

ANGULAR UNIVERSITY



PREMIUM QUALITY
ANGULAR TUTORIALS

angular-university.io

JWT: The Complete Guide to JSON Web Tokens

This post is the first part of a two-parts step-by-step guide for implementing JWT-based Authentication in an Angular application (also applicable to enterprise applications).

The goal in this post is to first start by learning how **JSON Web Tokens (or JWTs) work in detail**, including how they can be used for User Authentication and Session Management in a Web Application.

In Part 2, we will then see how JWT-based Authentication can be implemented in the specific context of an Angular Application, but *this post is about JWTs only*.

Why a JWT Deep Dive?

Having a detailed overview of JWTs is essential for:

- implementing a JWT-based authentication solution
- all sorts of practical troubleshooting: understanding error messages, stack traces
- choosing third-party libraries and understanding their documentation
- designing an in-house authentication solution
- choosing and configuring a third-party authentication service

Even when choosing a ready to use JWT-based Authentication solution, there will still be some coding involved, especially on the client but also on the server.

At the end of this post, you will know JWTs in-depth including a good understanding of the cryptographic primitives that they are based upon, which are used in many other other security use cases.

You will know when to use JWTs and why, you will understand the JWT format to the point that you can manually troubleshoot signatures, and know several online / Node tools to do so.

Using those tools you will be able to troubleshoot yourself out of numerous JWT-related error situations. So without further ado let's get started with our JWT deep dive!

Why JWTs?

The biggest advantage of JWTs (when compared to user session management using an in-memory random token) is that they enable the delegation of the authentication logic to a third-party server that might be:

- a centralized in-house custom developed authentication server
- more typically, a commercial product like a LDAP capable of issuing JWTs
- or even a completely external third-party authentication provider such as for example Auth0

The external authentication server *can be completely separate from our application server* and does not have to share any secret key with other elements of the network, namely with our application server - there is no secret key installed on our server to be accidentally lost or stolen.

Also, there is no need for any direct live link between the authentication server or the application server for authentication to work (more on that later).

Furthermore, the application server can be **completely stateless**, as there is no need to keep tokens in-memory between requests. The authentication server can issue the token, send it back and then immediately discard it!

Also, there is also **no need to store password digests** at the level of the application database either, so fewer things to get stolen and less security-related bugs.

At this point you might be thinking: I have an in-house internal application, are JWTs a good solution for that as well? Yes, in the last section of this post we will cover the use of JWTs in a typical Pre-Authentication enterprise scenario.

Table Of Contents

In this post we are going to cover the following topics:

- What are JWTs?
- Online tools for JWT validation
- What is the format of a JSON Web Token
- JWTs in a Nutshell: Header, Payload, Signature

- Base64Url (vs Base64)
- User Session Management with JWTs: Subject and Expiration
- The HS256 JWT Signature - How does it work?
- Digital Signatures
- Hashing functions and SHA-256
- The RS256 JWT Signature - let's talk about public key crypto
- RS256 vs HS256 Signatures - Which one is better?
- JWKS (JSON Web Key Set) Endpoints
- How to implement JWT Signature Periodic Key Rotation
- JWTs in the Enterprise
- Summary and Conclusions

What are JWTs?

A JSON Web Token (or JWT) is simply a JSON payload containing a particular claim. The **key property of JWTs** is that in order to confirm if they are valid we only need to look at the token itself.

We don't have to contact a third-party service or keep JWTs in-memory between requests to confirm that the claim they carry is valid - this is because they carry a Message Authentication Code or MAC (more on this later).

A JWT is made of 3 parts: the Header, the Payload and the Signature. Let's go through each one, starting with the Payload.

What does a JWT Payload look like?

The payload of a JWT is just a plain Javascript object. Here is an example of a valid payload object:

```
1
2 {
3   "name": "John Doe",
4   "email": "john@johndoe.com",
5   "admin": true
6 }
7
```

In this case, the payload contains identification information about a given user, but in general, the payload could be anything else such as for example information about a bank transfer.

There are no restrictions on the content of the payload, but it's important to know that **a JWT is not encrypted**. So any information that we put in the token is still readable to anyone who intercepts the token.

Therefore it's important not to put in the Payload any user information that an attacker could leverage directly.

JWT Headers - Why are they necessary?

