



# THE PARTS OF JWT SECURITY NOBODY TALKS ABOUT

---

DR. PHILIPPE DE RYCK

<https://PragmaticWebSecurity.com>

# DR. PHILIPPE DE RYCK

- Deep understanding of the web security landscape
- Google Developer Expert (not employed by Google)
- Course curator of the  **SecAppDev** course  
(<https://secappdev.org>)



## Pragmatic Web Security

High-quality security training for developers and managers

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

Consulting services on security, OAuth 2.0, OpenID Connect, ...

@PHILIPPEDERYCK

[HTTPS://PRAGMATICWEBSECURITY.COM](https://pragmaticwebsecurity.com)

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ  
zdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6I  
BoaWxpcHB1IER1IFJ5Y2siLCJyb2x1cyI6InVzZXIgc  
mVzdGF1cmFudG93bmVyIiwiaWF0IjoxNTE2MjM5MDI  
yfQ.KPjhyE9oi83uehgw6Lm\_0yAZzRuJhcUqXETD  
2AIrF2A



# Encoded

PASTE A TOKEN HERE

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpXVCJ9.eyJzdGF1cmFudG93bmVyaWwifQ.KPjhyE9oi83uehgw6Lm\_0yAZzRuJhcUqXETD2AIrF2A

Base64-encoded

Contains a set of claims

Integrity-protected with a signature

# 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
```



# JWT IS A PART OF THE JOSE FRAMEWORK

- JOSE stands for JavaScript Object Signing and Encryption
  - A collection of specifications to securely transfer claims between parties
  - JWT is the mechanism to represent claims
  - JWTs are augmented with signatures and encryption to offer additional security
- The JOSE specifications support different serializations of a data object
  - **Compact serialization** generates URL-safe strings of data
    - *JWTs always use the compact serialization*
    - This type of serialization is also mandated for tokens used in the OpenID Connect protocol
  - The alternative **JSON serialization** is intended for use outside of web requests / responses
    - It is not optimized for compactness
    - It also supports the specification of multiple signatures using different keys and algorithms

# JWTs ARE A WAY TO REPRESENT CLAIMS

- Claims are key value pairs in the payload of the JWT
  - Apart from a few reserved claims, the issuer can include arbitrary claims
- The compact serialization mandates that the JWT is base64-encoded
  - Base64 encoding makes data safe to use in HTTP requests and responses
  - It looks like scrambled data, but it is **only an encoding**
  - Anyone can decode the payload of a JWT

```
> atob("eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6ImlBoaWxpchB1IERlIFJ5Y2siLCJyb2xlcyl6InVzZXIgcmlVzdGF1cmFudG93bmVyIiwiaWF0IjoxNTE2MjM5MDIy")  
< '{"sub":"1234567890","name":"Philippe De Ryck","roles":"user restaurantowner","iat":1516239022}'
```



# Stop using JWT for sessions

13 Jun 2016

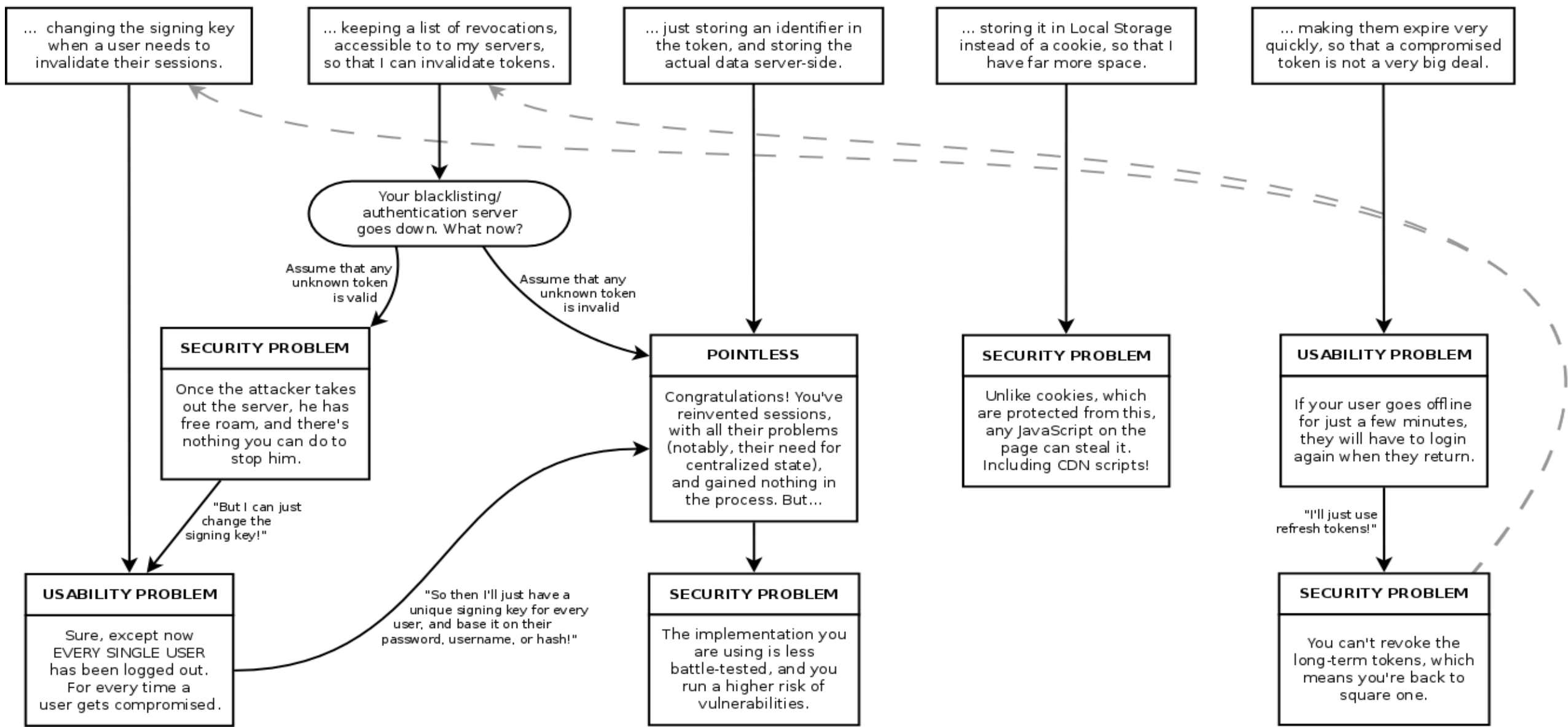
**Update - June 19, 2016:** A lot of people have been suggesting the same "solutions" to the problems below, but none of them are practical. I've [published a new post](#) with a slightly sarcastic flowchart - please have a look at it before suggesting a solution.

“ This article does *not* argue that you should *never* use JWT - just that it isn't suitable as a session mechanism, and that it is dangerous to use it like that. Valid usecases *do* exist for them, in other areas. ”

# Stop using JWT for sessions, part 2

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

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



# BEST PRACTICES AND LIMITATIONS

- JWTs are a mechanism to exchange claims in a trusted manner
  - Allows a single server to send out data, receive it back and verify its integrity
  - Allows different parties to exchange claims with integrity protection
- The main purpose of JWT is to exchange such claims
  - OpenID Connect is a good example of the use of a JWT to exchange claims
  - OAuth 2.0 architectures use JWTs to relay authorization information in the backend
- Using JWTs for session data is possible, if you address a couple of drawbacks
  - Think about how to handle revocation, and build your architecture to support it
  - Carefully think about which data needs to be stored in a JWT
    - Find the right balance between limiting the size and optimizing server-side processing



# Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

Valid according to  
the spec!

PAYLOAD: DATA

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



# VULNERABILITIES IN COMMON JWT LIBRARIES

- In 2015, people discovered two major vulnerabilities in JWT libraries
  - Some libraries accepted **none** as a valid signing algorithm
  - Some libraries got confused between symmetric and asymmetric signatures
- Accepting **none** as a valid signing algorithm
  - An attacker can craft his own JWT token without worrying about the signature
  - The library would perform its checks, note the **none** and simply decode the JWT
  - Using the data for sensitive operations resulted in authorization bypass attacks
- Tricking the library into mistaking asymmetric signatures for HMACs
  - The attacker can forge a token and add an HMAC using the server's public key as secret
  - The backend expects an asymmetric signature, and calls the library with the public key
  - The confused library verifies the HMAC with the public key as shared secret

