OAuth - Why it doesn't work, and how to Zero-day attack existing services

Table of Contents

Intro

What is OAuth?

Let me begin with a quick run down on what we're discussing. OAuth is a family of proposals that is being used for designing different kinds of authorization and authentication systems, and framing the usage for APIs and how different systems are integrated.

OAuth exists in many flavors, version 1, 1a, 2, and other specifications based on top of parts of it. Its exact usage also greatly differs across the many different implementations out there.

Disclaimer

I've written about OAuth in the past, and I've gotten a lot of feedback from readers all over. To avoid some criticism my past article received, I'd like to point out that this article will be focusing on how OAuth is typically used, and most of the points discussed here are true of nearly every major service which leverages OAuth in some way.

To put this differently, not every platform utilizing OAuth is necessarily broken. Due to the various flavors of OAuth, and the 76-page document on different possibilities with OAuth 2.0, it is possible to create something secure and sane to use which conforms to something OAuth based. Therefore your favorite flavor and design with OAuth may escape some or all of the problems discussed herein, even though the odds are that it does not.

Some of you may also argue that some things done with OAuth are misusing the specifications, or that OAuth doesn't specifically mandate things be done the way they are. Whatever the case may be, I'm not here to write about a particular OAuth based specification, but how OAuth is currently utilized by major players in the market, regardless if what these organizations are doing conforms with the specifications or not. This will impact many readers, either because they use services making use of OAuth in these ways, or they're maintaining OAuth-based services, and are building platforms similarly to how many others are building them.

Article structure

This article is going to be a long one, and in fact most sections in this article cover topics with enough material to make an entire article themselves, so let me give a brief outline on what this article is about and how it will be organized.

This article is the presentation of the culmination of research into what is currently being done in the market with OAuth. Not everything in this article is consistent with every product making use of OAuth, but this article will present what was found to be common practice and the biggest culprits behind insecure or useless services.

After this introduction, I'm going to begin by analyzing security flaws with OAuth. The general principles behind some of these flaws are well known by those in the security community, and already briefly analyzed in existing publications. However, I'm going to be covering some cases that have not yet been publicized, and also elaborating upon some points of well known flaws to make them easier to understand to the average developer or manager and emphasize why they need to be concerned.

After that I'm going to present an analysis of how some key components of OAuth are popularly implemented and how the popular implementations cripple services which choose to use OAuth by severely, inappropriately, and undesirably limiting what can be accomplished with them. Some techniques will be discussed that can be used as the basis for workarounds in a limited amount of cases, with a focus on the absurdity involved in implementing such. As part of the above, it will be frequently pointed out how those using OAuth are hurting themselves and their business.

Lastly I will briefly cover a few cases for when and how OAuth can be used properly, along with viable alternatives to OAuth which are currently in use. I will provide a survey of alternative techniques that will include what serious companies like Amazon and others are doing to achieve secure, usable, and reliable APIs.

Responsible Disclosure

Several of the flaws with OAuth *as is it popularly used* today can be exploited to carry out strong attacks against major services. Attacks against OAuth are nothing new, as IBM, Oracle, and other concerned members of the IETF issued a 71-page document describing 50 classes of attacks against OAuth based services over three

years ago, and I already discussed some of these points in my previous article. Despite this though, major security flaws with OAuth-based systems are extremely common.

I've spoken to executives and developers at several major corporations to point out security flaws with their OAuth-based systems (in one case, 4 years ago), and not one of them has done anything to fix their systems. It seems as though, given the popularity of OAuth, they haven't heard of any of the alternative *viable* solutions, and they therefore assume that OAuth must be the most secure thing available. It also seems as if they either have not heard or shrug off the information documenting the demonstrated attacks against the core principals of OAuth. I'm hoping that wider public dissemination of this information will give those affected the proper kick in the pants that they need, and serve as a wake up call to those designing or maintaining services.

Therefore, if you investigate and find that some or all of these flaws exist in your service which either provides or makes use of something OAuth-based, please act responsibly in dealing with this information. Update your services appropriately, or apply appropriate pressure to your business partners to address the relevant issues.

Although the various information mentioned here and linked to from here can be used to exploit existing services *today*, please be responsible and aim to improve and not destroy what belongs to others. This article is meant as a wake up call for those implementing their services improperly and an appeal to improve them, not a how-to guide for hackers looking to exploit them.

