42crunch

# Are You Properly Using JWTs?

**Philippe Leothaud**, CTO and Founder

Kristin Davis

Head of Marketing
**42Crunch**

Phil Leothaud

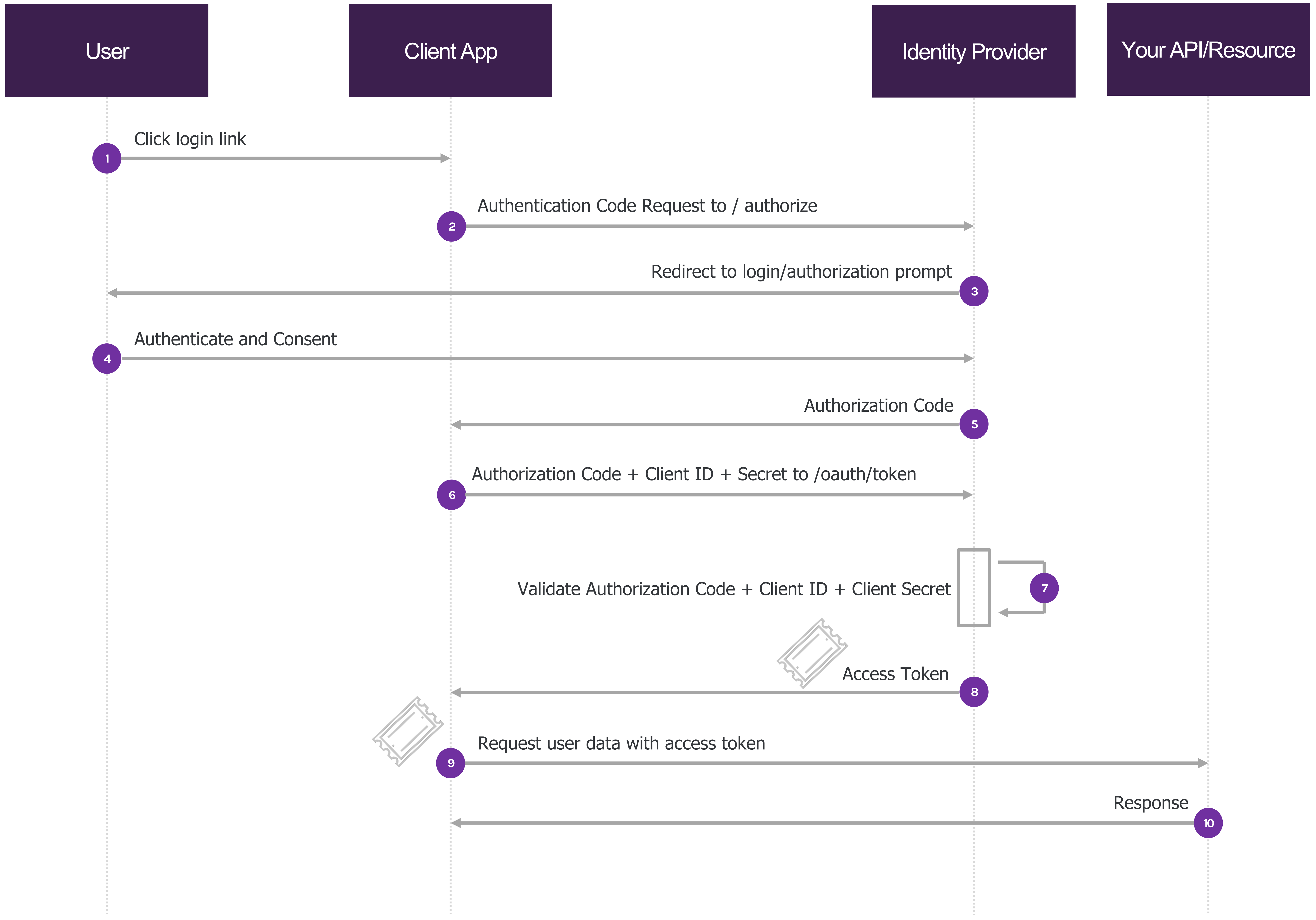
CTO and Founder
**42Crunch**

© COPYRIGHT 42CRUNCH

## What are JWTs The crypto behind it

© COPYRIGHT 42CRUNCH

## Why do we need tokens?


User
Client App
Identity Provider
Your API/Resource
1
Click login link
2
Authentication Code Request to / authorize
3
Redirect to login/authorization prompt
4
Authenticate and Consent
5
Authorization Code
6
Authorization Code + Client ID + Secret to /oauth/token
7
Validate Authorization Code + Client ID + Client Secret
8
Access Token
9
Request user data with access token
10
Response

© COPYRIGHT 42CRUNCH

## What are JWTs

- JWTs (RFC 7519) are a convenient way to transport over HTTP base64-URL encoded claims across parties in JSON format.
- Claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure.
- They are easy to use, supported by many libraries in more or less all programming langages, and therefore pervasive.

© COPYRIGHT 42CRUNCH

## Tokens can be anything

ec9f8fbb-a357-4fb6-a6af-de6ce54fb3d2

## Why JSON Web Tokens?

- Transported info right in the token
- No need for shared databases
- No extra API calls
- JSON is easy to use in code

```
{
  "user":“elmer@foodbeat.com",
  "is_admin":false,
  "twitter":"elmer.foodbeat",
  "iss":1579551140,
  "exp":1579551740
}
```

© COPYRIGHT 42CRUNCH

## Common Use Cases

- OAuth2
- OpenID Connect id_token
- Any JSON payload that needs to be protected and sent

