It's never been easier to expose services over HTTP. It's also never been easier to inadvertently expose security holes via those same services. This session is designed for the average developer/architect that wants a brief overview of API security without deep dives into cryptography or complex authentication frameworks. You'll learn about OAuth, API Keys, HMAC authentication and more. Don't worry if those things sound foreign; they'll be explained in a clear, practical way. Examples will use WebAPI but the talk will focus on the principles, not the tech.

Welcome to “Securing Your API Endpoints”

Opening:

I have a confession to make. A few years ago, I made a huge rookie mistake.

I was writing a feature and I wanted a piece of JavaScript to make an API call back to my server. Unfortunately, as it turns out RESTful APIs are normally stateless, so I couldn't rely on the server just automagically knowing which user account was attached to the API call. I obviously needed to do something extra in order to add authentication to that API call.

Has anyone here ever googled for “how to authenticate an API”? I found a pretty confusing mess. There were websites talking about bearer tokens and nonces and “federated identity” systems. There were arguments about the correct cryographic hash functions to use when signing requests. There were two different versions of OAuth that work in entirely different ways. There were things called “web tokens” that some people claimed were God’s gift to the Internet, and other people called a scourge on mankind. Everything seemed complex and I couldn’t find a comprehensive, easy to use guide that helped me choose what authentication technique was best suited for my use case.

So… I did a bad thing. I was in a hurry and I didn’t want to invest the time to really understand the options, so I cobbled together some things I saw on StackOverflow and shipped it. And, surprise surprise, I got it wrong and shipped a security defect.

**Click for “today’s goal”**

My goal today is to give you the coherent overview that I desperately needed a few months ago. I’m going to break down all of the complex terminology and compare and contrast the different techniques you can choose from. The next time that you need to choose how to secure an API endpoint, I want you to start from a place of knowledge so that you can make an educated decision and not a rookie mistake.

**Click for “What’s on the agenda”**

Here’s the basic structure of this talk.

1. First, we’re going to talk about the different concepts that go into authentication and why you, as an API author, should care about them.
2. Second, we’re going to use those concepts to compare and contrast the options you can choose from when authenticating your API calls. We’ll cover everything from HTTP Basic Auth to OpenID Connect.
3. Finally, we’ll wrap up with some suggestions for selecting a technology to match your use case.

**Click for “not advanced”**

Also, I want to be clear about what this session is NOT.

First, it’s not an advanced security session. If you already know the difference between OAuth 1 and 2, how to sign a request using HMAC, or how to use JSON Web Tokens to replace server-side sessions, then you’re probably in the wrong place. My intended audience is people who DON’T know those things, or even that those are the things they need to know about in the first place.

**Click for “not getting started”**

Second, this is not a “getting started with foo” style talk. I have a ton of information to cover so I won’t be able to show you the “hello world” for each technique.

In fact, there are things in this talk that I have no direct hands on experience with. We’re going to talk about OAuth in a bit but I’ve never personally written any OAuth code.

**Click for “understand your options”**

However, I *have* done a lot of research about OAuth and how it compares to the other options, and that’s what this session is about. I’ve packaged up all of that research and distilled it into the most coherent format I could create. I want to help you narrow the universe of possibilities to the one or two technologies that are most suitable for your use case.

I’m about to turn on the firehose, and if I go too fast please feel free to stop me and ask questions. This presentation, along with all of my speaker notes, is on my public Github if you want to refer to it later. I’ll give you a link at the end.

**Click for “Identity, Authentication and Authorization”**

The subtitle of this talk is “A practical guide to API authentication” but we’re actually going to talk about three different concepts: “identity”, “authentication”, and “authorization”. They are all related but they really are different things and you need to consider them separately when making design decisions.

Identity is your app’s concept of a user. If the request says “yo, this is Alice, give me data on Foo #42”, the identity associated with the request is “Alice”.

Authentication is the process through which we securely associate an identity with a request. Authentication is how we determine if it really is Alice making the request, someone working on her behalf, or someone trying to impersonate her.

Authorization is the process through which we validate PERMISSION to perform the request. Is Alice allowed to see FOO #42?

Just because a request is authenticated doesn’t mean its authorized, and just because a request is authorized doesn’t mean its authenticated. We’ll talk more about that when we get to OAuth.

**click**

Not all APIs care about all of these things. For instance, Google maps requires an API key to make requests. That key is how they establish IDENTITY, primarily for rate limiting, but they really don’t care about authorization. You don’t need any specific permission to access one address over another.

On the other hand, Twitter’s API cares about all three. It needs to know which account the tweet is attached to, the identity of the user making the tweet, and whether that user is allowed to tweet for that account.

The point is that before you can pick an authentication strategy, you need to understand what problems it needs to solve and which ones it doesn’t.

Lastly, no matter how you authenticate your API calls, your app will still be responsible for some amount of access control. At the end of the day, your custom business rules are the only thing that can know for sure if Alice is allowed to view data for Foo #42.

**Click for “contenders” image**

So, how do you authenticate and secure your API endpoints? Here are the things I’ll be discussion today.

Three of the techniques are built directly into the web server and require very little code.

Custom schemes require more code, but provide more flexibility. We’ll cover API Keys and JSON Web Tokens.

We’ll talk about 2 different flavors of oAuth and how they relate to OpenID Connect.