The content of the Payload is then validated by the receiver by inspecting the signature. But there are multiple types of signatures, so one of the things that the receiver needs to know is for example which type of signature to look for.

This type of technical metadata information about the token itself is placed in a separate Javascript object and sent together with the Payload.

That separate JSON object is known as the JWT Header, and here is an example of a valid header:

```
1
2 {
3   "alg": "RS256",
4   "typ": "JWT"
5 }
6
```

As we can see, its also just a plain Javascript object. In this header, we can see that the signature type used for this JWT was RS256.

More on the multiple types of signatures in a moment, right now let's focus on understanding what the presence of the signature enables in terms of Authentication.

JWT signatures - How are they used for Authentication?

The last part of a JWT is the signature, which is a Message Authentication Code (or MAC). The signature of a JWT can only be produced by someone in possession of both the payload (plus the header) and a given secret key.

Here is how the signature is used to ensure Authentication:

- the user submits the username and password to an Authentication server, which might be our Application server, but it's typically a separate server
- the Authentication server validates the username and password combination and creates a JWT token with a payload containing the user technical identifier and an expiration timestamp
- the Authentication server then takes a secret key, and uses it to sign the Header plus Payload and sends it back to the user

browser (we will cover later the exact details on how the signature works)

- the browser takes the signed JWT and starts sending it with each HTTP request to our Application server
- The signed JWT acts effectively as a temporary user credential, that replaces the permanent credential which is the username and password combination

And from there, here is what our Application server does with the JWT token:

- our Application server checks the JWT signature and confirms that indeed someone in possession of the secret key signed this particular Payload
- The Payload identifies a particular user via a technical identifier
- Only the Authentication server is in possession of the private key, and the Authentication server only gives out tokens to users that submit the correct password
- therefore our Application server can safely be sure that this token was indeed given to this particular user by the Authentication server, meaning that it's indeed the user as it had the right password
- The server proceeds with processing the HTTP request assuming that it indeed belongs to that user

The only way for an attacker to impersonate a user would be to either steal both its username and personal login password, or steal the secret signing key from the Authentication server.

As we can see, the signature is really the key part of the JWT!

The signature is what enables a fully stateless server to be sure that a given HTTP request belongs to a given user, just by looking at a JWT token present in the request itself, and without forcing the password to be sent each time with the request.

Is the goal of JWTs to make a server stateless?

Making the server stateless is a nice side effect, but the key benefit of JWTs is that the Authentication server that issued the JWT and the Application server that validates the JWT can be two completely separate servers.

This means that there is only the need for some minimal Authentication logic at the level of the application server - we only need to check the JWT!

It would be possible for a complete cluster applications to delegate login/signup to a single Authentication server.

This means that the Applications servers are simpler and safer, as a lot of the Authentication functionality is concentrated on the Authentication server and reused across applications.

Now that we know on a high level how JWTs enable stateless third-party Authentication, let's get into their implementation details.

What does a JSON Web Token look like?

Let's then have a look at an example of a JWT, taken from the online JWT validation tool available at jwt.io:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoiOnRydWV9.TjVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

You might be thinking, where have the JSON objects gone?? We will get them back in a moment. In fact, at the end of this post you will understand in-depth every aspect of this strange looking string.

Let's have a look at it: we can see that it does have 3 parts separated by dots. The first part before the first dot is the JWT Header:

```
JWT Header:  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

The second part, between the first dot and the second, is the Payload:

JWT Payload:

```
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0Ij06MTUyOTU0ODk5InQ
```

And the last part, after the second dot is the Signature:

JWT Signature:

```
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

If you want to confirm that the information is indeed there, simply copy the complete JWT string above to the official online JWT validation tool available at jwt.io.

But then what are all these characters, how can we read back the information in a JWT for troubleshooting? How does jwt.io get the JSON objects back?

Base64 in a Nutshell, or is it Base64Url?

Believe it or not, the Payload, the Header, and the Signature are still in there in readable form.

It's just that we want to make sure that when we send the JWT across the network we won't run into any of those nasty ("garbled text"

`qÃ@Ã¼Ã¶:Ã`) character encoding issues.

This problem happens because different computers across the world are handling strings in different character encodings, such as for example UTF-8, ISO 8859-1, etc.

