Signing JSON Tokens with RSA

RS256 vs HS256

When signing a JSON Web Token (JWT) from the server, two algorithms are supported for signing JSON Web Tokens (JWTs): RS256 and HS256. HS256 is the default for clients and RS256 is the default for APIs. When building applications, it is important to understand the differences between these two algorithms. To begin, HS256 generates a symmetric MAC and RS256 generates an asymmetric signature. Simply put HS256 must share a secret with any client or API that wants to verify the JWT. Like any other symmetric algorithm, the same secret is used for both signing and verifying the JWT. This means there is no way to fully guarantee Auth0 generated the JWT as any client or API with the secret could generate a validly signed JWT. On the other hand, RS256 generates an asymmetric signature, which means a private key must be used to sign the JWT and a different public key must be used to verify the signature. Unlike symmetric algorithms, using RS256 offers assurances that our server is the signer of a JWT since only one party has the private key.

Verifying RSA256

At the most basic level, the JWKS is a set of keys containing the public keys that should be used to verify any JWT issued by the authorization server. We set this up as a static endpoint on the backend server, something like *https://your-domain/api/.well-known/jwks,json* To create that I added a folder called certs/ to the base project directory and then added the following in a file called 'jwks.json':

```
{
"keys": [
 {
   "alg": "RS256",
   "kty": "RSA",
   "use": "sig",
```

```
    "x5c": [

"MIIC+DCCAeCgAwIBAgIJBIGjYW6hFpn2MA0GCSqGSIb3DQEBBQUAMCMxITAfBgNVBAMTGGN1c3RvbWVyLWRlbW9zLmF1dGgwLmNvbTAeFw0xNjExMjIyMjIyMDVaFw0zMDA4MDEyMjIyMDVaMCMxITAfBgNVBAMTGGN1c3RvbWVyLWRlbW9zLmF1dGgwLmNvbTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMnjZc5bm/eGIHq09N9HKHahM7Y31P0ul+A2wwP4lSpIwFrWHzxw88/7Dwk9QMc+orGXX95R6av4GF+Es/nG3uK45ooMVMa/hYCh0Mtx3gnSuoTavQEkLzCvSwTqVwzZ+5noukWVqJuMKNwjL77GNcPLY7Xy2/skMCT5bR8UoWaufooQvYq6SyPcRAU4BtdquZRiBT4U5f+4pwNTxSvey7ki50yc1tG49Per/0zA4O6Tlpv8x7Red6m1bCNHt7+Z5nSl3RX/QYyAEUX1a28VcYmR41Osy+o2OUCXYdUAphDaHo4/8rbKTJhlu8jEcc1KoMXAKjgaVZtG/v5ltx6AXY0CAwEAAaMvMC0wDAYDVR0TBAUwAwEB/zAdBgNVHQ4EFgQUQxFG602h1cG+pnyvJoy9pGJJoCswDQYJKoZIhvcNAQEFBQADggEBAGvtCbzGNBUJPLICth3mLsX0Z4z8T8iu4tyoiuAshP/Ry/ZBnFnXmhD8vwgMZ2lTgUWwlrvlgN+fAtYKnwFO2G3BOCFw96Nm8So9sjTda9CCZ3dhoH57F/hVMBB0K6xhklAc0b5ZxUpCIN92v/w+xZoz1XQBHe8ZbRHaP1HpRM4M7DJk2G5cgUCyu3UBvYS41sHvzrxQ3z7vIePRA4WF4bEkfX12gvny0RsPkrbVMXX1Rj9t6V7QXrbPYBAO+43JvDGYawxYVvLhz+BJ45x50GFQmHszfY3BR9TPK8xmMmQwtIvLu1PMttNCs7niCYkSiUv2sc2mlq1i3IashGkkgmo="
    ],
    "n": "yeNlzlub94YgerT030codqEztjfU_S6X4DbDA_iVKkjAWtYfPHDzz_sPCT1Axz6isZdf3lHpq_gYX4Sz-cbe4rjmigxUxr-FgKHQy3HeCdK6hNq9ASQvMK9LBOpXDNn7mei6RZWom4wo3CMvvsY1w8tjtfLb-yQwJPltHxShZq5-ihC9irpLI9xEBTgG12q5lGIFPhTl_7inA1PFK97LuSLnTJzW0bj096v_TMDg7pOWm_zHtF53qbVsI0e3v5nmdKXdFf9BjIARRfVrbxVxiZHjU6zL6jY5QJdh1QCmENoejj_ytspMmGW7yMRxzUqgxcAqOBpVm0b-_mW3HoBdjQ",
    "e": "AQAB",
    "kid": "NjVBRjY5MDlCMUIwNzU4RTA2QzZFMDQ4QzQ2MDAyQjVDNjk1RTM2Qg",
    "x5t": "NjVBRjY5MDlCMUIwNzU4RTA2QzZFMDQ4QzQ2MDAyQjVDNjk1RTM2Qg"
  }
]}
```


- alg: is the algorithm for the key
- kty: is the key type
- use: is how the key was meant to be used. For the example above, sig represents signature verification.
- x5c: is the x509 certificate chain
- kid: is the unique identifier for the key
- x5t: is the thumbprint of the x.509 cert (SHA-1 thumbprint)
- parameter n: Base64 URL encoded string representing the modulus of the RSA Key.
- parameter e: Base64 URL encoded string representing the public exponent of the RSA Key.
- parameter d: Base64 URL encoded string representing the private exponent of the RSA Key.
- parameter p: Base64 URL encoded string representing the secret prime factor of the RSA Key.
- parameter q: Base64 URL encoded string representing the secret prime factor of the RSA Key.

- parameter dp: Base64 URL encoded string representing the first factor CRT exponent of the RSA Key. d mod (p-1)
- parameter dq: Base64 URL encoded string representing the second factor CRT exponentof the RSA Key. d mod (q-1)
- parameter qi: Base64 URL encoded string representing the first CRT coefficient of the RSA Key. q^-1 mod p

And the I added the following routes:

```
static(".well-known") {
    staticRootFolder = File("certs")
    file("jwks.json")
}
```

I recommend the following links:

- Generate JWKS Token
- Change from JWKS and PEM format

Steps for validating the JWT Server-side:

1. Retrieve the JWKS and filter for potential signature verification keys.
2. Extract the JWT from the request's authorization header.
3. Decode the JWT and grab the kid property from the header.
4. Find the signature verification key in the filtered JWKS with a matching kid property.
5. Using the x5c property build a certificate which will be used to verify the JWT signature.
6. Ensure the JWT contains the expected audience, issuer, expiration, etc.

Revision #2
Created 17 April 2022 00:26:50 by Elkip
Updated 17 April 2022 01:02:09 by Elkip