And time permitting, I’ll talk briefly about SAML and WS-Security. My hunch is that most of you are most interested in the less enterprisey stuff so that’s where I’m going to spend most of our time today.

Again, this isn’t a tutorial on how to integrate these things into your app. I just want you to understand at a high level how each of these things works.

**Click for “Auth built into the web server”**

The simplest way to secure your API is to use the authentication features built directly into your web server platform. There are 3 such technologies supported by all major web servers and using them generally requires very little custom code. If you’re looking for a standards-based, easy-to-use solution, start with these.

**Click for “Client certificates”**

The first of these techniques is “client certificates”, which you can basically think of as “reverse TLS”. What I mean by that is this: when you make a secure connection to a web server using TLS, your browser uses the site’s security certificate to verify the identity of that website. You get that little lock icon in the browser that tells you the website you’re talking to is who it claims to be, and that you haven’t been intercepted or redirected to an imposter.

Client certs do the same thing, but in reverse. You install the certificate on your *browser*, and it proves your identity to the server. The same tech that tells you it’s safe to give your credit card data to an e-commerce site tells the server that a request is coming from a verified person and hasn’t been modified in transit.

The nice thing about client certs is that there are no login screens, no 3rd party redirects for authorization, every request you make is instantly authenticated.

The primary drawbacks are:

1. One, all of your users have to install security certs in their browsers. This is obviously not something you want to do for an internet scale website like Facebook.
2. Secondly, when using IIS, this is only a “simple” approach when authenticating against Active Directory because the tooling to link a client cert to a specific identity is built into Windows. If you want to authenticate against your custom user database it’s definitely possible, it just takes some more work.

These drawbacks mean that client certs are best suited for authenticating internal APIs on a secure network.

**Click for HTTP Basic Authentication (img)**

If you don’t want to deal with client certs, the next simplest approach is HTTP Basic Authentication.

This is an internet standard, is supported by all major browsers and servers, and is easy to implement.

Using Basic Auth, the username and password are concatenated together, Base64 encoded, and then sent along with each request as an HTTP header.

**Click for “decode”**

When the server receives the request, it Base64 *decodes* that string back into its original format and parses out the username and password which are then used to authenticate the request.

Remember that Base64 encoding is *not encryption*, so the server doesn’t need any special keys to convert this back into the account credentials. But since this header is being sent over the wire with every request, you *must* use TLS to secure every single connection when using Basic Auth. It also means you’re only as secure as the underlying TLS implementation; if TLS gets cracked, like SSL did before it, these credentials are at risk of being stolen.

Note that the “Authorization” header is poorly named. This is authentication, not authorization.

**Click for “setting up basic auth”**

Setting up Basic Auth is really easy. If you’re using IIS, you get authentication against a Windows domain “for free” with a simple web config setting. Once you do this the ASP.NET framework will automatically authenticate against a domain and give you access to the authenticated user data in your code.

For a public-facing website you’d probably want to authenticate against your custom user database. Most platforms make that very easy to do. For example, in ASP.NET or WebAPI you can write a tiny bit of middleware and override a few methods to provide the custom database queries that are needed.

**Click for drawbacks**

Basic Auth is really easy to set up, but it does have two main drawbacks.

1. First, unless you heavily customize it, the API client is using the primary account username and password. That means if there are two different clients using the same credentials, there’s no way to revoke access from Client 1 without changing the account password, which would then break Client 2.
2. Second, the credentials are passed over the wire, for every request, as essentially plain text. You must use TLS on all requests.

**Click for “HTTP Digest Authentication” diagram**

An alternative to Basic Auth is another standard called Digest Authentication.

The main difference between them is that with Digest, the password is never sent over the wire, so there’s less risk of it being compromised.

Here’s how it works:

1. The client requests a secured resource
2. The server responds with a “not authorized” response and includes a “nonce”, which is some random or unique string. Nonce is short for “number used once”.  
     
   **(click for next slide)**
3. The browser then prompts the user for credentials, concatenates the username, password, and nonce together, and creates an MD5 hash of the result.  
     
   **(click for next slide)**
4. Browser then resubmits request, passing the username, the nonce, and computed hash value *in clear text* as part of the Authorization header. (*Again, the header is poorly named – this is still authentication, not authorization*). Password itself is not sent over the wire.
5. The server then takes the username, looks up the user’s password, re-calculates the hash, and compares it to what the client sent.
6. The only way the hashes match will be if client and server used the same password to create the hash, which the server considers proof of authentication.

**Click for summary**

Just like with Basic Auth, this is very widely supported and very easy to implement.

And since the password itself is never sent over the wire, you can safely use Digest Auth without a secure connection. In fact, the purpose of the nonce is to make sure that every request results in a different hash value, so that an attacker can’t brute force attack the hashes to reverse engineer the password.

However, there are two significant drawbacks to using Digest authentication. The first is that MD5 has been broken, so those hashes aren’t as secure as we’d like.

There’s another issue too – so just to see if you’re awake, can anyone guess why Digest is not used very much any more?

**Click for “prevents storing passwords”**

The biggest issue with Digest authentication is that it prevents the use of strong password encryption in your database! This is because the server must be able to take a username and obtain its plain text password in order to verify the hash. But the whole point of modern password security is to make this impossible! The use of any one-way encryption method, such as salting and hashing passwords, will prevent you from using Digest Authentication.