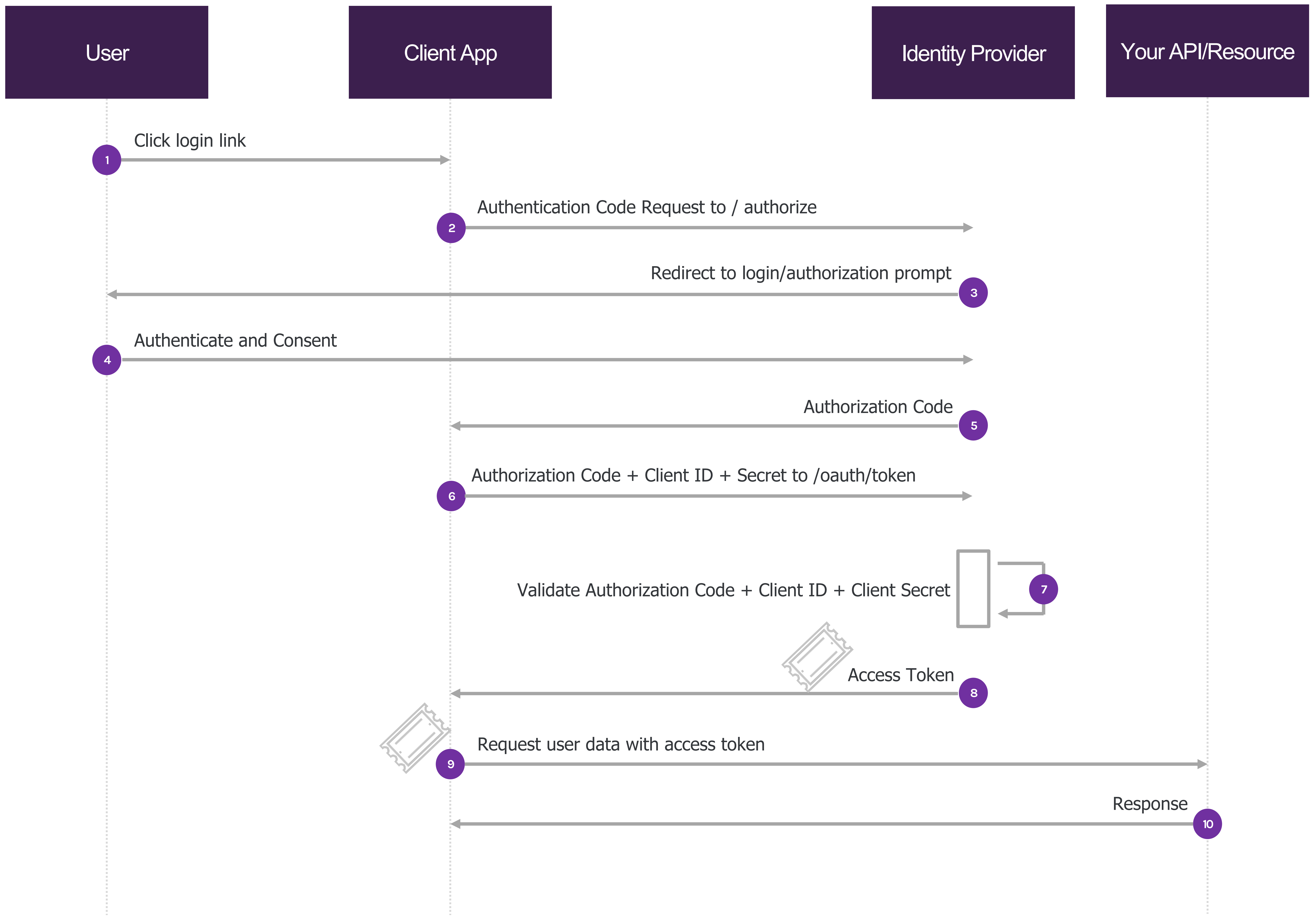
## Tokens are Encoded

- To pass them in URLs and headers
- Base64URL encoding is used
- Encoding != signing
- Encoding != encryption

```
POST /book HTTP/1.1
Content-Type: application/json
Accept: application/json
Host: resource.catalog.library
Authorization: Bearer
IUojlkoiaos298jkkdksdosiduIUiopo
{
"isbn":"9780201038019",
"author":"Donald Knuth",
"title":"The Art of Computer
Programming"
}
```

## How do you know that the token is from a specific source?

- They are signed/encrypted
- In AuthN scenarios IdP signs the new token:
  1. Calculates signature
  2. Appends it to token
- Client passes the token to resource as is
- Resource verifies the signature


User
Client App
Identity Provider
Your API/Resource
1 Click login link
2 Authentication Code Request to / authorize
3 Redirect to login/authorization prompt
4 Authenticate and Consent
5 Authorization Code
6 Authorization Code + Client ID + Secret to /oauth/token
7 Validate Authorization Code + Client ID + Client Secret
8 Access Token
9 Request user data with access token
10 Response

© COPYRIGHT 42CRUNCH

## (some) JOSE Header Parameters

- alg: the signing algorithm (used even when JWT is encrypted!). Can be none, RS256, HS256, … - full list in RFC7518
- enc: the Key encryption algorithm when using encryption
- jku: URI to a set of JSON-encoded public keys one of which corresponds to the key used to sign the token
- jwk: public key corresponding to the one used to sign the token
- kid: hint indicating which key was used
- x5u: URI for X.509 public key certificate or certificate chain
- x5c: X.509 public key certificate or certificate chain
- x5t: encoded SHA-1 thumbprint / digest of the DER encoding of X.509 certificate
- x5t#S256: SHA-256 thumbprint
- … and some more -> See RFCs from 7515 to 7519 ;-)

© COPYRIGHT 42CRUNCH

## Signing Process

1. Create JOSE header
2. Encode it
3. Create payload (does not have to be JSON)
4. Encode it too
5. Concatenate with . in between
6. Compute signature using alg
7. Base64URL-encode and append

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXV
CJ9.eyJ1c2VyIjoiZG1pdHJ5QDQyY3J1b
mNoLmNvbSIsImlzX2FkbWluIjpmYWxzZS
widHdpdHRlciI6IkRTb3RuaWtvdiIsIml
zcyI6MTU3OTU1MTE0MCwiZXhwIjoxNTc5
NTUxNzQwfQ.n34z-LWu4INXl8-Cgac-
Ues7r99xgbt_A4aHuCAZRLU
```

© COPYRIGHT 42CRUNCH

## Attacks and how to prevent

© COPYRIGHT 42CRUNCH

An API **should not blindly trust** anything it receives from the client.

## None Algorithm Attack

1. Attacker modifies or creates a token

