



# JWT SECURITY

---

DR. PHILIPPE DE RYCK

<https://PragmaticWebSecurity.com>

Internet Engineering Task Force (IETF)  
Request for Comments: 7519  
Category: Standards Track

M. Jones  
Microsoft  
J. Bradley

ISSN: 2070-1721  
Internet Engineering Task Force (IETF)  
Request for Comments: 7516  
Category: Standards Track

M. Jones  
Microsoft  
J. Hildebrand

ISSN: 2070-1721  
Internet Engineering Task Force (IETF)  
Request for Comments: 7515  
Category: Standards Track  
ISSN: 2070-1721

M. Jones  
Microsoft  
J. Bradley  
Ping Identity

Abstrac  
JSON  
clai  
are  
Web  
Encr  
sign  
(MAC  
JSON  
JSON  
for  
Web  
that  
Auth  
JSON

Internet Engineering Task Force (IETF)  
Request for Comments: 7517  
Category: Standards Track  
ISSN: 2070-1721

M. Jones  
Microsoft  
May 2015

### JSON Web Key (JWK)

Abstrac  
Abstract  
JSON  
sign  
data  
with  
Algo  
spec  
sepa

#### Abstract

A JSON Web Key (JWK) is a JavaScript Object Notation (JSON) data structure that represents a cryptographic key. This specification also defines a JWK Set JSON data structure that represents a set of JWKs. Cryptographic algorithms and identifiers for use with this specification are described in the separate JSON Web Algorithms (JWA) specification and IANA registries established by that specification.

Internet Engineering Task Force (IETF)  
Request for Comments: 7519  
Category: Standards Track

M. Jones  
Microsoft  
J. Bradley

ISSN: 2070-1721  
Internet Engineering Task Force (IETF)  
Request for Comments: 7516  
Category: Standards Track

M. Jones  
Microsoft  
J. Hildebrand

ISSN: 2070-1721  
Internet Engineering Task Force (IETF)  
Request for Comments: 7515  
Category: Standards Track  
ISSN: 2070-1721

M. Jones  
Microsoft  
J. Bradley  
Ping Identity

Abstrac

Internet Eng  
Request f  
Catego  
ISSN: 2

M. Jones  
Microsoft  
May 2015



Abstrac  
JSON  
clai  
are  
Web  
Encr  
sign  
(MAC  
JSON  
JSON  
for  
Web  
that  
Auth  
JSON

Abstrac

Abstrac

JSON  
sign  
data  
with  
Algo  
spec  
sepa

A JSON Object Notation (JSON) data structure with a public key. This specification also defines a structure that represents a set of JWKS. Cryptographic and identifiers for use with this specification are described in the separate JSON Web Algorithms (JWA) specification and IANA registries established by that specification.

# I am *Dr. Philippe De Ryck*



**Founder of Pragmatic Web Security**



**Google Developer Expert**



**Auth0 Ambassador / Expert**



**SecAppDev organizer**

## I help developers with security



**Academic-level security training**



**Hands-on in-depth online courses**



**Security advisory services**

*Not Jim* →



*Jim* →



<https://pragmaticwebsecurity.com>

1

JWT Signature Schemes

2

JWT Key Management

3

Ridiculous JWT  
vulnerabilities

4

Quiz & Summary



# JWT SIGNATURE SCHEMES



By default, JWTs are ...



**A** Base64 encoded

**B** Signed

**C** Encrypted



## Using the java-jwt library to decode a JWT

---

```
1 String token = getTokenFromUrl(); //"eyJhbGciOiJIU...";
2 try {
3     DecodedJWT jwt = JWT.decode(token);
4 }
5 catch (JWTDecodeException exception) {
6     //Invalid token
7 }
```

The *decode* function returns the claims of the JWT, but does not verify the signature