Because of this issue no one really uses Digest for API authentication. But it’s important to understand the concepts behind using hash values to avoid sending sensitive data over the wire because those patterns are going to come up again and again.

**Click for “Custom”**

Those first 3 things work great if you want the API client to present actual user credentials during authentication, and if you OK with the constraints those techniques include. But what if those constraints are too limiting?

If you need more flexibility, either in terms of what information you use to authenticate OR how that information is transmitted, then the next simplest approach is to implement a custom scheme using API Keys or JSON Web Tokens.

**Click for “API Keys”**

Let’s talk about API Keys first.

With Basic and Digest Authentication, the client provides the primary account username and password as proof of identity. And that’s perfectly fine when the owner of those credentials is logging into a website or something.

But it’s a good idea, for many different reasons, to NOT rely on the account username and password in your API calls. If you’re passing around the primary account credentials with each API call, and something goes sideways and those credentials get stolen, then the entire account is compromised. The user has to change their password, which then breaks any other APIs that are using those same credentials. And if that person is reusing their credentials across different websites, then their exposure to harm is even greater.

**Click for revocable**

If you assign a unique API Key to each integration point, and you use that key as proof of identity instead of the account credentials, then it’s simple to revoke access from one integration without impacting any others.

And since API Keys are generally system-assigned, they tend to be unique for each site. If an API Key does get compromised, the attacker can only use it to access *your* system. They can’t take that key to any other system and gain additional access.

There is no “standard” of what an API Key should look like, but in most cases they are GUIDs or some other long, random, unique string. Since we’re using one single value as proof of identity, these obviously need to be hard to attack with brute force.

There are two different ways to use API keys for authentication.

**Click for “API Keys as passwords”**

The simplest approach is to treat the key like a password and pass it over the wire with every request.

This is called a "bearer token" because anyone that has that API Key may use it to authenticate as a specific user; there's no additional security.

And since you're passing the raw account credential over the wire, you **MUST use TLS** on all requests to keep it secure.

**Click for “passing the key”**

You can pass the API Key in either the querystring or an HTTP header. Passing it in the querystring is really easy to do and is ideal for scripting scenarios where managing HTTP request headers is difficult.

Generally speaking, however, passing it via a header is preferred for two reasons.

1. First, any value you pass in the querystring is going to get captured in web server log files, automated error reports, etc. You wouldn’t normally want plain-text passwords to be stored in unencrypted log files, so you probably don’t want API Keys there either.
2. Second, passing API Keys in the querystring makes it easy for them to get leaked via copy/paste. Whether or not you want someone to be able to copy a URL out of Fiddler or the F12 dev tools and then paste it to reissue an authenticated request probably depends on the sensitivity of the data your API exposes.

**Click**

Using API Keys as bearer tokens is very easy, but there is one significant trade-off you need to think about. You can *either* have secure storage of API Keys *or* the ability to show users a list of their keys. Not both.

Since API Keys are basically account passwords, you should consider salting and hashing them when you store them in the database. If you store them as text, and someone were to get access to your database, they would gain access to everything they need to impersonate every user in your system.

But if you salt and hash the API Keys when you store them, it will prevent you from showing a user a list of their API Keys. That’s kind of the whole point; you want your system to be able to *verify* a bearer token API Key, but not decrypt it to plain text, just like with passwords.

If you decide to store these things as plain text, then you should at least implement an expiration policy to limit the vulnerability window if the database *is* compromised.

**Signatures and MACs**

The other way to use API keys is to digitally sign the HTTP request.

This is basically a custom version of Digest Auth, but we use an API key to sign the request instead of a password. This allows us to continue to protect the primary account password with full encryption.

**Click for HMAC diagram**

Here’s how it works:

1. The client prepares its message, which in our case is probably a URL like [*http://foo.com/bar?bat=42*](http://foo.com/bar?bat=42)
2. The client concatenates the secret value with that message and then runs it through a hashing algorithm to generate a signature
3. The client sends the original message to the server, *plus* the signature in an authorization header.
4. The server receives the message and repeats the same hashing operation, then compares the result with the value from the authorization header. If they match, it knows the message was created by someone that knows the secret value AND that the message wasn’t modified in transit. (If either of those things is false, the server would have computed a different hash)

Technically speaking the authorization header is a type of “message authentication code”, or MAC. Since this technique uses a hash it’s called a “hash-based message authentication code”, or HMAC.

**Click for next**

HMAC has a lot of benefits:

1. No credentials (passwords or API Keys) are being sent over the wire, which makes this safe to use *without* SSL.
2. Guarantees message integrity - An attacker can’t capture one MAC value and use it to authorize a different request – each combination of request data has a unique MAC.

**Click for HMAC Drawbacks**

The primary drawback to HMAC is complexity.

The client and server must compute the hash EXACTLY the same way. This usually requires publishing a detailed set of instructions that describe how to “canonicalize” the request data by specifying how to capitalize it, how to deal with multi-valued parameters, etc.

Also, if the hash includes header values then you need to ensure that the client and server work with the same list of included headers; for instance, if some middleware network component adds a cache-control header to the request, and the server includes that header when verifying the signature, then verification is going to fail.

This screenshot is just a tiny piece of the instructions for preparing just one portion of the canonicalized string for an AWS API call.