```
{
  "alg": "HS256",
  "typ": "JWT"
}.
{
  "user":"elmer@foodbeat.com",
  "is_admin":false
}.
X0Wglk3qxprPLVTw2cYzuwEcJEEfED2F5Xgm
TdQFY7A
```

© COPYRIGHT 42CRUNCH

## None Algorithm Attack

1. Attacker modifies or creates a token

```
{
  "alg": "HS256",
  "typ": "JWT"
}.
{
  "user":"elmer@foodbeat.com",
  "is_admin":true
}.
X0Wglk3qxprPLVTw2cYzuwEcJEEfED2F5Xgm
TdQFY7A
```

© COPYRIGHT 42CRUNCH

## None Algorithm Attack

1. Attacker modifies or creates a token
2. They set alg to None in the header
3. And send it without a signature
4. Since alg is None, this is a valid JWS

```
{
  "alg": "None",
  "typ": "JWT"
}.
{
  "user":"elmer@foodbeat.com",
  "is_admin":true
}.
```

```
eyJhbGciOiAiTm9uZSIsCiAgInR5cCI6ICJK
V1QifQ.
eyJ1c2VyIjoiZG1pdHJ5QDQyY3J1bmNoLmNv
bSIsImlzX2FkbWluIjp0cnVlfQ.
```

## HMAC Algorithm Attack

- HMAC is symmetric: same shared key used to sign & verify
- RSA is asymmetric: public & private keys
- Attacker:
  1. Puts HS256 instead of RS256
  2. Signs with public RS256 key
- API code blindly uses public RSA key with HMAC alg to verify signature

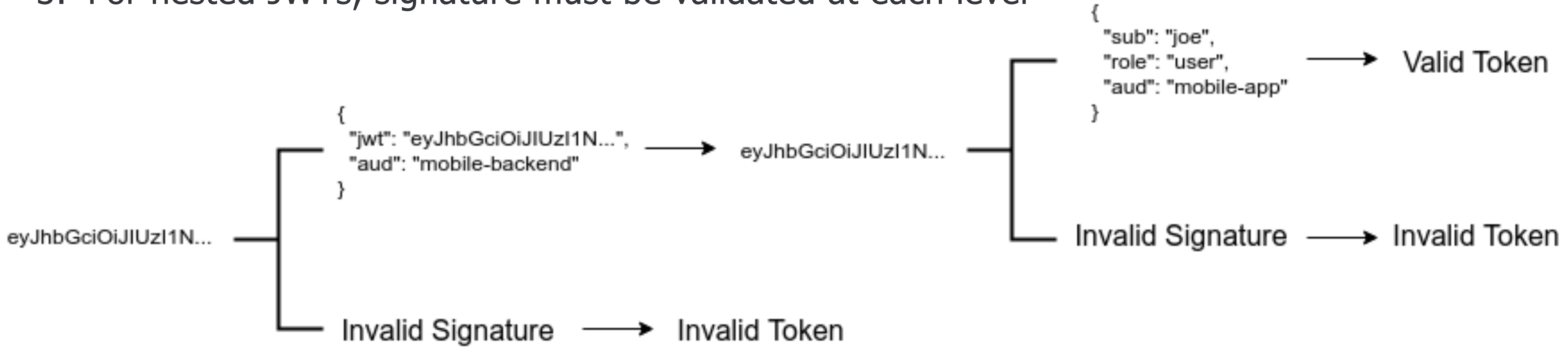
```
{
 "alg" : "RS256",
 "typ" : "JWT“
}.
{
  "user":“elmer@foodbeat.com",
  "is_admin":false
}.
RSA signature with RSA private key

Changed to:
{
 "alg" : “HS256",
 "typ" : "JWT“
}.
{
  "user":“elmer@foodbeat.com",
  "is_admin":true
}.
HMAC signature with RSA public key
```

© COPYRIGHT 42CRUNCH

## Lack of Signature Validation

1. Developers may not validate signature at all
2. They blindly trust the signature passed in the header
3. For nested JWTs, signature must be validated at each level



## Bruteforce Attack on Signature

1. Developers use a low entropy key
2. Attackers intercept a valid token
3. They now know the alg and have a token with valid signature
4. They can run a dictionary attack figure out the key
5. Once the signature matches they know your key and can forge tokens

```
signature = HMAC-
SHA256(base64urlEncode(header) +
'.' + base64urlEncode(payload),
"qwerty")
```

© COPYRIGHT 42CRUNCH

## Leaked Keys

- Source code repos
- Directory traversals
- XXE
- SSRF

```
peration == "MIRROR_X":
irror_mod.use_x = True
irror_mod.use_y = False
irror_mod.use_z = False
_operation == "MIRROR_Y":
irror_mod.use_x = False
irror_mod.use_y = True
irror_mod.use_z = False
_operation == "MIRROR_Z":
irror_mod.use_x = False
irror_mod.use_y = False
irror_mod.use_z = True

election at the end -add
_ob.select= 1
er_ob.select=1
ntext.scene.objects.acti
("Selected" + str(modifie
irror_ob.select = 0
 bpy.context.selected_ob
ata.objects[one.name].se

int("please select exactl

 OPERATOR CLASSES ----

types.Operator):
 X mirror to the selected
ject.mirror_mirror_x"
ror X"

ontext):
ext.active_object is not
```

## kid as a file path

1. Developers use a filepath for the key
2. Developers do not sanitize the value
3. Attackers specify any valid path with known content
4. They use symmetric alg and that known content

