Three New Attacks Against JSON Web Tokens

Tom Tervoort
Speaker intro
1. Background
   - Transferring identity claims
   - JSON Web Tokens
   - Prior attacks
   - Criticisms

2. New attacks
   - Sign/encrypt confusion
   - Polyglot token
   - Billion hash attack

3. Takeaways
Background
Transferring identity claims
Classic (stateful) approach

My token: 4EC72A4BFF14A8CB

Whose token is this?
4EC72A4BFF14A8CB

Name: Alice
E-mail: alice@example.com
Birthdate: ...
Registered devices: ....
Member of groups: ...

Claim store (e.g. session DB or IdP)
Transferring identity claims
Cryptographic approach
## Comparison

<table>
<thead>
<tr>
<th>Stateful tokens</th>
<th>Signed/encrypted claims</th>
</tr>
</thead>
<tbody>
<tr>
<td>Many central DB lookups needed</td>
<td>Fast to verify and easy to scale</td>
</tr>
<tr>
<td>Mutable claims</td>
<td>Claims fixed until expiration</td>
</tr>
<tr>
<td>Trivially revocable</td>
<td>No revocation before expire date</td>
</tr>
<tr>
<td>Secrets are ephemeral</td>
<td>Requires key management</td>
</tr>
<tr>
<td>Token leak: compromise 1 user</td>
<td>Key leak: compromise all users</td>
</tr>
<tr>
<td>Easy to build, given secure RNG</td>
<td>Involves complex cryptography</td>
</tr>
</tbody>
</table>

Common hybrid approach: cryptographic access token and stateful “refresh token”
Cryptography is hard
JSON Web Tokens

- Massive improvement over legacy standards
- Proper integrity protection
- Easy to read and debug
- Simple and concise claims
- > 100 implementations
- Used by OpenID Connect
- They’re everywhere
Some JSON Web Acronyms

JWT (JSON Web Token): JSON-based claims format using JOSE for protection

JOSE (Javascript Object Signing and Encryption): set of open standards, including:

JWS (JSON Web Signature): JOSE standard for cryptographic authentication

JWE (JSON Web Encryption): JOSE standard for encryption

JWA (JSON Web Algorithms): cryptographic algorithms for use in JWS/JWE

JWK (JSON Web Keys): JSON-based format to represent JOSE keys
Prior JWT attacks

• Bypass signature validation by providing a token signed with the “none” algorithm
• Bypass blocklist filter with “nOne”…
• **Algorithm confusion**: using an RSA public key as an HMAC secret key
• **Key injection/self-signed JWT**: putting your own key in the “jwk” header
• Classic crypto attacks against primitives: RSA padding oracle; CurveSwap

• Probably most common: **simple dictionary words** being used as cryptographic keys
Important design flaws
(personal opinion)

1. Deciding the decryption/validation algorithm based on untrusted ciphertext
2. Letting end users choose between cryptographic algorithms
3. … including one broken since 1998 (RSA PKCS#1 v1.5 encryption) and “none”
4. Some algorithms are interchangeable, some dramatically change security properties
5. Over-engineered: trying to support many (obscure) use cases at once
New attack: sign/encrypt confusion
JWT flavors

### JWT in JWS format
A collection of claims in JSON format

```
{
  "iss": ..., 
  "sub": ..., 

```

*BASE64URL encoding*

- **Header**
- **Payload**
- **Signature**

### JWT in JWE format
A collection of claims in JSON format

```
{
  "iss": ..., 
  "sub": ..., 
```

*Encrypt* 

```
???????
??????
??????
??????
```

*BASE64URL encoding*

- **Header**
- **Encrypted Key**
- **Vector**
- **Payload**
- **Tag**

### Nested JWT (JWS in JWE pattern)
A collection of claims in JSON format

```
{
  "iss": ..., 
  "sub": ..., 
```

*Encrypt* 

```
???????
??????
??????
??????
```

*BASE64URL encoding*

- **Header**
- **Payload**
- **Signature**
- **Tag**

<table>
<thead>
<tr>
<th></th>
<th>Symmetric JWS</th>
<th>Asymmetric JWS</th>
<th>Symmetric JWE</th>
<th>Asymmetric JWE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Authenticity</strong></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>❌</td>
</tr>
<tr>
<td><strong>Confidentiality</strong></td>
<td>❌</td>
<td>❌</td>
<td>✔️</td>
<td>✔️</td>
</tr>
</tbody>
</table>
# JWT flavors