Usage Scenarios

For this article, I'm going to focus on two usage scenarios and see how OAuth stacks up with them and why it doesn't work. It is important to keep these scenarios in mind, as I will be constantly returning to them throughout this article.

Let us imagine an Exciting Video Service (or *EVS*), where users can upload videos and share those videos with their friends, providing either public or restricted

access to their uploads. EVS also provides OAuth-based APIs for uploading and deleting videos, and managing permissions regarding who can view those videos.

While I will be focusing on this imaginary service for sake of example, the issues under discussion will apply to any service, whether it's for file and document storage, calendar management, online meetings, discussion groups, resource management, or anything else that provides an OAuth-based API. Also bear in mind that I'm not actually referring to any specific video service, even though some or all of these issues may apply to existing video services which make use of OAuth. It can be left as an exercise for the reader to determine which ones.

For our first usage scenario, let us imagine a camcorder manufacturer that would like to supply software with their camcorder which, among other things, allows videos recorded with the device to be uploaded to EVS. It is intended to allow users of this camcorder to plug their device into their computer, open the custom software, select the videos on the device they'd like to upload, and come back later knowing that all the selected videos will have been uploaded to EVS.

For our second scenario, let us imagine a small organization decides to purchase 50 accounts with EVS for their employees, so all employees can upload videos and have them shared with other employees from the same department. This organization is using A Friendly Custom Platform, for managing their employees and which sub-units they belong to, and would like to integrate their AFCP service with EVS. This organization will expect that when a manager assigns someone to the sales department using AFCP that the employee in question will automatically gain access to all videos that belong to members of the sales department. They will expect the reverse to occur if they remove someone from the sales department, and all similar scenarios.

The Issues

Security Related

Stealing credentials / Gaining elevated access

One of the most popular reasons why token-based authentication systems (a core premise of OAuth) are currently used is that *when implemented properly* they avoid the need to provide third party applications and services with individual users' credentials. It is undesirable to provide third parties with personal user credentials, because:

- It gives third parties more access than they require.
- It is another place personal credentials are stored and can be stolen from.
- It requires that API consumers be updated when users change their credentials.
- Access cannot be easily revoked from just one application without also revoking access from all other applications.
- User credentials can be too limited where additional authentication factors are in use.

The above list of problems can be avoided by any token-based authentication system, not just OAuth. While this is counted as a strength to OAuth, it is hardly unique, and viable alternatives can be used which carry the same strengths without OAuth's weaknesses.

However, despite being based on a solid premise, OAuth *as popularly implemented* attempts to avoid the above problems by providing a system with something along the lines of the following steps:

1. Users visit the third party application/service such as AFCP and informs it of their desire to integrate a particular service.
2. AFCP then brings up a special login page hosted by EVS where the user enters their EVS credentials.
3. EVS then asks the user to confirm they are sure they want the third party application/service, AFCP, to have all the levels of access specified.
4. EVS then provides AFCP with some kind of token or series of tokens it can then use to make the various API calls.

Since these tokens are not the user's credentials, and can be specific to every user and application combination, limited in their permissions, and later revoked, it seemingly avoids all the aforementioned problems this setup was designed to address. However, in reality, despite having a sound premise at its core, this particular usage flow used by popular implementations of OAuth does not address all the above enumerated issues.

This design begins from an insecure standpoint. The rule of thumb when it comes to designing a secure platform is that anything which begins from an insecure standpoint is already lost, it cannot be salvaged. Thanks to step #1, which begins on a service making use of EVS rather than EVS itself, users have already been man-in-the-middled from the very beginning. This is the computer-system equivalent of giving out personal or financial information over the phone to an incoming caller who claims to be from some utility service you use but whose number you either do not recognize or is blocked. There have been many such scams in recent times, and don't need to be elaborated upon here. The main point is that if you cannot trust the party which initiated a connection, then you cannot trust the connection at all. It is impossible to design a secure authentication system for API use which achieves the objectives enumerated above unless the first step begins from the EVS side itself.

