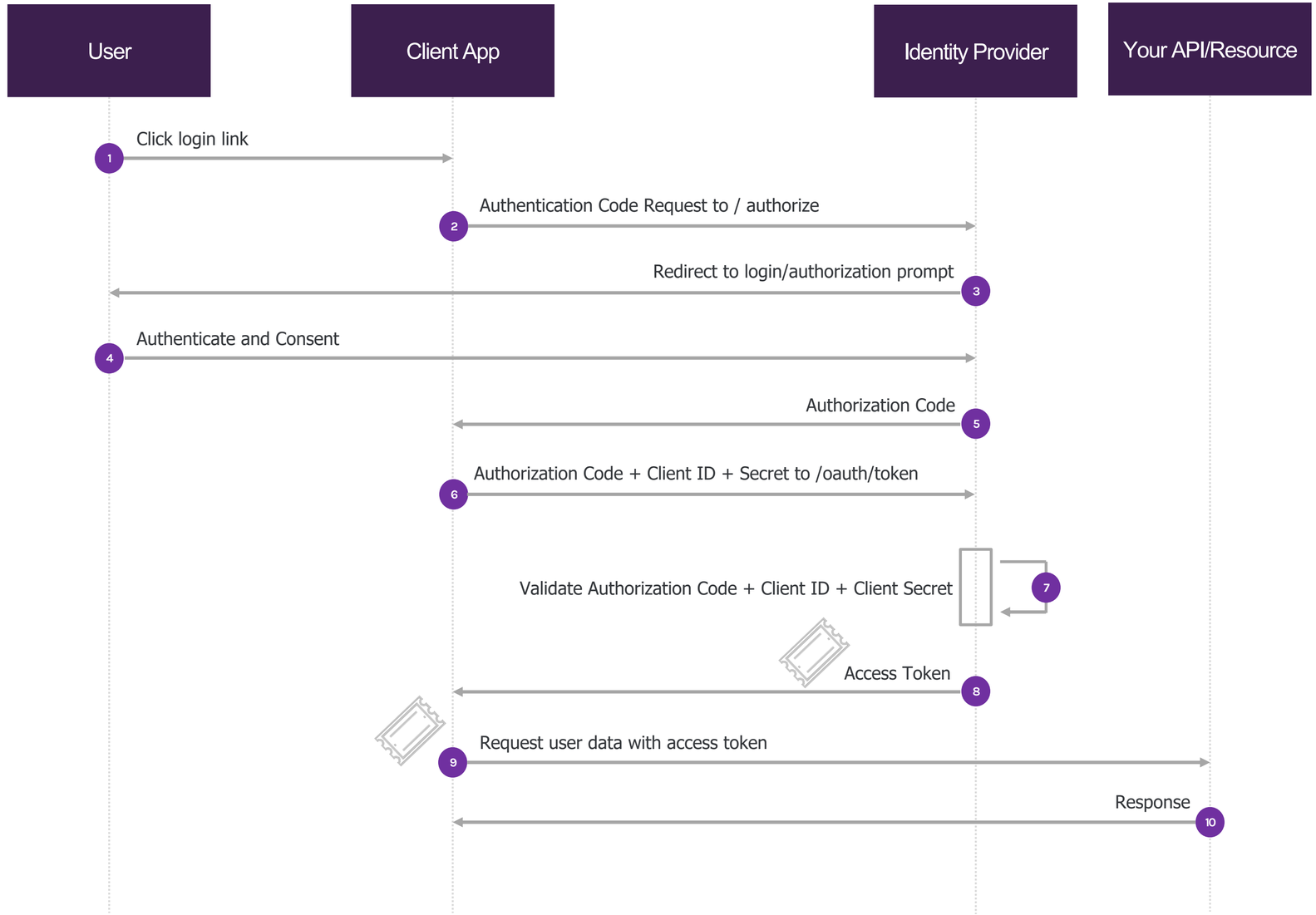
How to Best Leverage JWTs for API Security

Isabelle Mauny, 42Crunch Field CTO and co-founder
Dmitry Sotnikov, 42Crunch CPO, Curator of APIsecurity.io

Agenda

- 5-minute introduction to JWT ;)
- 2-minute introduction to API protection with 42Crunch
- Common JWT attacks and the way to protect against them

Why do we need tokens?