## JWT in JWE format

A collection of claims in JSON format:

```json
{
  "iss": ..., 
  "sub": ..., 
  ... 
}
```

Encrypt

```plaintext
????????
????????
????????
????????
```

BASE64URL encoding

<table>
<thead>
<tr>
<th>Symmetric JWS</th>
<th>Asymmetric JWS</th>
<th>Symmetric JWE</th>
<th>Asymmetric JWE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Authenticity</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Confidentiality</strong></td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

Image source: Takahiro Kawasaki
Should we expect developers to be crypto experts?

<table>
<thead>
<tr>
<th>&quot;alg&quot; Param Value</th>
<th>Key Management Algorithm</th>
<th>More Header Params</th>
<th>Implementation Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA1_5</td>
<td>RSAES-PKCS1-v1_5</td>
<td>(none)</td>
<td>Recommended</td>
</tr>
<tr>
<td>RSA-OAEP</td>
<td>RSAES OAEP using default parameters</td>
<td>(none)</td>
<td>Optional</td>
</tr>
<tr>
<td>RSA-OAEP-256</td>
<td>RSAES OAEP using SHA-256 and MDFI</td>
<td>(none)</td>
<td>Optional</td>
</tr>
<tr>
<td>A128KW</td>
<td>AES Key Wrap with default initial value using 128-bit key</td>
<td>(none)</td>
<td>Recommended</td>
</tr>
<tr>
<td>A192KW</td>
<td>AES Key Wrap with default initial value using 192-bit key</td>
<td>(none)</td>
<td>Optional</td>
</tr>
<tr>
<td>A256KW</td>
<td>AES Key Wrap with default initial value using 256-bit key</td>
<td>(none)</td>
<td>Recommended</td>
</tr>
<tr>
<td>d2I</td>
<td>Direct use of a shared symmetric key as the CEK</td>
<td>(none)</td>
<td>Recommended</td>
</tr>
<tr>
<td>ECDH-ES</td>
<td>Elliptic Curve Diffie-Hellman Ephemeral Static key agreement using Concat KDF</td>
<td>&quot;epk&quot;, &quot;apu&quot;, &quot;apv&quot;</td>
<td>Recommended</td>
</tr>
<tr>
<td>ECDH-ES+A128KW</td>
<td>ECDH-ES using Concat KDF and CEK wrapped with &quot;A128KW&quot;</td>
<td>&quot;epk&quot;, &quot;apu&quot;, &quot;apv&quot;</td>
<td>Optional</td>
</tr>
<tr>
<td>ECDH-ES+A192KW</td>
<td>ECDH-ES using Concat KDF and CEK wrapped with &quot;A192KW&quot;</td>
<td>&quot;epk&quot;, &quot;apu&quot;, &quot;apv&quot;</td>
<td>Optional</td>
</tr>
</tbody>
</table>

- Fine for JWTs
- Not suitable for JWTs!
What if we just avoid encrypted JWTs?

Key file:

```
{ "kty": "RSA", "n": "sEDRQ2skIS0rUYJawAPU4MFf6Y0xWymr6f6PQnCdnUaNPW14KDVJJ2XgNGG9Xodc-jRtCmp", "e": "A048", "d": "dSkR P7wQdjJYUEiQosFB4a2SKObH7WmQRbk1GQJzUMliZBGmKub_kn3653liKM8T8uoDQCF", "p": "2ubPBK8Rk9g81B0Mainf0fJpga4ZTCwntk1X4Rzb2JZ171TUYFeqT8G0FTRc0Tn1lW54c9jzI", "q": "sBDoJvNUNK7s-WDf1kr_69rXwJ1683jC2BxV3g2xY0y4bP7jYvnXemhH8KfNTqPbZ", "dp": "N2gJ-M4W2KuMBnNiVfVPUdkxklkE6qLJrNnu2KEBRFWk-g8Xdo0lWP8sEnzaJrW1-YqFva0w", "dq": "X0Fm9b08y1msQxbrLrjvZP6CwLmCUIp8YZ84p-2ot5ldEJqvvDDZbN1x0KjLoYY0hwVs", "q1": "10M5d-T18zL_0-NatMj3rFL8VI037Tr0Q6c1lp6p0W0KOL7BCyosYs06RvainM317vn"
}
```