## Using the java-jwt library to verify the HMAC and decode a JWT

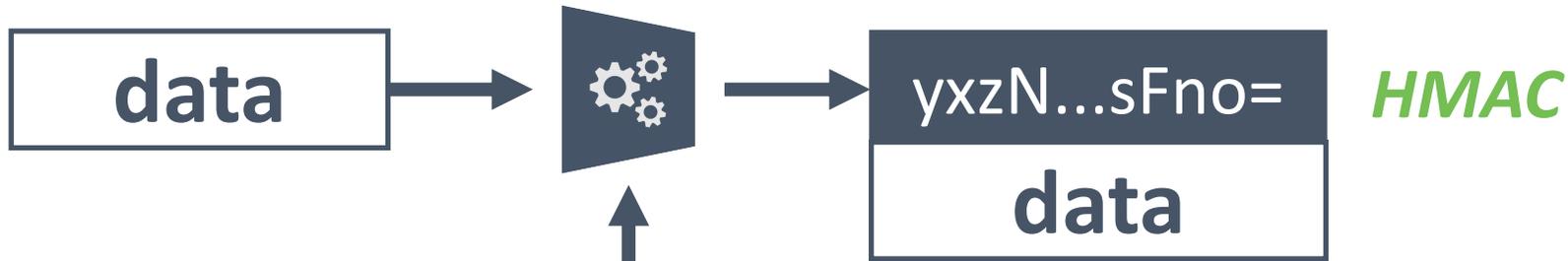
---

```
1
2 String token = getTokenFromUrl(); //"eyJhbGciOiJIU...";
3 try {
4     Algorithm algorithm = Algorithm.HMAC256("secret");
5     JWTVerifier verifier = JWT.require(algorithm).build();
6     DecodedJWT jwt = verifier.verify(token);
7 }
8 catch (JWTVerificationException exception) {
9     //Invalid signature/claims
10 }
```

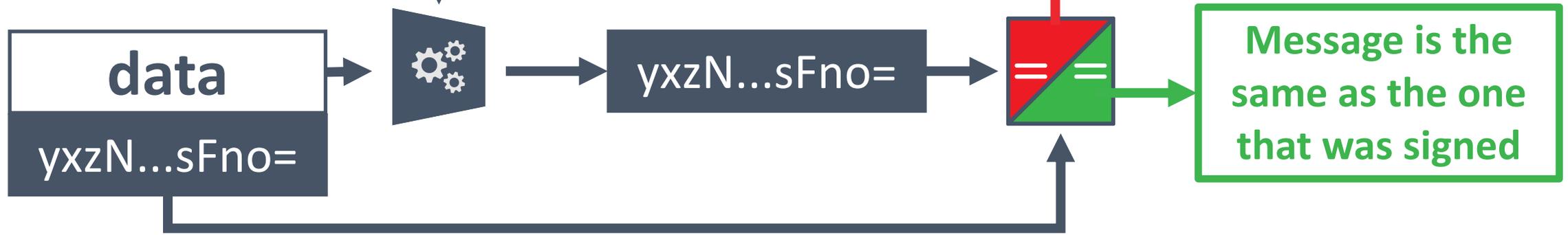
The *verify* function on a verifier will only return the claims when the signature is valid



**GENERATE HMAC**



**VERIFY HMAC**



WEB APPLICATION SECURITY

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

OCTOBER 1, 2020 - 2 MINS READ

```
02a7051bfdc0a8",  
18b1bee5f0428c0918",  
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**

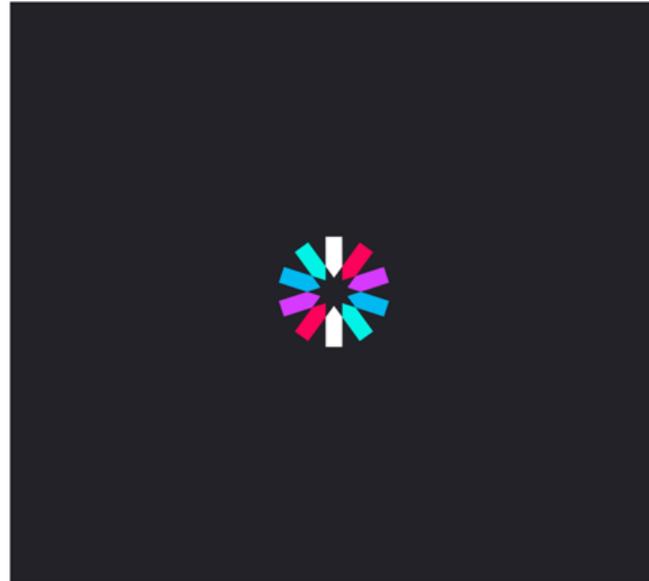
# Brute Forcing HS256 is Possible: The Importance of Using Strong Keys in Signing JWTs

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
```