And these problems are ubiquitous as much as strings are: whenever we have a string in any platform we have an encoding being used. Even if we didn't specify any encoding:

- either the default encoding of the operating system will be used
- or it will be taken from a configuration parameter in our server

We want to be able to send strings across the network without having to worry about these issues, so we choose a subset of characters that all common encodings handle the same way, and that is how the Base64 encoding format was born.

Base64 vs Base64Url

But what we see in a JWT is actually not Base64: instead its **Base64Url**.

It's just like Base64, but there are a couple of characters different so that we can easily send a JWT as part of a Url parameter, which is exactly what happens if for example we use a third-party login page that then redirects to our site.

So if we take the second part of this JWT (between the first and the second dot), we get the Payload which looks like this:

```
eyJzdWIIOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjYWRtaW4iOnRydWV9
```

Let's just run it through an online decoder, like for example [this one](#):

1	
2	{
3	"sub":"1234567890",
4	"name":"John Doe",
5	"admin":true
6	}
7	

We get back a JSON payload! This is great to know for troubleshooting. We did use a Base64 decoder though, more on this later, right now let's summarize what we have so far:

Summary: We now have a good overview of the content of a JWT Header and the Payload: they are just two Javascript objects, converted to JSON and encoded in Base64Url and separated by a dot.

This format is really just a practical way of sending JSON across a network.

Before moving into the signature, let's talk about what do we put in the Payload in the concrete use case of User Authentication.

User Session Management with JWTs: Subject and Expiration

As we have mentioned, a JWT payload could in principle be any claim, and not just user identification information. But using JWT for Authentication is such a common use case that there are a couple of specific properties of a payload defined for supporting:

- user identification
- session expiration

Here is a payload with a couple of the most commonly used JWT payload properties:

```
1
2 {
3   "iss": "Identifier of our Authentication Server",
4   "iat": 1504699136,
5   "sub": "github|353454354354353453",
6   "exp": 1504699256
7 }
8
```

Here is what these standard properties mean:

- `iss` means the issuing entity, in this case, our authentication server
- `iat` is the timestamp of creation of the JWT (in seconds since Epoch)
- `sub` contains the technical identifier of the user
- `exp` contains the token expiration timestamp

This is what is called a Bearer Token, and implicitly means:

The Authentication Server confirms that the bearer of this token is the user with the following ID defined by the `sub` property: let's give this user access

Now that we have a good understanding of how the Payload is used in the typical case of User Authentication, let's now focus on understanding the Signature.

There are many types of signatures for JWTs, in this post we going to cover two: HS256 and RS256. So let's start with the first signature type: HS256.

The HS256 JWT Digital Signature - How does it work?

Like most signatures, the HS256 digital signature is based on a special type of function: a cryptographic hashing function.

This sounds intimidating, but it's a concept well worth learning: this knowledge was useful 20 years ago and will be useful for a very long time. A lot of practical security implementation revolves around hashes, they are everywhere in web application security.

The good news is that its possible to explain everything that we need to know (as Web application developers) about Hashing in a few paragraphs, and that is what we will do here.

We are going to do this in two steps: first, we will talk about what a hashing function is, and after that, we will see how such function

together with a password can be combined to produce a Message Authentication Code (which **is** the digital signature).

In the end of this section, you will be able to reproduce the HS256 JWT signature yourself using online troubleshooting tools and an npm package.

What is a Hashing function?

A hashing function is a special type of function with some very unique properties: it has lots of practical useful use cases like digital signatures.

We are going to talk about 4 interesting properties of these functions, and then see why these properties enable us to produce a verifiable signature.

The function that we will be using here is called SHA-256.

Hashing Functions Property 1 - Irreversibility

A hashing function is a bit like a meat grinder: you put steaks on one end, and get hamburgers on the other - and there is no way to ever get back those steaks starting with the hamburger.

the function is effectively irreversible!

This means that if we take our Header and Payload and run it through this function, no one will be able to get the data back again just by looking at the output.

To see the output of for example SHA-256, try it out with this [online hash calculator](#), to see what a typical output that looks like this:

```
3f306b76e92c8a8fbae88a3ef1c0f9b0a81fe3a953fa9320d5d0281b05
9887c3
```

This also means that hashing is not encryption: encryption by definition is a reversible action - we do need to get back the original input from the encrypted output.