JWT signer:

```
from authlib.jose import jwt, JsonWebKey
from time import time
import json

with open('rsa-key.jwk', 'r') as keyfile:
    key = JsonWebKey.import_key(json.load(keyfile))

header = {'alg': 'RS256'}
payload = {'iss': 'secure-issuer', 'sub': username, 'exp': round(time()) + 3600}
token = jwt.encode(header, payload, key).decode()
```

JWT validator:

```
from authlib.jose import jwt, JsonWebKey
import sys, json

with open('rsa-key.jwk', 'r') as keyfile:
    key = JsonWebKey.import_key(json.load(keyfile))

claims = jwt.decode(token, key)
username = claims.validate()['sub']
```
What if we just avoid encrypted JWTs?

Key file:

```
{
  "kty": "RSA",
  "n": "sEFRQ02sk150rUY1aWAPUMMF66Y0xWymrbf6PQmnCdnULa5Pi64KDVJx2xNGg9X0dXjR1Cmp",
  "e": "A0AB",
  "d": "ds1r P7wQijYMYeIoqFB4a2SK0hTHWmQgrb1Igu12MkKub kn38D3t1iKNMT80uIDQHF",
  "p": "2ubP8IRxK4B8TOMain0fJpaGA4ZTUCObwntIX4Rz2I2I1THUFeTq880DFcRTnW5QcijM",
  "q": "z1BoJvUNK7s-5WDXlkr_69rxwL6rI83jC2BxV3x2y0rybP74yvxeM1D68k5FT7qPbZ2",
  "d": "NzgJ-Mm2YKuM6nNldgPvPUDkK7e68qLRnUkEBFryJx-g8Xd0O1WP83ezaJrWl-YqSFvzw",
  "q": "X0FmH98z1y1msQxbrlJrv2PzMcLmCIAPYrZBp4-2ot51bD7qzuVDDZ2b1x0KpJlyYyOxV5",
  "q": "10H5d-T1aZL_o-NalM3r SlyV10s1iTr0G2c1Ip6pW0KOL7BCyos7So0VtainM3117nv"
}
```

JWT signer:
```
from authlib.jose import jwt, JsonWebKey
from time import time
import json

with open('rsa-key.jwk', 'r') as keyfile:
    key = JsonWebKey.import_key(json.load(keyfile))

header = {'alg': 'RS256'}
payload = {'iss': 'secure-issuer', 'sub': username, 'exp': round(time()) + 3600}

token = jwt.encode(header, payload, key).decode()
```

JWT validator:
```
from authlib.jose import jwt, JsonWebKey
import json

with open('rsa-key.jwk', 'r') as keyfile:
    key = JsonWebKey.import_key(json.load(keyfile))

claims = jwt.decode(token, key)

username = claims.validate()['sub']
```

RSA JWK file usable for:
- Signing
- Validation
- Encryption
- Decryption

Decides algorithm based on JWT header. Accepts RSA-encrypted JWE!
Sign/encrypt confusion attack

Preconditions:
1. Library supports asymmetric JWTs
2. App uses JWS tokens with RSA or ECDSA (RS*/PS*/ES*)
3. Private key accessible by validation function
4. No specific algorithm or JWT wrapper type is enforced
5. Attacker can determine public key. E.g. by:
   - Reading it from OIDC endpoint /jwks.json
   - If alg is RS*, can compute it from two tokens (https://github.com/SecuraBV/jws2pubkey)
New attack: polyglot JWT
A dangerous pattern

What if library A and library B parse JWTs differently?
Maybe exploit JSON ambiguity?

See also: https://bishopfox.com/blog/json-interoperability-vulnerabilities
Or an alternative serialization format?