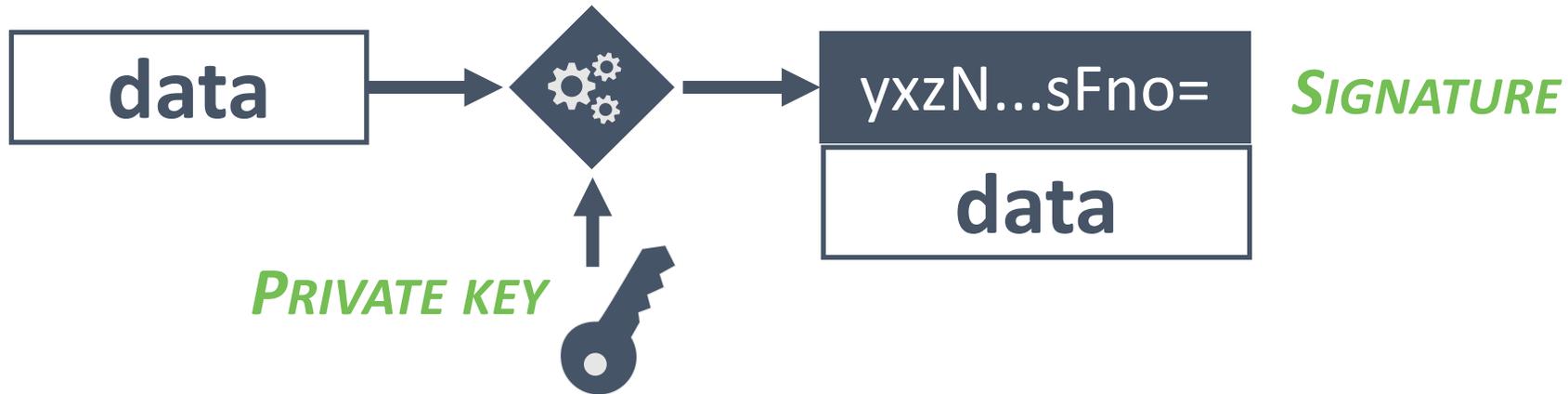
A key of the same size as the hash output (for instance, 256 bits for "HS256") or larger MUST be used with this algorithm.