```
verify(clientToken, serverRSAPublicKey)
```



draft-ietf-oauth-jwt-bcp-07 - JS x +

tools.ietf.org/html/draft-ietf-oauth-jwt-bcp-07#page-5

[[Docs](#)] [[txt](#)|[pdf](#)|[xml](#)|[html](#)] [[Tracker](#)] [[WG](#)] [[Email](#)] [[Diff1](#)] [[Diff2](#)] [[Nits](#)]

Versions: ([draft-sheffer-oauth-jwt-bcp](#)) [00](#) [01](#)  
[02](#) [03](#) [04](#) [05](#) [06](#) [07](#)

|   |                  |
|---|------------------|
| OAuth Working Group                             | Y. Sheffer       |
| Internet-Draft                                  | Intuit           |
| Updates: RFC <a href="#">7519</a> (if approved) | D. Hardt         |
| Intended status: Best Current Practice          |                  |
| Expires: April 15, 2020                         | M. Jones         |
|   | Microsoft        |
|   | October 13, 2019 |

**JSON Web Token Best Current Practices**  
**draft-ietf-oauth-jwt-bcp-07**

Abstract

JSON Web Tokens, also known as JWTs, are URL-safe JSON-based security



# Encoded

PASTE A TOKEN HERE

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpXVCJ9.eyJzdGF1cmFudG93bmVyaWwifQ.KPjhyE9oi83uehgw6Lm\_0yAZzRuJhcUqXETD2AIrF2A

Base64-encoded

Contains a set of claims

Integrity-protected with a signature

# Decoded

EDIT THE PAYLOAD AND SECRET

## HEADER: ALGORITHM & TOKEN TYPE

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

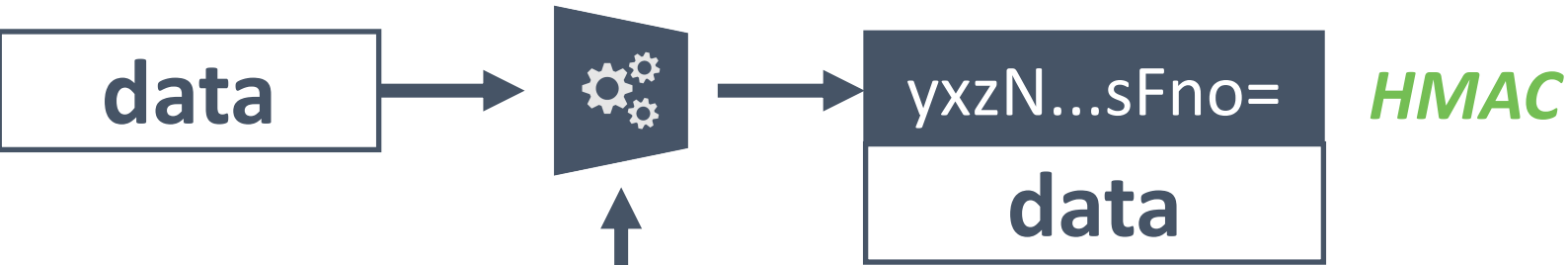
## PAYLOAD: DATA

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

## VERIFY SIGNATURE

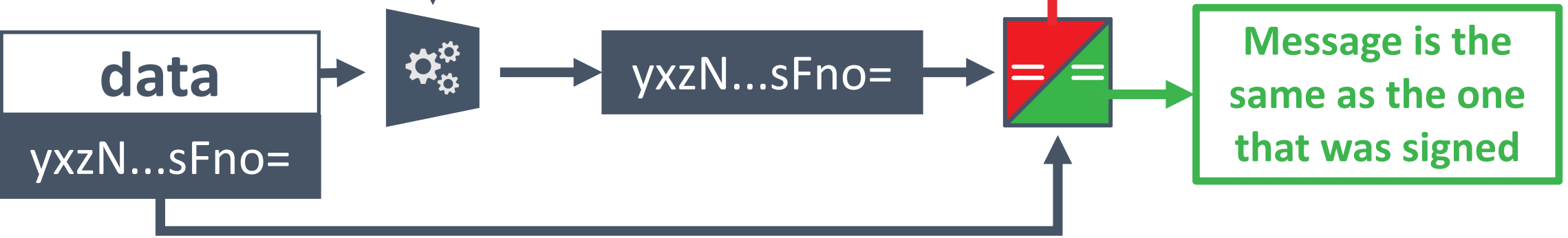
```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  SuperSecretHMACKey
) ☐ secret base64 encoded
```

GENERATE HMAC




SECRET KEY

VERIFY HMAC