JWS Compact Serialization

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJhbGljZSIsImlhdCI6MTUxNjIzOTAyMn0.rv61W60MY3WdNuyFrDb31rcbBpfuYWoS4fOi6Mmjeg

JWS Flattened JSON Serialization

{  "protected":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9",  "payload":"eyJzdWIiOiJhbGljZSIslhdCI6MTUxNjIzOTAyMn0",  "signature":"rv61W60MY3WdNuyFrDb31rcbBpfuYWoS4fOi6Mmjeg"}

JWT spec requires compact, but some libraries pass the JWT to a general JWS parser that accepts either type
Library mismatch

**python-jwt JWT validator**
(assumes compact)

```python
header, claims, _ = jwt.split('•')
parsed_header = json_decode(base64url_decode(header))
alg = parsed_header.get('alg')
if alg is None:
    raise _JWTError('alg header not present')
if alg not in allowed_algs:
    raise _JWTError('algorithm not allowed: ' + alg)
if not ignore_not_implmented:
    for k in parsed_header:
        if k not in JWSHeaderRegistry:
            raise _JWTError('unknown header: ' + k)
        if not JWSHeaderRegistry[k].supported:
            raise _JWTError('header not implemented: ' + k)

if pub key:
    token = JWT()
    token.allowed_algs = allowed_algs
    token.deserialize(jwt, pub key)
else
    if 'name' not in allowed_algs:
        raise _JWTError('no key but none alg not allowed')

parsed_claims = json_decode(base64url_decode(claims))
```

**jwcrypto JWS validator**
(first tries JSON; then compact)

```python
try:
jw = json_decode(raw jwt)
if 'signatures' in jw:
    o['signature'] = []
    for s in jw['signatures']:
        os = self._deserialize_signature(s)
        o['signature'].append(os)
        self._deserialize_base64(os, o, get('protected'))
else:
    o = self._deserialize_signature(jw)
    self._deserialize_base64(o, o, get('protected'))
if 'payload' in jw:
    if o.get('b64', True):
        o['payload'] = base64url_decode(str(jw['payload']))
    else:
        o['payload'] = jw['payload']

except ValueError:
c = raw jwt.split('•')
if len(c) == 3:
    raise ValueError('Unrecognized representation') from None
p = base64url_decode(str(c[0]))
if len(g) > 0:
    o['protected'] = p.decode('utf-8')
    self._deserialize_base64(p, o, 'protected')
    o['payload'] = base64url_decode(str(c[1]))
    o['signature'] = base64url_decode(str(c[2]))
```
A polyglot token

```json
{
  "AAAA": ".XXXX.",
  "protected": "AAAA",
  "payload": "BBBB",
  "signature": "CCCC"
}
```
A polyglot token

`jwcrypto` ignored unknown JSON fields:

```
{
    "AAAA": ".XXXX.",
    "protected": "AAAA",
    "payload": "BBBB",
    "signature": "CCCC"
}
```
A polyglot token

```python
import jwt

# Split on periods and ignore non-base64 characters

header = "AAAA": "XXXX",
"protected": "AAAA",
"payload": "BBBB",
"signature": "CCCC"
```

Given a token with a legitimate payload, the attacker can replace it with any spoofed claims.
New attack: billion hashes attack
Some interesting JWE "alg" values

<table>
<thead>
<tr>
<th>PBES2-HS256+A128KW</th>
<th>PBES2 with HMAC SHA-256 and &quot;A128KW&quot; wrapping</th>
<th>&quot;p2s&quot;, &quot;p2c&quot;</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBES2-HS384+A192KW</td>
<td>PBES2 with HMAC SHA-384 and &quot;A192KW&quot; wrapping</td>
<td>&quot;p2s&quot;, &quot;p2c&quot;</td>
<td>Optional</td>
</tr>
<tr>
<td>PBES2-HS512+A256KW</td>
<td>PBES2 with HMAC SHA-512 and &quot;A256KW&quot; wrapping</td>
<td>&quot;p2s&quot;, &quot;p2c&quot;</td>
<td>Optional</td>
</tr>
</tbody>
</table>

4.8. Key Encryption with PBES2

This section defines the specifics of performing password-based encryption of a JWE CEK, by first deriving a key encryption key from a user-supplied password using PBES2 schemes as specified in Section 6.2 of [RFC2898], then by encrypting the JWE CEK using the derived key.
What can go wrong?