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

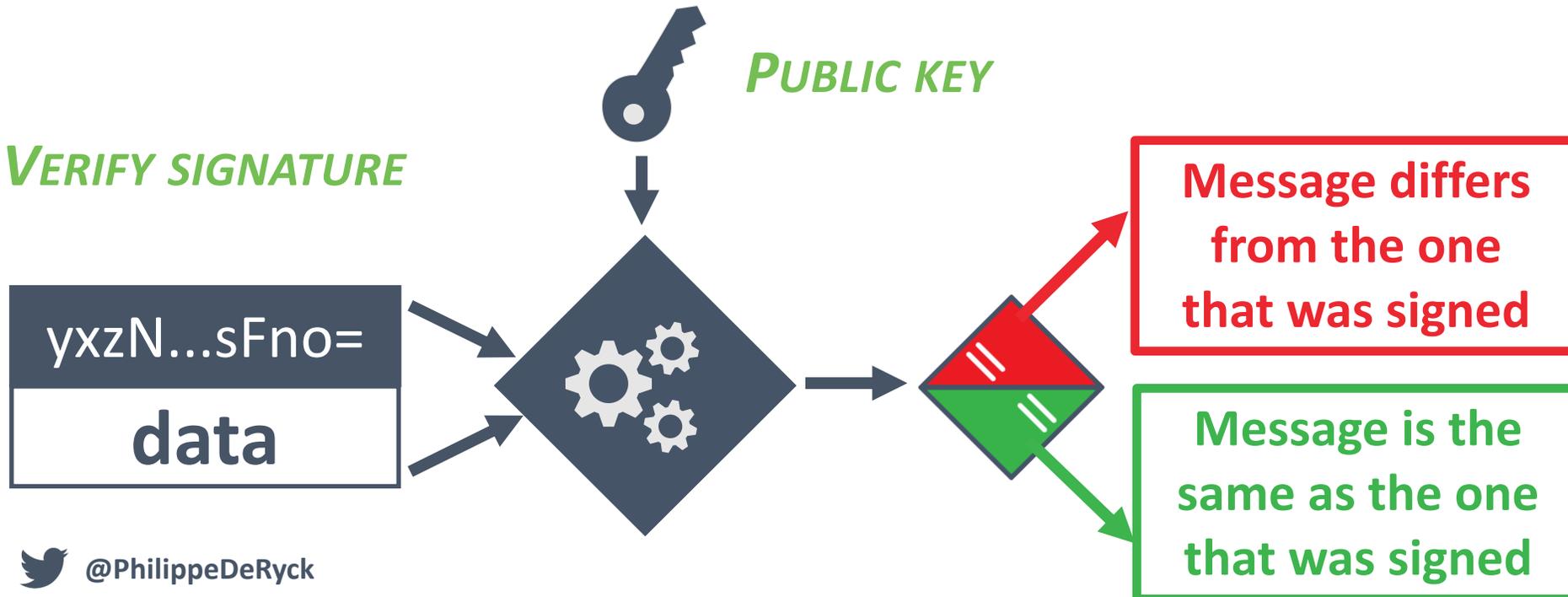


# ASYMMETRIC JWT SIGNATURES

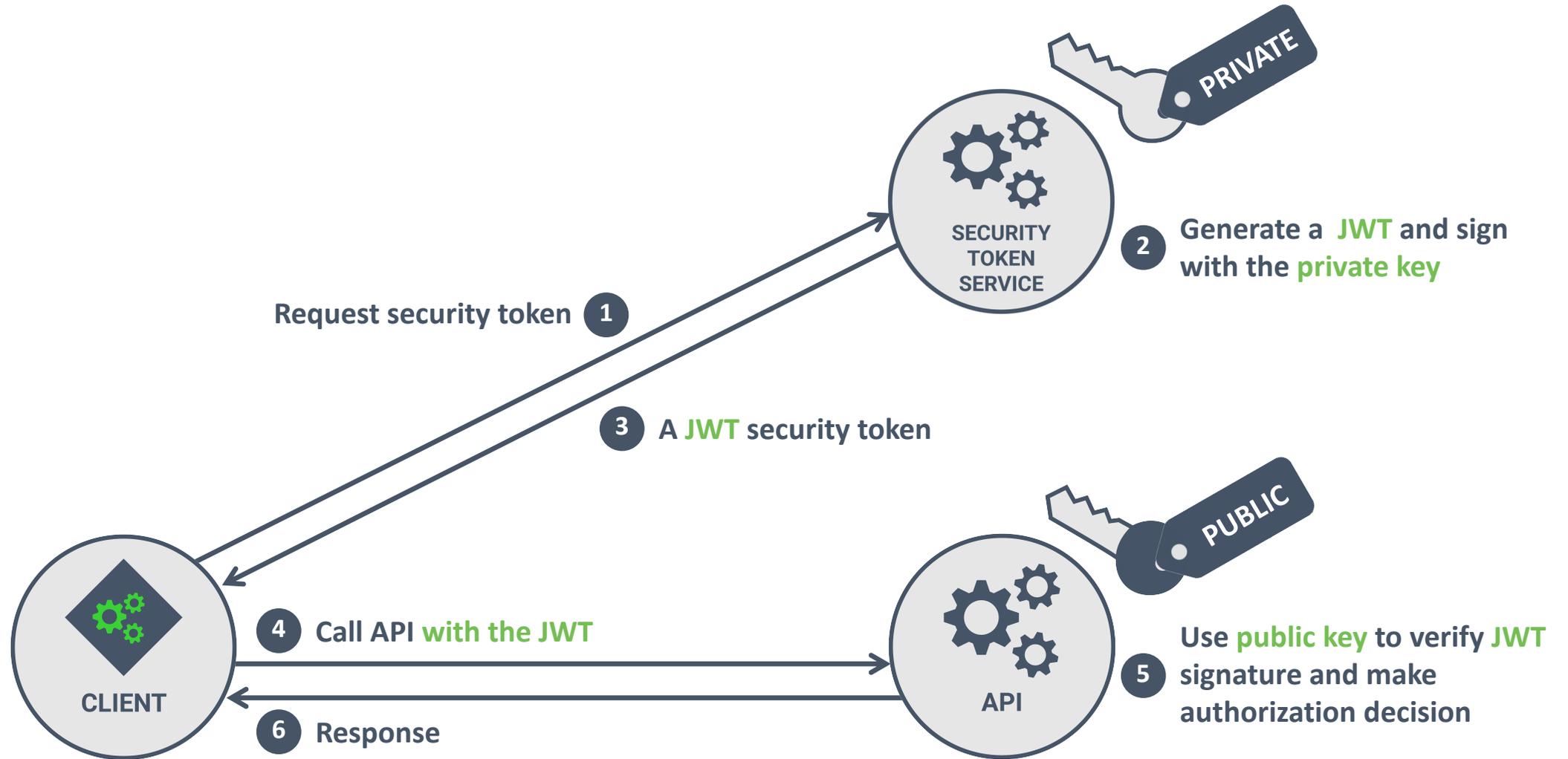
## GENERATE SIGNATURE



## VERIFY SIGNATURE



# A DISTRIBUTED JWT USE CASE



1

JWT Signature Schemes

2

JWT Key Management

3

Ridiculous JWT  
vulnerabilities

4

Quiz & Summary



# JWT KEY MANAGEMENT





Which of these key distribution mechanisms are used by JWTs?

- A** Static deployment (e.g., in an environment file)
- B** Embedding the key in a JWT
- C** Embedding the location of the key in a JWT
- D** Not using keys at all

## HEADER: ALGORITHM & TOKEN TYPE

---

```
{  
  "alg": "RS256",  
  "typ": "JWT"  
}
```

What if there are multiple possible keys?

How does the receiver know which key to use to verify the signature?



## HEADER: ALGORITHM & TOKEN TYPE

---

```
{  
  "alg": "RS256",  
  "typ": "JWT",  
  "kid": "d8cf3fa301a34c968502a7051bfdc0a8"  
}
```

---

The reserved *kid* claim represents a key identifier, helping the receiver to find the right key

Useful to retrieve a key from a centralized key store



## HEADER: ALGORITHM & TOKEN TYPE

---

```
{  
  "alg": "RS256",  
  "typ": "JWT",  
  "kid": "d8cf3fa301a34c968502a7051bfdc0a8",  
  "jku": "https://sts.restograde.com/keys.json"  
}
```

The reserved *jku* claim represents a URL pointing to a set of public keys that can be used to verify the signature

Since these keys are publicly available, the receiver can retrieve them from this location

The *kid* claim can be used to select the right key from the key set



## HEADER: ALGORITHM & TOKEN TYPE

---

```
{  
  "alg": "RS256",  
  "typ": "JWT",  
  "x5u": "https://sts.restograde.com/cert.pem"  
}
```

The reserved **x5u** claim represents the location of an X.509 certificate (TLS certificate)

Since the certificate is publicly available, the receiver can retrieve it from this location



## HEADER: ALGORITHM & TOKEN TYPE

---

```
{  
  "alg": "RS256",  
  "typ": "JWT",  
  "kid": "666",  
  "jku": "https://maliciousfood.com/evilkeyz.json"  
}
```

Without proper verification, a gullible backend will retrieve the attacker's keys and use them to verify a malicious JWT token

This setup allows an attacker to provide arbitrary JWT tokens that will be considered valid, causing a major vulnerability