Tokens can be anything

ec9f8fbb-a357-4fb6-a6af-de6ce54fb3d2

Why JSON Web Tokens?

- User info right in the token
- Decouple resources and IdP
- No need for shared databases
- No extra API calls
- JSON is easy to use in code

```
{
  "user":"dmitry@42crunch.com",
  "is_admin":false,
  "twitter":"DSotnikov",
  "iss":1579551140,
  "exp":1579551740
}
```

Common Use Cases

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

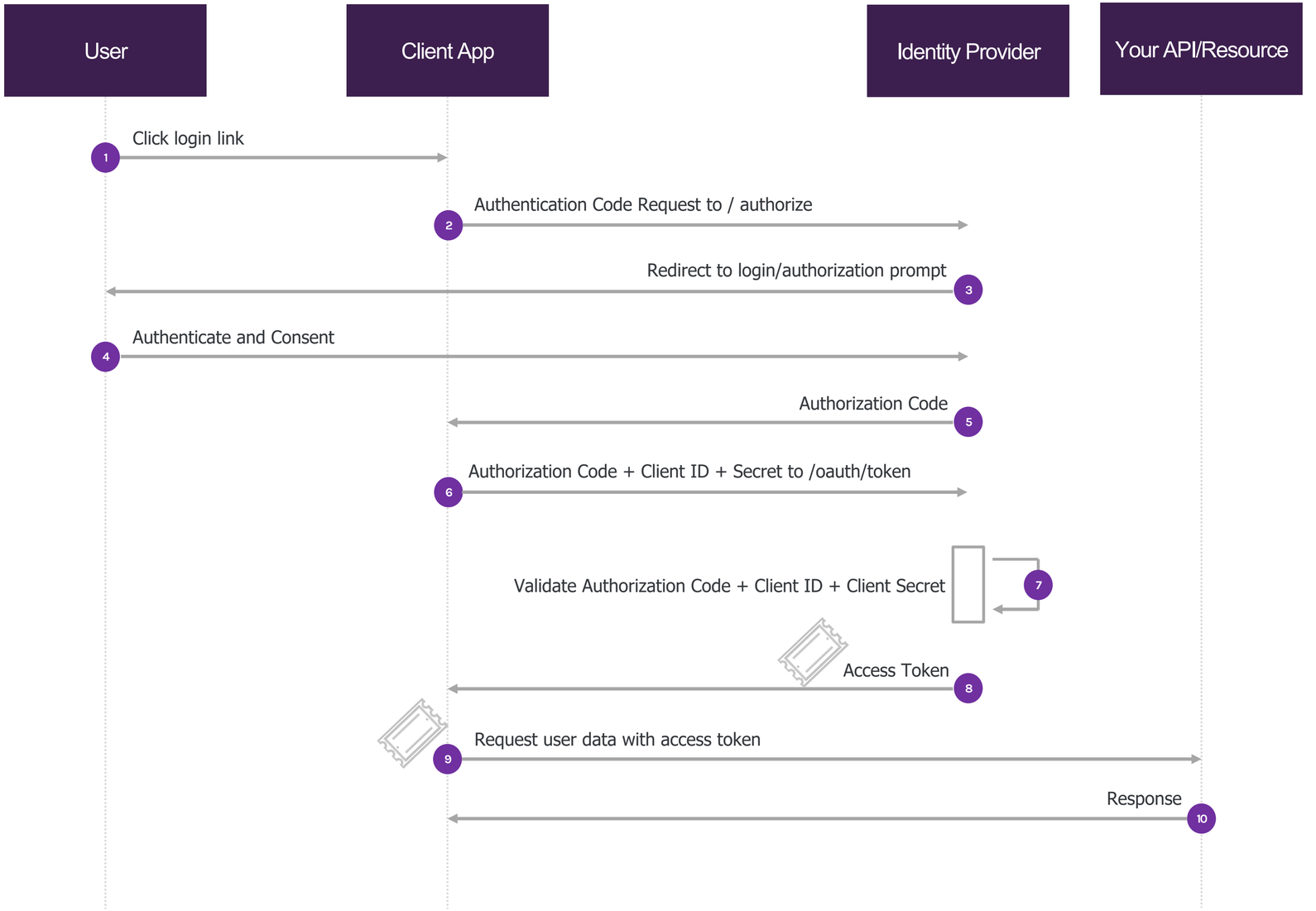
Tokens are Encoded

- 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 IdP?