Hashing Functions Property 2 - Reproducible

Another important thing to know about hashing is that it's reproducible, meaning that if we hash the same input header and payload multiple times, we will always get back the exact same result, bit by bit.

This means that given a pair of inputs and a hash output, we can always validate if a given output (for example a signature) is correct because we can easily reproduce the calculation - but only if we have all the inputs.

Hashing Functions Property 3 - No Collisions

Another interesting property of Hashing functions is that if we submit multiple values to it, we always get back a unique result per input value.

There are effectively no situations when two different inputs will produce the same output - a unique input produces a unique output.

This means that if we hash the Payload plus the Header, we always get back the exact same result, and not other input data could have produced the same hash output - the hash output is effectively a unique representation of the input data.

Hashing Functions Property 4 - Unpredictability

The last property that we will cover about hashing functions is that given a known output, it's not possible to guess the input using a successive incremental approximation method.

Let's say that we had the hash output just above and we were trying to find out the Payload that generated it, by guessing an input and inspecting the output to see if it's close to the expected result.

Then we simply tweak one character on the input and then check the output again to see if we got closer, and if so repeat the process until we manage to guess the input.

But there is only one problem:

With hashing functions, this strategy will not work!

This is because in a hashing function, if we change even a single character in the input (actually even a single bit), on average 50% of the output bits will change!

So even minimal differences in the input create a completely different output.

This all sounds interesting, but you are likely thinking at this point: how does a hashing function enable a digital signature then??

Can't the attacker just take the Header and Payload and forge the signature?

Anyone can apply the SHA-256 function and get to the same output and append it to the signature of the JWT, right?

How to use hashing functions to produce a signature?

That last part is true, anyone can reproduce the hash of a given Header and Payload.

But the HS256 signature is not only that: instead, what we do is we take the Header, the Payload and we add also a secret password, and then we hash everything together.

The result of that is a SHA-256 HMAC or Hash-Based Message Authentication Code, and one example of a function that does that is the HMAC-SHA256 function, which is used in HS256 signatures.

The result of that function can only be reproduced by someone in possession of the JWT Header, the Payload (which are readable by anyone that grabbed the token), **AND** the password.

This means that that resulting hash is effectively a form of digital signature!

This is because the hashed result proves that the Payload was created and then signed by someone in possession of the password: there would be no other way for someone to come up with that particular hash.

So the hash serves as an unforgeable digital proof that the Payload is valid.

The hash is then appended to the message, in order to allow the receiver to authenticate it: this hashed output is called an HMAC: Hash-Based Message Authentication Code, which is a form of digital signature.

And that is exactly what we do in the case of JWTs: that last part of the JWT (after the second dot) is the SHA-256 hash of the Header plus the Payload, encoded in Base64Url.

How to validate a JWT signature?

So when we receive a HS256-signed JWT on our server, we have to have that exact same password too, in order to be able to validate the signature and confirm that the token Payload is indeed valid.

To check the signature we simply take the JWT header plus the payload and hash it together with the password. This means in the case of HS256 that the JWT receiver needs to have the exact same password as the sender.

And if we get back the same hash as in the signature it means that the token must be valid, because only someone with the password could have come up with that signature.

And that, in general, is how digital signatures and HMACs work. Do you want to see this in action?

Manually confirming a SHA-256 JWT Signature

Let's take that the same JWT as above and remove the signature and the second dot, leaving only the Header and the Payload part. That would look something like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoiOnRydWV9
```

Now if you copy/paste this string to an online HMAC SHA-256 tool like [this one](#), and use the password `secret`, we get back the JWT signature!

Or almost, we will get back the Base64 version of it, which still has an `=` at the end, and this is close but not identical to Base64Url:

```
TJVA95OrM7E2cBab30RMhrHDcEfxjoYZgeFONFh7HgQ=
```

That equal sign would show up as `%3D` in a URL bar, so that is kind of messy, but it also explains the need for Base64Url, if we want to send JWTs as a URL parameter.

There aren't a lot of online Base64Url converters available, but we can always do that in the command line. So to really confirm this HS256 signature, here is an [npm package](#) that implements Base64Url, as well as conversion from/to Base64.

The base64url NPM Package

Let's then use it to convert our result to Base64 URL and completely confirm the signature and our understanding of how it works:

```
mkdir quick-test && cd quick-test
npm init
npm install base64url

node
> const base64url = require('base64url');
>
base64url.fromBase64("TJVA95OrM7E2cBab30RMhrHDcEfxjoYZgeFONFh7HgQ=")
```

```
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

So finally there we have it, this string is the HS256 JWT signature that we were trying to reproduce:

This is the exact same signature as the JWT above, character per character!