```
HMACSHA256(  
    base64UrlEncode(header) + "." +  
    base64UrlEncode(payload),  
    SuperSecretHMACKey  
)
```

 secret base64 encoded

HMACSHA256 (

base64UrlEncode(header) + "." +

base64UrlEncode(payload),

X7j1J1ygedeTzcI01HPxwU8XkpSCTYL4cjfF2hAj

) ☐ secret base64 encoded



# 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

March 23, 2017



@PhilippeDeRyck

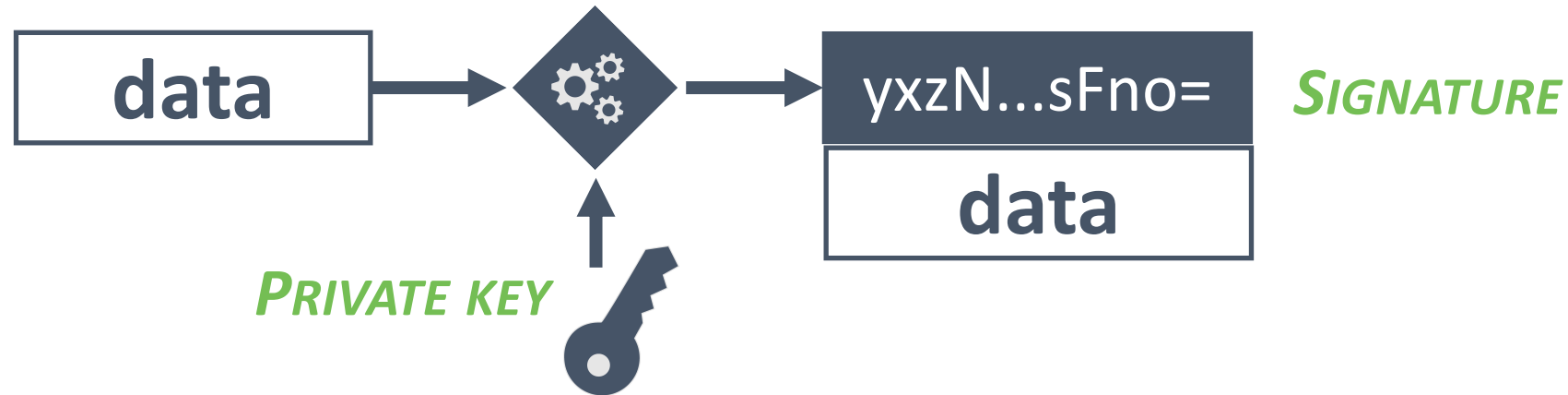
**A key of the same size as the hash output  
(e.g., 256 bits for "HS256") or larger  
MUST be used with this algorithm.**

*RFC 7518 - JSON Web Algorithms (JWA)*



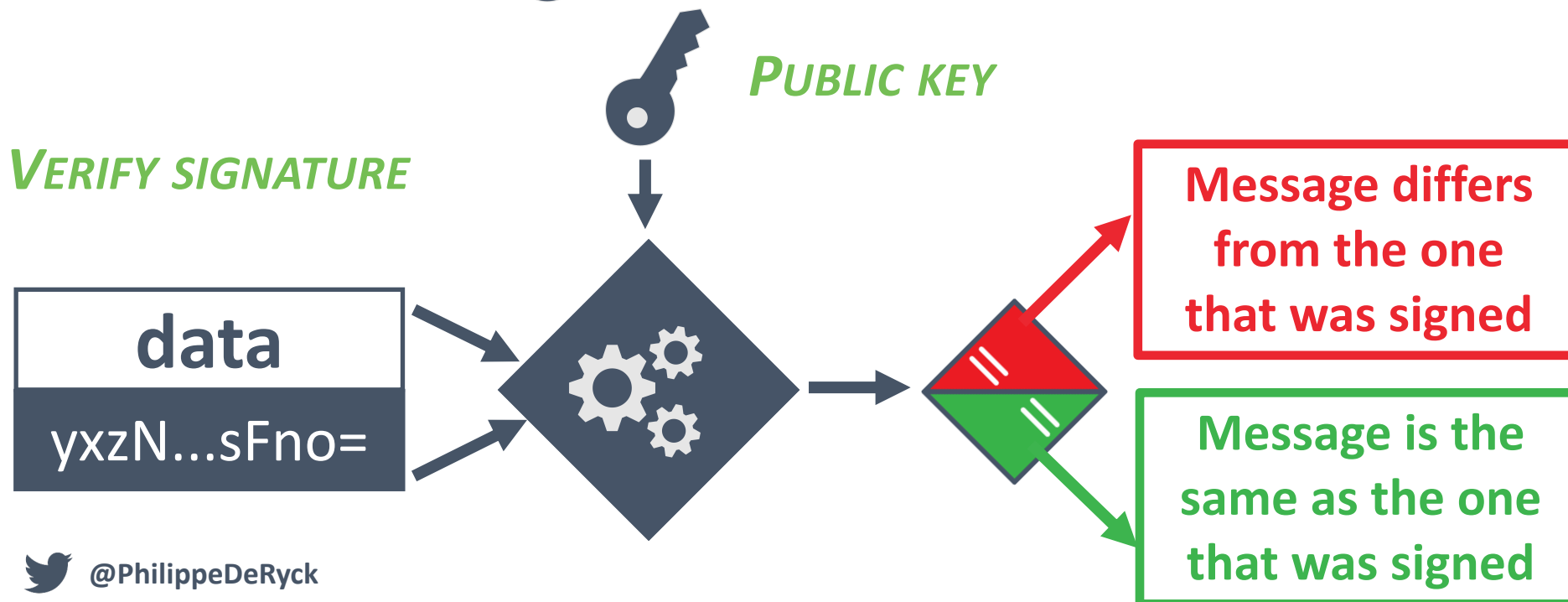
# ASYMMETRIC JWT SIGNATURES

## GENERATE SIGNATURE



## PUBLIC KEY

## VERIFY SIGNATURE



# JWT SIGNATURES

- JWTs support both symmetric HMACs and asymmetric signatures
  - Symmetric HMACs depend on a shared secret key
  - Asymmetric are digital signatures that depend on a public/private key pair
- Symmetric HMACs are useful to use within a single trust zone
  - Backend service storing claims in a JWT for use within the application
  - Not the right choice when other (internal) services are involved
    - *Never ever share your secret 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



# JWT SIGNATURES



*HMACs are only useful in an isolated application*

*The use of HMACs requires careful attention for key generation*

*Distributed scenarios should **always** use asymmetric signatures*



# KEY MANAGEMENT FOR VERIFYING SIGNATURES

- To verify a signed JWT, the receiver needs the proper cryptographic key
  - For symmetric keys, this is the same key as used by the creator of the JWT
  - For asymmetric keys, this is the public key of the creator of the JWT
- Key management is crucial to ensure the proper use of JWT tokens
  - Cryptographic keys need to be rotated frequently to ensure their security
  - When rotating keys, different tokens will be signed with different keys
  - Hardcoding keys is simple, but a really bad idea
- Key management for JWTs comes in various different flavors
  - Simplest mechanism is to use a key identifier to point to the right key
  - Complex setups can even exchange keys using the JWT data structure



## HEADER: ALGORITHM & TOKEN TYPE

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

Identify a key known by  
the receiver

## HEADER: ALGORITHM & TOKEN TYPE

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

**Provide a URL  
containing a set of keys**