Similar to the phone scam example, where the caller simply needs to sound as if they are calling from the company they're impersonating, to attack a user, a third-party application or service just has to provide something which looks like the special EVS login page for step #2. Most users will not be aware that the page displaying the EVS logo and asking for their username and password is not authentically from EVS. Once a third-party service gains the user's credentials in this manner, the application or service, and those behind it, now have more access than they should, and their access cannot be revoked without the user changing their credentials.

Since users have already been conditioned with OAuth-based login flows with all kinds of embedded frames with corporate logos, silly pop-ups, and redirects with ridiculous URLs on atypical domains, most users won't even notice any red flags which would alert them that the page into which they're currently entering their credentials may not be genuine. A colleague of mine put it so: *The companies*

themselves are training their users to be phishing targets. If URLs are needed to be displayed, the attacker can even register an official sounding domain (microsoft-authentication.us, ibm-secure.co.uk, gooogleauth.us, my-citrix.au, globalauth.world) and can even have redirection go through some URL which appears to be legit.

In the case of actual standalone applications, such as that provided with your camcorder, nothing outrageous even needs to be done. Since the web browser is being provided by a custom application, it can already capture every single input box and all data sent over the network with it, it doesn't even need to spoof some login form.

This class of attack is labeled in the aforementioned security document as 4.1.4. Threat: End-User Credentials Phished Using Compromised or Embedded Browser. The solutions offered? (emphasis mine)

> **Client applications should avoid directly asking users for their credentials**. In addition, end users could be educated about phishing attacks and best practices, such as only accessing trusted clients, as **OAuth does not provide any protection against malicious applications** and the end user is solely responsible for the trustworthiness of any native application installed.

Also:

> Client **developers should not write client applications that collect authentication information** directly from users and should instead delegate this task to a trusted system component, e.g., the system browser.

Essentially OAuth security guidelines say that developers making use of OAuth should not try to attack the users or do anything malicious. Relying on external developers not to do anything malicious is not a security model any sane service designer would rely on.

Nearly every major OAuth-based service I know of can be attacked with the method outlined here.

For those of you thinking OAuth is the new gold-standard for security, wake up! OAuth *as popularly implemented* is already defeated before it has begun. Many systems implemented way before OAuth ever existed are secure and work around this issue effectively. Unfortunately, and all too often, I've seen services transition themselves from something secure to an insecure OAuth model because someone told their developers or managers that OAuth was "*more secure*", "*forward thinking*", "*future proof*", or any number of other buzzwords which sound nice but lack anything in the way of substance. Most of the time these changes are implemented without even reviewing whether these changes address any existing problems or if the solution is any better than what they are replacing.

Masquerading as an OAuth-using service

A common mistake I see in OAuth-based service design is the supplying of an endpoint designed for a web browser which accepts as one of its parameters a *client_secret* (or something of the same concept). The OAuth *client_id* and *client_secret* parameters are essentially a third-party platform's equivalent of its own personal API username and password, and should therefore only be known to those developers making use of EVS's APIs. Since it is analogous to a password, the *client_secret* parameter should **never** be sent across a user's web browser (*hint: the word secret is in the name of the parameter*). If some user of an application or service can find out the *client_id* and *client_secret* of the application or service, that means they can masquerade as that service and potentially do something malicious. Also note that some services will sometimes name the *client_secret* parameter something else, so review the service you are working with carefully and see if any of their other parameters need to be kept secret. Unfortunately, since important variables are sometimes not indicative of their nature, this problem is more common than it should be. Additionally, some services will build an authentication flow on top of OAuth using the *client_id* alone. Be wary of these, because under certain circumstances such a *client_id* functions exactly like a *client_secret*.

Since OAuth *as popularly implemented* transfers users between multiple websites using a web browser, and that OAuth needs one website to send the other a *client_id* and *client_secret* (or equivalent), such as between AFCP and EVS, these are actually available to the user of the web browser if they monitor the browser's HTTP log. This is even possible in various custom browsers built into applications where a

simple right click allows access to an inspector of some sort with network logging capabilities.

In our case of AFCP utilizing EVS, this flaw would allow employees which have some access to AFCP to potentially gain more access than they should, and perhaps apply permissions they should not have access to. In a different example, if Facebook made use of an OAuth endpoint via a web browser for GMail where both the *client_id* and *client_secret* were transferred through the web browser, this would allow every user of Facebook to impersonate Facebook itself in this regard.

