OAuth - The Big Picture

apigee

Greg Brail & Sam Ramji

Table of Contents

OAuth - The Big Picture

Introduction

OAuth has taken off as a standard way and a best practice for apps and websites to handle authentication.

OAuth is an open protocol for allowing secure API authorization from desktop and web applications through a simple and standard method. It manages handshakes between applications and is used when an API publisher wants to know who is communicating with the system. Many of the largest API publishers have implemented OAuth to handle write access to their APIs.

We titled it *OAuth – The Big Picture* because it does not attempt to compete with sites about the protocols as defined by RFC 5849 (OAuth 1.0) or OAuth 2.0 or explain the architecture and in-depth technical and implementation details of OAuth. There are many great sites and discussion groups (including wiki.OAuth.net and OAuth Google groups) that delve into the details of OAuth and the evolving specification.

Rather, this e-book is designed for those who want to understand OAuth, its advantages and challenges, what role it might play for their products, without having to know the fine details of the protocol. We hope it will be a guide for members of the business development team, product managers, technical evangelists, product architects, and so on who make strategic decisions about their API products.

This e-book discusses what OAuth is, how it works, and how it fits with APIs and the emerging world of open platforms. We take a look at the evolving OAuth specification and why implementing OAuth can be complex. We provide some recommendations for how to approach implementing OAuth to ultimately deliver a secure and great user experience for web and mobile apps.

Open platforms and security

Businesses become platforms. Every market in history has had intermediaries. These intermediaries connect buyers and sellers by knowing what both want and creating convenient ways to transact.

Apps are the new intermediaries. They occupy many niches already and continue to multiply, as do devices. Companies cannot build for all these niches as each one requires expertise in design and development, and there are too many niches.

As Marc Andreessen observed, and as Annabelle Gawer described in *Platform Leadership*, the solution to these business problems across traditional industries has already been found: the platform business model. (Evans, Hagiu and Schmalensee also explored the issues deeply in *Invisible Engines* (2006))

In short, software is eating the world… We are in the middle of a dramatic and broad technological and economic shift in which software companies are poised to take over large swathes of the economy. (*Marc Andreessen, Aug 2011)*

Platforms are open

As we've learned from digital natives like Twitter and Facebook, and Intel and Microsoft before them, open platforms grow the fastest. In the API era of competition, speed is crucial because critical mass leads rapidly to market dominance.

Open platforms mean that developers can build apps quickly and without formal commitment to joint research, joint development, and joint marketing. They **decouple partners** from the platform business's cycles. The cost of innovation is significantly reduced, enabling many more experiments to be made more quickly, increasing the chance of a major improvement to the platform business, its customers, and the intermediaries. This is ***low-friction innovation***.

Open does not mean secure

This takes us to the stakes required for digital business in the API era.

For an intermediary to connect a buyer and a seller, there must be trust. The intermediary must be trustworthy, and the transaction must be trustworthy. In modern businesses buyers (users) have accounts with sellers (providers), filled with data as well as transaction privileges. For the system to function well, buyers must be able to fire their intermediary without breaking their relationship with the seller.

Historically and in traditional markets, reputations have been established through ongoing transactions and personal interactions and reference.

With apps as the intermediary, new dynamics exist on top of the historical foundation. Apps are new. They are often short-lived, and their business model depends on building a high volume of users. They must have some way to have their first transaction and be proven or else improved. This way must align with the loose coupling philosophy at the heart of an open platform – otherwise we've just secured our way back into old-fashioned closed businesses.

So how do you build a trustworthy system in an open world? It takes open security architecture.

Introducing OAuth

OAuth stands for Open Authorization. It's a free and open protocol, built on IETF standards and licenses from the Open Web Foundation, and is the right solution for securing open platforms.

The developers of OAuth set out to solve the problem that services and passwords don't mix well as you start to combine apps and mash them up.