```
1 // Library: com.nimbusds.nimbus-jose-jwt
2 JWSHeader header = new JWSHeader.Builder(JWSAlgorithm.RS256)
3     .jwkURL(new URI("https://restograde.com/jwks.json"))
4     .keyID(keyID)
5     .build();
6
7 JWTClaimsSet claimsSet = new JWTClaimsSet.Builder()
8     .issueTime(new Date())
9     .issuer("https://restograde.com")
10    .claim("username", "philippe")
11    .build();
12
13 JWSSigner signer = new RSASSASigner(privateKey);
14 SignedJWT jwt = new SignedJWT(header, claimsSet);
15 jwt.sign(signer);
16 result = jwt.serialize();
```





## HEADER: ALGORITHM & TOKEN TYPE

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

Provide a .X509  
certificate with a key

# KEY IDENTIFICATION IN JWTs

- Asymmetric algorithms use a key pair
  - The private key is used to generate a signature and is kept secret
  - The public key is used to verify a signature and can be publicly known
- Simple approach uses the *kid* parameter to identify the public key
  - The parameter could include a fingerprint of the public key
  - Of course, this still requires the receiver to obtain the public key one way or another
- But the public key is public, so it can also be included as part of the JWT token
  - The specification supports this through various parameters
  - The set of parameters are *jku*, *jwk*, *kid*, *x5u*, and *x5c*



```
1 // Library: com.nimbusds.nimbus-jose-jwt
2 JWSHeader header = new JWSHeader.Builder(JWSAlgorithm.RS256)
3     .jwkURL(new URI("https://restograde.com/jwks.json"))
4     .keyID(keyID)
5     .build();
6
7 JWTClaimsSet claimsSet = new JWTClaimsSet.Builder()
8     .issueTime(new Date())
9     .issuer("restograde.com")
10    .claim("username", "philippe")
11    .build();
12
13 JWSSigner signer = new RSASSASigner(privateKey);
14 SignedJWT jwt = new SignedJWT(header, claimsSet);
15 jwt.sign(signer);
16 result = jwt.serialize();
```



## HEADER: ALGORITHM & TOKEN TYPE