This issue is present any time an OAuth endpoint expects to be sent the *client_secret* in plain text via a user's web browser, or the API consumer is mislead into thinking doing so is a requirement and embeds a secret where they should not. A good indication that this vulnerability may be present is an endpoint where both the parameters for *client_secret* (or equivalent) and *redirect_uri* are expected (or even optionally allowed). The *redirect_uri* parameter is designed specifically for browser use to indicate to EVS where to send the user's browser after the login actions have been performed on the EVS side. As such, it means that if *redirect_uri* is in use for transferring on an endpoint, the flow is expected to be performed via the user's web browser. Neither of the major OAuth documents specify or call for the use of such a case where both the parameters for *client_secret* and *redirect_uri* are expected to be used like this.

A quick search online of such potentially offending OAuth-based APIs unfortunately shows many hits. Although Google offers many ways to use OAuth, they have a flow which advertises the two being used together:

Handling the response and making a token request

The response to the initial authentication request includes an authorization code (`code`), which your application can exchange for an access token and a refresh token. To make this token request, send an HTTP `POST` request to the `/oauth2/v4/token` endpoint, and include the following parameters:

Field	Description
`code`	The authorization code returned from the initial request.
`client_id`	The client ID you obtained from the Google Developers Console.
`client_secret`	The client secret you obtained from the Developers Console (optional for clients registered as Android, iOS or Chrome applications).
`redirect_uri`	The redirect URI you obtained from the Developers Console.
`grant_type`	As defined in the OAuth 2.0 specification, this field must contain a value of **`authorization_code`**.

The actual request might look like the following:

```
POST /oauth2/v4/token HTTP/1.1
Host: www.googleapis.com
Content-Type: application/x-www-form-urlencoded

code=4/v6xr77ewYqhvHSyW6UJ1w7jKwAzu&
client_id=8819981768.apps.googleusercontent.com&
client_secret=your_client_secret&
redirect_uri=https://oauth2-login-demo.appspot.com/code&
grant_type=authorization_code
```

Citrix makes this mistake:

After granting access the user will be redirected to the URL you passed in the redirect_uri parameter with an **authorization code** appended:

```
http://YOUR_URL?code=THE_AUTHORIZATION_CODE
```

You must then use the authorization code to obtain an access token. To authenticate your app, you must pass the authorization code and your client_secret to our endpoint. In return you will receive an access token. Make a **POST** like this:

URL	https://podio.com/oauth/token
Request body	grant_type=authorization_code&client_id=**YOUR_APP_ID**&redirect_uri=**YOUR_URL**&client_secret=**YOUR_APP_SECRET**&code=**THE_AUTHORIZATION_CODE**

Along with Cisco:

Getting an Access Token

If the user granted permission to your app, Spark will redirect the user's web browser to the `redirect_uri` you specified when entering the grant flow. The request to the Redirect URL will contain a `code` parameter in the query string like so:

`http://your-server.com/auth?code=[illegible]`

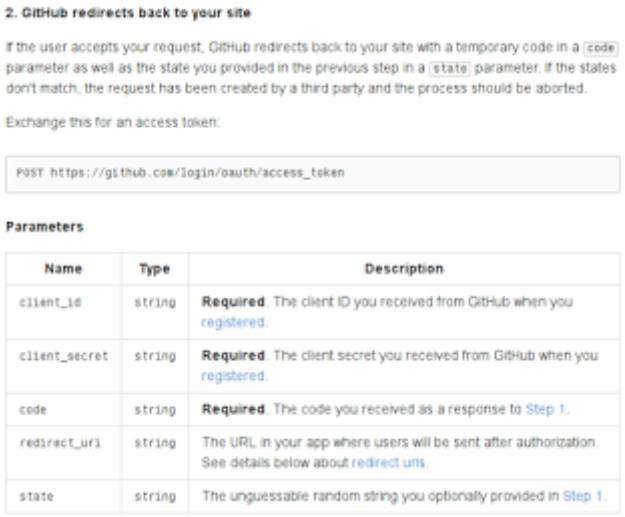
Your app will then need to exchange this Authorization Code for an Access Token that can be used to invoke the APIs. To do this your app will need to perform an HTTP POST to the following URL with a standard set of OAuth parameters. This endpoint will only accept an `x-www-form-urlencoded` body.