- You sign them
- 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



Signing Process

1. Create JOSE header
2. Encode it

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

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

JOSE Header Key Parameters

- alg: None, RS256, HS256, … - full list in RFC7518
- 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
- typ: media type of this token: e.g. JWT
- cty: media type of JWT content
- crit: lists extensions that must be understood and processed

Signing Process

1. Create JOSE header
2. Encode it

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

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

Signing Process

1. Create JOSE header
2. Encode it
3. Create payload (does not have to be JSON)
4. Encode it too

```
{
"user":"dmitry@42crunch.com",
"is_admin":false,
"twitter":"DSotnikov",
"iss":1579551140,
"exp":1579551740
}
```

```
eyJ1c2VyIjoiZG1pdHJ5QDQyY3J1bmNoLmNvbSI
sImlzX2FkbWluIjpmYWxzZSwidHdpdHRlciI6Ik
RTb3RuaWtvdiIsImlzcyI6MTU3OTU1MTE0MCwiZ
XhwIjoxNTc5NTUxNzQwfQ
```

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

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXV
CJ9.eyJ1c2VyIjoiZG1pdHJ5QDQyY3J1b
mNoLmNvbSIsImlzX2FkbWluIjpmYWxzZS
widHdpdHRlciI6IkRTb3RuaWtvdiIsIml
zcyI6MTU3OTU1MTE0MCwiZXhwIjoxNTc5
NTUxNzQwfQ
```

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

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiZG1pdHJ5QDQyY3J1bmNoLmNvbSIsImlzX2FkbWluIjpmYWxzZSwidHdpdHRlciI6IkRTb3RuaWtvdiIsImlzcyI6MTU3OTU1MTE0MCwiZXhwIjoxNTc5NTUxNzQwfQ.n34z-LWu4INXl8-Cgac-Ues7r99xgbt_A4aHuCAZRLU


DEMO: Decode a Token

¯_(ツ)_/¯

What could possibly go wrong?!

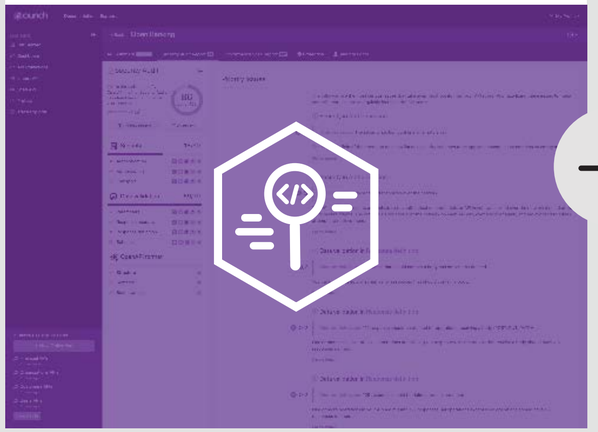
An API **should not blindly trust** anything it receives from the client.


42Crunch API Security
Platform

IDE & CI/CD Plugins

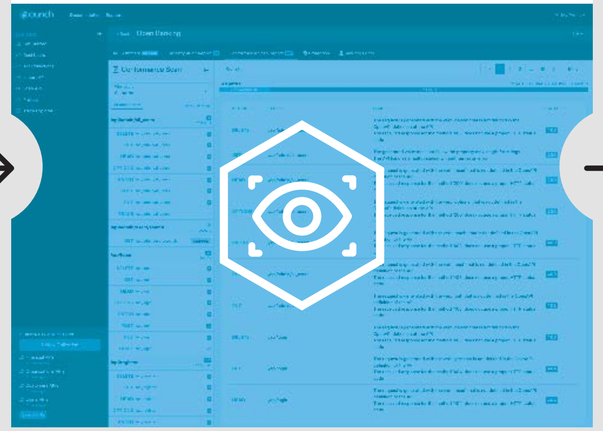
42Crunch API Security Platform

API Security Audit



- Discover existing and new APIs
- Perform 200+ security best practices checks
- Enforce corporate security policies

API Conformance Scan



- Dynamic security testing of API
- Discover implementation security flaws and discrepancies
- Unveil the full attack surface

API Firewall Protection

- Effective positive security API firewall
- Sub-millisecond overhead
- Works with existing API gateways & microservices deployments


Attacks and Remedy

None Algorithm Attack

1. Attacker modifies or creates a token