So congratulations, you now know in-depth how HS256 JWT signatures work and you will be able to troubleshoot yourself out of any situation using these online tools and packages.

Why other types of signatures then?

This, in summary, is how JWT signatures are used for Authentication, and HS256 is just an example of a particular signature type. But there other signature types, and the one most commonly used is: RS256.

What is the difference? We have introduced HS256 here mostly because it makes it much simpler to understand the notion of a MAC code, and you might very well find it in production in many applications.

But in general, it's **much** better to use something like the RS256 signature method instead, because as we are going to learn in the next section it has so many advantages over HS256.

Disadvantages of HS256 Signatures

HS256 signatures can be brute forced if the input secret key is weak, but that could be said about many other key based technologies.

Hash-based signatures are however particularly simpler to brute force when compared to other alternatives, for typical production key sizes.

But more than that, a practical disadvantage of HS256 is that it requires the existence of a previously agreed secret password between the server that is issuing the JWTs, and any other server machine consuming the JWTs for validation and user identification.

Unpractical Key Rotation

This means that if we want to change the password we need to have it distributed and installed to all network nodes that need it, which is not convenient, it's error prone and would involve coordinated server downtime.

This might not even be feasible if let's say one server is managed by a completely different team or even by a third-party organization.

No separation between token creation and validation

It all boils down to the fact that there is no distinction between the ability to create JWTs vs the ability of simply validating them: with HS256 everyone in the network can both create **and** validate tokens because they all have the secret password.

This means that there are many more places where the password can be lost or stolen by an attacker, as the password needs to be installed everywhere, and not all applications have the same level of operational security.

One way to mitigate this is to create one shared password per application, but instead, we are going to learn about a new signature

method that solves all these problems and that modern JWT-based solutions now use by default: RS256.

The RS256 JWT Signature

With RS256 we are still going to produce a Message Authentication Code just like before, the goal is still to create a digital signature that proves that a given JWT is valid.

But in the case of this signature, we are going to separate the ability to create valid tokens, that only the Authentication server should have, from the ability to validate JWT tokens, that only our Application server would benefit from doing.

The way that we are going to do that is that we are going to create two keys instead of one:

- There will still be a private key but this time it will be owned only by the Authentication server, used only to sign JWTs
- The private key can be used to sign JWTs, but it cannot be used to validate them
- There is a second key called the public key, which is used by the application server only to validate JWTs
- The public key can be used to validate JWT signatures, but it cannot be used to sign new JWTs
- The public key does not need to be kept private and it often is not, because if the attacker gets it there is no way to use it to forge signatures

Introducing The RSA encryption technology

RS256 signatures use a particular type of keys, called RSA Keys. RSA is the name of an encryption/decryption algorithm that takes one key to encrypt and a second key to decrypt.

Note that RSA is not a Hashing function, because by definition the output of encryption can be reversed and we can get back the initial result.

Let's see what an RSA *Public* Key looks like:

```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDdlatRjRjogo3WojgGHF
HYLugdUWAY9iR3fy4arWNA1KoS8kVw33cJibXr8bvwUAUparCwlvdbH6dv
EOfou0/gCFQsHUfQrSDv+MuSUMAe8jzKE4qW+jK+xQU9a03GUnKHkkle+Q
0pX/g6jXZ7r1/xAK5Do2kQ+X5xK9cipRgEKwIDAQAB
-----END PUBLIC KEY-----
```

Again it looks bit scary, but it's just a unique key generated by a command line tool like openssl or an online RSA key generation utility like [this one](#).

Again this key *could* be made public, and this is actually typically published, so the attacker does not need to guess this key: it usually already has it.