Imagine, in today's environment of web APIs and mobile apps, if every web site that used an API from another web site had to share and store that web site password. Soon you'd have the proliferation of your passwords all over the Internet with every service you've used from Facebook, to Twitter, to Skype, to your bank account. Do you trust them all with your password?

OAuth is another way to authenticate to a service - a security protocol that allows users to grant third-party access to their web resources without sharing their passwords.

The heart of OAuth is an authorization token with limited rights, which the user can revoke at any time should they become suspicious or dissatisfied with the app they're using to access your business.

The valet key metaphor

OAuth supports this "delegated authentication" between web apps using a security token called an "access token." Rather than relying on a single password as the master key for every app that accesses an API, OAuth uses this token. An OAuth token gives *one* app access to *one* API on behalf of *one* user.

Eran Hammer-Lahav, a spec author for OAuth, compares the token to a valet key. This is an apt metaphor.

For most people, their car is one of their most valuable possessions, valued in tens of thousands of dollars. They are convenient places to leave our other valuables like computers and clothing. Yet we are sometimes required to give them to a parking attendant or valet whom we've not met before. A valet key solves the problem – it's an access token with limited rights that can operate the vehicle but not grant access to the trunk or glove box.

For great information about the history of OAuth, see Eran Hammer-Lahav's guide.

Just enough permission

A corollary of the principle of least harm is the principle of "just enough permission". An app should have just enough permission to do the things the user wants it to. In OAuth, permissions can be gracefully upgraded as well – if the user tries to do something in an app and they haven't authorized the corresponding permission, the business can give the user the option to add that permission, using the bootstrapping sequence used to grant the token in the first place.

Just enough responsibility

App developers are not security experts. A developer's job is to make software that does what it is supposed to. A security expert's job is to make sure software never does what it is not supposed to do.

App developers do not want responsibility of holding a user's secret information. Usernames and passwords, credit card or banking information, or the lifetime history of everyone you've ever called or emailed – these are heavy secrets that require heavy security. The right place for these is within your own business, secured by your own experts and your own infrastructure investments.

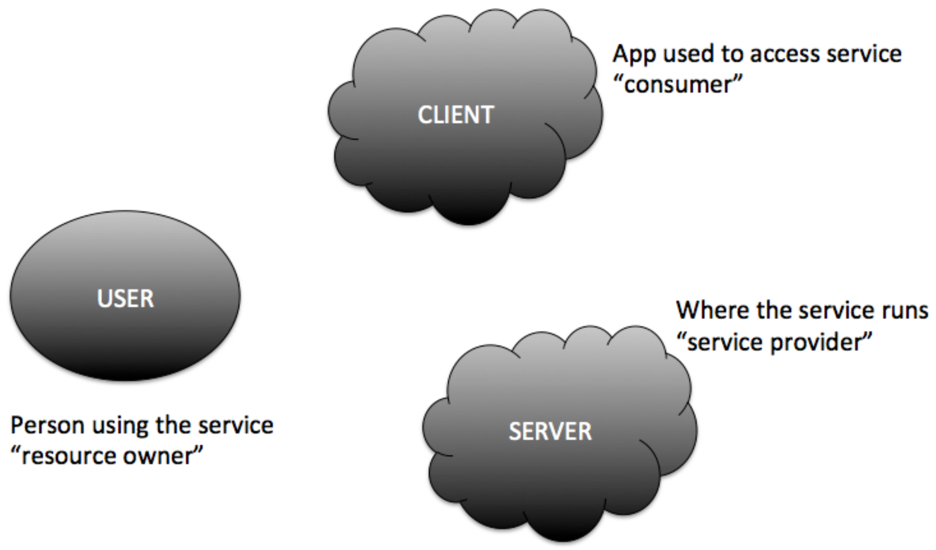
Decoupling partners (app developers) from these challenges keeps security consistent with the open platform potential for low-friction innovation.

How does OAuth work?

