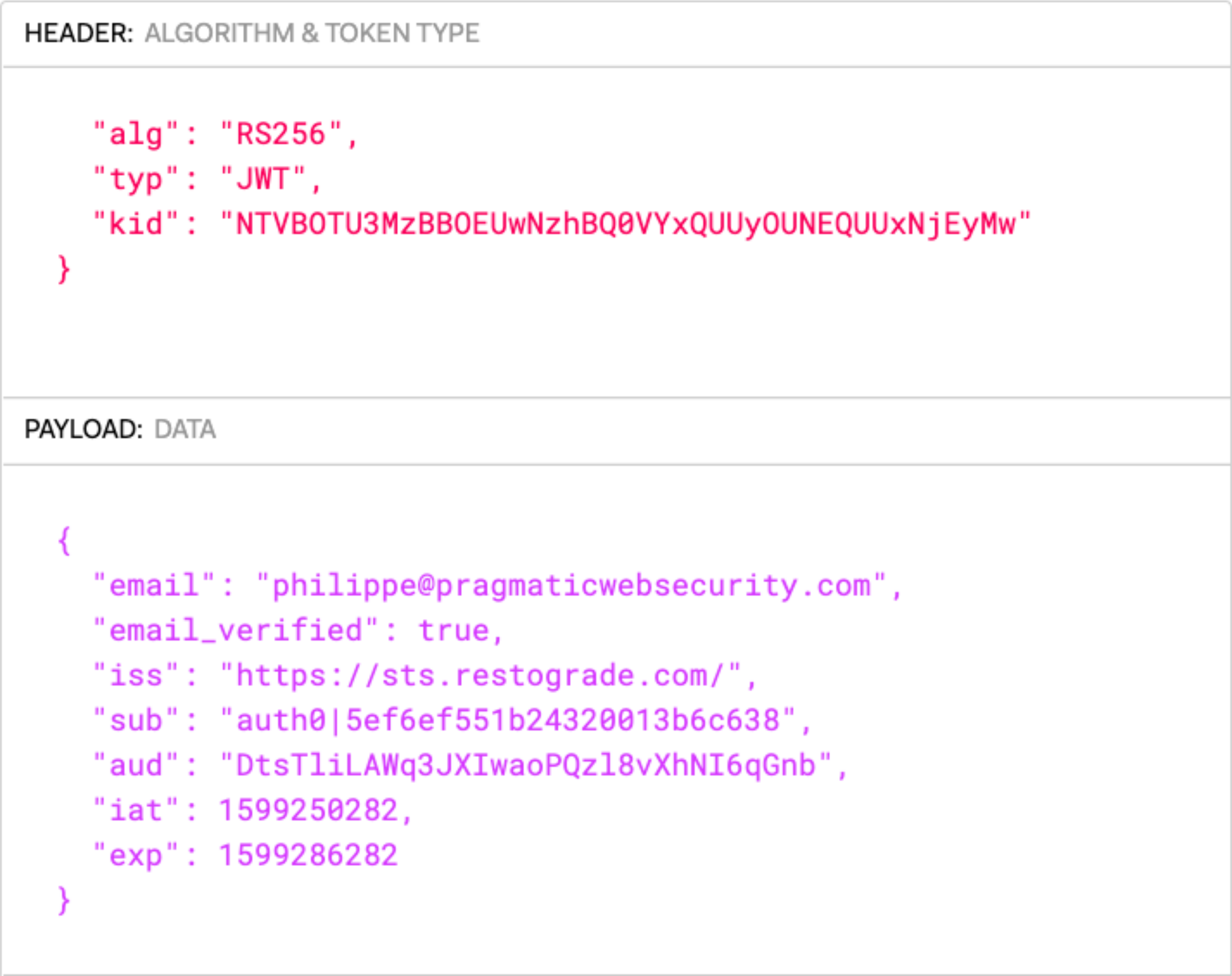


HEADER: ALGORITHM & TOKEN TYPE

```
  "alg": "RS256",
  "typ": "JWT",
  "kid": "NTVBOTU3MzBBOEUwNzhBQ0VYxQUUyOUNEQUUxNjEyMw"
}
```

JWTs should be explicitly typed. For an access token, the *typ* should be set to *at+jwt* to avoid token type confusion

```
  "iss": "https://sts.restograde.com/",
  "sub": "auth0|5ef6ef551b24320013b6c638",
  "aud": [
    "https://api.restograde.com",
    "https://restograde.eu.auth0.com/userinfo"
  ],
  "iat": 1599250282,
  "exp": 1599336682,
  "azp": "DtsTliLAWq3JXIwaoPQzl8vXhNI6qGnb",
  "scope": "openid email read:reviews delete:reviews"
}
```



HEADER: ALGORITHM & TOKEN TYPE

```
  "alg": "RS256",
  "typ": "JWT",
  "kid": "NTVBOTU3MzBBOEUwNzhBQ0VYxQUUyOUNEQUUxNjEyMw"
}
```

PAYLOAD: DATA

```
{
  "email": "philippe@pragmaticwebsecurity.com",
  "email_verified": true,
  "iss": "https://sts.restograde.com/",
  "sub": "auth0|5ef6ef551b24320013b6c638",
  "aud": "DtsTliLAWq3JXIwaoPQzl8vXhNI6qGnb",
  "iat": 1599250282,
  "exp": 1599286282
}
```

Which one is the OAuth 2.0 access token and which one is the OIDC identity token?