```
{  
  "alg": "RS256",  
  "typ": "JWT",  
  "kid": "KjrsfCS8cb9kJFkingu6FdCqogWXURu-rLTbbyrL7jo",  
  "jku": "https://evil.example.com/jwks.json"  
}
```



# TRUSTING THE KEY

- Trusting the key which is embedded in the JWT is a difficult problem
  - Your application should restrict which keys it accepts
  - The attacker can always provide a signed JWT containing a valid key
- Approving specific keys
  - The application can identify a set of valid keys using their fingerprints
  - Dynamic whitelisting can be done using backchannel requests to load keys
    - Only load keys from trusted sources
- Limiting valid sources of the keys
  - Dynamic JWK URLs can be whitelisted per valid domain (and path if possible)
  - Certificate-based keys should be checked for a valid **Common Name** in the certificate



.well-known/openid-configuration

pragmaticwebsecurity.eu.auth0.com X

https://pragmaticwebsecurity.eu.auth0.com/ 133%

JSON Raw Data Headers

Save Copy Collapse All Expand All Filter JSON

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





```
1 String domain = "pragmaticwebsecurity.eu.auth0.com";
2
3 // Get the proper key material
4 DecodedJWT insecureJwt = JWT.decode(identityToken);
5 String kid = insecureJwt.getKeyId();
6 Jwk jwk = getProvider(domain).get(kid);
7
8 // Verify the signature on the token
9 Algorithm algorithm = Algorithm.RSA256((RSAPublicKey)
10                                     jwk.getPublicKey(), null);
11 JWTVerifier verifier = JWT.require(algorithm)
12     .withAudience(clientId)
13     .withIssuer(issuer)
14     .withClaim("nonce", session.getAttribute("oidc.nonce").toString())
15     .build();
16 DecodedJWT jwt = verifier.verify(identityToken);
17
18 logger.info("Successfully verified identity token");
19 logger.debug(identityToken);
```



# JWT KEY MANAGEMENT



*Signing keys need to be rotated to ensure security*

*Key information is part of the header, but is **untrusted***

*Use an out-of-band channel to distribute public signing keys*



eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX  
VCJ9.eyJ1c2VyIjoiaUGhpbG1wcGUgRGU  
gUn1jayIsImFjY291bnQiOiJCRTk1MDg  
3MTE3ODc4MDc2IiwiaY3VycmVuY3kiOiJ  
FVVIiLCJiYWxhbmNlIjozMDAwMDAwfQ.  
TxXAP80sH21uwsCPfyHmnNjKCac9MpdL  
HIzWmXn45Nc