```
{
  "alg": "HS256",
  "typ": "JWT"
}.
{
  "user":"dmitry@42crunch.com",
  "is_admin":false
}.
X0Wglk3qxprLVTw2cYzuwEcJEEfED2F5XgmT
dQFY7A
```

None Algorithm Attack

1. Attacker modifies or creates a token

```
{
  "alg": "HS256",
  "typ": "JWT"
}.
{
  "user":"dmitry@42crunch.com",
  "is_admin":true
}.
X0Wglk3qxprPLVTw2cYzuwEcJEEfED2F5XgmTdQFY7A
```

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":"dmitry@42crunch.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":"dmitry@42crunch.com",
  "is_admin":false
}.
RSA signature with RSA private key

Changed to:
{
 "alg" : “HS256",
 "typ" : "JWT“
}.
{
  "user":"dmitry@42crunch.com",
  "is_admin":true
}.
HMAC signature with RSA public key
```


JWT Header Abuse

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:

Key.where("key = #{kid}").first


Attack value:

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

Command Injection

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

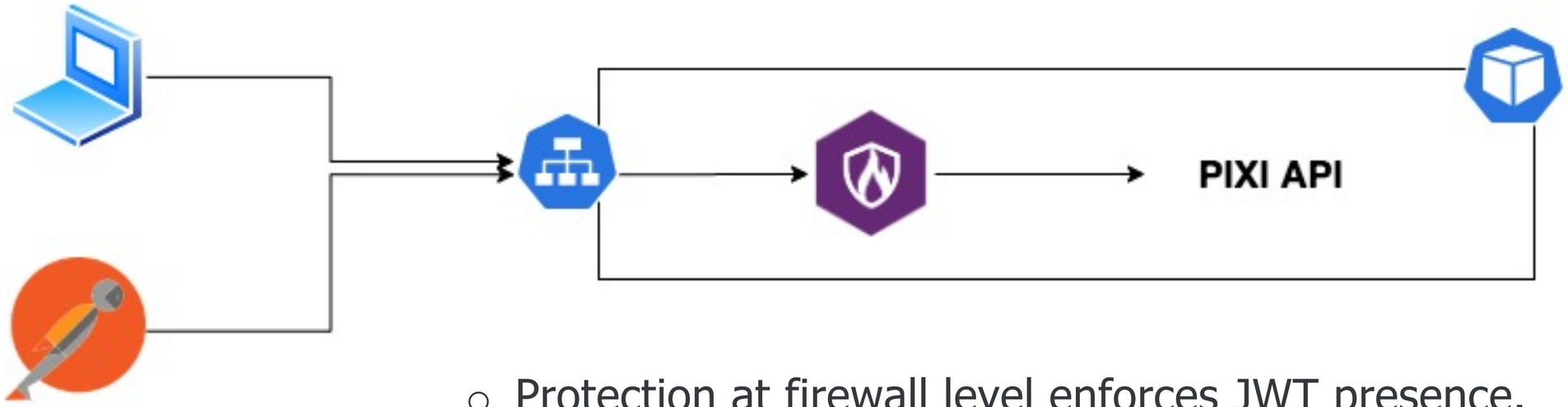
2. Attackers send an injection string and get their commands executed on the server

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


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

Demo:
Protecting JWT Header

Demo Setup



- Protection at firewall level enforces JWT presence.
- JWT is validated according to internal protection rules and custom header/payload schemas.

Demo: Header Validation

1. Algorithm: overall type (HMAC, RSA, EDCH) and custom type (ex PS256 only)
2. Type (set as JWT)
3. Kid : is it present ? And if yes, if it valid ?