```
{
 "alg" : "HS256",
 "typ" : "JWT",
 "kid" : "secret/hmac.key“
}

change to:
{
 "alg" : "HS256",
 "typ" : "JWT",
 "kid" : "../../styles/site.css“
}
```

## kid with SQL Injection

1. Developers use unsafe code to retrieve key from database

2. Attackers supply invalid key ID with a SQL injection resulting in known result

```
Unsafe SQL retrieval:

SELECT Key WHERE ("key = ${kid}")


Attack value:

"kid": "blah' UNION SELECT 'key';--"
```

© COPYRIGHT 42CRUNCH

## Command Injection

1. Developers use header parameter as a filename and unsafe operation to read the file

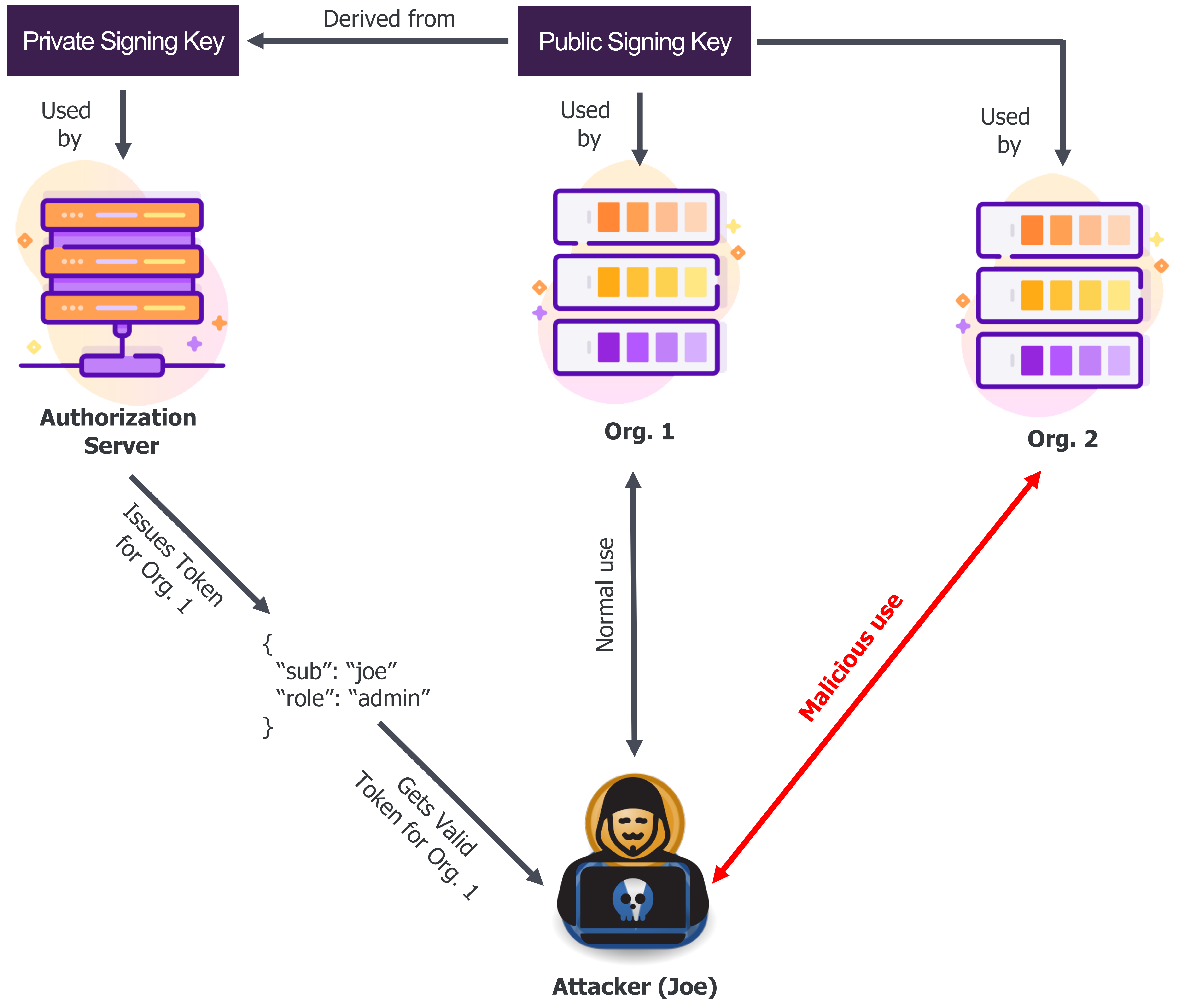
2. Attackers send an injection string and get their commands executed on the server

```
File.open(key_filename), system(),
exec(), etc.