Simply, there are three entities (legs) to consider for an OAuth scenario:

- The user of a service – let's call him Bob
- A client (a Web app, a mobile app, or a server) – let's take the URL-shortening service bit.ly as an example client
- The server (where the service runs) – let's take Twitter as an example

OAuth - The Big Picture



How does our user Bob interact with Twitter through his [bit.ly]() account?

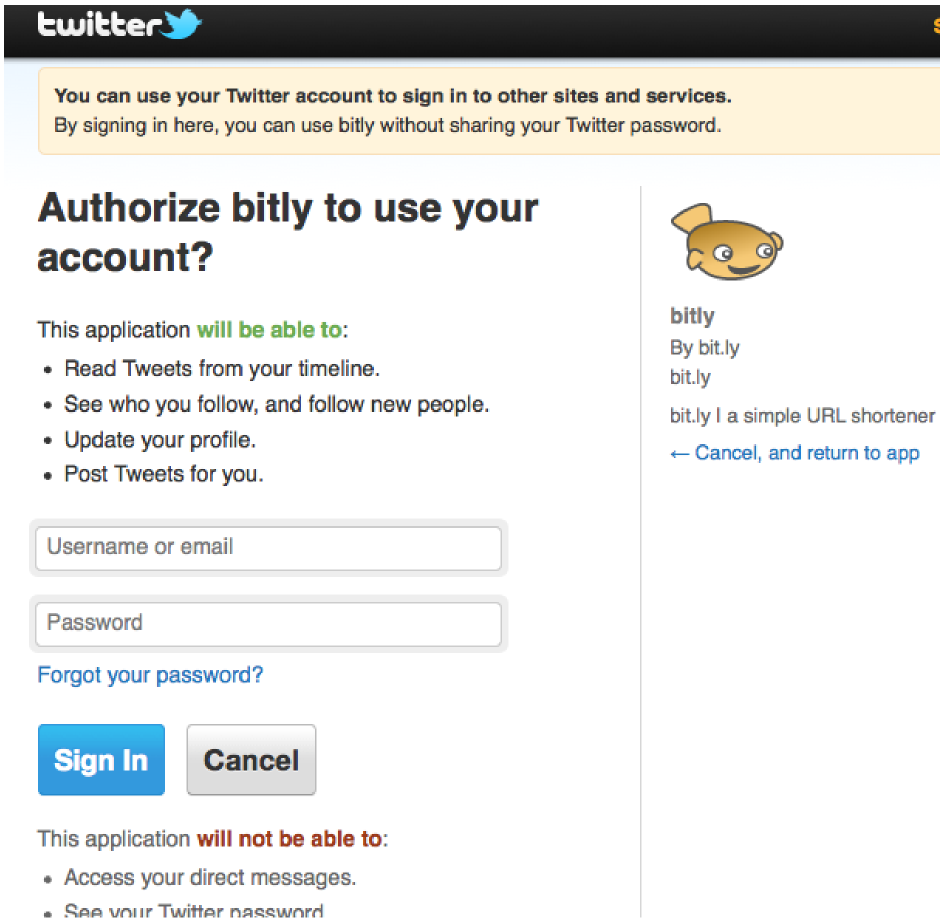
1. Bob visits bit.ly on the web, which uses a service provided by Twitter. Bob already has logins for bit.ly and Twitter.

2. Behind the scenes, bit.ly uses it's OAuth credentials to begin the authentication process for Bob.

   Bit.ly redirects Bob temporarily to twitter.com to log in. (bit.ly never sees Bob's Twitter password).

   The page hosted by Twitter that asks if an application can access Twitter on your behalf is probably familiar to most of us by now.

   Note that Twitter shows Bob what rights the app is asking for, and importantly what the app will not be able to do – in this case, see that bit.ly cannot access Bob's direct messages or see his Twitter password.


You can use your Twitter account to sign in to other sites and services.
By signing in here, you can use bitly without sharing your Twitter password.

Authorize bitly to use your account?