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

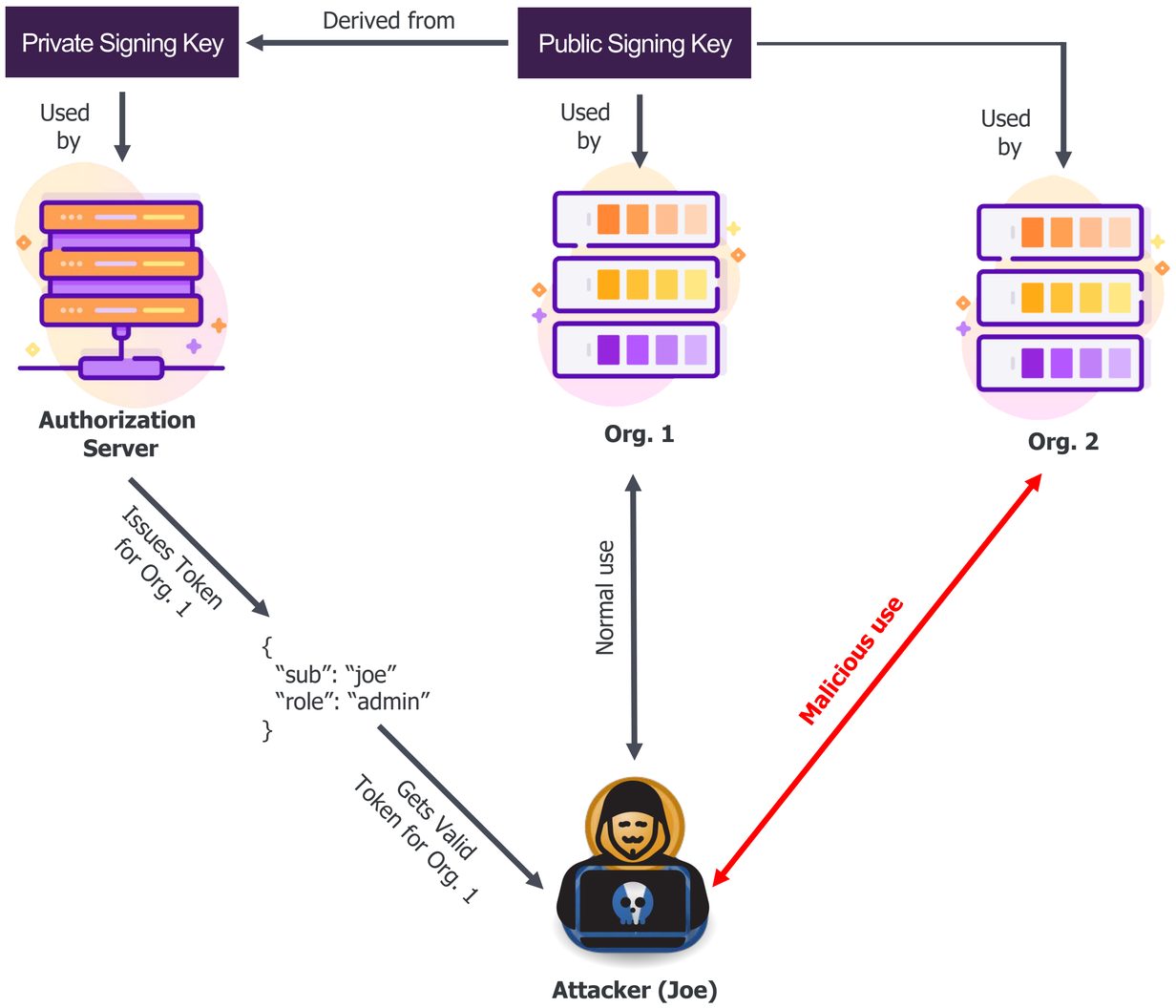
Bruteforce Attack on Signature

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")
```

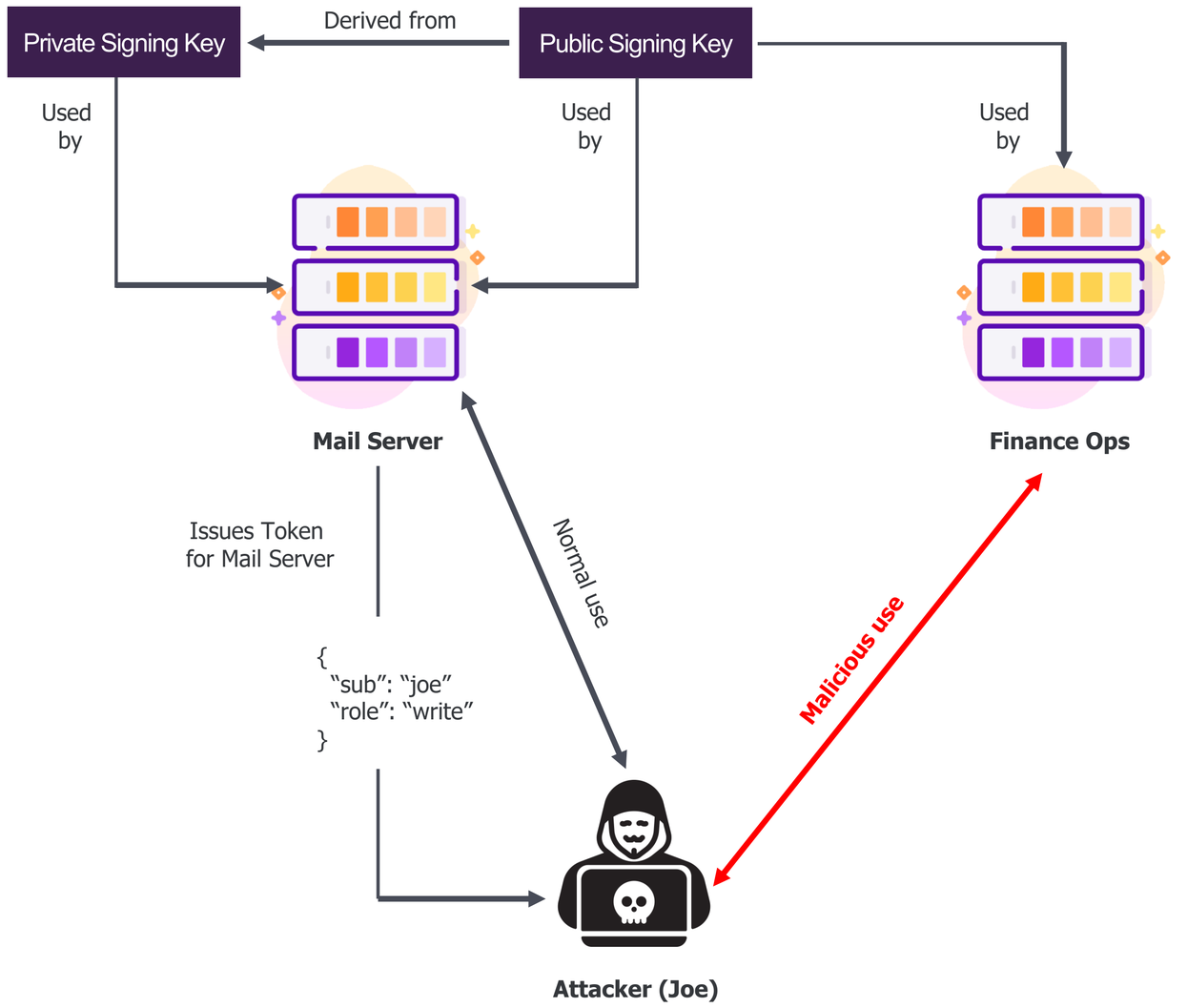
Substitution Attack: Different Recipient

- 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