`https://api.ciscospark.com/v1/access_token`

The required parameters are:

`grant_type`	This should be set to "authorization_code"
`client_id`	Issued when creating your app
`client_secret`	Remember this guy? You kept it safe somewhere when creating your app
`code`	The Authorization Code from the previous step
`redirect_uri`	Must match the one used in the previous step

So does Github:



2. GitHub redirects back to your site

If the user accepts your request, GitHub redirects back to your site with a temporary code in a `code` parameter as well as the state you provided in the previous step in a `state` parameter. If the states don't match, the request has been created by a third party and the process should be aborted.

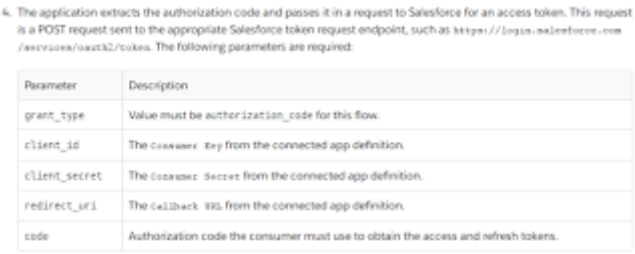
Exchange this for an access token:

```
POST https://github.com/login/oauth/access_token
```

Parameters

Name	Type	Description
client_id	string	**Required**. The client ID you received from GitHub when you registered.
client_secret	string	**Required**. The client secret you received from GitHub when you registered.
code	string	**Required**. The code you received as a response to Step 1.
redirect_uri	string	The URL in your app where users will be sent after authorization. See details below about redirect urls.
state	string	The unguessable random string you optionally provided in Step 1.

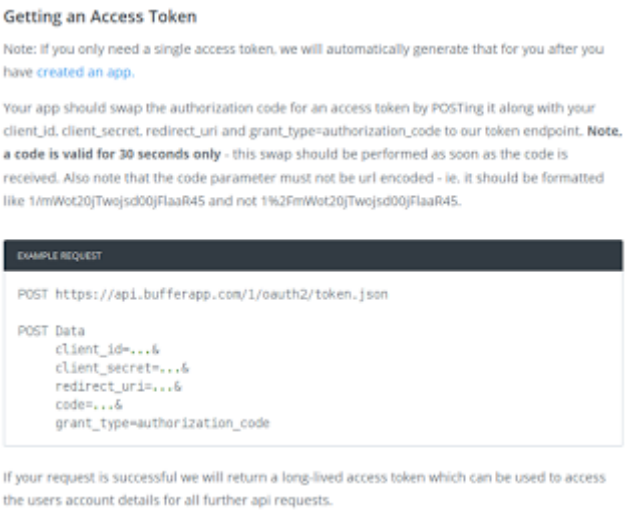
And Salesforce:



4. The application extracts the authorization code and passes it in a request to Salesforce for an access token. This request is a POST request sent to the appropriate Salesforce token request endpoint, such as `https://login.salesforce.com/services/oauth2/token`. The following parameters are required:

Parameter	Description
grant_type	Value must be `authorization_code` for this flow.
client_id	The `Consumer Key` from the connected app definition.
client_secret	The `Consumer Secret` from the connected app definition.
redirect_uri	The `Callback URL` from the connected app definition.
code	Authorization code the consumer must use to obtain the access and refresh tokens.

As well as Buffer:



Getting an Access Token

Note: If you only need a single access token, we will automatically generate that for you after you have created an app.

Your app should swap the authorization code for an access token by POSTing it along with your client_id, client_secret, redirect_uri and grant_type=authorization_code to our token endpoint. **Note, a code is valid for 30 seconds only** - this swap should be performed as soon as the code is received. Also note that the code parameter must not be url encoded - ie, it should be formatted like 1/mWot20jTwojsd00jFlaaR45 and not 1%2FmWot20jTwojsd00jFlaaR45.

EXAMPLE REQUEST

```
POST https://api.bufferapp.com/1/oauth2/token.json

POST Data
     client_id=...&
     client_secret=...&
     redirect_uri=...&
     code=...&
     grant_type=authorization_code
```

If your request is successful we will return a long-lived access token which can be used to access the users account details for all further api requests.