But then there is the corresponding RSA *Private* Key:

```
-----BEGIN RSA PRIVATE KEY-----
MIICWwIBAAKBgQDdlatRjRjogo3WojgGHFHYLugdUWAY9iR3fy4arWNA1K
oS8kVw33cJibXr8bvwUAUparCwlvdbH6dvEOfou0/gCFQsHUfQrSDv+MuS
```

```
UMAE8jzKE4qW+jK+xQU9a03GUnKHkkle+Q0pX/g6jXZ7r1/xAK5Do2kQ+X
5xK9cipRgEKwIDAQABAOGAD+onAtVye4ic7VR7V50DF9bOnwRwNXrARcDh
q9LWNRrRGEleSYyTQ6EbatXS3MCyjjX2eMhu/aF5YhXBwkppwxg+E0mXeh
+MzL7Zh284OuPbkg1AaGhV9bb6/5CpuGb1esyPbYW+Ty2PC0GSzfIXkXs7
6jXAu9TOBvD0ybc2YlkCQQDywg2R/7t3Q2OE2+yo382CLJdr1SLVROWKwb
4tb2PjhY4XAwV8d1vy0RenxTB+K5Mu57uVSTHtrMK0GAtFr833AkeEA6avx
200Ho61Yela/4k5kQDtjEf1N0LfI+BcWZtxsS3jDM3i1Hp0KSu5rsCPb8a
cJo5RO26gGVrfAsDcIXKC+bQJAZZ2XIpsitLyPpuiMOvBbzPavd4gY6Z8K
WrfYzJoI/Q9FuBo6rKw14BFoToD7WIUS+hpkagwWiz+6zLoX1dbOZwJACm
H5fSSjAkLRi54PKJ8TFUeOP15h9sQzydI8zJU+upvDEKZsZc/UhT/SySDO
xQ4G/523Y0sz/OztSWcol/UMgQJALesy++GdvoIDLfJX5GBQpuFgFenRiR
DabxrE9MNUZ2aPFaFp+DyAe+b4nDwuJaW2LURbr8AEZga7oQj0uYxcYw==
-----END RSA PRIVATE KEY-----
```

The good news is that there is no way an attacker could guess that!

Again let's remember the two keys are linked, what one key encrypts the other and only the other can decrypt. But how do we use that to produce a signature?

Why not just encrypt the payload with RSA?

Here is one attempt to create a digital signature using RSA: we take the Header and the Payload, and encrypt it using RSA with the private key, then we send the JWT over.

The receiver gets the JWT, decrypts it with the Public Key, and checks the result.

If the decryption process works and the output looks like a JSON payload, this means that it must have been the Authentication Server

that created this data and encrypted it. So it must be valid, right?

That is indeed true, and it would be sufficient to prove that the token is correct. But that is not what we do due to practical reasons.

The RSA encryption process is relatively slow compared to for example a hashing function. For larger payload sizes this could be an issue, and this is only one reason.

So what do then, how do HS256 signatures actually use RSA in practice?

Using RSA and SHA-256 to sign a JWT (RSA-SHA256)

In practice what we do is we take the Header and the Payload and we hash them first, using for example SHA-256.

This is something very fast to do, and we obtain a unique representation of the input data that is much smaller than the actual data itself.

We then take the hash output, and encrypt that instead of the whole data (header plus payload) using the RSA private key, which gives us the RS256 signature!

We then append it to the JWT as the last of the 3 parts, and we send it.

How does the receiver check RS256 Signatures?

The receiver of the JWT will then:

- take the header and the payload, and hash everything with SHA-256
- decrypt the signature using the public key, and obtain the signature hash
- the receiver compares the signature hash with the hash that he calculated himself based on the Header and the Payload

Do the two hashes match? Then this proves that the JWT was indeed created by the Authentication server!

Anyone could have calculated that hash, but only the Authentication server could have encrypted it with the matching RSA private key.

Do you think that there has to be more to it? Let's confirm this then, and learn how to troubleshoot RS256 signatures along the way.