Substitution Attack: Cross JWT

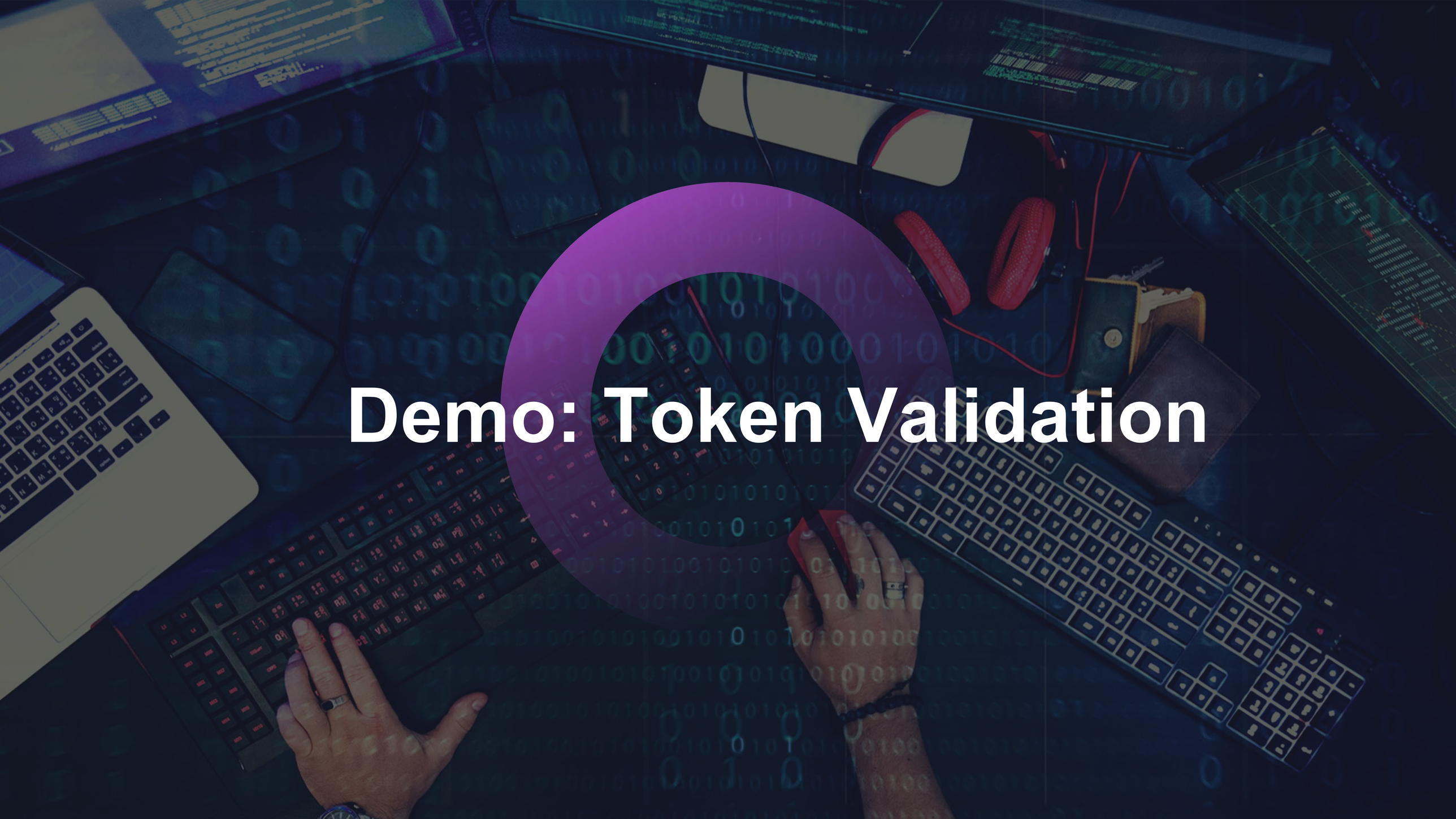
- Lack of exact matching within the same organization
    - E.g. check for "aud": "myorg/*" instead of "aud":"myorg/finance-ops"
- Can also happen in multitenancy, site hosting, or any subdomains with any user content
- Use exact matching to protect yourself