And Zendesk:

Step 3 - Get an access token from Zendesk

If your application received an authorization code from Zendesk in response to the user granting access, your application can exchange it for an access token. To get the access token, make a POST request to the following endpoint:

```
https://{subdomain}.zendesk.com/oauth/tokens
```

Include the following required parameters in the request:

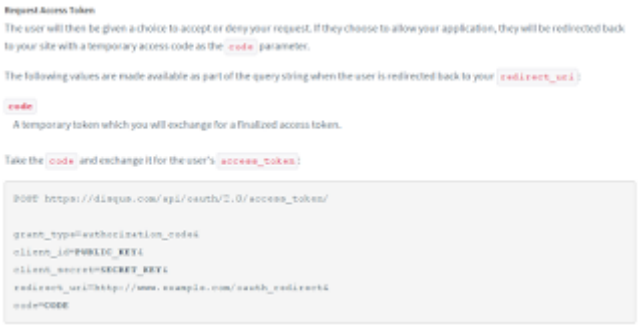
- **grant_type** - Specify **authorization_code** as the value.
- **code** - Use the authorization code you received from Zendesk after the user granted access.
- **client_id** - Use the unique identifier you received when you registered your application with Zendesk.
- **client_secret** - Use the Secret value you received when you registered your application with Zendesk.
- **redirect_uri** - The same redirect URL as in step 3. For ID purposes only.
- **scope** - Specify **read** as the value.

The request must be over https and the parameters must be formatted as JSON.

Using curl

```
curl https://{subdomain}.zendesk.com/oauth/tokens \
  -H "Content-Type: application/json" \
  -d '{"grant_type": "authorization_code", "code": "{your_code}",
    "client_id": "{your_client_id}", "client_secret": "{your_client_secret}"
    "redirect_uri": "{your_redirect_url}", "scope": "read" }' \
  -X POST
```

And the popular Disqus:



Request Access Token

The user will then be given a choice to accept or deny your request. If they choose to allow your application, they will be redirected back to your site with a temporary access code as the `code` parameter.

The following values are made available as part of the query string when the user is redirected back to your `redirect_uri`:

`code`

A temporary token which you will exchange for a finalized access token.

Take the `code` and exchange it for the user's `access_token`:

```
POST https://disqus.com/api/oauth/2.0/access_token/

grant_type=authorization_code&
client_id=PUBLIC_KEY&
client_secret=SECRET_KEY&
redirect_uri=http://www.example.com/oauth_redirect&
code=CODE
```

I got this list after searching with Google for only two minutes. I'm sure readers can find many more with a little more effort. Note, the above list is not an indication that any of these services are directly vulnerable or that it's too easy to misuse them. Due to various factors, where for example, Zendesk specifically says their *redirect_uri* parameter is not actually used for redirection *in this particular case*, and that they even show that the endpoint should be called from an application using curl - not a full fledged web browser, developers will hopefully not be misled into trying to do something dangerous with it. However, inexperienced developers in their applications may try to load one of these endpoints with a custom web browser. Furthermore, this combination simply being common in the wild is lowering developer's defenses against a potentially nasty misuse, where even the

more experienced developers of OAuth-based services are blindly applying in similar situations, especially where the *client_secret* is named something else and the idea of keeping a secret is lost on them.

A good indication that a service is broken in this regard is when several popular OAuth libraries fail to work with this service. Such services will generally offer their own "SDK" which will work with them, and point third party developers to the official SDK if they complain their favorite library is unable to make use of their frankenstein-OAuth. These kinds of *customizations* often goes unnoticed, as the majority of developers prefer to make use of an advertised SDK when provided, and forgo rolling their own combination of software to utilize a service.

This class of attack is labeled in the aforementioned security document as 4.1.1. Threat: Obtaining Client Secrets. However, it doesn't even mention the specific attack case where a server is requiring use of a web browser for passing both the *client_id* and *client_secret* (or something of similar use). The authors of the document probably didn't expect anyone to design a service which could be this stupid nor developers using such APIs to make use of them with a custom web browser or SDK. These developers are mixing and matching separate components from different parts of the OAuth specifications in unspecified ways and expecting their platforms to remain secure without considering what new issues may be introduced by this patchwork approach. This is unfortunately the manner in which most of the OAuth players today function, and the already rampant problem only perpetuates itself as more and more providers jump on the bandwagon and copy the approaches they see *or* ***think*** *they see* being used by others.