Manually confirming an RS256 JWT signature

Let's start by taking an example of a JWT signed with RS256 from jwt.io:

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0Ij0iOnRydWV9.EkN-DOsnsuRjRO6BxXemmJDm3HbxrbRzXglbN2S4sOkopdU4IsDxTI8jO19W_A4K8ZPJijNLis4EZsHeY559a4DFOd50_OqgHGuERTqYZyuhtF39yxJPAjUESwxk2J5k_4zM30-vtdlGhyo4IbqKKSy6J9mTniYJPenn5-HIirE
```

As we can see, there is no immediate visual difference when compared to an HS256 JWT, but this was signed with the same RSA Private Key shown above.

Now let's isolate the Header and the Payload only, and remove the signature:

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoiOnRydWV9
```

All we have to do now is to hash this with SHA-256, and encrypt it with RSA using the RSA private key shown above.

The result should be the JWT signature! Let's confirm if that is the case using the Node built-in Crypto module. There is no need for an external installation, this comes built-in with Node.

This module comes with a built-in [RSA-SHA256 function](#) and many other signature functions that we can use to try to reproduce signatures.

To do that, the first thing that we need is to take the RSA private key and save it to a text file, named `private.key`.

Then on the command line, we run the node shell and execute this small program:

```
1
2  const crypto = require('crypto');
3  const fs = require('fs');
4  const sign = crypto.createSign('RSA-SHA256');
5
6  // copy / paste here the header and the payload of your JWT only
7  sign.write('eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9
8           .eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoiOnRydWV9
9  sign.end();
10
11 var privateKey = fs.readFileSync('./private.key');
```

```
12
13 console.log(sign.sign(privateKey, 'base64'));
14
```

If you are using a different JWT than the test JWT that we have been using, then you need to copy/paste only the two parts to the `write` call, without the JWT signature.

And here is what we get back:

```
EkN+DOSnsuRjRO6BxXemmJDm3HbxrbRzXglbN2S4sOkopdU4IsDxTI8jO
19W/A4K8ZPJijNLis4EZsHeY559a4DFOd50/OqgHGuERTqYZyuhtF39yxJ
PAjUESwxk2J5k/4zM3O+vtdlGhyo4IbqKKSy6J9mTniYJPenn5+HIirE=
```

Which is completely different than the JWT signature !! But wait a second: there are slashes here, equal signs: this would not be possible to put in an URL without further escaping.

This is because we have created the Base64 version of the signature, and what we need instead is the Base64Url version of this. So let's convert it:

```
bash$ node
const base64url = require('base64url');
base64url.fromBase64("EkN+DOSnsuRjRO6BxXemmJDm3HbxrbRzXglb
N2S4sOkopdU4IsDxTI8jO19W/A4K8ZPJijNLis4EZsHeY559a4DFOd50/O
qgHGuERTqYZyuhtF39yxJPAjUESwxk2J5k/4zM3O+vtdlGhyo4IbqKKSy6
J9mTniYJPenn5+HIirE=");
```

And here is what get back:

```
EkN-  
DOsnsuRjRO6BxXemmJDm3HbxrbRzXglbN2S4sOkopdU4IsDxTI8jO19W_A  
4K8ZPJijNLis4EZsHeY559a4DFOd50_OqgHGuERTqYZyuhtF39yxJPAjUE  
Swxk2J5k_4zM3O-vtdlGhyo4IbqKKSy6J9mTniYJPenn5-HIirE
```

Which is exactly, bit by bit the RS256 signature that we were trying to create!

This proves our understanding of RS256 JWT signatures, and we now know how to troubleshoot them if needed.

In summary, RS256 JWT signatures are simply an RSA encrypted SHA-256 hash of the header plus payload.

So we now know how RS256 signatures work, but why are these signatures better than HS256?

RS256 Signatures vs HS256 - Why use HS256?

With RS256, the attacker can easily perform the first step of signature creation process which is to create the SHA-256 hash based on the values of a stolen JWT header and payload.

But from there to recreate a signature he would have to brute force RSA, which for a good key size is unfeasible.

But that is not the most practical reason why we would want to choose RS256 over HS256, for most applications.

With RS256, we also know that the private key that has the power of signing tokens is only kept by the Authentication server, where it's much safer - so RS256 means fewer chances of losing the signing private key.

But there is a much bigger practical reason to choose HS256 - simplified key rotation.

How to implement key rotation

Let's remember that the public key used to validate tokens can be published anywhere, and the attacker cannot in practice do anything with it.

After all, what good would it do for an attacker to be able to validate a stolen JWT? An attacker wants to be able to forge JWTs, not validate them.

