

# Pragmatic Web Security

Security training for developers



## THE PARTS OF JWT SECURITY NOBODY TALKS ABOUT

# WHAT DO YOU KNOW ABOUT JSON WEB TOKENS?



```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpXVCJ9.eyJzdGF1cmFudG93bmVyaWwifQ.KPjhyE9oi83uehgw6Lm_0yAZzRuJhcUqXETD2AIf2A
```



# Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6ImlBoaWxpcHBlIERlIFJ5Y2siLCJyb2x1cyI6InVzZXIgcmlvZGF1cmFudG93bmVyIiwiaWF0IjoxNTE2MjM0MDIyfQ.KPjhyE9oi83uehgw6Lm_0yAZzRuJhcUqXETD2AIf2A
```

# 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



# JSON WEB TOKENS (JWT)

## THE TECHNICALITIES OF JWT

## USING JWTs FOR SESSION DATA

## ADVANCED JWT TOPICS

## SECURITY CONSIDERATIONS

## CONCLUSION



# 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("eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IjE2MzQ1Njc4OTUuIiwiaWF0IjoiMTUxNjIzOTAwMjIuInQ")
```

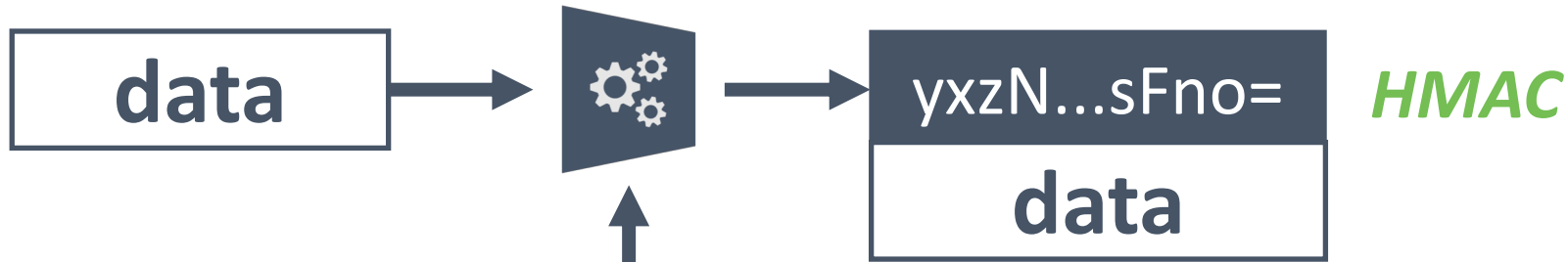
```
< '{"sub":"1234567890","name":"Philippe De Ryck","roles":"user restaurantowner","iat":1516239022}'
```



HOW CAN YOU ENSURE THE  
SECURITY OF JWT CLAIMS?