{
  "kid":"'filename' | whoami;“
}
```

## Substitution Attack: Different Recipient

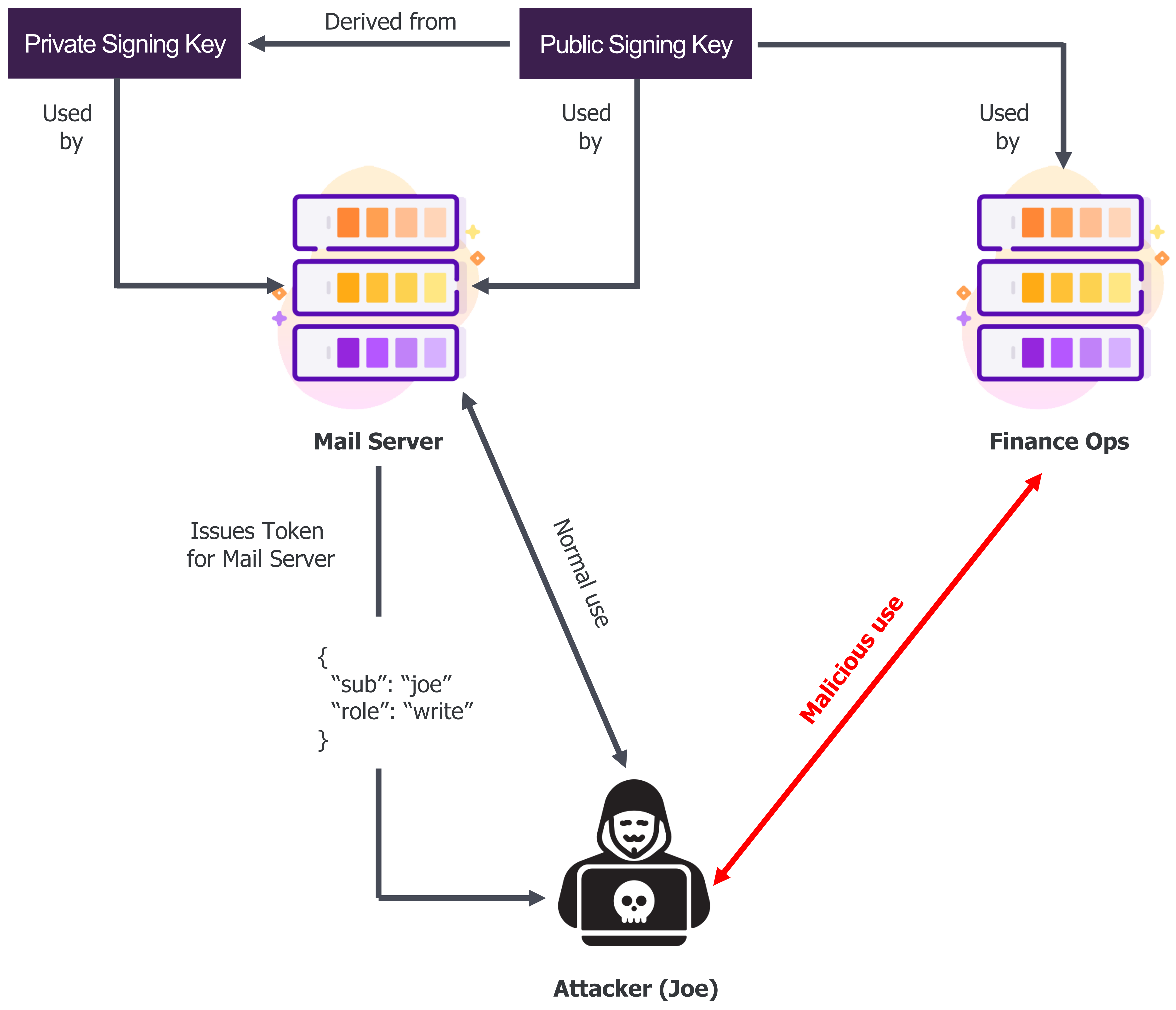
- Attacker gets a valid token for one organization / resource and uses it with another
- To prevent this, make each token specific to the issuer, subject, resource:
  - iss: URL of the IdP
  - sub: to whom it was issued
  - aud: audience for the token



© COPYRIGHT 42CRUNCH

## Substitution Attack: Cross JWT

- Lack of exact matching within the same organization
    - E.g. check for "aud": "myorg/*" instead of "aud":"myorg/finance-ops"
- Can also happen in multitenancy, site hosting, or any subdomains with any user content
- Use exact matching to protect yourself



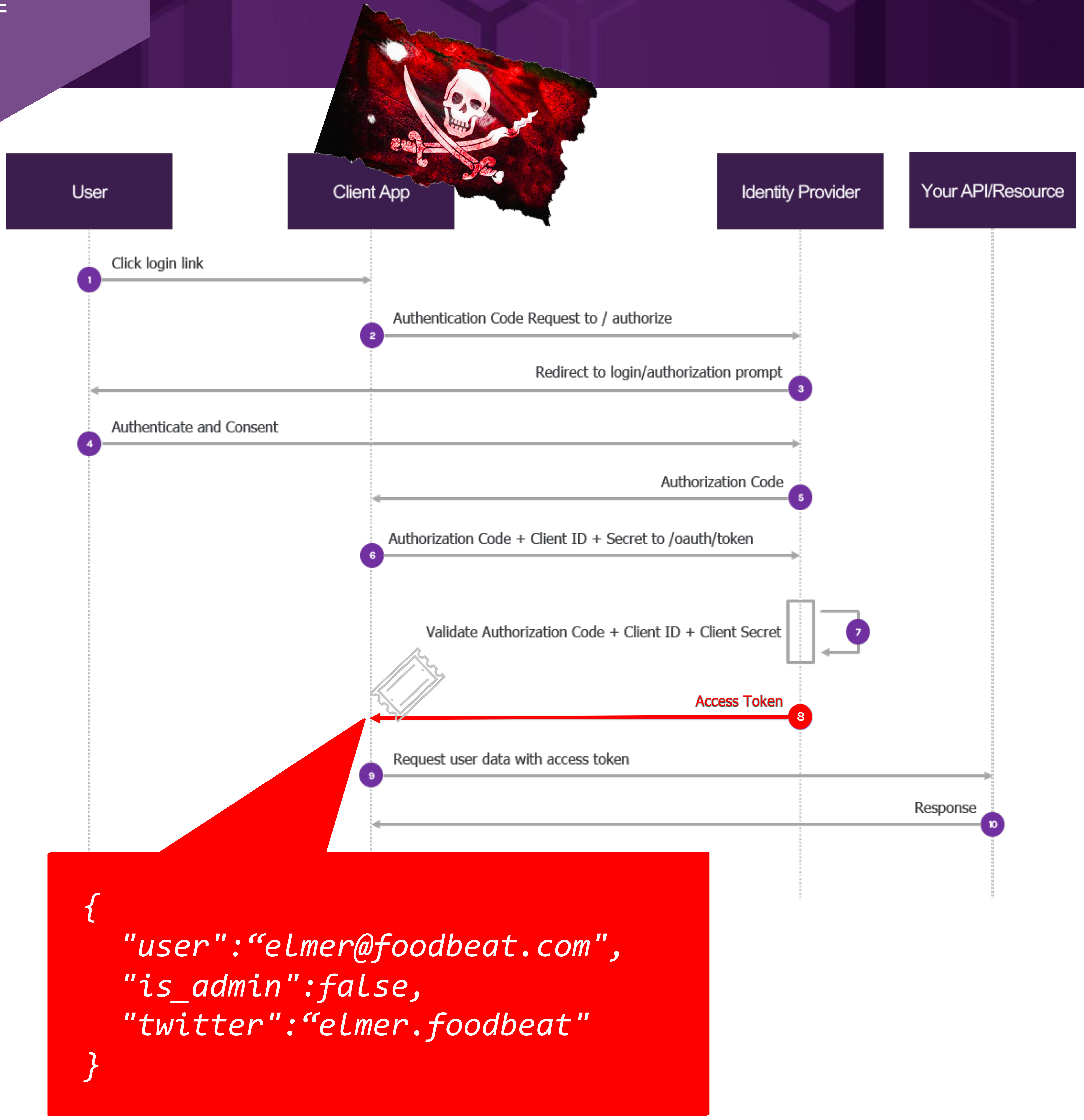
## Intercept and Reuse

- Attacker gets a hold of the token
- Since this is a bearer token with no time limits – they just keep using it as long as they want
- Set short time limits: exp, nbf
- Set minimal scopes
- Tie JWT to a specific client