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

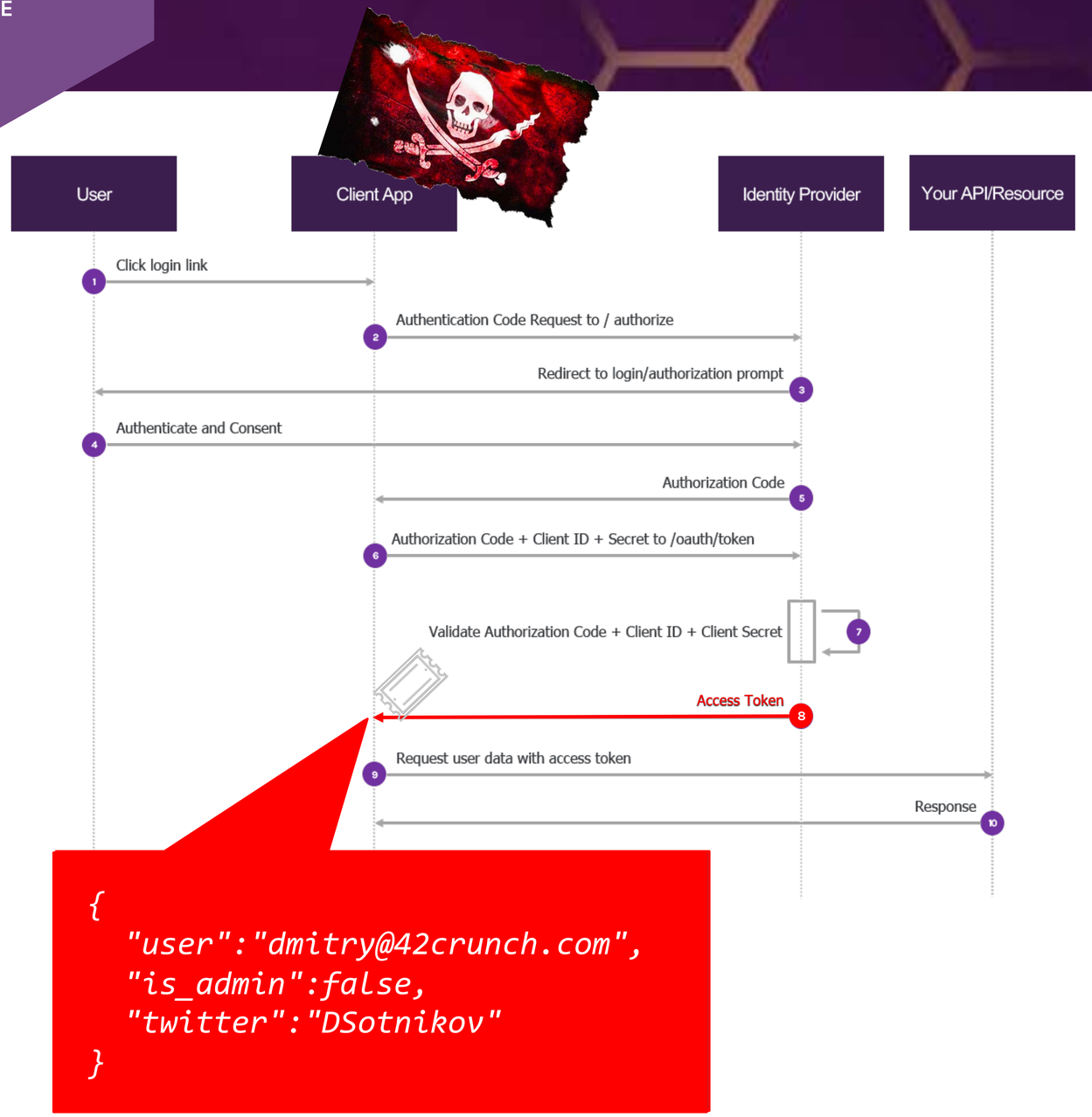

```
{
  "user":"dmitry@42crunch.com",
  "is_admin":false,
  "twitter":"DSotnikov",
  "iss":1579551140,
  "exp":1579551740
}
```



Demo: Token Validation

These Tokens are not Opaque

- Client 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



Summary

- Externalize JWT policies
- Do not trust the tokens
- Document and strictly enforce token schemas

Additional Resources

- jwt.io
- 42Crunch.com
- Documentation on 42Crunch JWT Protections
- APIsecurity.io
- JWT Validation Best Practices: https://datatracker.ietf.org/doc/rfc8725/

THANK YOU

- questions -