This complexity is a necessary part of using HMAC. It’s the price you pay for the increased security that signed requests provide.

**Click for “signed requests” diagram**

In order for signing to work, the request must include at least two things:

1. The signature created with the user’s secret value
2. An indicator of the requestor’s identity

The identity indicator is required so that the server knows which user’s secret value to use when verifying the signature. The signature by itself isn’t verifiable; we need to know which key it was created with.

This means that if you build an HMAC system you’ll have to choose what to use as the identifier and what to use as the secret value.

If you’re building an internal API, or if you’re building both the client and the server, then you might be able to use a username or a user ID value as the identifier. If you’re building something for public use, you’ll probably want to use API Keys as the public identifier, just like we saw in the last section. In any case, the identifier is **public**; it needs to be something you’re OK transmitting, in clear text, over an unsecure connection.

But if we use an API Key as the public identifier, what should we use as the secret value?

**Click for “What to use”**

Whatever value you choose to use as the secret, it must be stored in plain text or using reversible encryption. The app must be able to obtain the raw value to verify the signature.

This means you can’t use the user’s password as the secret value. The whole point of secure password storage is for the app to ONLY know the encrypted password, and to prevent the app from being able to obtain the raw value.

A popular approach is to issue API Keys as a **pair** of values: a public API key, that is transmitted over the wire with each request, and a private key that is only known by the client and server.

This isn’t without drawbacks, of course. Storing the secret keys in plain text or in a reversible manner means that any attacker that compromises the database can begin to impersonate users.

If your API isn’t that sensitive, then this might be a risk you’re willing to take. After all, accessing the plain text secret keys is not the same as accessing plain text passwords. If you’re concerned about this, however, then you will probably want to implement an expiration policy for the secret keys. That way, even if the database is compromised and you’re unaware, there would be a limited window of time during which the compromised keys would be usable.

**Click for “HMAC for server clients”**

The last detail of an API Key implementation that you need to think about it, how does the client come to know the key in the first place?

Regardless of whether you’re using bearer tokens or HMAC, the client must know the secret value.

If you’re writing a server-based client it’s easy. Some person, likely a programmer, obtains the API key and secret value using some secure mechanism, such as logging into a secure website over TLS, and then puts it into the source code or config file for the client. As long as the config file is secured, the key is secure.

**Click for “HMAC for JS clients”**

It’s more complicated for a JavaScript client because there’s no way to pre-load the key onto the client up front. Users can log in from any browser in the world.

Generally speaking, in these cases you need the user to actively authenticate themselves by logging in. Once authenticated, an API or encryption key could be securely transmitted *back* to the client. However, a JS client *cannot securely store* the key. That’s because JS is not a secure environment – crypto functions can be monkey patched, local storage is vulnerable to cross-site scripting, etc. As a rule, you should assume that anything you expose to JS is open for inspection.

So if we can’t store the API or encryption keys in JS, how do we secure an API for a JS client?

**Click for “JWT”**

One answer is JSON Web Tokens, pronounced “JOT”.

JSON Web Tokens are an open, industry standard method for securely representing claims between two parties.

“Claims” are nothing more than pieces of data; they might be demographic such as a name, they might be user preferences, or they might be statements about what the user is allowed to do in the system. And since this standard is called “JSON” Web Tokens, the claims are obviously expressed using JSON.

Once those claims are generated, the server cryptographically signs the claims just like we saw with HMAC and creates a token. It gives that token to the browser, and the browser resends the token back to the server with each request. The server then validates the token and then uses the claims as needed, for instance to perform authorization.

**Click for example**

Let’s walk through that again in a little more detail.

Using JWT, the user still needs to securely authenticate themselves by logging in. And, it goes without saying, that login is performed over a secure connection so that the user credentials are kept safe in transit.

**Click for “create token”**

Once the server validates the credentials and confirms the identity of the request, it then creates a JSON document indicating that the user has authenticated and specifying the permissions they have in the system.

The values in this document are called “claims”.

**Click for HMAC**

The server then uses its private encryption key to sign that token, just like we saw with request signing using HMAC. In this case though, the encryption key is known *only* to the server, not the client.

**Click for return to client**

The server then responds to the login request, sending both the token and the signature. The client might store the token in memory or in a cookie; we’ll talk about that in a minute.

**Click for re-submit**

Finally, the client re-submits the token and signature with every subsequent request. Upon receiving a request that contains a token, the server re-computes the signature using its private key and validates the result against the signature provided by the client.

If the signature matches then the server knows those claims are valid; they could only have been signed by someone in possession of that private key. If the claims in the token had been modified in any way the signature wouldn’t match.

Again, this is very similar to signing API keys, except in this case *only the server knows the secret key*. The client’s job is just to store and re-submit the token.

**Click** **for “Format of JWT Token”**

JWT tokens consist of 3 pieces of data, separated by a period.

First is a standard header that typically indicates the type of token and the hashing algorithm being used. In this example we’re using HMAC SHA256, but other options exist.

In fact, the ability for the client to choose which hashing algorithm should be used is a core piece of the JSON Web Token standard. It's also one of the reasons that some security experts don't like JWT, although a discussion of those concerns is out of scope for this talk.

**Click for Payload**

The second piece of data is the payload, which contains the actual claims you’re making.