This application will be able to:

- Read Tweets from your timeline.
- See who you follow, and follow new people.
- Update your profile.
- Post Tweets for you.

Username or email

Password

Forgot your password?

Sign In Cancel

This application will not be able to:

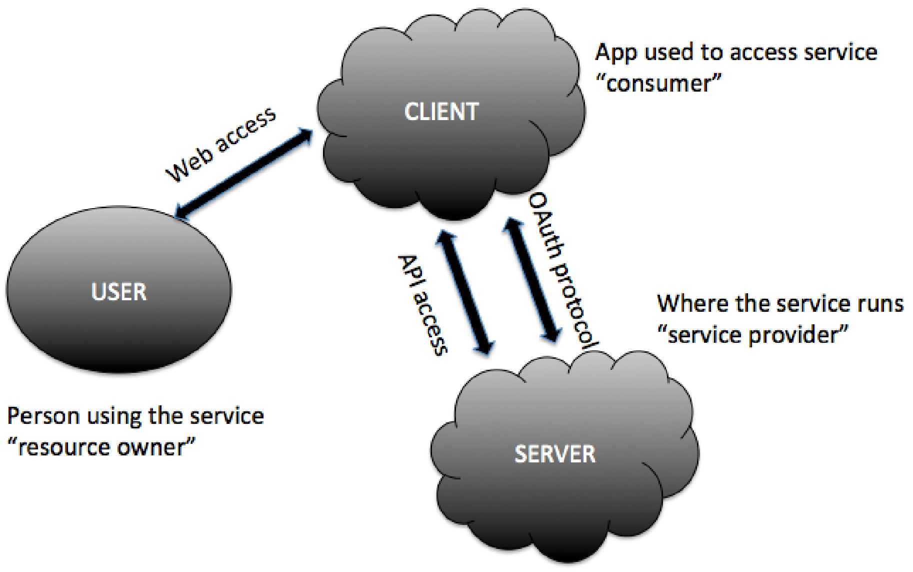
- Access your direct messages.
- See your Twitter password

bitly
By bit.ly
bit.ly
bit.ly I a simple URL shortener
← Cancel, and return to app

3. If this sign in is successful, bit.ly uses its own OAuth credentials (token) to retrieve credentials for Bob (that's the valet key that allows bit.ly to use Twitter on Bob's behalf).

4. Bit.ly stores Bob's credentials along with Bob's account. They allow him to use bit.ly and only bit.ly to access Twitter.

OAuth - The Big Picture



Why is OAuth important?

What if bit.ly is hacked and someone steals all the passwords? Bob will have lost his bit.ly password but the thieves don't have his Twitter password.

The Twitter API team, knowing that bit.ly has been hacked, can decide to revoke bit.ly OAuth credentials. Now everyone that uses bit.ly can't use Twitter.

This is extreme but can be important when an app is severely comprised so you don't have to change your passwords.

On the other hand, if Bob decides to change his Twitter password, the token that bit.ly has for Bob that allows him to access Twitter need not be affected.

Bit.ly never had Bob's Twitter password so he doesn't have to do anything at bit.ly – and he will be able to access Twitter through bit.ly next time he tries.