# .well-known/openid-configuration

The screenshot shows a web browser window with the URL `https://sts.restograde.com/well-known/openid-configuration`. The browser's developer tools are open to the JSON tab, displaying the following configuration:

```
{  "issuer": "https://sts.restograde.com/",  "authorization_endpoint": "https://sts.restograde.com/authorize",  "token_endpoint": "https://sts.restograde.com/oauth/token",  "device_authorization_endpoint": "https://sts.restograde.com/oauth/device/code",  "userinfo_endpoint": "https://sts.restograde.com/userinfo",  "mfa_challenge_endpoint": "https://sts.restograde.com/mfa/challenge",  "jwks_uri": "https://sts.restograde.com/.well-known/jwks.json",  "registration_endpoint": "https://sts.restograde.com/oidc/register",  "revocation_endpoint": "https://sts.restograde.com/oauth/revoke",  "scopes_supported": [...],  "response_types_supported": [...],  "code_challenge_methods_supported": [...],  "response_modes_supported": [...],  "subject_types_supported": [...],  "id_token_signing_alg_values_supported": [...],  "token_endpoint_auth_methods_supported": [...],  "claims_supported": [...]
```



Browser window showing a REST client interface for the URL `https://sts.restograde.com/.well-known/jwks.json`. The interface displays the JSON response in a tree view under the "keys:" section.

```
keys:
  0:
    alg: "RS256"
    kty: "RSA"
    use: "sig"
    n: "yjFkdj-bAfynJHa8mTAmasVQiYJfwunLBvYUoPBF4tWE8sKa1nR9DnNhnwU3pi_p5PtAvXqC1m8uKdMBpimc6YBR0LY3FxJk3Yc7cJaLvsNWQm5-8iM6w3j3hxbHPUtw6QWLHm6UPmsx96a3fen402xBLlKlXZafQY62uSaiKE6Pd87p_n4"
    e: "AQAB"
    kid: "NTVB0TU3MzBB0EUwNzhBQ0VGMkQ0QUU5QTYxQUUyOUNEQUUxNjEyMw"
    x5t: "NTVB0TU3MzBB0EUwNzhBQ0VGMkQ0QUU5QTYxQUUyOUNEQUUxNjEyMw"
    x5c:
      0: "MIIDCTCCAfGgAwIBAgIJUH0n/jbd+B2BMA0GCSqGSIb3DQEBCwUAMCIxIDAeBgNVBAMTF3Jlc3RvZ3JhZGUuZXUuYy9KckdryZMCZqxVCJgl/C6csG9hSg8EXi1YTywprWdH0c2GfBTemL+nk+0C9eoLWby4p0wGmKZzpgFHTmjZg7g0E22/CWMxJFaqHhxyZEbypngpKWlnSUFi5rQ2hy5TPM/3HrxMvXLd//v+hNi/mmp6LaIY618V1I+ZUEgaYpxgq5cP7zAgMl/wQEAwIChDANBgkqhkiG9w0BAQsFAA0CAQEAXv6sxBw6W6NFKGlP0ufvmTi3ZoEgVGBNM8z92gnmkP8d7bMzUcVXKm/7Dzic0lsnZfDARIJCMYl9Rrz/n34vlkRKi9bvWSDXwfqad3zpVlGzM45PqC+e8A/qdcwHy4vbwkQQ8a/bL6bSobN8"
  1:
    alg: "RS256"
```



1

JWT Signature Schemes

2

JWT Key Management

3

Ridiculous JWT  
vulnerabilities

4

Quiz & Summary



# RIDICULOUS JWT VULNERABILITIES



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  
}
```



# Critical Vulnerabilities Affect JSON Web Token Libraries

Author:

Chris Brook

April 1, 2015 / 2:58 pm

3:30 minute read

Share this article:



```
forgedToken = sign(tokenPayload, serverRSAPublicKey, 'HS256')
```

JavaScript	VULNERABLE (?)
<input checked="" type="checkbox"/> Sign	<input checked="" type="checkbox"/> HS256
<input checked="" type="checkbox"/> Verify	<input checked="" type="checkbox"/> HS384
<input checked="" type="checkbox"/> iss check	<input checked="" type="checkbox"/> HS512
<input checked="" type="checkbox"/> sub check	<input checked="" type="checkbox"/> RS256
<input checked="" type="checkbox"/> aud check	<input checked="" type="checkbox"/> RS384
<input checked="" type="checkbox"/> exp check	<input checked="" type="checkbox"/> RS512
<input checked="" type="checkbox"/> nbf check	<input checked="" type="checkbox"/> ES256
<input checked="" type="checkbox"/> iat check	<input checked="" type="checkbox"/> ES384
	<input checked="" type="checkbox"/> ES512

PHP	VULNERABLE (?)
<input checked="" type="checkbox"/> Sign	<input checked="" type="checkbox"/> HS256
<input checked="" type="checkbox"/> Verify	<input checked="" type="checkbox"/> HS384
<input checked="" type="checkbox"/> iss check	<input checked="" type="checkbox"/> HS512
<input checked="" type="checkbox"/> sub check	<input checked="" type="checkbox"/> RS256
<input checked="" type="checkbox"/> aud check	<input checked="" type="checkbox"/> RS384
<input checked="" type="checkbox"/> exp check	<input checked="" type="checkbox"/> RS512
<input checked="" type="checkbox"/> nbf check	<input checked="" type="checkbox"/> ES256
<input checked="" type="checkbox"/> iat check	<input checked="" type="checkbox"/> ES384
<input checked="" type="checkbox"/> jti check	<input checked="" type="checkbox"/> ES512

“

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>

How Many Days Has It Been Since... x +  
howmanydayssinceajwtalgnonevuln.com

# It has been 90 days since the last alg=none JWT vulnerability.

The UK NHS COVID-19 contact tracing app for Android was accepting alg=none tokens in venue check-in QR codes. [Write-up here.](#)

Out of date? [@ me on Twitter](#)  
© 2021



# JSON Web Token Attacker

**JOSEPH** - JavaScript Object Signing and Encryption Pentesting Helper

This extension helps to test applications that use JavaScript Object Signing and Encryption, including JSON Web Tokens.

## Features

- Recognition and marking
- JWS/JWE editors
- (Semi-)Automated attacks
  - Bleichenbacher MMA
  - Key Confusion (aka Algorithm Substitution)
  - Signature Exclusion
- Base64url en-/decoder
- Easy extensibility of new attacks