The JWT spec defines a handful of predefined claims called “registered claims”. These are optional but are recommended if you want your tokens to be more easily used between disparate systems. For instance, in this example here I’m using the “subject” claim to represent the ID of the user that logged in.

In addition to those predefined claims you can also create private claims to fit your own needs. In this example “name” and the “is\_admin” flag are private claims.

**Click for signature**

The third piece of data is the signature, which is calculated by combining the header and payload together and then running them through the hashing function.

**Click for token**

Finally, the base64-encoded header, the base64-encoded payload, and the signature are all concatenated together with periods, resulting in the final token.

This string right here is what gets sent back to the browser, and is then echoed back to the server on subsequent requests, usually in an Authorization header.

The really cool thing about this is that these claims make JWT tokens self-contained and stateless; rather than just saying “the user ID is 12” and relying on the server to figure out what permissions user 12 has, the JWT token itself can state those permissions. This can really help with performance because everything the server needs for authentication *and* authorization can be contained directly in the token, avoiding the need for additional database lookups to authorize a request.

**Click for warning**

It’s important to note that even though the token looks like gibberish, the header and payload are encoded but not encrypted. This means that anyone that gets access to that token will be able to read the claims that it contains.

This brings up some interesting trade-offs regarding where you store the token between requests, and what sort of access your JS code has to it.

**Click for storage**

Once the server sends your JS client a JWT token, you can either store it in LocalStorage or in a cookie.

The advantage of LocalStorage is that your JS code can access the claims. This is useful if the token contains data that you need for purposes other than API authentication and authorization, such as the user’s preferences or email address or something. The downside is that information stored in LocalStorage is vulnerable to cross-site scripting attacks, so you should only store tokens there if the payload doesn’t contain sensitive values.

Alternatively, you could put the token into an httpOnly secure cookie. This way, the token is protected in transit by TLS, and by definition an httpOnly cookie will be inaccessible to Javascript. However, that means that the token can *only* be used for server-side authentication and authorization and you can’t use the token to make data available to your JS app.

**OAuth**

At this point we’ve discussed some authentication schemes that are supported natively by the web server itself and we’ve discussed some custom systems based around API Keys. The next thing on the agenda is to talk about OAuth.

One of the reasons that we talked about those other thing first is because OAuth uses many of the same concepts and it’s easier to understand once you understand the underlying patterns.

We’re going to discuss a few different things: the difference between 2- and 3-party authorization, the differences between OAuth versions 1.0 and 2.0, and what it means for OAuth to be an authorization framework, not an authentication one.

**Click for “2-legged”**

OAuth was originally designed to solve the problem of “delegated authorization” in a 3-party scenario.

To explain that, let’s first review the traditional 2-party scenario you see here.

In this model, the client uses its credentials to access its resources on the server. As far as the server is concerned, the credentials belong to the client. The server doesn’t care where those credentials came from or if the client is acting on behalf of some other entity, as long as the authentication succeeds then the request is processed. In OAuth parlance, this is a “2-legged” model because there are two entities involved. One scenario that uses this model is server-to-server communication.

**Click for 3-legged**

However, there are times when the client is acting on behalf of another entity, such as a person that’s interacting with it. In those cases, client is not accessing its own resources but those of the user.

One way to implement this is for the user to share their credentials with the client so that the client can use them to make the authenticated request.

In some cases this is fine. If you log into a trusted website that uses a JS front end, and that front end makes API calls to the server on your behalf, it’s not a big deal if that front end uses your credentials. In fact, if you *log in* to a site, you’re explicitly giving that site your credentials in order to authenticate yourself.

**Click for next**

But what if you DON’T trust the client with your credentials? Let’s say you have some photos that you’ve uploaded to Facebook and you want to use an online photo printing service to create greeting cards using those photos. However, you really don’t want to trust the photo service with your Facebook username and password. In this case, the photo printing service is an **untrusted client** because you understandably don’t want to give it your full account details.

**Click for “Delegated Auth (3-legged)”**

This is the exact scenario that OAuth was originally designed for. It’s called a “3-legged” model because there are 3 parties involved: the Resource Owner that owns the content, the Service Provider that hosts the content, and the Client that accesses the content. OAuth allows the Resource Owner to authorize the Client to access the data client on their behalf, but without sharing the owner’s credentials with the client.

**transition**

There are two versions of OAuth and they solve this problem in very different ways. Unfortunately, it’s not universally accepted that the newer version is best. I’m going to talk about both versions so that you can make an informed decision between them.

OAuth 1.0 was published as an RFC in April 2010. A short while later a session fixation attack was found and version “1.0a” was published to address it. From this point forward, when I say “OAuth 1.0” I really mean “1.0a”.

**Click for OAuth diagram**

**TODO: Flow explanation**

**Click for auth screenshot**

One of the nice parts about OAuth is that it’s an *authorization* protocol, which means that it’s designed to let you precisely control what sort of access is being granted to the client. These permissions are called “scopes” in OAuth terminology and they are defined by the Resource Server.

**Click for OAuth 1.0a recap**

In technical terms, OAuth 1.0 is implemented using signed requests. Just like we saw with HMAC, this means that it does NOT require TLS and that the server can be sure that the request wasn’t modified in transit. The original 3-legged “flow” I just showed you works best with web-based clients because the user must be redirected to the resource owner’s website in order to authorize access.