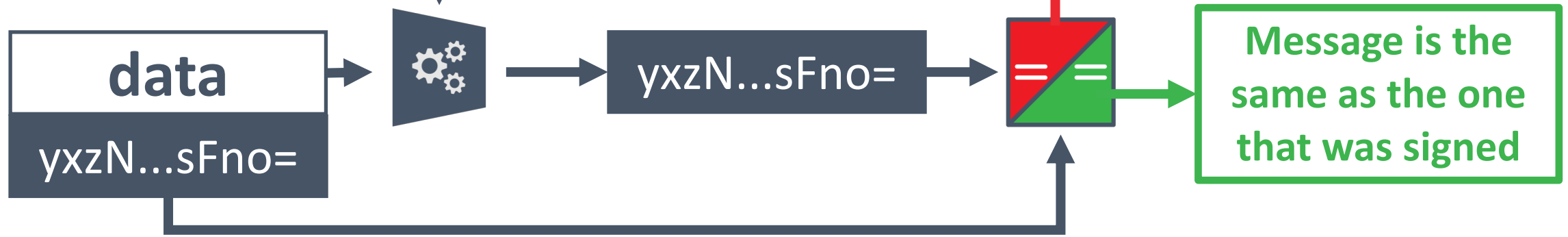


**GENERATE HMAC**



**SECRET KEY**

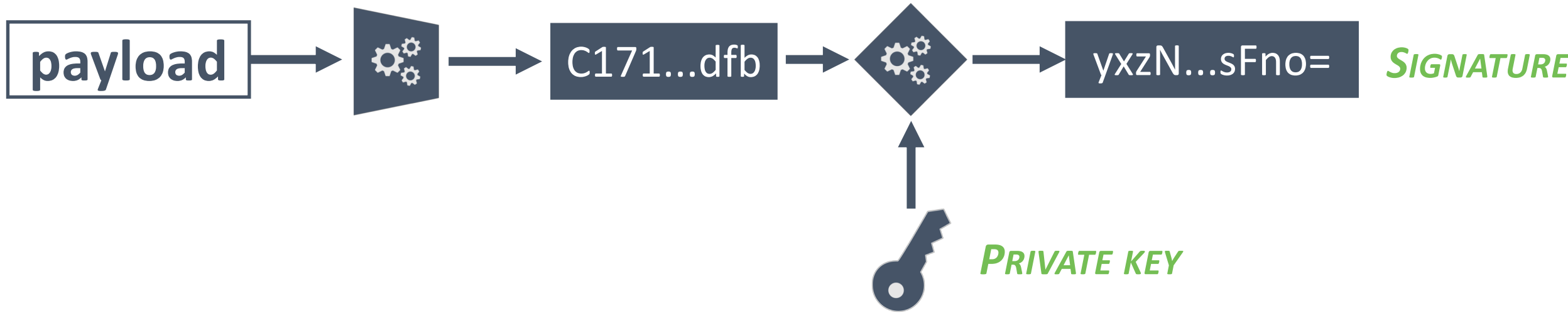
**VERIFY HMAC**



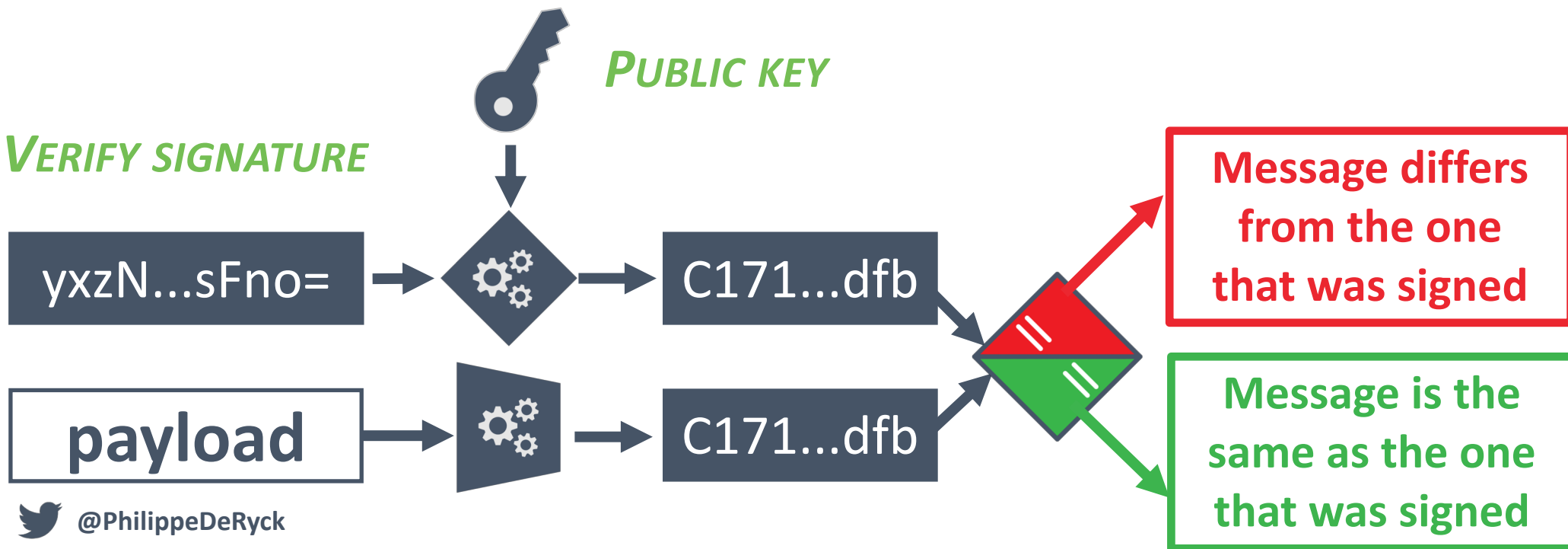
WHAT IS THE PROBLEM  
WITH HMAC-PROTECTED  
JWTs?



**GENERATE SIGNATURE**



**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



# JSON WEB SIGNATURE (JWS)

RFC 7515

- The JWS specification describes the signing mechanism of JWTs
  - The spec covers how to generate, embed and verify signatures
  - It also covers the details on how to provide proper key information
- The header of the JWT contains all information needed to verify the signature
  - The *typ* parameter specifies the media type of the data that has been signed
    - In this context, this parameter has the value JWT
  - The *alg* parameter specifies the algorithm used to sign the header and payload
  - Additional fields can be used to relay key information to the receiver





# HANDLING JWTs IN THE BACKEND

- JWTs are designed to transfer a set of claims
  - Backend applications can use these claims to make authorization or business decisions
  - Before any claims are used, the integrity of the JWT token needs to be verified
- For a signed JWT, this means checking the signature before using the data
  - A valid signature indicates that someone with the proper key has generated the JWT
  - Once the signature has been verified, the claims can be used to make decisions
- Implementing cryptographic operations correctly is difficult
  - Virtually every language has a well-supported a JWT library
    - Check the full list of supported libraries and their features on <https://jwt.io>
  - Use a well-vetted library to generate and validate JWTs





## Java

- ✔ Sign                   ✔ HS256
- ✔ Verify                 ✔ HS384
- ✔ iss check             ✔ HS512
- ✔ sub check             ✔ RS256
- ✔ aud check             ✔ RS384
- ✔ exp check             ✔ RS512
- ✔ nbf check             ✔ ES256
- ✔ iat check             ✔ ES384
- ✔ jti check             ✔ ES512
- ✔ PS256
- ✔ PS384
- ✔ PS512

Brian Campbell

View Repo

maven: org.bitbucket.b\_c / jose4j / 0.6.3



## Java

- ✔ Sign                   ✔ HS256
- ✔ Verify                 ✔ HS384
- ✔ iss check             ✔ HS512
- ✘ sub check             ✔ RS256
- ✔ aud check             ✔ RS384
- ✔ exp check             ✔ RS512
- ✘ nbf check             ✔ ES256
- ✘ iat check             ✔ ES384
- ✘ jti check             ✔ ES512
- ✔ PS256
- ✔ PS384
- ✔ PS512

connect2id

View Repo

maven: com.nimbusds / nimbus-jose-jwt / 5.7



## Java

- ✔ Sign                   ✔ HS256
- ✔ Verify                 ✔ HS384
- ✔ iss check             ✔ HS512
- ✔ sub check             ✔ RS256
- ✔ aud check             ✔ RS384
- ✔ exp check             ✔ RS512
- ✔ nbf check             ✔ ES256
- ✔ iat check             ✔ ES384
- ✔ jti check             ✔ ES512
- ✔ PS256
- ✔ PS384
- ✔ PS512

Les Hazlewood   ★ 4323

View Repo


maven: io.jsonwebtoken / jjwt / 0.9.0

# CAN YOU SPOT A PROBLEM HERE?




```
1 String token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IWRXb2kEE";
2 try {
3     DecodedJWT jwt = JWT.decode(token);
4 } catch (JWTDecodeException exception){
5     //Invalid token
6 }
```

```
1 String token = "eyJhbGciOiJIUzI1NiIsInR5cC...ZWfOkEE";
2 try {
3     DecodedJWT jwt = JWT.decode(token);
4 } catch (JWTDecodeException exception) {
5     //Invalid token
6 }
```



Decoding only

```
1 String token = "eyJhbGciOiJIUzI1NiIsInR5cC...ZWfOkEE";
2 try {
3     Algorithm algorithm = Algorithm.HMAC256("secret");
4     JWTVerifier verifier = JWT.require(algorithm)
5         .build(); //Reusable verifier instance
6     DecodedJWT jwt = verifier.verify(token);
7 } catch (JWTVerificationException exception) {
8     //Invalid signature/claims
9 }
```



Signature verification

# ADDITIONAL BACKEND SECURITY CONSIDERATIONS

- JWT tokens support a number of reserved claims to hold token metadata
  - Examples are *iss*, *exp*, *nbf* and *aud*
  - All of these claims are optional, but it is highly recommended to use them
- Checks that need to be done by the backend
  - The *iss* claim should match an expected issuer of JWT tokens
  - 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 *aud* claim indicates the intended target audience, which should match the backend
- The backend is responsible for checking these claims
  - Some libraries support the enforcement of a set of constraints



```
1 // Library: com.auth0.java-jwt
2 String token = "eyJ0eXAiOi...StzssyYXtJZs";
3 try {
4     JWTVerifier verifier = JWT.require(algorithm)
5         .withIssuer("restograde.com")
6         .withAudience("restograde.com")
7         .build();
8     DecodedJWT jwt = verifier.verify(token);
9 } catch (JWTVerificationException exception) {
10     //Invalid signature/claims
11 }
```

[com.auth0.jwt.exceptions.TokenExpiredException](#):

The Token has expired on Sat Feb 14 03:45:33 CET 2009.