**Author** Dennis Detering

**Version** 1.0.2

**Rating**  ☆☆☆☆

**Popularity** 

**Last updated** 08 February 2019

You can install BApps directly within Burp, via the BApp Store feature in the Burp Extender tool. You can also download them from here, for offline installation into Burp.



@PhilippeDeRyck

1

JWT Signature Schemes

2

JWT Key Management

3

Ridiculous JWT  
vulnerabilities

4

Quiz & Summary



# SUMMARY

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

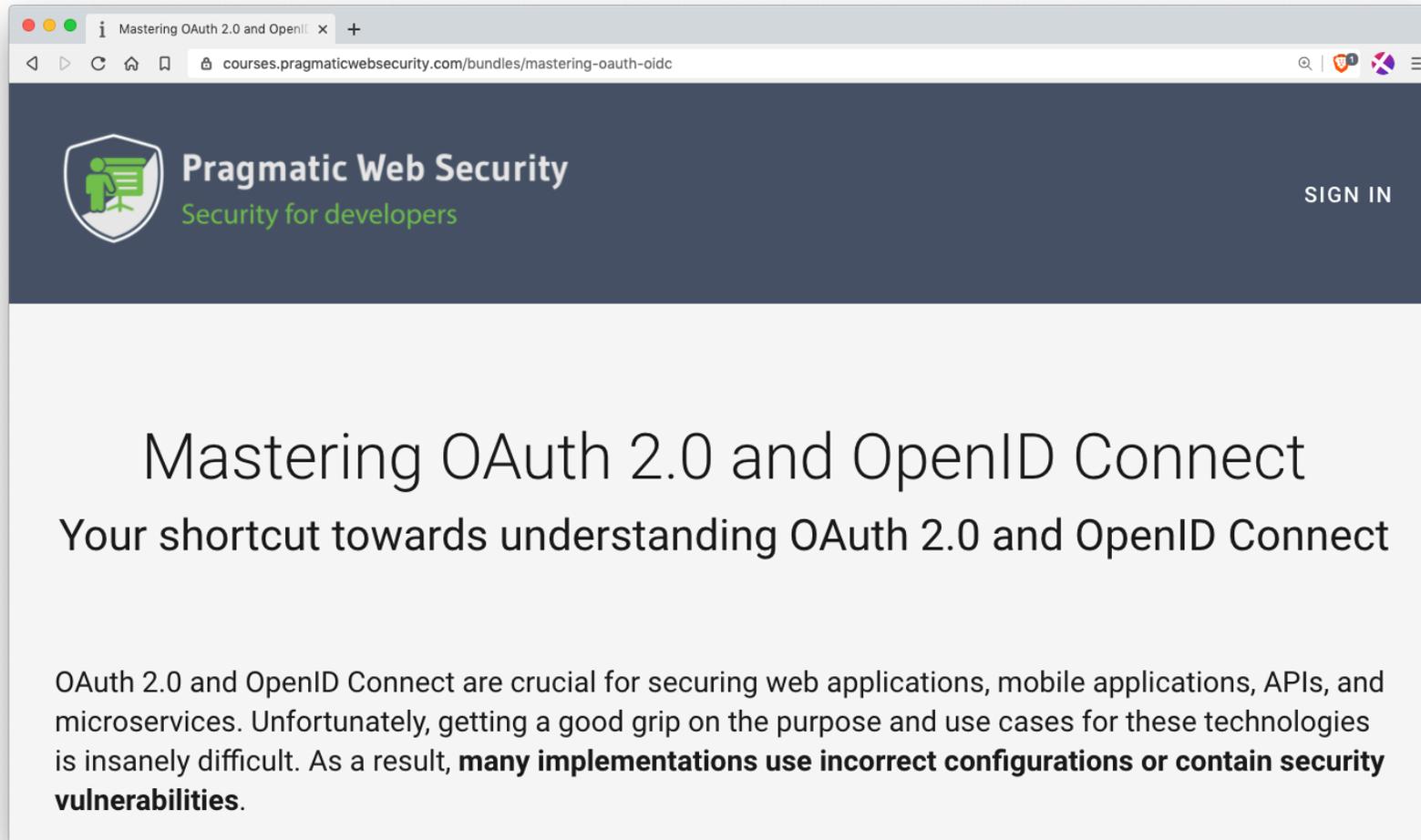
JSON Web Tokens, also known as JWTs, are URL-safe JSON-based security tokens that contain a set of claims that can be signed and/or encrypted. JWTs are being widely used and deployed as a simple security token format in numerous protocols and applications, both in the area of digital identity and in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



# BEST PRACTICES JWT SECURITY

- Choose the proper signature algorithm
  - HMACs are only useful internally in an application
  - All other scenarios should rely on asymmetric signatures
    - Make sure you have a secure way to obtain the public keys of the sender
- Follow JWT security recommendations
  - Explicitly type your JWTs
  - Use strong signature algorithms
  - Use reserved claims and their meaning
- Explicitly verify the security of the backend application
  - Libraries should be actively supported and up to date
  - JWTs with *none* signatures should be rejected case-insensitively
  - JWTs with invalid signatures should be rejected

# This online course condenses dozens of confusing specs into a crystal-clear academic-level learning experience



The screenshot shows a web browser window with the URL `courses.pragmaticwebsecurity.com/bundles/mastering-oauth-oidc`. The page header features the Pragmatic Web Security logo (a shield with a person and a screen) and the text "Pragmatic Web Security" and "Security for developers". A "SIGN IN" button is visible in the top right. The main content area has the title "Mastering OAuth 2.0 and OpenID Connect" and the subtitle "Your shortcut towards understanding OAuth 2.0 and OpenID Connect". Below this, a paragraph states: "OAuth 2.0 and OpenID Connect are crucial for securing web applications, mobile applications, APIs, and microservices. Unfortunately, getting a good grip on the purpose and use cases for these technologies is insanely difficult. As a result, **many implementations use incorrect configurations or contain security vulnerabilities.**"

**25% discount**

*Use coupon code*

**VIRTUAL\_OWASP**

*Offer expires Feb 25th, 2021*



# Thank you for watching!

Connect on social media for more  
in-depth security content

← *Still not Jim*



**@PhilippeDeRyck**



**/in/PhilippeDeRyck**