There are two primary drawbacks to OAuth 1.0. The first is complexity. The OAuth handshake is very involved and there’s a lot of stuff going on to support the request signing. This is mitigated somewhat by widespread development of client libraries that handle those details for you, but the complexity of the implementation remains a consideration for OAuth 1.0.

The second drawback is that the protected resource endpoints must have access to the client credentials in order to validate the request. This is the same issue we discussed with both Digest Auth and HMAC: in order for the API server to verify the request signature it must have access to the secret keys used to sign it. This may not be a good fit for large enterprises where a centralized authorization server is used to issue credentials and a separate server is used for API calls.

(<http://hueniverse.com/2010/05/15/introducing-oauth-2-0/>)

**Click for OAuth 2.0**

OAuth 2.0 was designed to address those drawbacks, specifically the complexity issue. It is an entirely different implementation than 1.0 and the two are not compatible with each other.

The major difference between 1.0 and 2.0 is the lack of request signing. Instead, OAuth 2.0 uses “bearer tokens” that contain the authorization information. This is very similar to the “API Key” systems we discussed earlier and it means that unlike OAuth 1.0, 2.0 requires SSL for all requests. In essence, it obtains simplicity by totally delegating security to the transport layer.

OAuth 2.0 also has better support for 3-legged authorization in non-web clients and may be more enterprise friendly.

**Click for OAuth 2.0 diagram**

**TODO: Diagram explanation**

**Click for “OAuth 2.0 security concerns”**

One trade-off for this simplicity and convenience is security. Since bearer tokens are essentially a password, OAuth 2.0 requires TLS to protect them in transit.

Also, bearer tokens don’t tell you as much about the caller. With a signed request, you’re assured that the person making the request is either the resource owner or has access to the resource owner’s secret key, which by design is never transmitted over the wire with a request. With bearer tokens, all you know is that the caller has the token, but since they are passed around with each request there’s more potential for them to be leaked or stolen.

In addition, the 2.0 spec is considered a *framework* rather than a *protocol*. This means that a lot of decisions are left up to implementers, and thus interoperability suffers; if you take the code you’ve written against one OAuth 2.0 implementation and try to point it at a different authorization service by just changing a few URLs, it’s unlikely to “just work”. OAuth 1.0 offers a much higher degree of interoperability.

**Click for “road to hell**”

These aren’t trivial concerns. Eran Hammer, who was the lead author for the OAuth working group, withdrew his name from the 2.0 specification prior to launching it. He wrote a blog post called “OAuth 2.0 and the Road to Hell” in which he claimed that “OAuth 2.0 is a bad protocol. WS-\* bad”. He said that, compared to OAuth 1.0, the 2.0 spec is “more complex, less interoperable, less useful, more incomplete, and less secure”.

This article, which I highly suggest you read before choosing OAuth 2.0, goes on to say that the 2.0 spec is a designed-by-committee patchwork of compromises that mostly serves the enterprise at the expense of its two original goals of security and interoperability.

<http://hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell/>

**~~So which OAuth to use?~~**

~~So which version should you use?~~

~~Consider 1.0a if there’s a mature client library for your platform, you don’t want or can’t require SSL for API calls, you want to be as secure as possible, or the ability to easily switch between authorization providers is important to you.~~

~~On the other hand, consider OAuth 2 if you’re comfortable using SSL on all requests, if you’re more concerned about scaling with a single provider than interacting with multiple providers, and if you’re willing to trade a little security for a simpler implementation.~~

**Click for “road to hell #2”**

Eran Hammer also says “If you consider yourself a security expert, use 2.0 after careful examination of its features. If you are not an expert, copy the implementation of a provider you trust to get it right or make sure you have some security experts on site to figure it out for you”.

If security is really important to you, I recommend you do some additional research before settling on OAuth 2.0.

**Click for “Not authentication”**

<http://www.thread-safe.com/2012/01/problem-with-oauth-for-authentication.html>

Let’s talk about authentication vs authorization again.

OAuth is a standard for delegating authorization to an API. It’s about you, as a Resource Owner, authorizing one application to access your data from another application.

It is not an authentication protocol, for a few reasons.

**Click for “Access tokens != proof of authentication”**

User authentication is the first step in issuing an access token, so it’s tempting to think that possession of an access token is proof that authentication has occurred.

However, authentication is about two things:

1. Knowing the identity of the current user
2. Knowing if the user is present or not

An OAuth access token by itself does not give you either of those things.

It doesn’t give you the identity of the current user because, by definition, OAuth tokens are opaque to clients. A client obtains a token from the authorization server and relays it to the protected resource, but the token itself is a black box; the client can’t parse it or extract any data from it. That means the token, by itself, doesn’t tell the client anything about the user’s identity.

OAuth access tokens don’t tell you if the user is present or not because OAuth was specifically designed to allow delegated access in scenarios where the user *isn’t* present.

Without those two things, possession of an access token alone does not authenticate a user.

**“Access of protected API != proof of authentication”**

Let’s assume you have an access token that authorizes you to call an identity API to get user information. Since the token can be traded for identity information, you might think this is enough to prove the user is authenticated.

This is true when the token was created in the context of a user being authenticated via the redirect. However, in OAuth 2.0 there are other ways to get access tokens without the user being present, such as refresh tokens which allows a previously authorized client to renew their access. Also, the access token is generally usable long after the user is no longer present. If a client wants to make sure that authentication is still valid, it can’t simply re-call the identity API because neither the client nor the API has a good way to tell if the user is still there.

