Three New Attacks Against JSON Web Tokens

By: Tom Tervoort, Secura BV

JSON Web Tokens (JWTs) have become omnipresent tools for web authentication, session management and identity federation. However, some have criticized JWT and associated Javascript Object Signing and Encryption (JOSE) standards for cryptographic design flaws and dangerous levels of unnecessary complexity. These have arguably led to severe vulnerabilities such as the well-known "alg":"none" attack.

When examining JOSE standards myself I also noticed a few potential "foot guns" that might result in JWT library implementers introducing vulnerabilities if they were to interpret the RFC in certain ways. This prompted me to investigate various JWT libraries for vulnerabilities. As a result, I managed to identify three new classes of JWT attacks affecting at least six different implementations. Two of these attacks ("sign/encrypt confusion" and "polyglot token") can allow complete token forgery, allowing authentication bypasses or privilege escalation in applications using an affected library and configuration. The third ("billion hashes") attack can be leveraged for a denial-of-service attack against token-processing servers.

In this whitepaper, I will outline these new vulnerability classes (and one other finding) and show how to exploit them. While each could be considered to be purely implementation bugs, I argue that they are also the result of understandable RFC interpretations and that these problems are indicative of broader issues with the JOSE standards.

What's wrong with JWTs?

Nowadays, JSON Web Tokens (JWTs) are by far the most popular mechanism for cryptographically protected "tokens" that transfer identity and privilege information ("claims") about an application user or client. They are generally stored by the parties whose identity the tokens describe. To prevent users from altering their own tokens in order to impersonate others or elevate privileges, cryptographic integrity protection is used. Only those who possess the right secret key should be able to issue tokens.

JWT claims are encoded as a JSON object that is subsequently wrapped in a JSON Web Signature (JWS) or JSON Web Encryption (JWE) object, or a combination of the two. JWS and JWE are two of the Javascript Object Signing and Encryption (JOSE) standards. These define a variety of cryptographic objects with syntax based on familiar encoding mechanisms like JSON or base64.

But is the cryptography any good? I would say that JWS and JWE schemes offer a large improvement of legacy standards such as XML Encryption (which is pretty much fundamentally broken), or shoddy proprietary solutions from Oracle or IBM. Nonetheless, cryptographers still love to complain about JWT cryptography. Personally, I particularly agree with the following criticisms:

1. The `"alg"` parameter, which indicates which cryptographic algorithm to use, is part of the token itself. Because the token could have been spoofed by an attacker, this basically means it's the attacker who tells the verifier what kind of cryptography to use (opening op a huge array of possibilities for cross-protocol attacks, like the the "alg":"none" and HMAC/RSA confusion attacks). While it is possible to also restrict algorithms in the configuration or store the algorithm along the key, this is not enforced.
2. Instead of having one strong set of cryptographic primitives selected by experts, it is left up to developers (who are probably not cryptographic experts) to make a choice from many different options (including one algorithm that was effectively broken 17 years before the RFC was published!).
3. What makes points 1 and 2 worse is that different algorithms can have widely different security properties. Based on the `"alg"` value, your token could for example be protected with a Message Authentication Code (MAC), signed, encrypted with a shared secret or encrypted with a public key. An then there is the none value, which indicates no cryptographic protection should be performed at all!
4. Besides many options for cryptographic algorithms, a large amount of optional features are defined ranging from compression to X.509 certificate processing. Instead of trying to solve one problem well the JOSE specifications attempts to support many (obscure) use cases, thereby introducing more complexity and attack surface.

Due to issues like this, it is easy for developers using JWTs or those building JWT libraries to make subtle mistakes which in turn could have high impact. JWT validation is generally a security-critical process and any exploits in that area could lead to privilege escalation as well as full authentication bypasses. Because of this, I decided to take a closer look at the JWT RFCs as well as the source code of a variety of open source JWT libraries, which resulted in the discovery of three novel attack techniques, which I will describe here.

New attack 1: Sign/encrypt confusion

Theory

RFC 7519 allows JWTs to either be signed (the JWS form) or encrypted (the JWE form). When using JWS the claims within the JWT are not hidden from the owner but the token is still protected against tampering; when using JWE the token contents are hidden from them. It is also possible to combine both forms by embedding a JWS inside a JWE, but claims that are directly embedded in a JWE are also allowed.