Explicit typing for JWTs

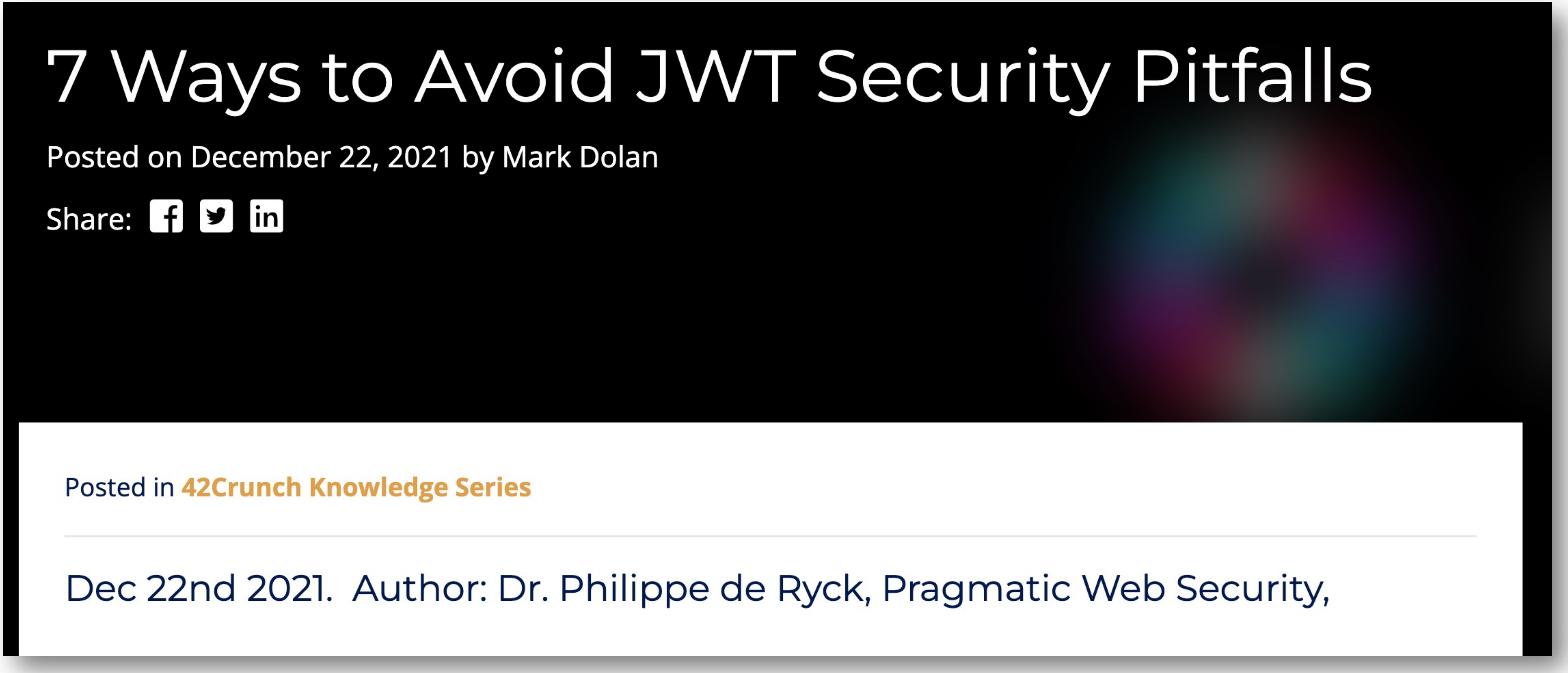
- JWTs are just a data representation and can be used for different scenarios
  - Due to reserved claims, many JWTs contain similar values
  - It can become tricky to differentiate between JWTs from the same service
    - OAuth 2.0 access tokens and OIDC identity tokens are issued by the same server
    - While both tokens contain similar claims, they serve a completely different purpose
    - An attacker could gain API access by using an identity token, which should never happen

- JWT best practices recommend explicit JWT typing
  - Instead of the generic ***JWT*** type, applications should use a custom type
  - E.g., the recommendation for OAuth 2.0 access tokens is to use ***at+jwt***

- Explicit typing is highly recommended for custom JWTs
  - Only accept JWTs with proper typing and reject everything else

JWT TESTING GUIDE

- A JWT with a modified payload (and thus an invalid signature)
- A JWT signed with the wrong key
- A JWT with ***alg: none***
- A JWT with ***alg: nOnE*** (to bypass case-sensitive checks)
- A JWT with an HMAC using the public key as the secret
- A JWT with the wrong algorithm (e.g., ***RS256*** instead of ***PS256***)
- A JWT with the wrong ***typ*** header
- A JWT with an invalid ***iss*** value
- A JWT with an invalid ***aud*** value
- A JWT with an ***exp*** timestamp in the past
- A JWT with an ***nbf*** timestamp in the future



7 Ways to Avoid JWT Security Pitfalls

Posted on December 22, 2021 by Mark Dolan

Share:

Posted in **42Crunch Knowledge Series**

Dec 22nd 2021. Author: Dr. Philippe de Ryck, Pragmatic Web Security,

https://42crunch.com/7-ways-to-avoid-jwt-pitfalls/

Follow JWT Best Current Practices

Use JWTs sensibly and write a battery of tests to verify that your code/frameworks/libraries handle JWT tokens the way you expect

What happens when 💩 goes wrong?

Compartmentalize your APIs

Many APIs combine sensitive features and mundane application logic into a single service.

Compartmentalization helps limit the impact of a vulnerability.

KEY TAKEAWAYS

1. The best authorization policy is understandable and auditable
2. Analyze your APIs for data leakage and brute force attack vectors
3. Perimeter security cannot be your only defense

Thank you!

Reach out to discuss
how I can help you with security

https://pragmaticwebsecurity.com