**Lack of audience restriction**

The last reason that OAuth should not be used for authentication is that most OAuth APIs do not provide any way to restrict access tokens to a specific client. This means it’s possible for a client to use an access token that was originally generated for a different client.

Here’s an example of why it’s dangerous to use OAuth as an authentication protocol.

Let’s say there is a website that lets people “log in with Facebook”. A user goes to Foo.com and follows the normal OAuth flow, which means they get redirected to Facebook where they **authenticate against Facebook** and then authorize Foo.com to access their data, and get redirected back to Foo.com with an access token. Foo.com then makes an API call to Facebook, gets their data, and logs them in.

In this scenario, the user happens to still be present and Foo.com can consider the user as authenticated.

**Click for example 2**

Let’s say Foo.com isn’t trustworthy. It turns around and make a request against Bar.com, which also allows Facebook logins. But instead of actually going through the authentication process to obtain a new access token, Foo just sends the access token it already has. At this point, Foo.com is “logged in” to Bar as the original user.

In this scenario, neither Foo nor Bar have any way to access more data from Facebook than were originally authorized. However, if Bar uses the presence of the access token to treat Foo as an authenticated user, then Foo might be able to access data from Bar belonging to the user.

**Click for final slide**

This is why it’s dangerous to use possession of an OAuth access token as proof that authentication has occurred. In both scenarios the website had an access token that it could exchange for data from Facebook, but in only one scenario was there as authenticated user involved.

To avoid these issues, you should only use an access token to access the specified resource. If you implicitly treat resource access as authentication, you are vulnerable.

**Transition**

To solve this problem, the access token needs some way to assert which client it is authorizing and which identity it is for. That way, only that one specific client would be able to use the token, and the token itself would serve as proof of authentication.

**Click for OpenID Connect**

That’s the basic concept behind OpenID Connect. It’s an open standard that defines an interoperable identity layer on top of OAuth 2.0. It allows you to perform authentication using OAuth by closing some of the gaps we just mentioned.

It does that by adding a few things to OAuth.

1. First, it adds ID Tokens. These are signed JSON Web Token that are given to the client app in addition to the regular OAuth access token. The ID token contains a set of claims about the authentication session including an identifier for the user, an identifier for the identity provider that issued the token, and the identifier of the client for which the token was created. It also includes information such as the token’s valid lifetime and other metadata about the authentication context. Unlike the original OAuth token, which is opaque to clients, these ID tokens have a well-known format so that clients can parse them without relying on an external service.
2. Also, systems implementing OpenID Connect must implement a standard, protected resource for returning claims about the current user. One of the reasons that interoperability suffers with OAuth 2.0 is that so many details are left up to the implementer. OpenID Connect standardizes how identity information should be shared to improve interoperability between providers.

OpenID Connect is a new standard. It is similar to, but separate from, OpenID 2.0, which is falling into disuse.

**Click for “OpenID Connect vs OAuth (for authentication)”**

Here’s a graphic depiction of the difference between OAuth and OpenID Connect.

With OAuth, the client says to the user “hey, give me a restricted access to your account, so I know you own it”. The user then gets that restricted access key and hands it over. This is the sort of pseudo-authentication we talked about earlier, where all the client knows *for sure* is that it was given a restricted access key to the account.

**Click for OpenID Connect graphic**

With OpenID Connect, the client gets the digital equivalent of a notarized referral letter confirming the user’s identity.

Both approaches give the client access to the user’s name, email, etc. But only OpenID Connect tells you that the user IS the account owner.

**Click for “Enterprisey options”**

To wrap up our review of authentication options I want to touch very briefly on two additional techniques that you might want to be aware of.

The first is SAML, which stands for “Security Assertion Markup Language”. It provides similar functionality as JSON Web Tokens in that it allows for the secure exchange of authentication and authorization data between different parties, but using SOAP and XML rather than JSON over HTTP. SAML is significantly older and more complex than JWT, but it does offer some additional functionality such as additional transport protocols and different types of encryption. SAML is very commonly used for enterprise single-sign-on scenarios, although JWT is starting to see some adoption in this area.

The second is WS-Security, and honestly I don’t know much about it. If you’re dealing with very complex, enterprise-grade authentication scenarios then you might want to hire a security expert to help. It’s probably safe to say that nobody attending my 101-level intro to authentication systems has any business implementing WS-Security in PROD without a little help.

**Click for “So what should YOU use?”**

OK, so I just threw a metric crapton of information at you. We talked about a lot of different authentication choices with a lot of different trade-offs. Like most things in this industry, the correct answer to this question is “it depends”.

**Click for client certificates**

Client certificates are pretty easy to use, IF you can get your users to install them. If you’re securing a private API on a trusted network, and you’re authenticating against Active Directory, then this is a really nice way to avoid login prompts altogether.

Client certificates are also a nice option for server-to-server authentication.

**Click for Basic Auth**

If you're authenticating server-to-server API calls against a standard user database such as ActiveDirectory, then Basic Auth is worth considering.

Basic Auth is *not* a good idea for a pure JS client because it requires storing the credentials somewhere in browser memory, which is generally a bad idea.

**Click for Digest Auth**

There’s no good reason to use Digest Auth. Anyone that tells you to do this is about 15 years out of date.