# ADDITIONAL JWT CLAIMS

- JWTs support three more reserved claims
  - The *sub* claim identifies the principal that is the subject of the JWT
    - E.g., a particular user in a Single-Sign On scenario
    - The subject must be unique per issuer
  - The *iat* claim contains the date a token was issued
    - Merely informative, as the spec does not require validation of this property
  - The *jti* claim contains a unique JWT identifier
    - The spec does not specify a format, but it should be a unique case-sensitive string
- Apart from these reserved claims, issuers can add arbitrary claims
  - A claim is simply a JSON key/value pair
  - Claims can represent user information, authorization roles, ...
  - JWTs can also be used to contain integrity-protected application data



# JWT BEST PRACTICES

- Choose the proper signature scheme for your deployment
  - Symmetric key signatures have a very narrow set of supported use cases
  - Asymmetric key signatures work well in distributed scenarios
- Ensure that your backend properly enforces JWT validity
  - The signature needs to be verified before any data can be used
  - If present, the *iss*, *exp*, *nbf*, *aud* claims need to be verified against expected values
  - Write explicit tests to ensure that these constraints are indeed enforced
- Use a well-vetted JWT library
  - Libraries make JWT generation and verification a lot easier
  - Helps avoid common vulnerabilities and mistakes





# JSON WEB TOKENS (JWT)

THE TECHNICALITIES OF JWT

**USING JWTs FOR SESSION DATA**

ADVANCED JWT TOPICS

SECURITY CONSIDERATIONS

CONCLUSION

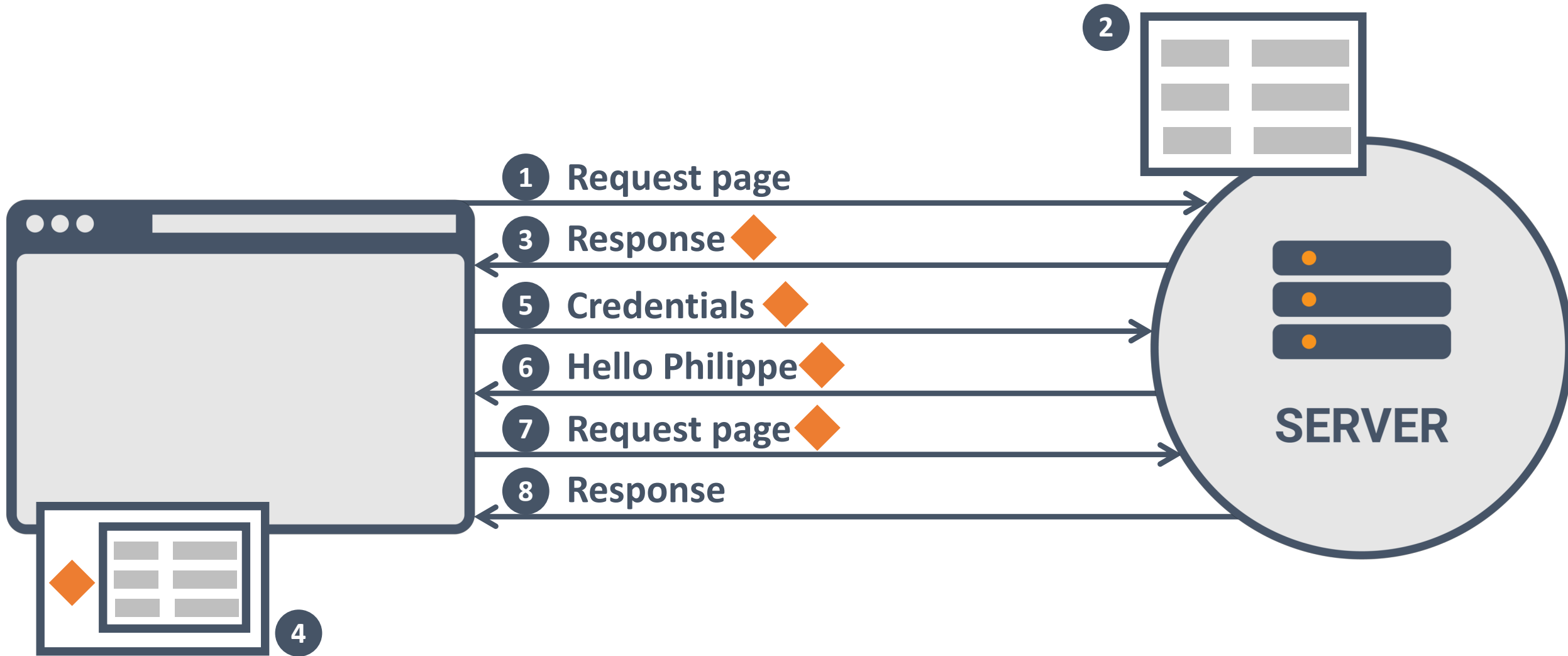


# SERVER-SIDE SESSION DATA

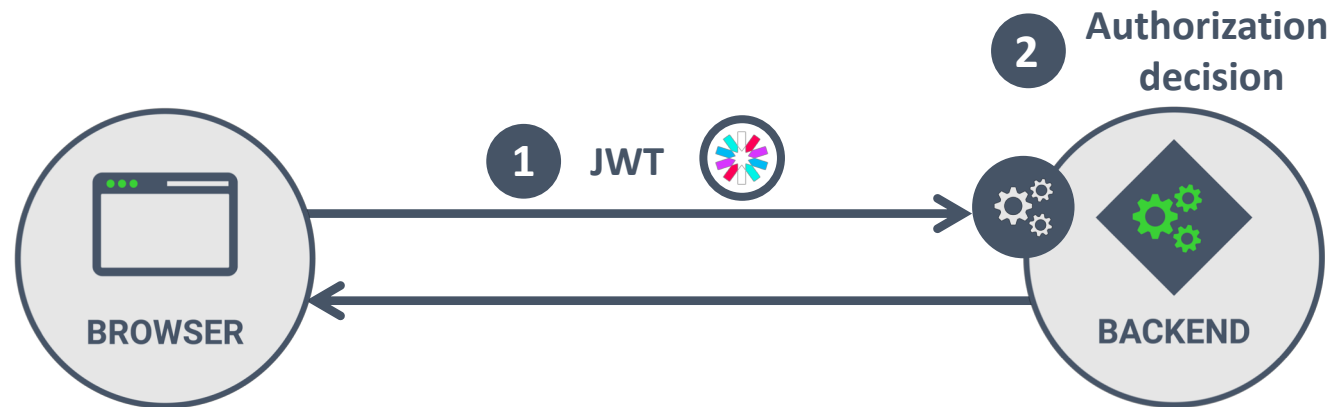
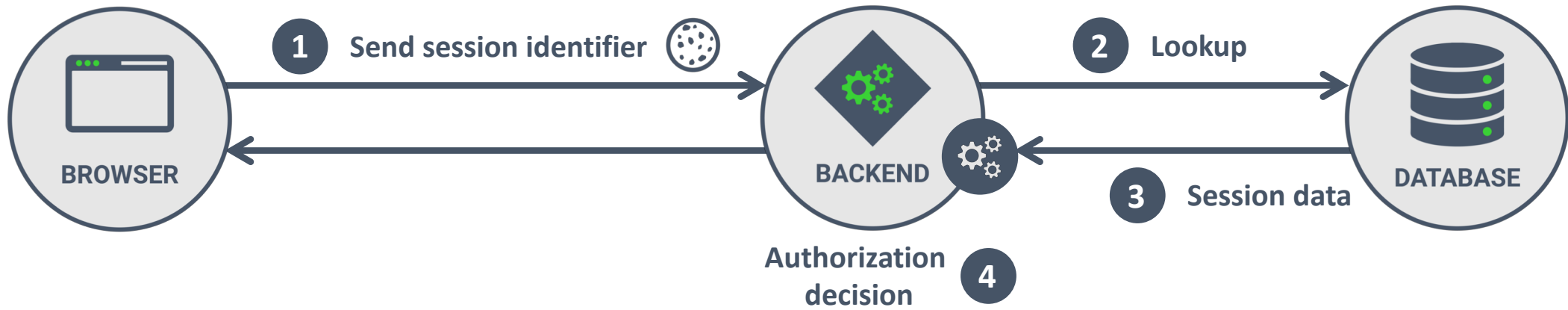


```
Set-Cookie: id=1234...; Secure; HttpOnly
```

# CLIENT-SIDE AUTHORIZATION STATE



Authorization: Bearer eyJhbGciOiJIU...