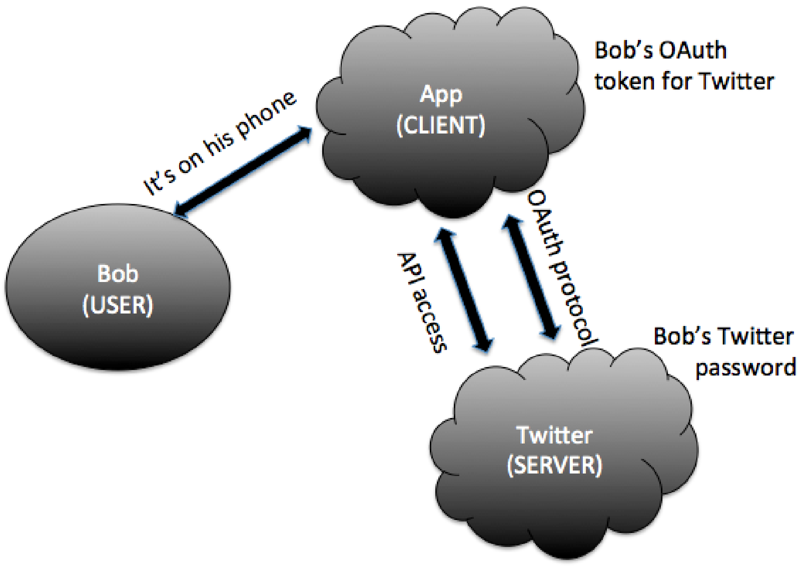
OAuth flow for mobile apps

In the previous section we looked at how OAuth allows users to grant third-parties access to their web services without sharing their passwords.

In that previous example, our user (Bob) accessed his Twitter account through the bit.ly web site.

What happens when Bob is using a mobile app instead of a web app? It's pretty much the same for a mobile app as a web app. When you fire up your mobile app and want to access Twitter, you don't want to have to give the mobile app your Twitter password and store it on your phone.

In the same way as happens for the web app, the phone app opens a browser window and directs Bob to login to Twitter (and authorize bit.ly to use his Twitter account) - using OAuth.



The phone app never sees Bob's Twitter password.

The phone app uses its OAuth credentials to retrieve credentials for Bob. It stores these locally. In this way,

OAuth allows this *one* phone app access to the Twitter service on behalf of *one* user (Bob).

This all happens through your browser. There are both advantages and disadvantages to this. The advantage of this is that the app never sees your password.

You don't have to trust the person that built the app or worry about losing your phone (as much).

If Bob loses his phone, he can log into Twitter and revoke the credentials that Twitter gave the phone app. (A thief cannot uncover the password, only the token.)

The disadvantage is that the app has to open up a browser window. This possibly breaks the flow of a nicely designed UI for a mobile app.

Twitter and Facebook for example, have made the choice to have everybody log in through the browser. This is how they can ensure that third-party developers of mobile applications never get access to the user's real password. Since the OAuth token can be stored on the mobile device, it's not necessary for the user to go through this step every time they launch the app, but only the first time.

On the other hand, opening up a browser window to log in a user, even once, can seem a bit cumbersome. That's why OAuth also makes it possible for a mobile app to ask the user for the password directly, fetch the OAuth token, and then erase the password from memory. This requires more trust in the mobile application, but some API providers, especially those who build mobile applications in-house, can use this technique and still retain the other advantages of OAuth, such as not having to store the password permanently on the device.

Why is OAuth good for API providers?

In previous sections we talked about why OAuth is good for users - how OAuth allows users to grant third-parties access to their web services or mobile apps without sharing their passwords.

We believe that OAuth is also good for API providers whether they are exposing APIs for web apps or APIs designed for mobile apps.

OAuth means that Web apps that expose APIs don't have to share passwords.

There are two alternatives: First, the security risk of typing your password into every single web app you use – an unacceptable risk! Secondly, some sort of universal single sign-on mechanism (run by a third party that everyone agrees upon and trusts), which of course doesn't exist!

Therefore, for all **web apps that expose APIs to other web apps** -- we recommend **OAuth**.

OAuth eliminates the need to store a password on a mobile device, adding a layer of security when an API is used by mobile apps built by untrusted developers for a public API.

Also, an OAuth token is hard to guess (because it is much longer and more random than a password.) It is tied to a particular app and device. It can be revoked without affecting other devices and apps.

OAuth allows the API provider to revoke tokens for an individual user, for an entire app, without requiring the user to change their original password. This is critical if a mobile device is compromised or if a rogue app is discovered.

OAuth also *future proofs* the authentication process. Because users' browsers are redirected to a place that's under the control of the API team to get the OAuth token, the API team can control what to do at that point in the workflow.

