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. Two years ago, I made a huge rookie mistake.

My team develops a fairly standard server-rendered web app. We eventually want to build a mobile app, so we’ve started taking a more API-based approach with some of our newer features. We figured that if the website is basically a Javascript and HTML interface on top of some APIs then we’d be in a good position to build a native app on top of those same APIs.

Of course, RESTful APIs are normally stateless, so we couldn’t rely on the server tracking a session or automatically knowing which user was making a given request. We had to introduce some sort of API authentication scheme to do that.

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. 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 and which authentication options require SSL and which don’t, or if you already know how to sign a request using HMAC, 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 separate things and you need to differentiate them when making security 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 and determine if the user is present on the connection or not. 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.

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 about that in a minute.

**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 IDENTITY is posting a tweet, it needs to AUTHENTICATE the validity of that identity, and it needs to AUTHORIZE the client to tweet on behalf of that identity.

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 I’ll briefly mention, but not discuss in detail, some of the really “enterprisey” options.

Again, this isn’t a tutorial on how to implement any of these things, I just want you to understand at a high level how they all work.

**Click for “Secure Connection Terminology”**

Quick sidebar: I assume everyone here understands the idea of a secure connection in which communications between a client and server are encrypted. Some of the techniques I’ll discuss require a secure connection, some don’t.

The important thing to note is that the old protocol for doing this, SSL, is broken. Make sure your servers are set up to use TLS instead.

If I say SSL at any other time of this session, it’s just an old habit. I mean TLS.

**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 SSL”. 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 this allows the server to verify *your* identity when you make an API call. 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, client certs work best when you’re authenticating against Active Directory because the tooling to link a client cert against a user account already exists. I’m sure it’s possible to link client certs against a custom user database but I haven’t done the research.

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 credentials for accessing a resource are Base64 encoded and sent along with each request as an HTTP header.

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.

For a public-facing website you’d probably want to authenticate against your custom membership system. In ASP.NET or WebAPI you can do that by writing some simple middleware and overriding a few methods, and I assume that other web server platforms support similar extensibility.

**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, in the clear. Remember that Base64 encoding is reversible, it isn’t encryption, so anyone that sniffs the network traffic could obtain the credentials. Basic Authentication should always be used with TLS on every secured request, because TLS encrypts the request as it travels over the wire. This also means that basic auth is only as secure as the TLS implementation.

**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”.
3. The browser uses the nonce to create an MD5 hash of the username and password.
4. The browser re-requests the page passing the username and nonce in clear text, plus the hashed value.
5. The server looks up the user’s password, uses nonce to re-calculate the hash, and verifies the request.
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**

Since the password itself is never sent over the wire, you can safely use Digest Auth without a secure connection.

And just like with Basic Auth it’s very easy to integrate with other standards-based systems.

However, there is one significant drawback to Digest in that it prevents the use of strong password encryption in your database! This is because the server must have access to the raw password in order to verify the hash. The use of any one-way encryption method, such as salting and hashing passwords according to modern best practices, will prevent you from using Digest Authentication.

Because of this issue no one really uses Digest for API authentication, but important to understand the concept because we’re going to see a derivatives of it a bit later.

**Click for “Custom”**

Those 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.

The idea is that the client sends something *other than* the primary username/combo during authentication. This not only keeps the primary credentials secure, but allows them to be easily revoked. For instance, if you use two different Twitter clients, you could give them separate API keys. If you decide you no longer trust Client 1 you could delete its key without any impact on Client 2.

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. These need to be hard to brute force attack for obvious reasons.

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

**Click for “API Keys as passwords”**

The simplest way to use API Keys is to treat them like a password and pass them to the server, with each request, in plain text. This is analogous to passing a plain text password using Basic Auth and requires the use of TLS.

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, even though both approaches send the key over the wire in plain text, using headers are more secure because 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**

When you use API Keys like this they are called “bearer tokens” and are basically the same thing as a password. Whomever possesses the token may use it, and there’s little that you can do to verify the identity of the caller.

That means that at a minimum, if you’re passing an API Key in the querystring OR in a header, you must use TLS on all requests, to keep the key encrypted in transit. You’re also only as secure as the TLS implementation. If there’s a bug in your platform’s TLS library then your keys could be compromised.

Another drawback is that there’s no way to verify message integrity. If a request is made without TLS, or over a compromised TLS connection, there’s no way for your server to verify that the request you received is the same as the client sent.

In addition, you should consider salting and hashing your API keys when you store them in the database, just like you do with passwords. If you store them as text, and someone were to compromise your database, they would gain access to everything they need to impersonate every user in your system. That’s basically what happened with the Buffer attack I mentioned during my intro.

If you do end up salting and hashing your API Keys, however, it will mean you can no longer show users a list of their API Keys. That’s kind of the whole point; you want your system to be able to *verify* an API Key, but not decrypt it to plain text, just like a password.

AWS and Octopus Deploy both use API Keys in this way and have begun storing hashed keys. When you create a key on those systems you can associate it with a human readable name, so that you can differentiate your keys, but once the key is generated there is no way for the system to recover it.