WHAT DO YOU PUT IN A  
JWT TO TRACK  
AUTHORIZATION STATE?



```
{
  "jti": "eedf40bc-5461-4e3f-a62c-bb01555464ef",
  "exp": 1535378374,
  "nbf": 0,
  "iat": 1535371174,
  "iss":
  "https://keycloak.restograde.com/auth/realms/Restograde",
  "aud": "com.restograde.viewer",
  "sub": "eb88c689-5f33-43a2-b990-3510b58a4bae",
  "typ": "Bearer",
  "azp": "com.restograde.viewer",
  "nonce":
  "wZUqJiE79LrJ3gCNDGsqnWqSEJgiQ5s8dG3hYLU0",
  "auth_time": 1535371174,
  "session_state": "50017794-5b14-4904-bd88-5dbbee662e87",
  "acr": "1",
  "allowed-origins": [
    "https://viewer.restograde.com"
  ],
```

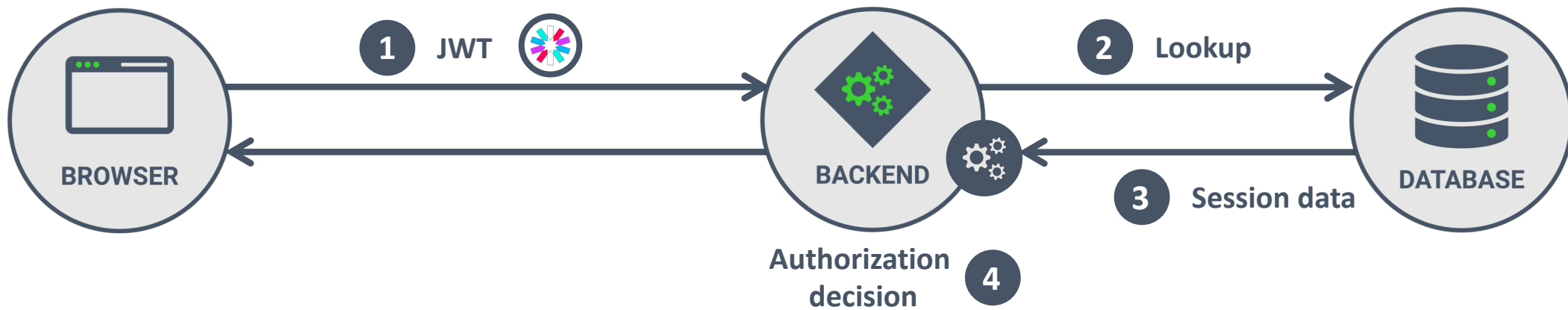
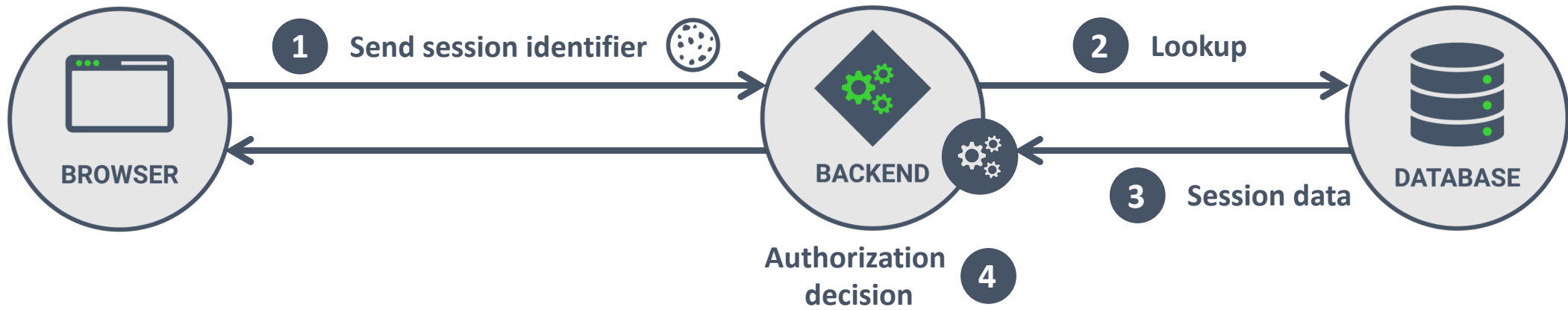
```
  "realm_access": {
    "roles": [
      "offline_access",
      "uma_authorization"
    ]
  },
  "resource_access": {
    "account": {
      "roles": [
        "manage-account",
        "manage-account-links",
        "view-profile"
      ]
    }
  },
  "scope": "openid reviews.read profile email",
  "email_verified": false,
  "name": "Philippe De Ryck",
  "preferred_username": "philippe",
  "given_name": "Philippe",
  "family_name": "De Ryck",
  "email": "philippe@pragmaticwebsecurity.com"
}
```



# JWTs ARE QUITE VERBOSE

- JWT tokens are a lot more verbose than simple session identifiers
  - Note that JWT already uses the compacted notation
  - E.g., an regular OAuth 2.0 access token as a JWT is approximately 1,5 Kb
- The size of requests by itself is not a big problem
  - It is however a bit contradictory to all the optimization we do for JS/CSS files
- To reduce the size of a JWT, you can reduce the claims in the JWT
  - Some people even go as far as only storing an identifier in the JWT
  - Even then, the JWT still contains a header and a signature, increasing its size
    - For simple data (e.g., session ID), JWTs are approximately 50 times larger







# THE PARADOX OF STATELESSNESS

- To make the backend stateless, the client needs to provide authorization state
  - Keeping such data in a JWT is the most common option
  - Long lifetimes can cause the JWT to contain stale authorization state
    - e.g., changed permissions
- Reducing the data in a JWT forces the backend to fetch authorization state
  - Closely resembles a stateful backend with server-side sessions
  - In the end, we're just mimicking session management with a JWT
- JWTs as plain session objects are getting a lot of critique
  - They don't really seem to solve anything, so what's the point ...