```
{
  "user":"elmer@foodbeat.com",
  "is_admin":false,
  "iss":1579551140,
  "exp":1579551740
}
```

## These Tokens are not Opaque

- Client actually gets the token
- The tokens are not encrypted
- Rogue client can decode the token and get valuable info from it:
  - PII or other exposed info
  - Information about internals


User
Client App
Identity Provider
Your API/Resource
1 Click login link
2 Authentication Code Request to / authorize
3 Redirect to login/authorization prompt
4 Authenticate and Consent
5 Authorization Code
6 Authorization Code + Client ID + Secret to /oauth/token
7 Validate Authorization Code + Client ID + Client Secret
8 Access Token
9 Request user data with access token
10 Response

```
{
  "user":“elmer@foodbeat.com",
  "is_admin":false,
  "twitter":“elmer.foodbeat"
}
```

© COPYRIGHT 42CRUNCH

## Solution #1: Encryption

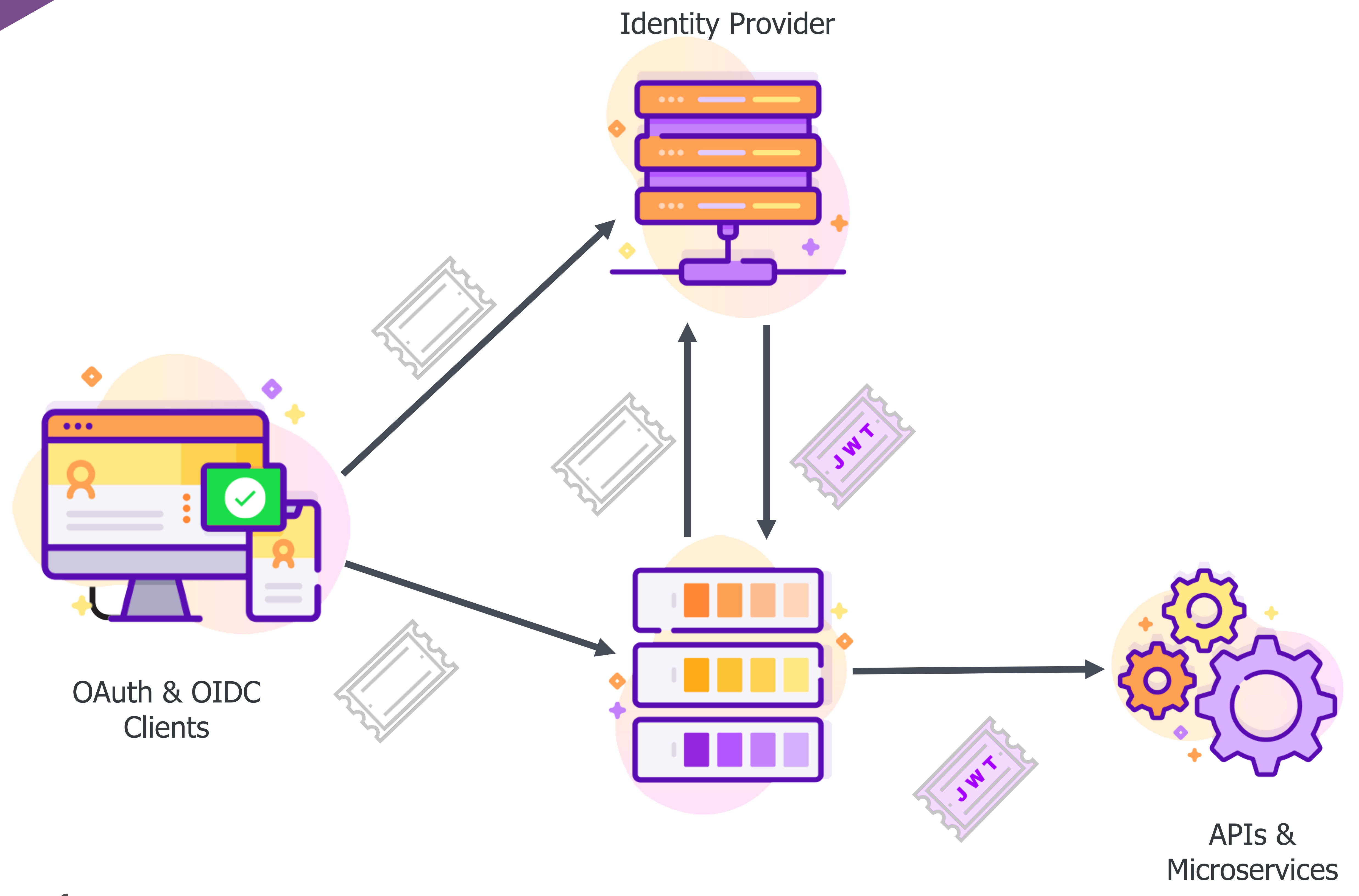
- JOSE header gets 2 extra parameters:
    - enc: algorithm for content encryption
    - zip: optional compression algorithm
- Algorithms used provide both integrity and confidentiality