Odds are you'll be able to find many systems making use of the above services which are exploitable due to this problem. It's especially common in desktop applications, where a secret can be directly pulled out of a compiled application binary, even if the service being used isn't requiring anything insecure. It is important to note that Google offers many ways to use OAuth, and only one of them has an endpoint which receives both *client_secret* and *redirect_uri*. In the case of Google at least, they aren't recommending this endpoint for web browser based applications despite the presence of *redirect_uri*, but I'm sure that doesn't stop anybody from using it with a custom web browser or copying this flow into their

service for an endpoint intended for regular browser use. In addition, Google appears to be the exception which proves the rule, and there's still the stupidity of all the other OAuth-based services out there which do not allow for secure OAuth-flows in such cases, as they require the *client_secret* (or equivalent) to always be passed, even when the flow is via a web browser. Worse, many of these services can **only** be used via a user's web browser, a point which will be further elaborated upon below.

The aforementioned security document mentions a couple of malicious things one can do once they steal an application's credentials (*client_id* and *client_secret*). I'll cover some issues below that can be coupled with this attack to allow one to do some nasty things which have not, to my knowledge, been previously covered. I'm sure my creative readers will also find additional ways to exploit stealing what is supposed to be kept *secret*.

Insecure Tokens

Token based authentication is a new concept to many developers. Therefore, it is also commonly misunderstood. Many developers designing something like EVS think if they simply follow some design guidelines (such as OAuth) on how APIs should work, or copy what other platforms are doing, then their platform is inherently secure. However, in order for something to be secure, it requires that every single component be secure, that the combination of the components be secure, and that the overall framework be secure. Remember, security is only as strong as its weakest link, and it is not enough to rely solely upon adhering to some overall framework and assume that any and all use of it is therefore secure. The OAuth-based framework in and of itself provides very little in the way of ensuring the security of the underlying components (if it's not outright counter-secure for certain components).

In order for a token-based system to have any semblance of security, the tokens generated must use a cryptographically secure pseudo-random number generator, a topic which is not so well understood. Even worse, APIs for good CSPRNGs in popular scripting languages is severely lacking, yet such scripting languages are often the foundation of what is being used to design many popular modern services.

If the tokens generated are predictable, that means that an attacker can impersonate users and perform malicious activities with their account simply by guessing at what a token might be. I saw one OAuth-based service from a major fortune 500 company which just uses some constantly incrementing ID (perhaps a database field?) for its tokens. I found another where I noticed the tokens generated all seemed to be the output of some monotonic-function. With further research, I found it was an extremely simple algorithm based on the current real world time. On these systems, I could log in as myself, see what the current token IDs are, predict what the next series are going to be, and then use that as part of a token exchange or other operation on behalf of the next random user. Combined with other techniques, even more targeted attacks can be accomplished.

This class of attack is labeled in the aforementioned security document as 4.5.3. Threat: Obtaining Refresh Token by Online Guessing and 4.6.3. Threat: Guessing Access Tokens. While this issue is remediable, the amount of services currently making this mistake, and the ease with which this mistake is made, does not bode well for proving the security of a given OAuth-based service from an external review.

Certain attacks against randomness, which is a whole other topic, can be used to utterly destroy an OAuth-based server if a security-hardened CSPRNG is not in use. While such issues are extremely problematic with other systems too, the issues are much more pronounced with popular OAuth implementations whose entire method of functioning is based around the concept of handing out random numbers. These tokens are generated server-side by EVS, and when used constantly as OAuth *as popularly implemented* does, drains the reliability of the server and increases predictability of all tokens involved. If you don't have a security-hardened CSPRNG for your environment which can protect against modern attacks, you're probably better off with another protocol which is more forgiving in this regard.

It should however be noted that some OAuth-based implementations structure things in such a way to move randomness requirements to the client-side. While in many ways this is just moving a problem elsewhere, it does reduce the attacks surface from the service-side of things. This at least allows for more educated consumers of OAuth-based services to use a service they can trust in a secure

manner, even if the less educated consumers may still be vulnerable. Applying this to our example, this setup would mean that the developers of AFCP can ensure its use of EVS is secure, and can't be exposed to such threats from EVS itself, even if ABC or XYZ do not use EVS securely.