HOW WOULD YOU **REVOKE**  
A JWT CONTAINING  
AUTHORIZATION STATE?



```
{
  "jti": "eedf40bc-5461-4e3f-a62c-bb01555464ef",
  "exp": 1535378374,
  "nbf": 0,
  "iat": 1535371174,
  "iss":
"https://keycloak.restograde.com/auth/realms/Restograde",
  "aud": "com.restograde.viewer",
  "sub": "eb88c689-5f33-43a2-b990-3510b58a4bae",
  "typ": "Bearer",
  "azp": "com.restograde.viewer",
  "nonce":
"wZUqJiE79LrJ3gCNDGsqnWqSEJgiQ5s8dG3hYLU0",
  "auth_time": 1535371174,
  "session_state": "50017794-5b14-4904-bd88-5dbbee662e87",
  "acr": "1",
  "allowed-origins": [
    "https://viewer.restograde.com"
  ],
```

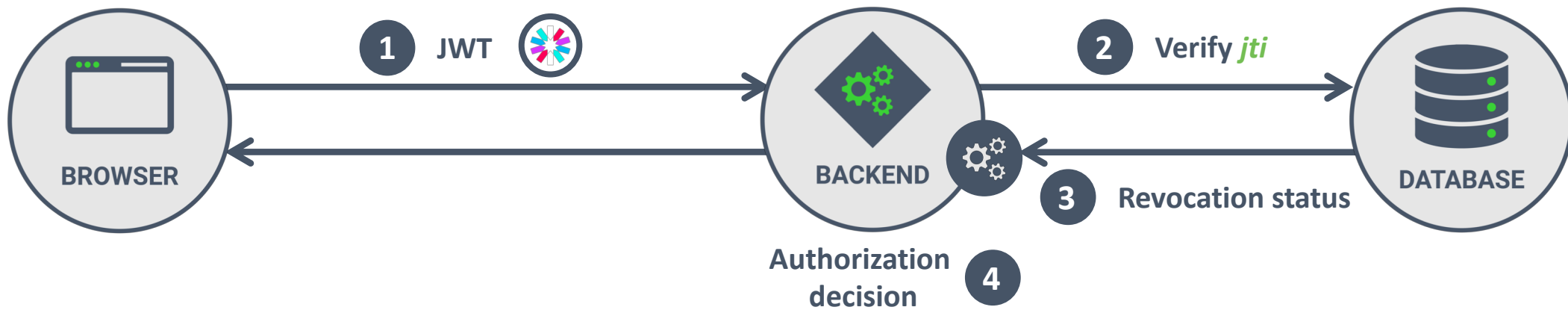
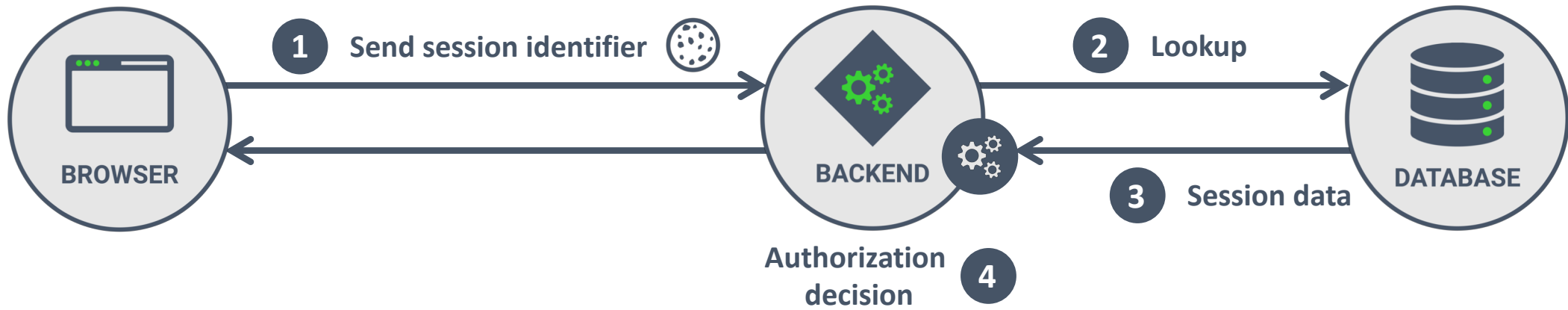
```
"realm_access": {
  "roles": [
    "offline_access",
    "uma_authorization"
  ]
},
"resource_access": {
  "account": {
    "roles": [
      "manage-account",
      "manage-account-links",
      "view-profile"
    ]
  }
},
"scope": "openid reviews.read profile email",
"email_verified": false,
"name": "Philippe De Ryck",
"preferred_username": "philippe",
"given_name": "Philippe",
"family_name": "De Ryck",
"email": "philippe@pragmaticwebsecurity.com"
}
```



# JWT REVOCATION

- A common revocation pattern uses the JWTs unique identifier
  - Keeping a list of invalid identifiers enables the backend to reject revoked JWTs
- Revoking a specific token for a specific device is challenging
  - The backend needs to keep a list of all issued *jti* claims
  - These identifiers need to be correlated to users and devices
- Verifying incoming JWTs against a revocation list requires explicit action
  - Depends on a centralized list of invalid identifiers
  - Check needs to happen on each incoming request
  - Adds a form of state to an otherwise stateless backend

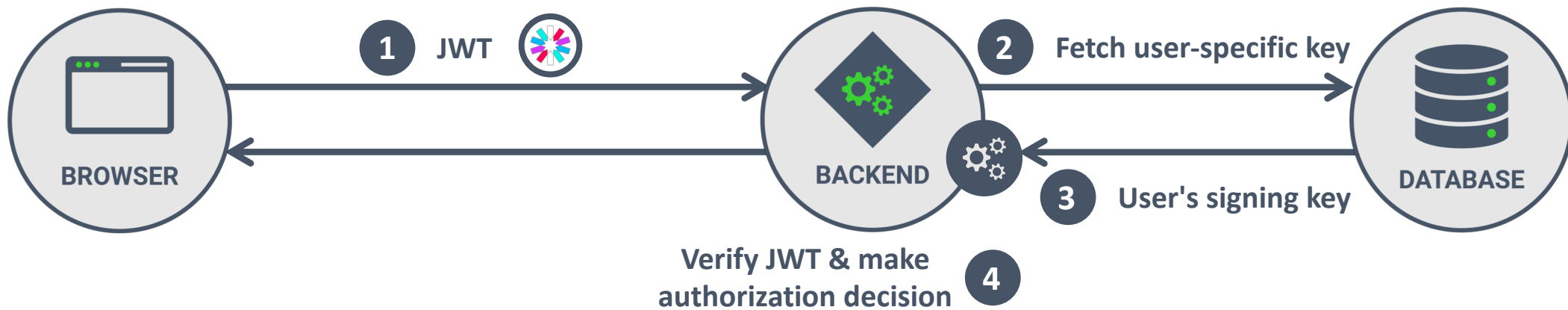
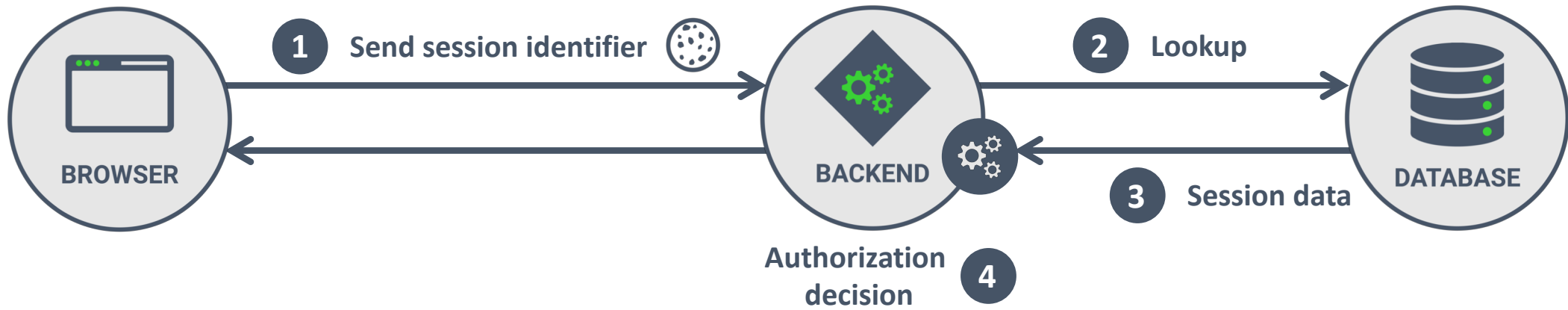