```
BASE64URL-ENCODE(UTF8(JOSE Header)).
BASE64URL-ENCODE(JWE Encrypted Key).
BASE64URL-ENCODE(Initialization
Vector).
BASE64URL-ENCODE(Cyphertext).
BASE64URL-ENCODE(Authentication Tag)
```

© COPYRIGHT 42CRUNCH

## Solution #2: Phantom Tokens

- Give clients tokens with bare minimum information
- At the edge, exchange that phantom token to a full JWT that is only used within internal, server-side network



```
{
  "sub":"ec9f8fbb-a357-4fb6-
a6af-de6ce54fb3d2"
}
```

```
{
  "user":“elmer@foodbeat.com",
  "is_admin":false,
  "twitter":“elmer.foodbeat“
}
```

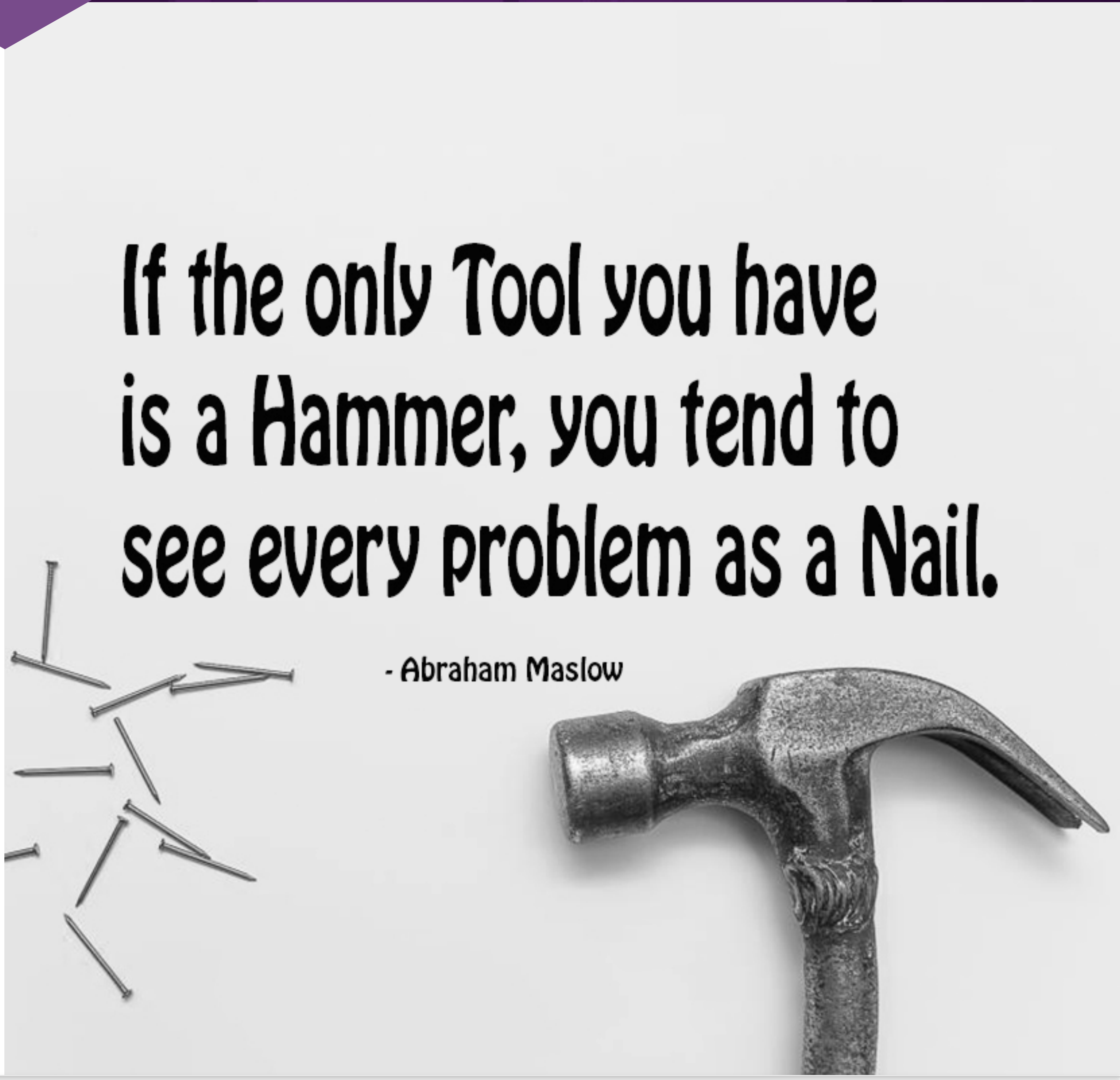
© COPYRIGHT 42CRUNCH

## Pattern and anti-patterns

© COPYRIGHT 42CRUNCH

## Is JWT right for you?

- HTTP headers have size limits (~ 8KB)
- Use of JWT can be an overkill
- They are a poor replacement of web app session tokens:
  - Much larger than a cookie
  - DB is likely used anyway
  - Web frameworks typically load users on incoming requests anyway
  - Cookies are signed and protected anyway
  - Caching and other site optimization will be a better solution than trying to persist all data and state in a JWT in a cookie


If the only Tool you have is a Hammer, you tend to see every problem as a Nail.
- Abraham Maslow

## Recommendations: Shared Secrets

- Safely store and retrieve
- Must be complex
- Key sets identified by kid
- Keys required for encrypted content validation

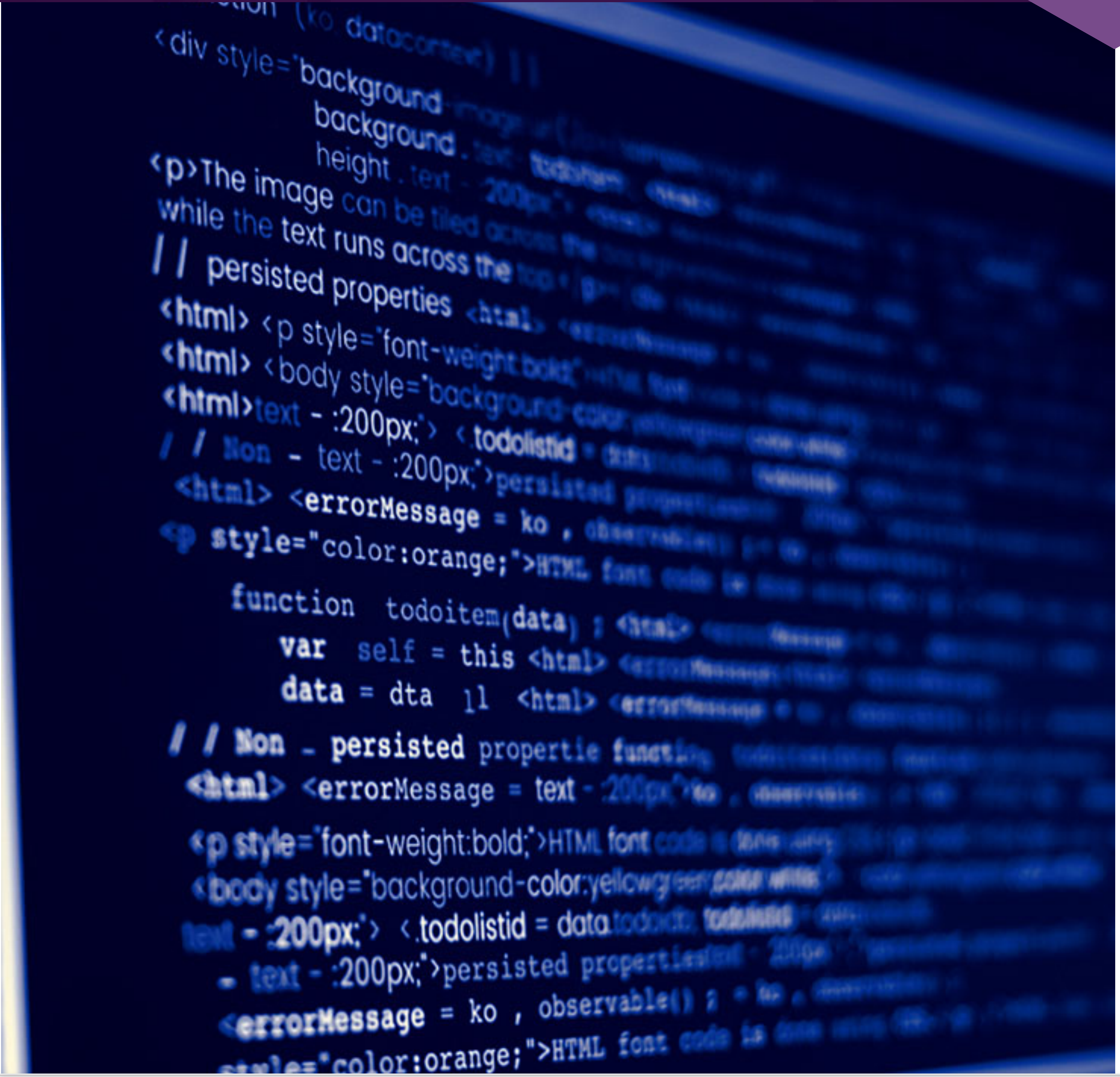
## Recommendations: Keys handling

- Safely store and retrieve
- Key sets identified by kid
- Keys required for encrypted content validation
- A key should be used for one and one only algorithm

© COPYRIGHT 42CRUNCH

## Recommendations: Internal Traffic and Encryption

- Keep external tokens opaque
- Internal tokens can carry payloads
- Encrypt sensitive information



## Recommendations: Token Validation

- Follow standards for all claims and processes
