Critical vulnerabilities in JSON Web Token libraries

Which libraries are vulnerable to attacks and how to prevent them.

If you are using node-jsonwebtoken, pyjwt, namshi/jose, php-jwtor jsjwt with asymmetric keys (RS256, RS384, RS512, ES256, ES384, ES512) please update to the latest version. See jwt.io for more information on the vulnerable libraries. (Updated 2015-04-20)

This is a guest post from Tim McLean, who is a member of the Auth0 Security Researcher Hall of Fame. Tim normally blogs atwww.timmclean.net.

Recently, while reviewing the security of various JSON Web Token implementations, I found many libraries with critical vulnerabilities allowing attackers to bypass the verification step. The same two flaws were found across many implementations and languages, so I thought it would be helpful to write up exactly where the problems occur. I believe that a change to the standard could help prevent future vulnerabilities.

For those who are unfamiliar, JSON Web Token (JWT) is a standard for creating tokens that assert some number of claims. For example, a server could generate a token that has the claim "logged in as admin" and provide that to a client. The client could then use that token to prove that they are logged in as admin. The tokens are signed by the server's key, so the server is able to verify that the token is legitimate.

JWTs generally have three parts: a header, a payload, and a signature. The header identifies which algorithm is used to generate the signature, and looks something like this:

```
header = '{"alg":"HS256","typ":"JWT"}'
```

`HS256` indicates that this token is signed using HMAC-SHA256.

The payload contains the claims that we wish to make:

```
payload = '{"loggedInAs":"admin","iat":1422779638}'
```

As suggested in the JWT spec, we include a timestamp called `iat`, short for "issued at".

The signature is calculated by base64url encoding the header and payload and concatenating them with a period as a separator:

```
key = 'secretkey'

unsignedToken = encodeBase64(header) + '.' + encodeBase64(payload)

signature = HMAC-SHA256(key, unsignedToken)
```

To put it all together, we base64url encode the signature, and join together the three parts using periods:

```
token = encodeBase64(header) + '.' + encodeBase64(payload) + '.' + encodeBase64(signature)

# token is now:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJsb2dnZWRJbkFzIjoiYWRtaW4iLCJpYXQiOjE0MjI3Nzk2Mzh9.gzSraSYS8EXBxLN_oWnFSRgCzcmJmMjLiuyu5CSpyHI
```

Great. So, what's wrong with that?

Well, let's try to verify a token.

First, we need to determine what algorithm was used to generate the signature. No problem, there's an `alg` field in the header that tells us just that.

But wait, we haven't validated this token yet, which means that we haven't validated the header. This puts us in an awkward position: in order to validate the token, we have to allow attackers to select which method we use to verify the signature.

This has disastrous implications for some implementations.

Meet the "none" algorithm

The `none` algorithm is a curious addition to JWT. It is intended to be used for situations where the integrity of the token has already been verified. Interestingly enough, it is one of only two algorithms that are mandatory to implement (the other being `HS256`).

Unfortunately, some libraries treated tokens signed with the `none`algorithm as a valid token with a verified signature. The result? Anyone can create their own "signed" tokens with whatever payload they want, allowing arbitrary account access on some systems.

Putting together such a token is easy. Modify the above example header to contain `"alg": "none"` instead of `HS256`. Make any desired changes to the payload. Use an empty signature (i.e. `signature = ""`).

Most (hopefully all?) implementations now have a basic check to prevent this attack: if a secret key was provided, then token verification will fail for tokens using the `none` algorithm. This is a good idea, but it doesn't solve the underlying problem: attackers control the choice of algorithm. Let's keep digging.

RSA or HMAC?

The JWT spec also defines a number of asymmetric signing algorithms (based on RSA and ECDSA). With these algorithms, tokens are created and signed using a private key, but verified using a corresponding public key. This is pretty neat: if you publish the public key but keep the private key to yourself, only you can sign tokens, but anyone can check if a given token is correctly signed.

Most of the JWT libraries that I've looked at have an API like this:

```
# sometimes called "decode"


verify(string token, string verificationKey)


# returns payload if valid token, else throws an error
```

In systems using HMAC signatures, `verificationKey` will be the server's secret signing key (since HMAC uses the same key for signing and verifying):

```
verify(clientToken, serverHMACSecretKey)
```

In systems using an asymmetric algorithm, `verificationKey` will be the public key against which the token should be verified:

```
verify(clientToken, serverRSAPublicKey)
```

Unfortunately, an attacker can abuse this. If a server is expecting a token signed with RSA, but actually receives a token signed with HMAC, **it will think the public key is actually an HMAC secret key**.

How is this a disaster? HMAC secret keys are supposed to be kept private, while public keys are, well, public. This means that your typical ski mask-wearing attacker has access to the public key, and can use this to forge a token that the server will accept.

Doing so is pretty straightforward. First, grab your favourite JWT library, and choose a payload for your token. Then, get the public key used on the server as a verification key (most likely in the text-based PEM format). Finally, sign your token using the PEM-formatted public key as an HMAC key. Essentially:

```
forgedToken = sign(tokenPayload, 'HS256', serverRSAPublicKey)
```

The trickiest part is making sure that `serverRSAPublicKey` is identical to the verification key used on the server. The strings must match exactly for the attack to work -- exact same format, and no extra or missing line breaks.

End result? Anyone with knowledge of the public key can forge tokens that will pass verification.

Recommendations for Library Developers

I suggest that JWT libraries add an `algorithm` parameter to their verification function:

```
verify(string token, string algorithm, string verificationKey)
```

The server should already know what algorithm it uses to sign tokens, and it's not safe to allow attackers to provide this value.

Some might argue that some servers need to support more than one algorithm for compatibility reasons. In this case, a separate key can (and should) be used for each supported algorithm. JWT conveniently provides a "key ID" field (`kid`) for exactly this purpose. Since servers can use the key ID to look up the key and its corresponding algorithm, attackers are no longer able to control the manner in which a key is used for verification. In any case, I don't think JWT libraries should even look at the `alg` field in the header, except maybe to check that it matches what was the expected algorithm.

Anyone using a JWT implementation should make sure that tokens with a different signature type are guaranteed to be rejected. Some libraries have an optional mechanism for whitelisting or blacklisting algorithms; take advantage of it or you might end up at risk. Even better: have a policy of performing security audits on any open source libraries that you use to provide mission-critical funtionality.

Improving the JWT/JWS standard

I would like to propose deprecating the header's `alg` field. As we've seen here, its misuse can have a devastating impact on the security of a JWT/JWS implementation. As far as I can tell, key IDs provide an adequate alternative.

This warrants a change to the spec: JWT libraries continue to be written with security flaws due to their dependence on `alg`.

JWT (and JOSE) present the opportunity to have a cross-platform suite of secure cryptography implementations. With these fixes, hopefully we're a little bit closer to making that a reality.