# JWT REVOCATION USING KEY ROTATION

- Forcing a change in signing key turns every existing JWT signature invalid
  - Previously issued tokens will no longer be accepted, resembling revocation
  - Keys can be rotated globally, or on a per-user basis
- Global key rotation is only useful for emergency incident response
  - Rotating an application-wide signing key causes *all* JWTs to become invalid
  - Doing this impacts every device of every user of the application
- Using per-user keys enables more granular rotation of keys
  - By changing a single user's signing key, all tokens of that user can be revoked
  - Impact remains limited to that single user, making this option seem viable





# 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.

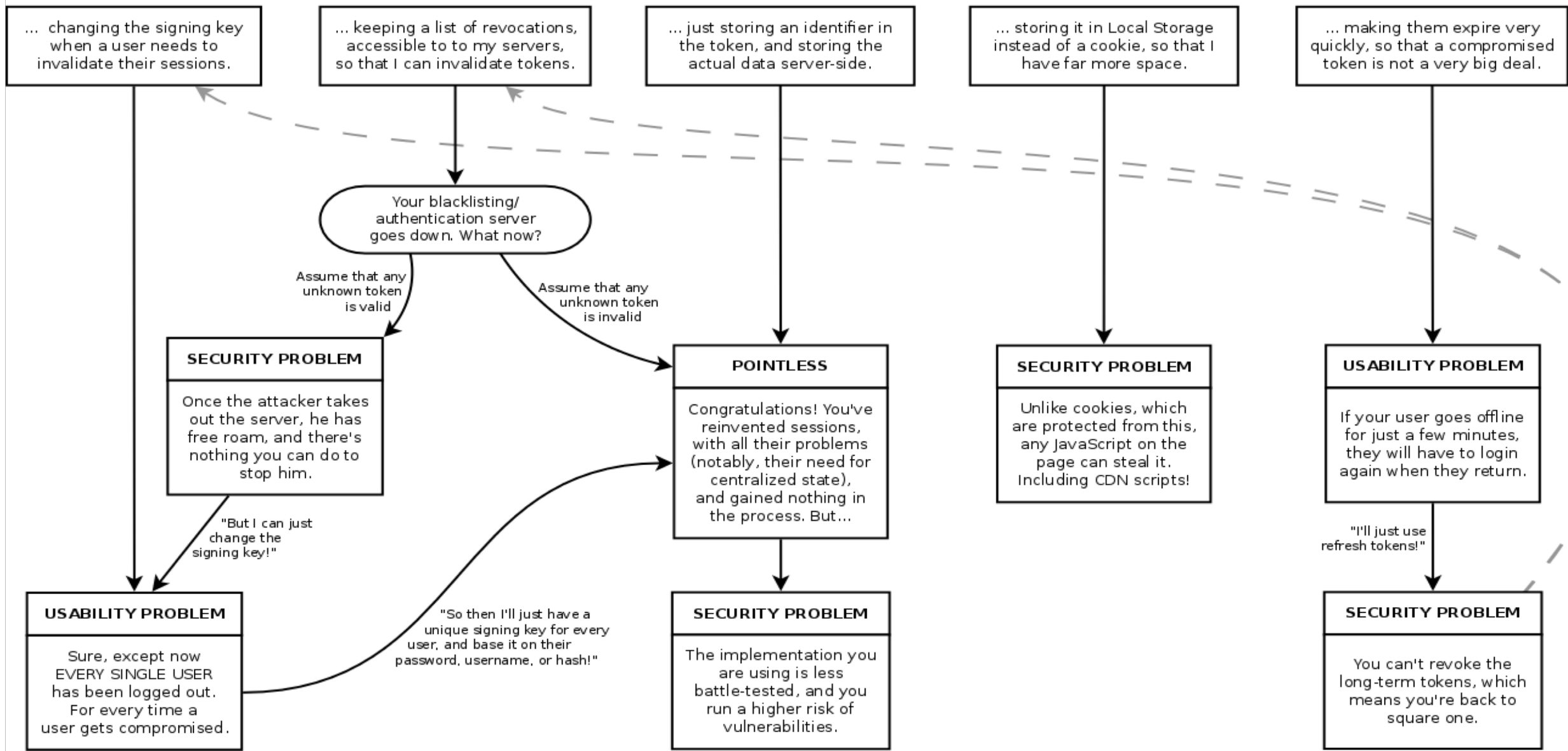
“ This article does *not* argue that you should *never* use JWT - just that it isn't suitable as a session mechanism, and that it is dangerous to use it like that. Valid usecases *do* exist for them, in other areas. ”