In other words, OAuth is flexible enough to support any of your company's specific workflows (think multi-factor authentication, terms of service, changes to terms of service, and so on…) without requiring a change to the client or server.

Therefore, for **APIs designed for mobile and native apps –** we recommend **OAuth.**

OAuth is complicated - is it worth the effort?

Because OAuth eliminates password sharing between web apps, and password storage on mobile devices, and importantly for the sake of your end-users' experience and security, we believe it is worth the effort.

There are reasons why OAuth seems more cumbersome and complicated for developers than plain passwords. Let's explore them as well as some recommendations to help you make OAuth decisions.

How many versions of OAuth and which one should you use?

OAuth 1.0 The first "production" version of OAuth 1.0 didn't actually do authentication delegation correctly and as a result, the spec itself had to be patched. No one should be using OAuth 1.0 now.

OAuth 1.0a Not an IETF standard but stable and well understood.
Uses a form of a cryoptographic hash code to sign each API request and exchange credentials between client and server. This means that you get API security without SSL (not only is the password never exchanged but the OAuth token is never exchanged – it's encrypted inside the signature). But coding the signature right is tricky.

OAuth 2.0 Actively under development in the IETF.
OAuth 2.0 is quite stable now – core flows are unlikely to change. It supports a 'bearer token', which is easier to code than the signature but requires SSL. (However, most APIs should be using SSL by default anyway.) Optional specs support signatures, SAML, and so on, but those specs are not yet stable.

The authentication dance is painful

There are a lot of great sites where you can get the details of the OAuth 1.0a authentication flow chart, so I'm not going to tackle that here. It's not simple. It gets a little simpler with bearer tokens in OAuth 2.0 but because of the security requirements and the fact that you have to exchange identity without a password, it's always going to be a little complex.

Signatures are painful

The good news is that in OAuth 2.0, signatures (the signing of each API call required in OAuth 1.0) are optional. You can do SSL and use a bearer token. As was mentioned above, getting the OAuth 1.0a signature algorithm right is difficult. If you're going to use OAuth 1.0a, we recommend you use one of the many libraries that are available.

Where do you store the credentials on the client?

It is important for the security of the mobile app that there is no way for an attacker to access to the device to access raw OAuth tokens. Unfortunately, there is no magic way to do this that is safe from every possible attack. For instance, you could encrypt th token, but then you need access to the key that decrypts it, and you can't encrypt that. One technique that makes life more difficult for an attacker is to encrypt the token using a key that is broken into several pieces and scattered in different parts of your application's files or source code.

What should you do?

Be patient. If your API can require HTTPS, use the latest draft of OAuth 2.0 with bearer tokens. Otherwise, use OAuth 1.0a. But the bottom line is that with your end-users' experience and security top-of-mind, we believe OAuth is worth the effort.

Implementing OAuth 2.0

If your API can require HTTPS, you should use OAuth 2.0. Otherwise, use OAuth 1.0a. It's possible to build an effective API right now using either a current draft of OAuth 2.0 (see Facebook) or OAuth 1.0a (see Twitter).

There are three types of credentials in OAuth 2.0.

BEARER TOKEN: A bearer token is a big random number – if the client has the token, then it's authenticated, and since the number is big and random guessing is pretty much impossible. Bearer tokens are easier for developers than OAuth 1.0a because they don't have to write the signature code. However, since it's transmitted as part of each API request in an HTTP header or a query parameter, bearer tokens provide no network security unless all the API calls are encrypted using SSL – although APIs that do anything sensitive ought to be requiring SSL anyway. This is the most straightforward of the credential types to implement. Use it!

MAC TOKEN: Like OAuth 1.0a, uses signature instead of SSL. The spec is still changing so we recommend that you wait. Once it is more stable, the use of the "MAC token" makes it possible to securely authenticate users without encrypting all traffic – this will be a good option for APIs that need the security of OAuth but handle very large requests or responses where SSL is inefficient.