Both JWS and JWE allow for both symmetric and asymmetric algorithms. These have different properties, however: for example, when using a symmetric JWS algorithm (like HS256) the JWT can both be created and validated with the same shared secret; meanwhile, when using an asymmetric algorithms (like RS256) the JWT can only be created by the private key owner but validated by everyone. When using a symmetric JWE you need access to a secret key in order to either encrypt or decrypt it. Because authenticated encryption is used, this is fine: when someone doesn't know the key they will not be able to tamper with claim contents.

But what about JWE objects with asymmetrically encryption? In that case a public key is used to encrypt and a private key is used to decrypt. In the context of JWTs that would mean the private key owner could inspect or validate tokens, but no secrets are needed to issue tokens! See also the table below. Clearly allowing anyone to forge a token goes against the main goal of JWTs.

	"none" JWS	symmetric JWS	asymmetric JWS	symmetric JWE	asymmetric JWE
Needs secret to read	no	no	no	yes	yes
Needs secret to change	**no**	yes	yes	yes	**no**

While RFC 7519 allows encrypted tokens to be wrapped in a single JWE object it does not forbid the use of asymmetric algorithms. In fact, it doesn't mention them at all. This results in many implementations adding support for these types of asymmetrically encrypted JWTs. Perhaps there is some obscure use case where this may be useful, but in most situations where you use JWTs you probably do not want everyone to be a token issuer.

For this reason, a developer well-versed in cryptography will probably not choose to use asymmetric JWE. However not every developer may be aware of this distinction: during pentests I have seen multiple cases of signing and public-key encryption being confused, resulting in security properties opposite to what was expected. This is quite a dangerous feature to support.

Even more interestingly, it turns out that even when a developer does not choose to use encryption at all, they may still be vulnerable to a new type of cross-protocol attack under the following conditions:

1. They are using a library that accepts both JWS or JWE wrapped JWTs. This is explicitly allowed by the RFC, which describes an automatic method of distinguishing between both types of JWT.
2. The library accepts JWE JWTs encrypted with a public key.
3. The developer is issuing asymmetrically signed tokens, using a private/public key pair.
4. The same private/public key pair is supplied to the validation routine (while technically only the public key is needed for validation this is common in practice when the same server both issues and validates tokens).

5. The developer does not enforce a specific validation algorithm, and the library does not require this by default.

In this situation the application will be issuing signed tokens that an attacker can not tamper with directly. However, what an attacker can do is encrypt a token with the same public key that is used for signing. A vulnerable library will then decrypt this JWE object with the private key and consider it as authentic, even though the developer did not intend to use encrypted tokens in the first place.

This attack would require the attacker to first figure out the public key that is in use. Since this key does not need to be kept secret (hence the name) it is often published somewhere, such as in a standard OpenID Connect Endpoint. Even when the public key is not published, some algorithms (including the highly common RS256, RS384 and RS512 options) allow it to be computed from just two signatures. In that case the attacker just has to obtain two different samples of legitimate signed tokens, after which they can start forging encrypted ones.

Exploitation

I identified three libraries against which this attack was possible. A simple example of a vulnerable token validator using one of these libraries (Authlib) is as follows:

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

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

def validate(token):
    claims = jwt.decode(token, key)
    claims.validate()
```

This code seems innocuous and is similar to the library's sample code. However, when the key file being loaded here happens to contain a private key along with its public key, it becomes vulnerable to the attack.

To exploit this, an attacker first needs to figure out the public key. I published a script that can compute it relatively efficiently in case the RS256, RS384 or RS512 algorithm (RSA with PKCS#1 padding) is used.

Once the public key is computed or otherwise obtained, the attack becomes trivial: just take an existing token's claims and adjust them as you like (or just write claims from scratch). Then you can use any

JWE encryption tool to make a forged JWT. Personally, I like to use the Burp Suite plugin JWT Editor for this purpose.

Affected libraries

- Authlib before version 1.1.0 (CVE-2022-39174)
- JWCrypto before version 1.4 (CVE-2022-3102)
- JWX before version 0.12.0

New attack 2: Polyglot token

Theory

There are several ambiguities in JSON standards and implementations, that can cause different parsers to implement the same string differently. When the same JSON object is passed processed by different systems, these parser inconsistencies can lead to vulnerabilities.

Because JWTs are based on JSON, I decided to look around for situations where a token was passed around between multiple parser implementations and where a parser inconsistency might cause a vulnerability. Consider a case where parser A is used to validate a JWT, but where parser B actually interprets its contents after A does not yield an error. If there is an inconsistency between these parsers, an attacker might construct an input that seems identical to a legitimate JWT to parser A, but will be interpreted to contain different attacker-chosen claims by parser B. This would result in a token tampering vulnerability.

While I did not manage to find an exploitable JSON parser inconsistency, I did find a token spoofing attack caused by another parser ambiguity: namely on which JWS representation to use. A JWS can actually be represented using three different kinds of syntax: *compact serialization*, *general JSON serialization* and *flattened JSON serialization*. The JWT RFC specifies that only the compact serialization should be used. However, many JWT libraries make the understandable decision to pass on their token to a general JWS library, and that JWS library may actually support more formats.

Such an inconsistency was present between the `python-jwt` JWT validator and the `jwcrypto` JWS validator: an attacker could forge an input that appeared towards `jwcrypto` as a JSON serialized JWS with a valid signature. However, when interpreted as compact serialization by `python-jwt`, a different payload would be parsed than the one over which the signature was validated. This resulted in a general forgery attack against `python-jwt`.

Exploitation

The vulnerable `python-jwt` code was as follows:

```
def verify_jwt(jwt,
               pub_key=None,
               allowed_algs=None,
               iat_skew=timedelta(),
               checks_optional=False,
               ignore_not_implemented=False):
    [...]

    header, claims, _ = jwt.split('.')

    parsed_header = json_decode(base64url_decode(header))

    [...]

    if pub_key:
        token = JWS()
        token.allowed_algs = allowed_algs
        token.deserialize(jwt, pub_key)
    elif 'none' not in allowed_algs:
        raise _JWTError('no key but none alg not allowed')

    parsed_claims = json_decode(base64url_decode(claims))

    [...]

    return parsed_header, parsed_claims
```

This code assumes the JWT is encoded using the compact serialization method, and therefore splits it on periods. Then the header and claims components are decoded, the entire JWT is validated using `jwcrypto` (the `deserialize` method will throw an exception when the signature is invalid) and finally the header and claims as originally parsed are returned.

To attack this, an attacker first has to obtain some legitimate (unprivileged) token with a valid signature. This uses compact serialization, which looks like this:

```
AAAA.BBBB.CCCC
```

Here the A's represent the JWS header, B's are the claims and C's are the signature. The "flattened" JSON equivalent of this JWS object is as follows:

```
{
    "protected": "AAAA",
    "payload": "BBBB",
    "signature": "CCCC"
}
```

This JWS would also be accepted by `jwcrypto`'s `deserialize` method and treated identically to the former one. However, when supplied to the `python-jwt` verifier an exception would be raised by the line `jwt.split('.')`, because there is no period in this string.

However, an attacker can modify this representation as follows:

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

From the perspective of `jwcrypto`, this is a valid JSON serialized JWS with all the necessary field. The additional AAAA field has no meaning in JWS objects and is simply ignored. `jwcrypto`'s validator will therefore not raise an exception.

`python-jwt` would however split this JSON object on dots and arrive at the following values for claims and header:

```
header: {"AAAA:"
claims: XXXX
```

The `header` contains a number of characters that are not part of the url-safe base64 alphabet. Luckily for the attacker, however, the parser used by the `base64url_decode` function used here will simply discard these invalid characters and decode the header in the same way as AAAA.

Finally, XXXX will be returned as the validated claims from the JWT. These were however not covered by any signature and completely falsified by the attacker. The attacker can therefore spoof their identity and any other claim asserted to by the JWT.

Affected libraries

- python-jwt before version 3.3.4 (CVE-2022-39227)

New attack 3: Billion hashes attack

Theory

Besides key-based symmetric and asymmetric cryptography, the JWE standard also supports password-based encryption through the PBES2 algorithms. Instead of a key, these algorithms take a password as a secret parameter and apply the password-based key derivation PBKDF2 to it in order to derive an encryption key.

Because passwords are often picked by humans, using predictable patterns, password-based encryption is vulnerable to offline *dictionary* and *brute-force* attacks where an attacker tries out many potential passwords in succession until they find the one with which they can decrypt a ciphertext. In order to somewhat mitigate this risk, the PBKDF2 function is parametrized with an *iteration count* parameter that defines how many successive cryptographic hash operations need to be carried out in order to turn a password into a key. The higher the iteration count, the slower the function becomes.

The idea behind intentionally slowing down this cryptographic operation is to make password guessing attacks more difficult, but you can't set this parameter too high or else it will become to slow for regular use. The implementer therefore has to try to find an iteration count value that has an appropriate security/performance trade-off.

In JWE, this iteration count is defined in the token header `p2c`. This is fine for some password-based encryption use cases, but creates a problem when an attacker can supply JWE objects that are being processed automatically: in that case an attacker could set `p2c` to an extremely high value in order to cause denial of service at the server doing the processing. Changing this value will invalidate the JWE authentication tag. However this tag can only be validated after the key is derived, and the expensive PBKDF2 computation has therefore already been carried out at that point.

PBES2 algorithms are typically not used for JWTs, however various libraries do support these algorithms and I found two instances where PBES2 is allowed by default, even if the user did not specify a password. Instead, these libraries would treat the configured encryption key as a password whenever a PBES2 algorithm was indicated, and perform the PBKDF2 function every time a new PBES2 token is provided. This allowed an unauthenticated resource exhaustion attack against these libraries.

Exploitation

To exploit this, simply create a JWE with the following header:

```
{
    "alg": "PBES2-HS512+A256KW",
    "p2s": "8Q1SzinasR3xchYz6ZZcHA",
    "p2c": 2147483647,
    "enc": "A128CBC-HS256"
}
```

Here, p2c is set to the maximal value of a signed 32-bit integer (the highest value accepted by both vulnerable implementations), and p2s is some arbitrary random string.

The JWE payload can be set to any arbitrary value. The encrypted key, IV and authentication tag fields can also be arbitrary random byte sequences, as long as they have appropriate lengths. An example of a full token using this header us as follows:

```
eyJhbGciOiJQQkVTMi1IUzUxMitBMjU2S1ciLCJwMnMiOiI4UTFTemluYXNSM3h
jaFl6NlpaY0hBIiwicDJjIjoyMTQ3NDgzNjQ3LCJlbmMiOiJBMTI4Q0JDLUhTMj
U2In0.YKbKLsEoyw_JoNvhtuHo9aaeRNSEhhAW2OVHcuF_HLqS0n6hA_fgCA.VB
iCzVHNoLiR3F4V82uoTQ.23i-Tb1AV4n0WKVSSgcQrdg6GRqsUKxjruHXYsTHAJ
LZ2nsnGIX86vMXqIi6IRsfywCRFzLxEcZBRnTvG3nhzPk0GDD7FMyXhUHpDjEYC
NA_XOmzg8yZR9oyjo6lTF6si4q9FZ2EhzgFQCLO_6h5EVg3vR75_hkBsnuoqoM3
dwejXBtIodN84PeqMb6asmas_dpSsz7H10fC5ni9xIz424givB1YLldF6exVmL9
3R3fOoOJbmk2GBQZL_SEGllv2cQsBgeprARsaQ7Bq99tT80coH8ItBjgV08AtzX
FFsx9qKvC982KLKdPQMTlVJKkqtV4Ru5LEVpBZXBnZrtViSOgyg6AiuwaS-rCrc
D_ePOGSuxvgtrokAKYPqmXUeRdjFJwafkYEkiuDCV9vWGAi1DH2xTafhJwcmywI
yzi4BqRpmdn_N-zl5tuJYyuvKhjKv6ihbsV_k1hJGPGAxJ6wUpmwC4PTQ2izEm0
TuSE8oMKdTw8V3kobXZ77ulMwDs4p.ALTKwxvAefeL-32NY7eTAQ
```

This token can then simply be sent to a vulnerable implementation as is. Conditions for exploitability are as follows:

1. The JWT library supports JWE-wrapped JWTs and PBES algorithms by default.
2. The JWT library does not use a separate API for password-based encryption, but instead treats encryption keys and passwords in the same manner.
3. The library user did not configure a specific permissible algorithm for JWT validation.

Affected libraries

- jose before versions 1.28.1, 2.0.5, 3.20.3 or 4.9.1 (CVE-2022-36083)
- jose-jwt before version 4.1

Not-so-new attack: Key injection

Theory

RFC 7515 defines a rather odd header parameter:

```
4.1.3.  "jwk" (JSON Web Key) Header Parameter

The "jwk" (JSON Web Key) Header Parameter is the public key that
corresponds to the key used to digitally sign the JWS.  This key is
represented as a JSON Web Key [JWK].  Use of this Header Parameter is
OPTIONAL.
```

So a JWS can contain a parameter called `jwk` that contains the public key with which the JWS was signed. This seems rather pointless, as a JWS validator should already be aware of the public key in order to validate the token. The RFC does not specify what an implementation is supposed do with this header. I guess the validator could support multiple keys and look up the key from a list, but that is more easy to implement with the `kid` (Key ID) parameter.

However, it is not hard to foresee a way this header could be used wrong: namely if the public key in the `jwk` parameter were to be used to also validate the token. In this case the attacker could *inject* their own key in this header and sign a completely spoofed token themselves. This is actually not a new type of attack: around five years ago a vulnerability was found in a Cisco JOSE library that worked exactly like this. By adding a `jwk` header with a custom key, an attacker could spoof tokens.

During my research I found this same type of vulnerability in the Python library `authlib`, although it was only exploitable if the library user would call the JWS API directly. The JWT and OAuth/OIDC APIs were not vulnerable.

This authlib bug shows that this attack vector has not yet been completely eliminated. I'd argue that this is another good argument as to why the definition of this header value is a design flaw. Like the none algorithm, implementations should not use it.

Exploitation

This is not a new attack, and tools already exist to find or exploit this issue. For example: the The JSON Web Token Toolkit includes a check for this vulnerability class and the Burp Suite plugin JWT Editor has an "Embedded JWK" option that automates this attack.

Affected libraries

- Authlib before version 1.1.0 (CVE-2022-39175)

Conclusions and recommendations

While JWT and JOSE standards provide a huge improvement over older cryptographic token formats, there are still a number of design flaws present that in my opinion increase the chance of implementation bugs. Since I could not review every single JWT library (and only focused on open source ones) it is not unlikely that the attack classes described here might also work against other libraries that are not listed. Furthermore, due to the complexity and attack surface of JOSE standards it would not surprise me if more new attack classes would be discovered in the future.

To defend against these types of issues, I would make the following recommendations:

- For JWT/JOSE library developers:
  - Less is more: I found it easier to find vulnerabilities in highly feature-complete libraries than those implementing a narrow subset of necessary functionality. Avoid supporting features that don't have a clear use case and turn them off by default when the use case is rare.
  - Don't use the `alg` parameter to decide what algorithm to use for validation. Instead, determine it beforehand based on the user configuration or metadata of the key.
  - Never accept JWTs that are encrypted with a password or public key, or which use a non-compact serialization format.
  - When delegating token validation to another library, do not attempt to also parse the token yourself. Instead, operate on the validated payload returned by that library.
- For JWT implementers:
  - Reconsider whether your application really requires cryptographic tokens, and isn't better off with plain random tokens. Besides sidestepping cryptographic issues, stateful random tokens are also easier to revoke and don't require key management.
  - Consider if you can also use a JWT alternative with a stronger cryptographic design, such as PASETO tokens, Macaroons or Biscuits.
  - If JWTs are required, explicitly configure the validation algorithm you intend to use. Practically all JWT libraries allow you to do this.
- For the JOSE working group:
  - Specify security recommendations that help implementers avoid the types of vulnerabilities discussed here.

- Restrict the subset of JWE/JWS algorithms and features that are appropriate for use in JWTs. For example: specify that public-key encryption (without signing) or password-based encryption should not be used to protect JWT claims.
- Discourage the use of untrusted `alg` header values for deciding a cryptographic validation algorithm.