**Click for API Keys as bearer tokens**

Using API Keys as “bearer tokens”, where you pass the key itself with each request, is really easy to implement so it’s great for rapidly standing up a new API when you don’t need top-notch security.

The primary drawbacks to bearer tokens are that you’re sending the credentials over the wire with every request, and there’s no way to validate the authenticity of a message that you receive. Using TLS can help by protecting credentials in transit, but that may not be enough if security is very important to your API.

**Click for API Keys for request signing**

If you want more security than you get with bearer tokens, then use API Keys to sign requests.

Remember that creating the signature can be complex and requires that the client and server create the hash in exactly the same way. If you’re dealing with 3rd party clients, you may find that documenting and supporting the canonicalization process is more trouble than its worth, so rolling your own HMAC approach might be best if you’re writing *both* the client and server yourself.

**Click for JWT**

JSON Web Tokens might be a good fit if you’re writing a JS client or you want a standards-based, self-contained, stateless alternative to in-memory sessions on the application server. If API performance is a mission critical concern for you then JWT may help you reduce your database lookups by storing commonly-queried data in the token itself.

Remember that this does require a secure login with user-entered credentials to initialize the token, so JWT isn’t a good fit for automated, server-to-server interaction. (Or more accurately, JWT is a fine way to handle *authorization* in a server-to-server model, but it relies on something else to handle the initial *authentication*)

**Click for OAuth 1.0a**

OAuth is a good fit if you need to support delegated access to user data.

You should consider the 1.0 version if you can’t, or don’t want to, rely on TLS for security, or if you care about client/provider interoperability. For instance, if you want to support clients that can connect to multiple API providers with just a few configuration changes.

The primary drawbacks with OAuth 1.0 are the complexity involved in making signed requests and limited support for non-browser clients, so make sure you’re prepared to deal with that complexity. 3rd party libraries can help with that.

**Click for OAuth 2.0**

If you care more about flexibility and simplicity than operability and security, and you can require TLS on all requests, then OAuth 2.0 is a better fit than 1.0. 2.0 is also a better fit if you want to support a wider set of devices and authentication flows.

The primary drawbacks to OAuth 2.0 are reduced security relative to 1.0, and less interoperability as a result of having greater flexibility. Writing a client that can interact with multiple authentication providers will be harder using 2.0 than with other options.

**Click for OpenID Connect**

Remember that OAuth by itself is for authorization only. If you want to authenticate against 3rd party data then use OpenID Connect on top of OAuth 2.0. This will let your users authenticate to your API with their Google accounts, for example.

You can also use OpenID Connect if you have multiple internal systems and want to set up your own centralized identity and authentication provider.

**Click for enterprisey**

Lastly, you should use SAML or WS-Security if you have a sick love affair with XML, or if you have one of the very complex scenarios that *needs* the extra complexity they entail.

If you’re creating an API for your own internal use, or for public use on the open internet, these things are overkill and you should stick to something simpler.

**Click for 3 key things**

Research shows that attendees of a talk such as this will only remember 3 things. That means that most of the information I just shared with you will vanish quickly if you don’t act on it.

So here are the 3 most important things I want you to remember:

1. You can use custom API keys as bearer tokens over TLS for a quick and easy approach, or you can use API Keys to sign request using HMAC if you need additional security.
2. JSON Web Tokens are a secure, stateless way to share non-sensitive data. Remember that by default tokens are encoded, not encrypted, so anything you put in them can be decoded by the client. You also need to make sure your tokens aren’t vulnerable to cross-site scripting or cross-site request forgery.
3. Lastly, OAuth is for authorization, not authentication. Use OpenID Connect if you need both.

**Click for last slide**

Actually, you all look like smart people, so I want you to try really hard to remember a 4th thing, and that’s my website. From there you can get to my Github where these slides are kept and you can also get to my email, my Twitter, my LinkedIn, etc.

Please feel free to reach out with feedback, questions, comments, etc. I’d love to hear from you.

THANK YOU!

What is JWT?

* Securely communicate JSON objects
* Header, payload, and signature – self contained
* The header is a JSON object consisting of type and algo sender used to encrypt
* The payload is a JSON object containing user-defined attributes (public claims) and attributes defined in the standard (reserved claims)
* JWT signature is the encoded header and payload, signed with a secret.
  + Provides identity of sender
  + Ensures message has not changed [due to base 64]
* Finsihed token looks like [encoded header].[encoded payload].[signature]
* Advantages
  + Compact (can send via URL, POST, or header)
  + Easy to sign [vs XML]
  + Easy to parse [vs XML]
* Store tokens in HTTPS-only cookies
  + Local storage vuln to XSS – don’t store anything of value in it, or trust info in web storage
  + Don’t transmit over non-HTTPS connections, prone to MITM
  + ARE susceptible to CSRF, so use CSRF token strategy
  + App loses visibility into the token – can’t pull out claims
    - Can include payload only in response body that sets the secure cookie
    - Use token to call endpoints

<https://stormpath.com/blog/jwt-the-right-way>

<http://stackoverflow.com/questions/35291573/csrf-protection-with-json-web-tokens>

<http://programmers.stackexchange.com/questions/298973/rest-api-security-stored-token-vs-jwt-vs-oauth>

<http://blog.prevoty.com/does-jwt-put-your-web-app-at-risk>