```
{  
  "user": "Philippe De Ryck",  
  "account": "BE95087117878076",  
  "currency": "EUR",  
  "balance": 1000000  
}
```



Totally real data

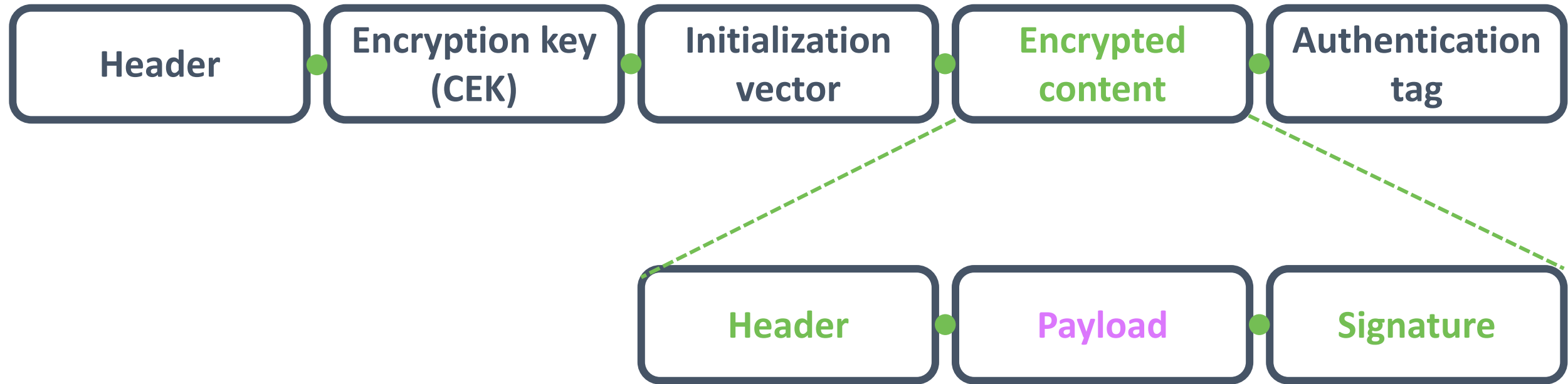


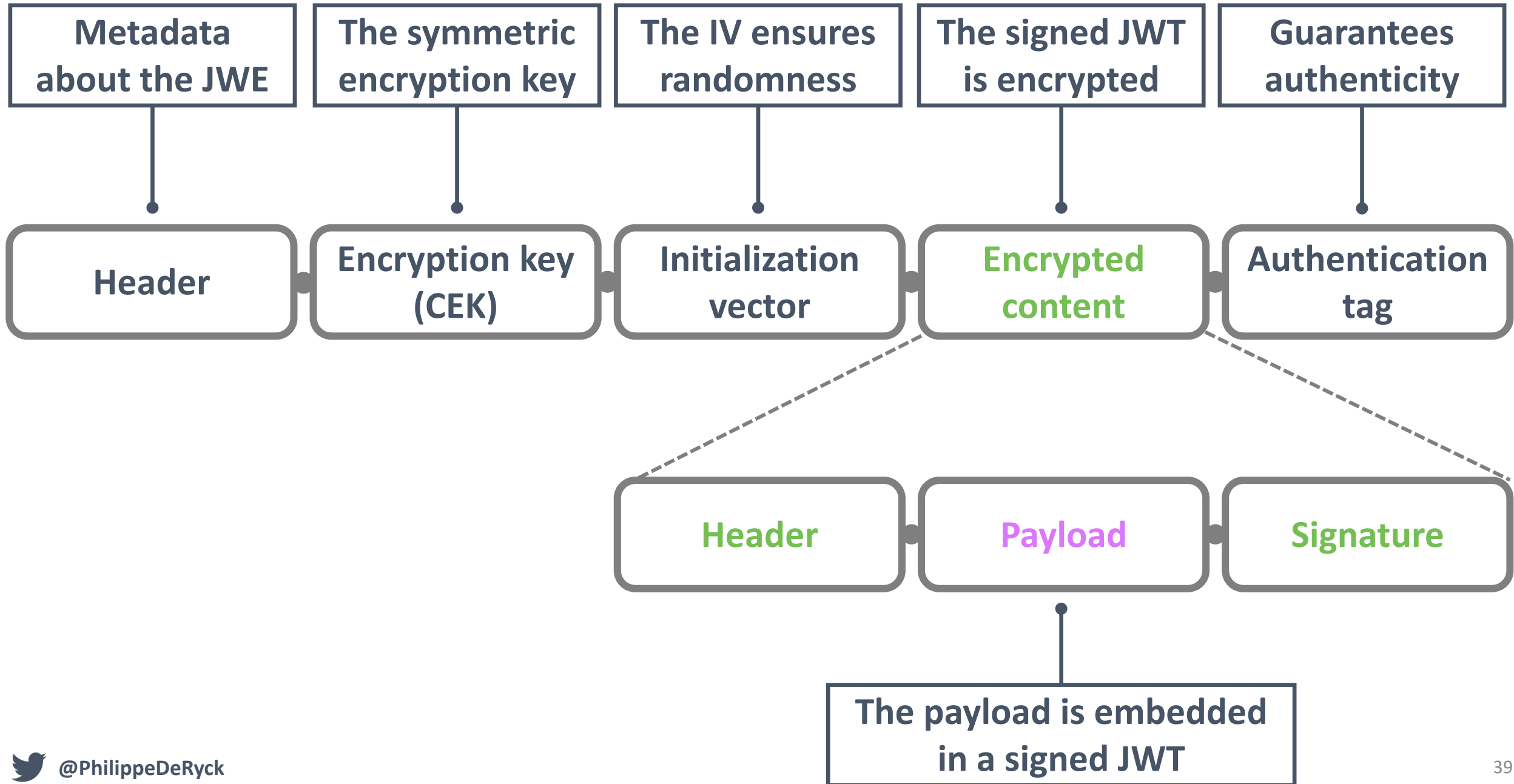
# JSON WEB ENCRYPTION (JWE)

RFC 7516

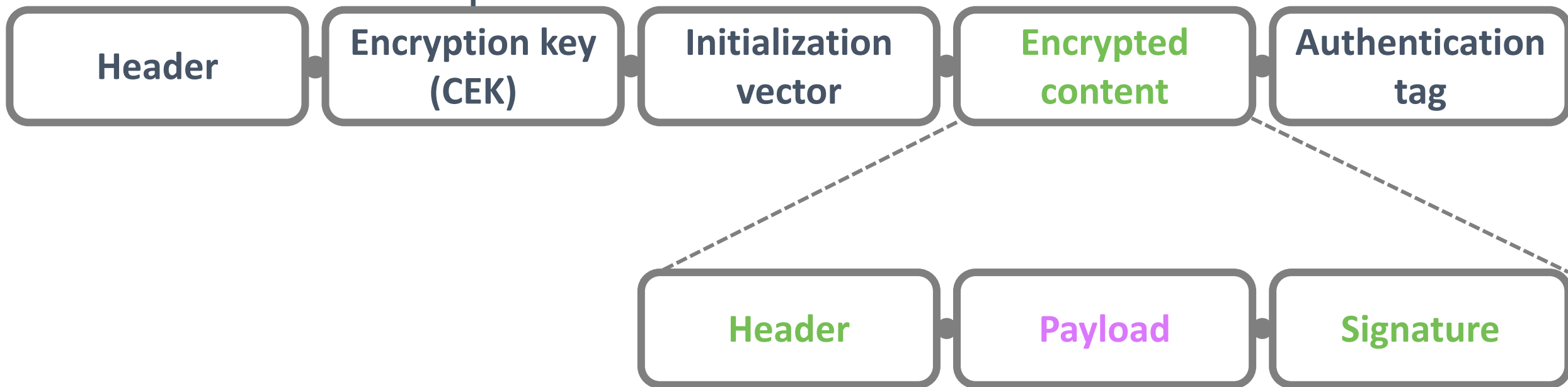
- The JWE specification describes the encryption mechanism of JWTs
  - The spec covers how to encrypt and decrypt the payload of a JWT
  - It also covers the details on how to provide proper key information
- JWE requires the use of *Authenticated Encryption* algorithms
  - These algorithms offer confidentiality, integrity and authenticity
  - Crudely put, these algorithms offer symmetric encryption with a built-in HMAC signature





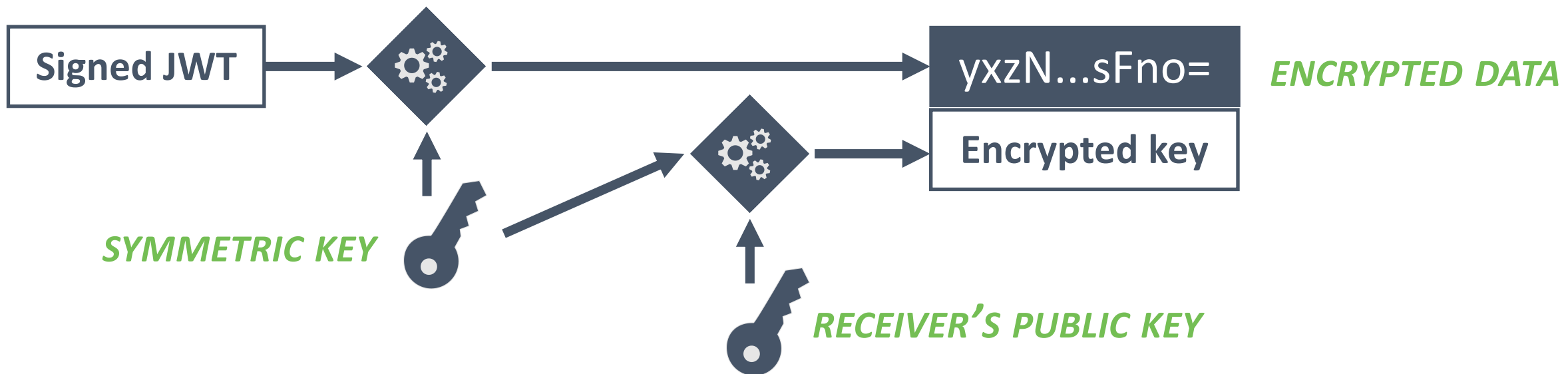


- Contains a symmetric key for encrypting/decrypting the data
- Typically, the key is encrypted with the receiver's public key
- When a pre-shared symmetric key is used, this field is empty

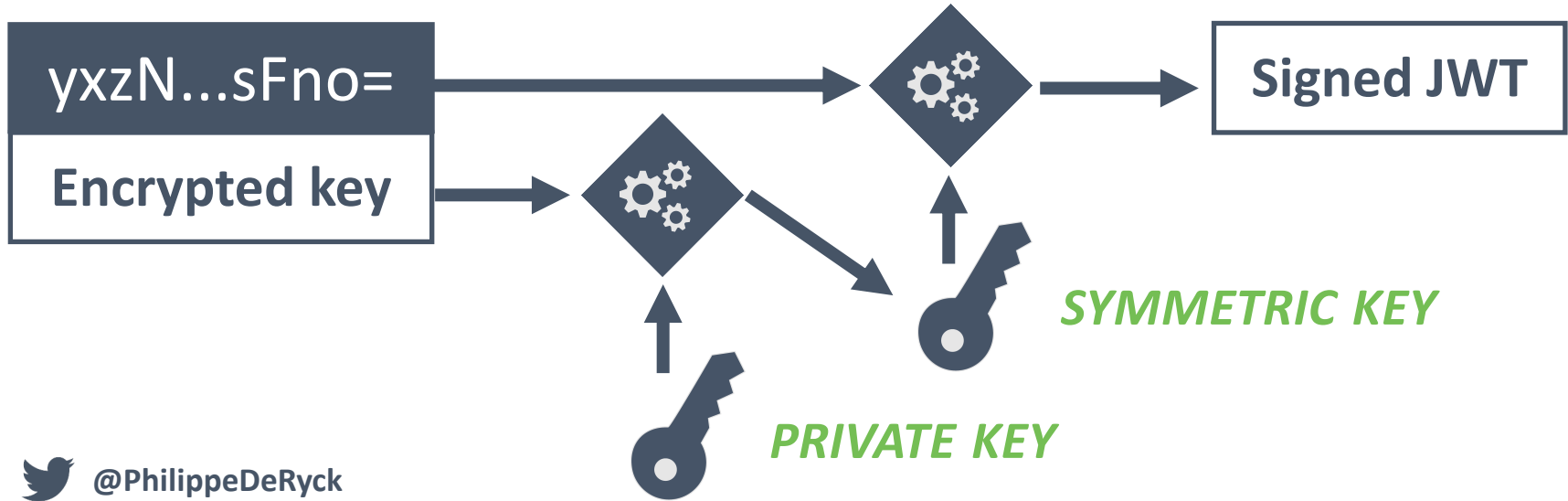




Encrypting a JWT



Decrypting a JWT



# JSON WEB ENCRYPTION (JWE)

RFC 7516

- The content is encrypted by the Content Encryption Key (CEK)
  - The **CEK** is part of the token, but is in turn encrypted with a separate key
  - The **initialization vector** is used to bootstrap the encryption algorithm
  - The **authentication tag** is used to verify the integrity of the content
- The header contains all the information to perform a proper decryption
  - The **typ** parameter specifies the media type of the data that has been signed
    - In this context, this parameter has the value JWT
  - The **enc** parameter specifies how the content of the JWT is encrypted
  - The **alg** parameter specifies how the content encryption key (CEK) is encrypted



# JWE AND KEY MANAGEMENT

- Key management is used to find the right key to decrypt the **CEK** in the token
- JWE supports similar key management mechanisms as JWS
  - The use of a shared symmetric key can be achieved using the **kid** parameter
    - In this case, the **alg** parameter is **dir** to indicate direct encryption
  - The use of a public/private key pair is supported through JWK or X.509
    - In these cases, the **alg** parameter indicates how the embedded CEK is encrypted
    - JWKs are supported through the **jwk** and **jku** parameters
    - X.509 certificates are supported through the **x5c** and **x5u** parameters
- The parameters are the same as for JWS, but their meaning differs slightly
  - JWE key parameters identify the public key used to encrypt the content of the token
  - With this public key, the receiver can identify the right private key for decryption