# Stop using JWT for sessions, part 2

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

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



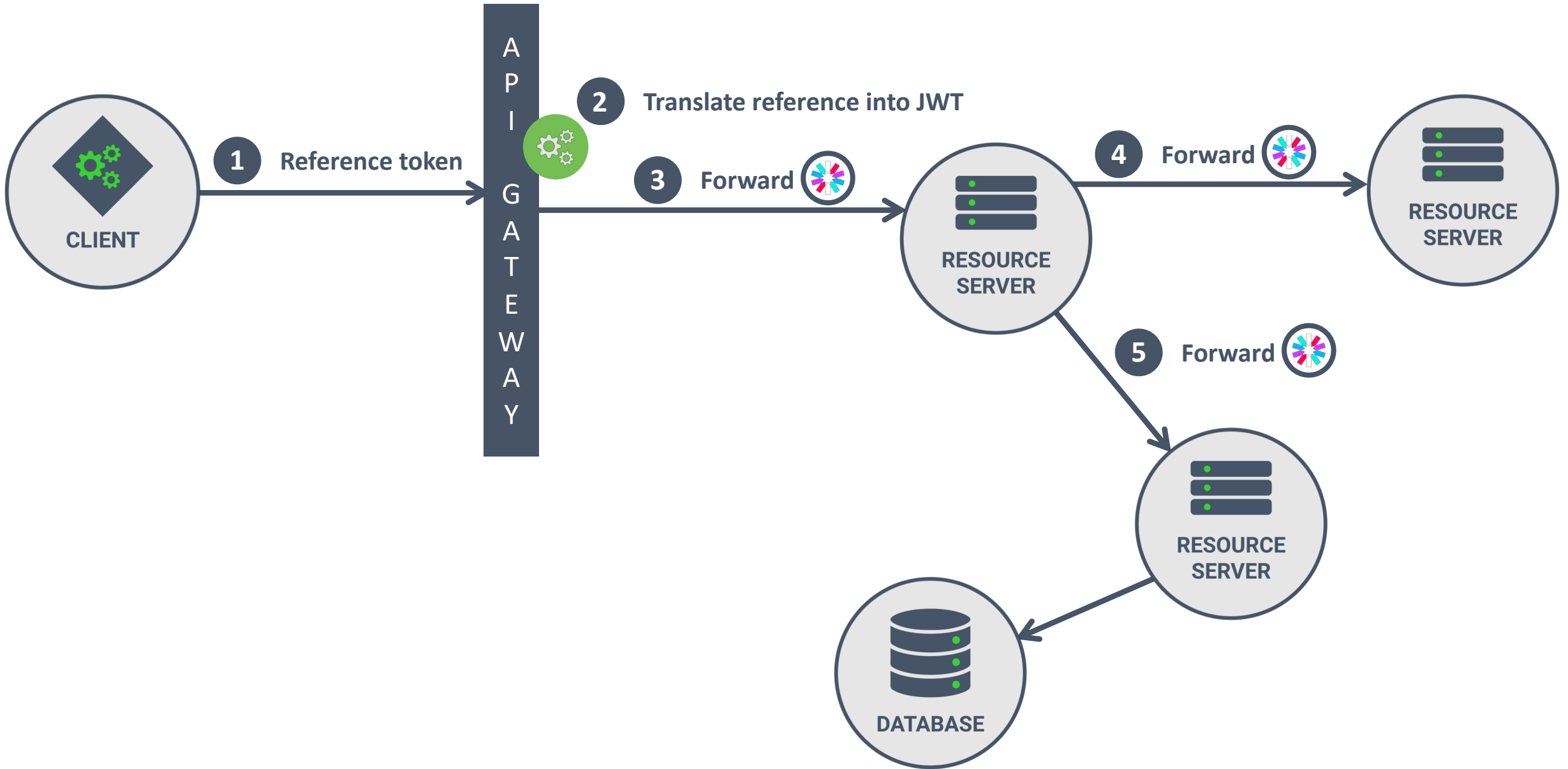
WHICH FRAMEWORK AND  
PROTOCOL ARE HEAVILY  
BASED ON JWT?



# JWT IN OAUTH 2.0 AND OPENID CONNECT

- OAuth 2.0 supports both reference tokens and self-contained tokens
  - Reference tokens refer to state kept by the authorization server
  - Self-contained tokens are formatted as a JWT signed by the authorization server
- OAuth 2.0 self-contained tokens are JWT-formatted claims
  - Access tokens are supposed to be short-lived (minutes to an hour)
  - Refresh tokens enable long-term access by getting a new access token
    - Refresh tokens are coupled to client credentials to avoid abuse
- OpenID Connect issues JWT-formatted identity tokens
  - Intended for single-time consumption by the client





# 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



# JSON WEB TOKENS (JWT)

THE TECHNICALITIES OF JWT

USING JWTs FOR SESSION DATA

**ADVANCED JWT TOPICS**

SECURITY CONSIDERATIONS

CONCLUSION



HOW WOULD YOU SOLVE  
THE KEY MANAGEMENT  
PROBLEM WITH JWKS?



# 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





# KEY IDENTIFICATION IN JWTs

- The JWT header supports a *kid* parameter
  - This parameter is designed to hold a key identifier
  - Its value is unspecified, so it can contain anything
- One example scenario is the use of symmetric keys to generate HMACs
  - Each key in use has a unique identifier (e.g., a UUID)
  - When generating a new JWT, the *kid* parameter contains the UUID of the signing key
  - During verification, the *kid* can be used to retrieve the right key

HEADER: ALGORITHM & TOKEN TYPE

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



# 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*



# DISTRIBUTING THE RIGHT KEY

- The JOSE suite also includes a JSON Web Key specification (RFC 7517)
  - JWK offers a way to represent cryptographic keys in a JSON format
  - A JWK can be included in a JWT token as a way to distribute a public key
- Including JWK information in a JWT can be done in two ways
  - Directly embedding the JWK using the *jwk* parameter
  - Embedding a URL that points to a set of JWK values using the *jku* parameter
    - In this case, the *kid* parameter is used to refer to a particular key

HEADER: ALGORITHM & TOKEN TYPE

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



```
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();
```



# DISTRIBUTING THE RIGHT KEY

- The alternative to using JWK is using X.509 certificates
  - X.509 certificates are used to transfer key information when using TLS
  - A certificate is issued by a CA and establishes the authority of a public key
- The key pair associated with the certificate can also be used to sign JWTs
  - Embed the certificate containing the public key directly using the **x5c** parameter
  - Use a URI pointing to the certificate containing the public key in the **x5u** parameter

HEADER: ALGORITHM & TOKEN TYPE

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



## 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/v 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);
```

WHICH OF THESE SECURITY PROPERTIES IS NOT YET COVERED?

A. Confidentiality

B. Integrity

C. Authenticity



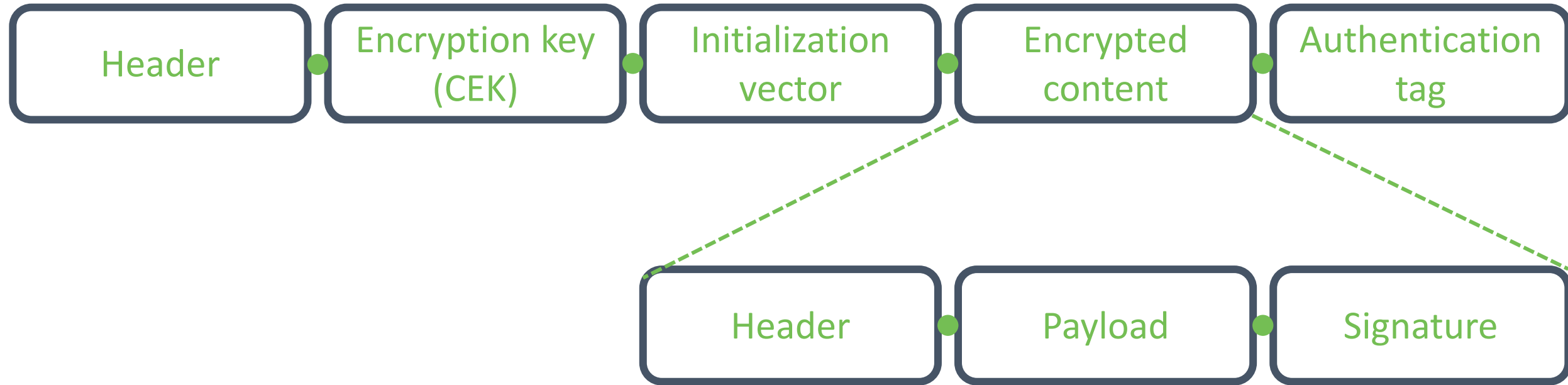
# JSON WEB ENCRYPTION (JWE)

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



# A JWE CONTAINS A NESTED JWT TOKEN