- Perform Algorithm Verification
- Use Appropriate Algorithms
- Validate All Cryptographic Operations
- Ensure Cryptographic Keys have Sufficient Entropy
- Validate Issuer and Subject
- Use and Validate Audience
- Do Not Trust Received Claims (injections and SSRF from kid and jku)

© COPYRIGHT 42CRUNCH

## How much of that can be externalized?

- OpenAPI Specification provides a standard machine-readable contract for APIs
- Includes: paths, operations, payloads, responses, authentication, scopes
- Can be enforced by API Gateways and API Firewalls
- JWT policies are not a part of OAS3

OPENAPI INITIATIVE

```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Swagger Petstore
servers:
  - url: http://petstore.swagger.io/v1
paths:
  /pets:
    get:
      summary: List all pets
      operationId: listPets
      tags:
        - pets
      parameters:
        - name: limit
          in: query
```

© COPYRIGHT 42CRUNCH

## OpenAPI Security-as-Code extensions from 42Crunch

- Define JWT server-side validation policies
- Can include any parameters and their values
- Can be applied across APIs, within a particular API, to a particular operation
- Can be audited during static code analysis
- Can be enforced by API Firewalls

```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Swagger Petstore
  servers:
  - url: http://petstore.swagger.io/v1
paths:
  /pets:
    get:
      x-42c-local-strategy:
        x-42c-strategy:
          protections:
          - jwt-validation_0.1:
              header.name: x-access-token
              jwk.envvar: JWK_PUBLIC_RSA_KEY
              authorized.algorithms:
              - RS256
              - RS384
      parameters:
       - name: limit
         in: query
```

© COPYRIGHT 42CRUNCH

## Additional Resources

- jwt.io
- Phantom tokens
- 42Crunch.com
- APIsecurity.io

© COPYRIGHT 42CRUNCH

# THANK YOU

- questions -