```
1 // Library: com.nimbusds.nimbus-jose-jwt
2 JWSHeader header = ...
3 JWTClaimsSet claimsSet = ...
4
5 JWSSigner signer = new RSASSASigner(privateKey);
6 SignedJWT jwt = new SignedJWT(header, claimsSet);
7 jwt.sign(signer);
8
9 JWEObjcet encryptedJWT = new JWEObjcet(
10     new JWEHeader.Builder(JWEAlgorithm.DIR,
11         EncryptionMethod.A256GCM)
12         .contentType("JWT") // required to signal nested JWT
13         .build(),
14     new Payload(jwt));
15 encryptedJWT.encrypt(new DirectEncrypter(encKey.getEncoded()));
16 result = encryptedJWT.serialize();
```



# JWS, JWE AND JWK

- The JWS specification describes the signature part of JWTs
  - The main challenge to overcome is to identify the right key to verify the signature
  - The ***kid*** parameter is a straightforward way to identify a known key
  - JWK or X.509 key representations can be used to send a public key to the receiver
- The JWE specification describes how to encrypt the contents of a JWT
  - The main challenge is again key management
  - The ***kid*** parameter is a straightforward way to identify a known key
  - JWK or X.509 key representations can be used to send a public key to the receiver
    - With this public key, the receiver can find the proper private key to decrypt
- All of these details should be hidden by using proper libraries



# JSON WEB ENCRYPTION (JWE)



*Typical JWTs are base64-encoded, without confidentiality*

*JWTs can be encrypted using a JWE container*

*Encrypting JWTs requires careful key management*



# SECURITY CONSIDERATIONS FOR JWT



*JWTs are a way to represent claims securely, nothing more*

*JWT security heavily relies on cryptography, so get that right*

*The most common JWT mistake is a lack of proper key management*



# SECURITY CONSIDERATIONS FOR JWT

- JWTs heavily rely on cryptography
  - Getting the security of JWT right requires a lot of attention to details
  - Fortunately, the libraries encapsulate most of the details in standard use cases
- Using cryptography requires you to think about a few things up front
  - Key sizes, key management and key rotation
  - Additional processes (e.g., combining compression with encryption causes issues)
- JWTs further complicate security because they contain metadata about crypto
  - The header informs the library how it needs to handle the token
  - But the header is untrusted, since an attacker can also manipulate the header
  - The header should not be trusted before the token is verified, which requires the header





# FREE SECURITY CHEAT SHEETS FOR MODERN APPLICATIONS

**Pragmatic Web Security**  
Security training for developers

**SECURITY CHEAT SHEET**  
Version 2018.002


## ANGULAR AND THE OWASP TOP 10

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

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


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

**1 USING DEPENDENCIES WITH KNOWN VULNERABILITIES**  
OWASP #9

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

**2 BROKEN AUTHENTICATION**  
OWASP #2

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

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

**SERVER-SIDE SESSION STATE**

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

**CLIENT-SIDE SESSION STATE**

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

**COOKIE-BASED SESSION STATE TRANSPORT**

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

**AUTHORIZATION HEADER-BASED SESSION STATE TRANSPORT**

- ☐ Only send the authorization header to whitelisted hosts  
Many custom interceptors send the header to every host

**3 CROSS-SITE SCRIPTING**  
OWASP #7

**PREVENTING HTML/SCRIPT INJECTION IN ANGULAR**

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

**PREVENTING CODE INJECTION OUTSIDE OF ANGULAR**

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

**4 BROKEN ACCESS CONTROL**  
OWASP #5

**AUTHORIZATION CHECKS**

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

**CROSS-ORIGIN RESOURCE SHARING (CORS)**

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

**5 SENSITIVE DATA EXPOSURE**  
OWASP #3

**DATA IN TRANSIT**

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

**DATA AT REST IN THE BROWSER**

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

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

https://cheatsheets.pragmaticwebsecurity.com/

**Pragmatic Web Security**  
Security training for developers




**SECURITY CHEAT SHEET**  
Version 2018.001

## JSON WEB TOKENS (JWT)

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

**INTRODUCTION**

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

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

**JWT INTEGRITY VERIFICATION**

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

**SYMMETRIC SIGNATURES**



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

**ASYNCHRONOUS SIGNATURES**



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

**BEST PRACTICES**

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

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

**VALIDATING JWTs**

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

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

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

**CRYPTOGRAPHIC KEY MANAGEMENT**

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

- ☐ Store key material in a dedicated key vault service  
Keys should be fetched dynamically, instead of being hardcoded
- ☐ Use the `kid` claim in the header to identify a specific key  
Keys should be fetched dynamically, instead of being hardcoded
- ☐ Public keys can be embedded in the header of a JWT  
The `pub` claim can hold a JSON Web Key-formatted public key  
The `x5c` claim can hold a public key and X.509-certificate
- ☐ Validate an embedded public key against a whitelist  
Failure to whitelist will cause an attacker's JWT to be accepted
- ☐ The header can also contain a URL pointing to public keys  
The `jku` claim can point to a file containing JSON Web Keys  
The `x5u` claim can point to a certificate containing a public key
- ☐ Validate a key URL against a whitelist of URLs / domains  
Failure to whitelist will cause an attacker's JWT to be accepted

**USING JWTs FOR AUTHORIZATION STATE**

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

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

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

https://cheatsheets.pragmaticwebsecurity.com/

<https://cheatsheets.pragmaticwebsecurity.com/>

49



**Pragmatic Web Security**

Security for developers

# THANK YOU!

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



**@PhilippeDeRyck**