SAML: OAuth 2.0 is also being extended to support the Security Assertion Markup Language (SAML) which is a very big and sophisticated security standard. Some big organizations love SAML and use it all over the place – in that case, it makes sense to use with OAuth. Otherwise, if you and your potential API developers don't understand SAML or know what it is, that's a signal to stay away. Also, the OAuth 2.0 specs around SAML are still fairly immature.

In short, we recommend that you use bearer tokens, as it is the simplest to implement.

How many legs?

OAuth is a protocol that involves three parties – the end user (who eventually holds and uses the access token), the "client" web app or mobile app that makes API calls, and the server that receives the API calls. Because there are three parties involved, many people use the term "three-legged" to describe OAuth.

Not all APIs need to authenticate the user – think of a "store locator" API or one that returns geographical information that is open to everyone. However, often these APIs wish to authenticate the application and developer who built the app, either for security purposes or to better track usage to make for a better API program.

Often this kind of application-specific validation is done using a unique "API key," but in the early days of OAuth many people also used the signature standard from OAuth 1.0a to do this. (The advantage here is that it allows a set of credentials to be securely transmitted from client to server without requiring SSL.) Since there are only two parties involved, this technique came to be known as "two-legged OAuth."

In the specific case of an API that requires application verification, but not end-user authentication, and can't handle the overhead of SSL, "two-legged OAuth" is a fine way to go. Otherwise, adding a unique "API key" to each request as a header or query parameter is simpler to implement and just as effective.

(One more pedantic note here – neither the OAuth 1.0 or 2.0 specs say anything at all about "legs." Whether "two-legged OAuth" is even OAuth at all is a fun subject for debate.)

Is OAuth all you need for API security?

OAuth has become the best practice and is essential for enabling secure user access and providing smooth user experiences. We strongly recommend OAuth for API providers when they are exposing APIs for web or mobile apps, but there are a couple of scenarios in which OAuth might be overkill or not the best solution.

For server-to-server APIs - APIs designed to be used only by a small number of servers – OAuth is overkill. Having a separate set of authentication credentials for each app is a nice feature of OAuth, but for server-to-server use, the need to log in securely using a browser, or to implement other steps in the OAuth "dance," gets in the way.

Instead, using a simple security standard like HTTP Basic authentication and assigning a unique password to each app is sufficient. Two-way SSL is another good, albeit cumbersome approach that has the advantage of stronger, more traceable authentication.

However, think ahead! Are those APIs really only going to be used by servers forever? In the future perhaps they'll be used by web apps or mobile clients, or the number of servers will grow far beyond what you first imagined. Furthermore, OAuth 2.0 has a number of ways for servers to securely get tokens that don't require a browser-based login. OAuth might seem like overkill today, but emerge as totally critical a few months down the road.

Use SSL for everything sensitive. Unless your API exclusively has open and non-sensitive data, support SSL and consider enforcing it by redirecting any API traffic on the non-SSL port to the SSL port. It makes other authentication schemes more secure, and keeps your user's private API data from prying eyes—and it's not all that hard to do.

Use API keys only for non-sensitive, read-only data. If you have a public API—which exposes data you'd make public on the Internet anyway–consider issuing non-sensitive API keys. These are easy to implement and still give you a way to identify applications and developers. Armed with an API key, you have the option of establishing a quota for your API, or at least monitoring usage by application. Without one, the only way to track usage is through IP addresses, which are not reliable. For example, the Yahoo Maps Geocoding API issues API keys so it can track its users and establish a quota, but the data that it returns is not sensitive, so it's not critical to keep the key secret. ("Two-legged OAuth" is another way to do this that protects the API key on the network without requiring SSL.)