**Signatures and MACs**

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

Just like we saw with Digest Auth, the idea behind “signing” a request is that if the server and the client both know some secret value, that is not known by anyone else, then they can use it to communicate securely without ever transmitting that secret value over the wire.

**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 SSL, 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”.

In this approach, the user still needs to securely authenticate themselves by logging in. Once their credentials are validated, the server creates a token indicating that the user has authenticated. The server uses its private encryption key to sign the token and then returns the token and the signature back to the browser.

On each request, browser re-submits token + signature, which server verifies it by re-computing the signature.

This is very similar to HMAC signing the request. If the token and signature match, then the server knows the token is legit and wasn’t modified in transit.

As long as the token itself doesn’t contain any sensitive values, there’s no risk.

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

From a high level, this is what you need to know about JWT tokens.

1. It’s a standard for the secure transmission of JSON objects.
2. These objects contain “claims” about the user, which are really just data properties of a JSON object.
3. Some of those properties are defined by the standard, but you can add custom claims as well. In this example, I’m creating a claim for the user’s name and a claim that they have authenticated as an admin.

That last piece (authenticated as an admin) is important. JWT tokens are not encrypted, so they should not contain sensitive values. Instead of sending the user’s API key, and then having the server look up the permissions for that key, a JWT token can directly communicate the actual permissions.

Instead of a token that says “here’s the user’s ID, go figure out what they can do”, it’s a token that says “here’s what the user can do”.

This makes JWT self-contained and stateless. All the information that the API needs to know can be provided as claims, and the token contains everything the server needs to validate the authenticity of those claims.

**Click for “Format diagram”**

The reason that the server can trust the claims is because of the signature.

A JWT token is sent over the wire in this format:

1. A standard header is base64 encoded
2. The token itself, called the payload, is base 64 encoded
3. The encoded header, encoded payload, and secret key are used to create a hash
4. The encoded header, encoded payload, and the hash are concatenated together with dots

When the server receives the token, it decodes the header and payload, re-calculates the hash using its own secret key, and compares the hash to what was included in the token. If they match, the server knows that the token has not been modified and is authentic.

**Click for “Storing JWT”**

Once the server sends your JS client a JWT token, you need to store it somewhere. Common places are in LocalStorage, or in a cookie.

The advantage of LocalStorage is that your application has access to the data in the token payload. This is useful if the token contains information that you need for purposes other than API authentication. 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 cookie will be protected in transit by TLS, and because its httpOnly the cookie will be inaccessible to Javascript. However, that means that you can’t use the token for anything *except* authentication, because your JS app code won’t be able to read it.

**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. These technologies are designed for SOAP web services and/or for complex authentication scenarios that occur in the enterprise.

The first is SAML, which stands for “Security Assertion Markup Language”. It’s an “XML-based, open standard data format for exchanging authentication and authorization data between parties”. Although technically designed for applications on the internet, SAML is generally used in enterprise SSO scenarios.

The second is WS-Security. In terms of complexity, this is the undisputed king. It is designed for very complex scenarios that I’m guessing no one in this room needs to deal with. All I’m going to say is that if your use cases involve things like this then you should be seeking out some professional consulting services. My freebie session at CodeMash ain’t gonna cut it for you.

**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”.

To try and tie up all these details into a nice little package, let’s briefly recap each technology and talk about the scenarios they are best suited for.

**Click for client certificates**

Client certificates are pretty easy to use, IF you can get your users to install them. They work best when you’re securing a private API on a trusted network. In Windows land, the sweet spot is when you’re using IIS and authenticating against Active Directory.

**Click for Basic Auth**

Basic Auth is ideal if you want to write very little code and you can tolerate TLS on all requests.

This works really well for server-to-server API calls. It’s not ideal for a pure JS client because it requires storing the credentials somewhere in browser memory.

**Click for Digest Auth**

There’s really no good reason to use Digest Auth. If you can tolerate SSL on all requests then just use Basic Auth. If you can’t, then use signed requests or JWT.

**Click for API Keys as bearer tokens**

You can use an API key as a “bearer token”, where you pass the key itself with each request, when you own both the client and the API, and you can require TLS for all requests.

There is a slight security risk here because your keys are only as safe as the TLS implementation. If TLS gets cracked, your keys could be sniffed over the wire.

**Click for API Keys for request signing**

If you own both the client and the API, but 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.

**Click for JWT**

If you’re writing a JS client for your own application, such as the front-end of a SPA, then JSON Web Tokens might be a good fit.

Remember that this does require a secure login with user-entered credentials to initialize the token. If you need to support server-to-server interaction then this probably won’t work for you.

**Click for OAuth 1.0a**

If you’re supporting 3rd party clients then oAuth is worth a look. Consider version 1.0a if you can’t or don’t want to rely on SSL for security.

You might also consider 1.0a 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.

**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 literally have no other choice, and/or have a sick love affair with XML. If you choose either of these, please seek professional help.

**Click for 3 key things**

It is commonly believed 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. Requests must use EITHER SSL/TLS, OR be signed with a MAC. If you’re passing sensitive data over the wire then use SSL. If you want to verify message integrity, use a MAC.
2. If you own the client AND server, consider API Keys or JWT. Use OAuth for 3rd party clients.
3. 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>