```
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 JWEObjekt encryptedJWT = new JWEObjekt(
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();
```



# 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



# 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 TOKENS (JWT)

THE TECHNICALITIES OF JWT

USING JWTs FOR SESSION DATA

ADVANCED JWT TOPICS

**SECURITY CONSIDERATIONS**

CONCLUSION



# 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

# SECURITY CONSIDERATIONS FOR JWT, JWS AND JWE

- 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)

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



# SECURITY CONSIDERATIONS FOR JWT, JWS AND JWE

- **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



# IS THIS A VALID JWT TOKEN?

## Decoded

EDIT THE PAYLOAD AND SECRET

### HEADER: ALGORITHM & TOKEN TYPE

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

### 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)
```

# SECURITY CONSIDERATIONS FOR JWT SIGNATURES

- **Update your dependencies**
  - These vulnerabilities were fixed in all libraries, so keep them up to date
  - Enable automatic dependency checking to flag future security vulnerabilities
- **Hardcode the signature algorithm if possible**
  - Works well for internal use of JWTs, where you decide how to generate JWTs
  - For third-party JWTs, this might break if the third party changes its signing algorithm
- **For third-party JWTs, you should restrict signatures to asymmetric only**
  - This prevents the attacker from forging a token with a symmetric signature
- **Some people advise the use of headless JWTs to avoid these issues**
  - A headless JWT is a regular JWT without the header part



# HEADLESS JWTs

- The application removes and re-attaches the header part of the JWT
  - The header is never sent to the client, but remains on the server
  - This avoids vulnerabilities where the attacker tampers with the header
- This proposal addresses a symptom, but does not work very well
  - Only works within one application, where you can also hardcode the algorithm
  - Makes everything more complicated, and deviates from the standard

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXUyJ9.eyJpc3MiOiJhdXRoMCJ9.AbIJTDMFc7yUa5MhvcP03nJPYCPzZtQcGep-zWfOkEE
```

```
String header = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXUyJ9";  
String token = header +  
"eyJpc3MiOiJhdXRoMCJ9.AbIJTDMFc7yUa5MhvcP03nJPYCPzZtQcGep-zWfOkEE"
```



# JWT SECURITY BEST PRACTICES

- The concept of a JWT is not inherently insecure
  - Security depends on how you use JWTs within your application
  - Security also depends on libraries, which have matured quickly
- Many security vulnerabilities occurred in library implementations
  - These vulnerabilities were known, but now occur in a new context
  - JWTs have suffered similar vulnerabilities than XML did a decade ago
- Carefully review your implementation for your use of JWTs
  - Follow best practices
  - Ensure that your application can deal with changes in the use of JWTs
    - This enables you to push out updates in case further security issues are discovered



# JSON WEB TOKENS (JWT)

THE TECHNICALITIES OF JWT

USING JWTs FOR SESSION DATA

ADVANCED JWT TOPICS

SECURITY CONSIDERATIONS

**CONCLUSION**



# RECAP

- The essence of a JWT is a mechanism to securely represent claims
  - Most common is a base64-encoded JWT with an embedded signature
  - JWTs support both symmetric and asymmetric signatures
  - The data embedded in JWTs can be encrypted using JSON Web Encryption (JWE)
- The claims of a JWT are a set of key/value pairs
  - The specification defines a set of reserved claims with JWT metadata
  - The application can define its custom claims within the token
- JWT generation and validation is handled by libraries
  - Libraries used to suffer from vulnerabilities, so use up-to-date libraries
  - Ensure that you are using in a secure way (yes, read the documentation!)



# BEST PRACTICES

- **Isolated applications can use JWTs with symmetric signatures**
  - Signature is generated using an HMAC and a server-side secret key
  - The secret key should be large enough, and kept secret
  - Anyone with the key can verify tokens, but also generate tokens
- **Distributed applications should use asymmetric signatures**
  - The private key is used to generate a signature
  - Everyone with the public key can verify the validity of the JWT
  - The default mechanism to represent identity data in the OpenID Connect protocol
- **The most crucial aspect of using JWT is server-side validation of the token**
  - The signature needs to be validated, preferably without relying on the header
  - The reserved claims need to be checked to ensure the token is being used correctly



# FREE SECURITY CHEAT SHEETS FOR MODERN APPLICATIONS

Pragmatic Web Security  
Security training for developers

SECURITY CHEAT SHEET  
Version 2018.002

## ANGULAR AND THE OWASP TOP 10

The OWASP top 10 is one of the most influential security documents of all time. But how do these top 10 vulnerabilities resonate in a frontend JavaScript application?  
This cheat sheet offers practical advice on handling the most relevant OWASP top 10 vulnerabilities in Angular applications.

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

### 1 USING DEPENDENCIES WITH KNOWN VULNERABILITIES

OWASP #9

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

### 2 BROKEN AUTHENTICATION

OWASP #2

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

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

### SERVER-SIDE SESSION STATE

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

### CLIENT-SIDE SESSION STATE

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

### COOKIE-BASED SESSION STATE TRANSPORT

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

### AUTHORIZATION HEADER-BASED SESSION STATE TRANSPORT

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

[1] [https://www.owasp.org/index.php/Session\\_Management\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Session_Management_Cheat_Sheet)

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

Pragmatic Web Security  
Security training for developers

SECURITY CHEAT SHEET  
Version 2018.001

## JSON WEB TOKENS (JWT)

JSON Web Tokens (JWTs) have become extremely popular. JWTs seem 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.

### ASYMMETRIC SIGNATURES

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

### BEST PRACTICES

- Always verify the signature of JWT tokens
- Avoid library functions that do not verify signatures
  - Example: The `decode` function of the `auth0 Java JWT library`*
- Check that the secret of 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 `jwk` claim can hold a JSON Web Key-formatted public key*
  - The `x5c` claim can hold a public key and X509-certificate*
- Validate an embedded public key against a whitelist
  - Failure to whitelist will cause an attacker's JWT to be accepted*
- The header can also contain a URL pointing to public keys
  - The `jku` claim can point to a file containing JSON Web Keys*
  - The `x5u` claim can point to a certificate containing a public key*
- Validate a key URL against a whitelist of URLs / domains
  - Failure to whitelist will cause an attacker's JWT to be accepted*

### USING JWTs FOR AUTHORIZATION STATE

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

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

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

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



# Web Security Essentials

*2-day training course*

*Modern-day best practices*

*Hands-on labs on a custom  
training application*

*April 25<sup>th</sup> – 26<sup>th</sup>, 2019*

*Leuven, Belgium*

<https://essentials.pragmaticwebsecurity.com>



# Pragmatic Web Security

Security training for developers



[/in/PhilippeDeRyck](https://www.linkedin.com/in/PhilippeDeRyck)



[@PhilippeDeRyck](https://twitter.com/PhilippeDeRyck)

[philippe@pragmaticwebsecurity.com](mailto:philippe@pragmaticwebsecurity.com)