Sanitize incoming and outgoing data. This will prevent malicious code or content from entering your system or being executed by clients that read your data. This practice should be employed for all APIs, but may be especially important for writable ones. For example, for a writable API, you do not want to allow insertion of JavaScript that could be used for cross-site scripting attacks when users visit your website. For example, if a parameter value is known to be numeric, it should be validated as numeric before being passed. Again, this practice should be applied for all APIs, but may be more needed for writable APIs.

Are there other scenarios for which OAuth is unnecessary or even a bad idea?

Just like the server-to-server scenario, we think that OAuth doesn't make sense in the following scenarios:

- Anything that requires commercial levels of trust. For example, when your security model requires the capabilities of a PKI infrastructure. Digitally signing each API call is slow and cumbersome but if your business requires it, then there's no substitute.

- One-time tokens. OAuth is a lot of complexity and machinery to make one API call.

Are there other OAuth anti-patterns out there?

Bad ideas include creating your own 'version' of OAuth or creating something that's *like* OAuth but different. Sticking with standards, and focusing your development efforts on creating great apps seems like a better idea than rolling your own security scheme.

Conclusions

As more businesses follow the platform imperative and add APIs, there is an imperative for the healthy growth of the market through the new intermediaries (apps). The imperative is to make it easy for developers to build great, secure apps that can delight users and grow businesses.

Teams who build APIs generally have to make many technical choices— SOAP or REST? XML or JSON? OAuth or something else?

Because it prevents password propagation around the Internet, while supporting a variety of ways to authenticate the end user, standardizing on OAuth is the clear choice for security.

Resources

http://oauth.net/

OAuth: The Big Picture. Apigee Webinar video and slides

Apigee API Tech & Best Practices Blog

API Craft Google Group

About Greg Brail and Sam Ramji

Gregory Brail, Technology

Gregory Brail brings a variety of experience to Apigee from more than 18 years in the industry. Prior to Apigee, Greg worked at BEA as technical lead for the WebLogic JMS team, and kicked off BEA's "Core Engine" initiative, which built a modular "micro-kernel" using core WebLogic technology. Prior to BEA, Greg served as Principal Architect for TransactPlus, a JPMorgan spinoff that offered guaranteed message delivery over the Internet. Greg began his career developing custom middleware libraries for Citibank, and then spent his formative years with transaction-processing pioneers Transarc, where he worked in the field, deploying production systems using Transarc technology at JPMorgan and elsewhere. Greg holds a degree in Computer Science from Brown University. Greg is a co-author of *APIs: A Strategy Guide*.

Sam Ramji, Strategy

Sam brings over 17 years of industry experience in enterprise software, product development, and open source strategy. Prior to Apigee, Sam led open source strategy across Microsoft, where he and his team worked with Bill Gates, Ray Ozzie, and contributed code to the Linux kernel. He was a founding member of BEA's AquaLogic product team and has built large-scale enterprise and Web-scale applications, leading the Ofoto engineering team through its acquisition by Kodak. Other experience includes hands-on development of client, client-server and distributed applications on Unix, Windows and Macintosh at companies ranging from Broderbund to Fair Isaac. Sam holds a B.S. in Cognitive Science from the University of California at San Diego, and is a member of the Institute for Generative Leadership. He holds positions on the boards of the Outercurve Foundation for open source, on the Open Cloud Initiative, and is a contributing editor to the ACM's journal Ubiquity.

Both Greg and Sam are frequent contributors on the Apigee API Tech & best practices blog, the Apigee YouTube channel, the API Craft Google Group, and Webinars.

About Apigee

Apigee is the leading provider of API products and technology for enterprises and developers. Hundreds of enterprises like Comcast, GameSpy, TransUnion Interactive, Guardian Life and Constant Contact and thousands of developers use Apigee's technology. Enterprises use Apigee for visibility, control and scale of their API strategies. Developers use Apigee to learn, explore and develop API-based applications. Learn more at http://apigee.com.

Accelerate your API Strategy

Scale Control and Secure your Enterprise

Developers – Consoles for the APIs you ❤