This opens up the possibility of publishing the public key in a server under our control.

The application servers then only needs to connect to that server to fetch the public key, and check again periodically just in case it has changed, either due to an emergency or periodic key rotation.

So there is no need to bring the application server and the authentication server down at the same time, and update the keys everywhere in one go.

But how can the public key get published? Here is one possible format.

JWKS (JSON Web Key Set) Endpoints

There are many formats to publish public keys, but here is one that will feel familiar: JWKS which is short for Json Web Key Set.

There are some very easy to use npm packages to consume these endpoints and validate JWTs, as we will see on part two.

These are endpoints that can publish a series of public keys, not just one.

If you are curious to know what this type of endpoints looks like, have a look at this [live example](#), here is what we receive in response to an HTTP GET request:

```
1
2 {
3   "keys": [
4     {
5       "alg": "RS256",
6       "kty": "RSA",
7       "use": "sig",
8       "x5c": [
9         "MIIDJTCCAg2gAwIBAgIJUP6A/iwWqvEDMA0GCSqGSIb3DQEBCwUAMDaxLjAsBgNVBA
10    ],
11     "n": "wUvZ-4dkT2nTfCDIwyH9K0tH4qYMGcW_KDYeh-TjBdASUS9cd741C0XMvmVSYGRP0BOLe
12     "e": "AQAB",
13     "kid": "QzY0NjREMjkyQTI4RTU2RkE4MUJBRDE4NzY1MUY1N0I4QjFCODlB0Q",
14     "x5t": "QzY0NjREMjkyQTI4RTU2RkE4MUJBRDE4NzY1MUY1N0I4QjFCODlB0Q"
15   }
16 ]
17 }
18
```

The `kid` property is the key identifier, and `x5c` is the representation of one particular public key.

The great thing about this format is that its standard, so we just to have the URL to the endpoint, and a library that consumes JWKS - this should give us a ready to use public key for validating JWTs, without having to install it on our server.

JWTs are often associated to public internet sites, and to social login solutions. But what about intranet, in-house applications?

JWTs in the Enterprise

JWTs are applicable also to the enterprise, as a great alternative to the typical Pre-Authentication setup which is a known security liability.

In the Pre-Authentication setup that many companies use, we run our application servers behind a proxy on a private network, and simply retrieve the current user from an HTTP header.

The HTTP header that identifies the user is usually filled in by a centralized element of the network, usually a centralized login page installed on a proxy server that will take care of the user session.

That server will block access to the application if the session is expired, and will authenticate the users after login.

After that, it will forward all requests to the application server and simply add an HTTP header to identify the user.

The problem is that with that setup in practice anyone inside the network can easily impersonate a user just by setting that same HTTP header!

There are solutions for that, like white-listing the IP of the proxy server at the level of the application server, or using a client certificate but in

practice, most companies don't have these measures in place.

A better version of the Pre-Authentication Setup

The Pre-Authentication idea is good though because this setup means that the Application Developer does not have to implement the authentication features itself on each application, sparing time and avoiding potential security bugs.

Wouldn't it be great to have the convenience of the Pre-Authentication setup, without having to compromise security and make our Application authentication easily bypassable, even if only inside a private network?

This is simple to do if we put JWT into the picture: instead of just putting the username in the header as we usually do in Pre-Authentication, let's make that HTTP header a JWT.

Let's then put the username inside the payload of that JWT, signed by the Authentication server.

The Application server, instead of just taking the username from the header, will first validate the JWT:

- if the signature is correct, then the user is correctly authenticated and the request goes through
- if not, the application server can simply reject the request

The result is that we now have Authentication working correctly, even inside the private network!

We no longer have to trust blindly the HTTP Header containing the username. We can make sure that that HTTP header is indeed valid and

issued by the proxy, and it's not some attacker trying to log in as another user.

Conclusions

In this post we got an overall idea of what JWTs are, and how they are used for Authentication. JWTs are simply JSON payloads with a easily verifiable and unforgable signature.

Again, there is nothing about JWTs that is authentication-specific, we could use them to send any claim across the network.

Another common security related use for JWTs is Authorization: we can for example put in the Payload the list of Authorization roles for the user. Read Only User, Administrator, etc.

In the next post of this series, we are going to learn how to implement Authentication in an Angular application using JWTs.