- Standard designer wants versatility: includes useful PBES algorithms
- Library implementer wants feature-completeness: implements all JWE algorithms
- Library implementer wants simple and clean interface: same API for all algorithms
- User decodes token with default settings, assuming these must be secure

- Result: application will try to decrypt JWTs claiming to be encrypted with a password, even though that doesn’t really make sense

- But if there’s no token spoofing cross-protocol attack between PBES and other algorithms this should not be a problem, right?
A PBES header parameter

4.8.1.2. "p2c" (PBES2 Count) Header Parameter

The "p2c" (PBES2 count) Header Parameter contains the PBKDF2 iteration count, represented as a positive JSON integer. This Header Parameter MUST be present and MUST be understood and processed by implementations when these algorithms are used.

The iteration count adds computational expense, ideally compounded by the possible range of keys introduced by the salt. A minimum iteration count of 1000 is RECOMMENDED.
DoS with a token header

```
{
    "alg": "PBES2-HS512+A256KW",
    "p2s": "AAAAAAAAAAAAAAAAAAAAA",
    "p2c": 2147483647,
    "enc": "A128CBC-HS256"
}
```

- Rest of the JWE can consist of bogus strings.
- The server needs to perform more than 4 billion SHA512 hashes to derive the token encryption key in before it can determine that this JWT is invalid.
- **Unauthenticated**: attacker does not need to know what a valid token looks like.
- It has to do this for **every request** with a JWT!
Takeaways
JWT library research

- Focus on popular open source libraries. Could not cover all 100+ JWT libraries!
- Vulnerabilities mainly found in highly featured libraries.
- Responsible disclosure very pleasant: fast and excellent response in each case
- Vulnerabilities found and mitigations implemented in the following libraries:

<table>
<thead>
<tr>
<th>Library</th>
<th>Language</th>
<th>Affected versions</th>
<th>Vulnerability</th>
<th>CVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authlib</td>
<td>Python</td>
<td>&lt; v1.1.0</td>
<td>Sign/encrypt confusion</td>
<td>CVE-2022-39174</td>
</tr>
<tr>
<td>JWCrypto</td>
<td>Python</td>
<td>&lt; v1.4</td>
<td>Sign/encrypt confusion</td>
<td>CVE-2022-3102</td>
</tr>
<tr>
<td>JWX</td>
<td>PHP</td>
<td>&lt; 0.12.0</td>
<td>Sign/encrypt confusion</td>
<td></td>
</tr>
<tr>
<td>Python-jwt</td>
<td>Python</td>
<td>&lt; v3.3.4</td>
<td>Polyglot token</td>
<td>CVE-2022-39227</td>
</tr>
<tr>
<td>Jose</td>
<td>JavaScript</td>
<td>&lt; v1.28.1, v2.0.5, v3.20.3, v4.9.1</td>
<td>Billion hashes</td>
<td>CVE-2022-36083</td>
</tr>
<tr>
<td>Jose-jwt</td>
<td>.NET</td>
<td>&lt; v4.1</td>
<td>Billion hashes</td>
<td></td>
</tr>
</tbody>
</table>
Recommendations for JWT library developers

• Less is more: don’t implement features with rare use cases, or turn them off by default.

• Don’t use the “alg” parameter in the token to decide the algorithm. Instead force users to make this explicit in their code or key file.

• Don’t support JWTs using asymmetric or password-based encryption.

• Avoid validate-then-parse-again patterns.
Recommendations for the JOSE working group

- Specify security recommendations to avoid the issues discussed here.
- Explicitly list which JWS and JWE algorithms are allowed for JWTs. Exclude the likes of “none”, PBES and public key encryption.
- Encourage existing methods to enforce that a key is only used with a single algorithm.
- Ideally, remove “alg” from token headers altogether.
Reconsider if you really need encrypted claims. Boring old random tokens have many advantages!

• Consider JWT alternatives like PASETO, Macaroons or Biscuits.
• When using JWT, always explicitly configure the validation algorithm.
• A JWT validation library is a critical dependency. Don’t forget to patch them!
Thank you!