Cross-site request forgeries

Before I get more into this one, let me just point out that despite the name, CSRF attacks are not necessarily originated from a third party site. CSRF attacks can also be initiated from within sites themselves that allow users to post their own links, such as various online discussion and messaging software.

There are many different techniques and frameworks designed to combat CSRF in various ways. OAuth-based integration can make many of these systems unusable or prompt various unsafe practices which can open sites up to attack.

One defense mechanism against CSRF is to ensure that the referer (sic) sent by the browser does not point to an external site. Since many OAuth implementations require that users are directed to certain pages from an external site, this defense cannot be enforced. Since the full scope of what pages and multitude of third party domains an OAuth server will perform redirection through, and since these URLs and domains involved are undocumented and may see periodic changes, the full scope of EVS domains and pages cannot be whitelisted.

Also one needs to consider whether it’s possible for those offering EVS to turn around and try to attack AFCP. One of the principles behind OAuth is that OAuth-based services are not expected to trust their consumers, yet at the same time, require that the consumers fully trust them by allowing them blanket CSRF avoidance. An ideal authentication system would ensure a mutual level of distrust, not a one-way street.

Partial whitelisting for either sources or destinations may also prove problematic. Depending on the anti-CSRF framework in use, tools for disabling certain features may be all-or-nothing, and it may not be possible to disable features on specific pages or for specific sources, forcing those who make use of EVS to stop using their anti-CSRF framework altogether.

OAuth specifically defines the optional *state* parameter to be used to specify CSRF tokens to prevent CSRF attacks. However, I found that OAuth-based services commonly have limitations on length or allowed characters in *state*, and may not return it verbatim as required. Therefore, due to weird compatibility issues, many consumers of OAuth-based services find themselves forced to turn off CSRF protection altogether on redirection endpoints. This is labeled under 10.14. Code Injection and Input Validation. Another consideration with the *state* parameter is that anyone with access to it on the EVS side can alter the request before sending the browser back to AFCP with an otherwise *valid* state parameter.

OAuth-based API consumers are further limited by being required to define a URI or series of URIs up-front upon registration of their application or service, which is a white-list of URIs that can be used for *redirect_uri*. Putting aside for the moment major usability issues with such a system which will be discussed below, this limitation forces developers to start getting creative with the *state* parameter or other potentially dangerous ideas which can lead to a whole slew of issues. Many OAuth-based servers allow only a single white-listed URI, or require an exact match with *redirect_uri* and disallow additional parameters appended to it. This causes developers to stop using their anti-CSRF framework and start trying to shove all sorts of dangerous stuff into the *state* parameter or create other poorly thought out systems. The end result is some combination of *redirect_uri* and *state* that can be used to push users to some page they should not be pushed to. This is labeled under 10.15. Open Redirectors.

The problem with such redirection can potentially be exploited by itself, due to the combination of parameters not being fully authenticated, or this exploit can be combined with the issue - *Masquerading as an OAuth-using service* documented above which can be used to wreak havoc upon users. By making use of a stolen *client_id* and *client_secret*, a malicious party can create a redirection flow which appears genuine even to AFCP, by the fact that the authentication was performed with AFCP's own credentials. A malicious user of AFCP may also be able to find a way to use or modify the *state* parameter, perhaps with stolen client credentials, in order to gain permissions they should not have within AFCP. All in all, due to poor design by OAuth *as popularly implemented*, and the difficulties external developers

face with poor education on certain topics, attacking OAuth-based consumers is often much easier than it should be.

Important reading here is also 3.5. Redirect URI, 3.6. "state" Parameter, and 4.4.1.8. Threat: CSRF Attack against redirect-uri.

Section conclusion

In terms of security, OAuth *as popularly implemented* does quite poorly. OAuth fails to achieve many of the security objectives it supposedly sets out to achieve. Further, some OAuth-based services outright require that their consumers open themselves up to various attacks in order to use them. Even in cases where OAuth-based services can be used securely, which isn't always known (due to important service-side security details such as token generation methods being undocumented and closed source), OAuth still leads to many poor programming practices. OAuth does little to protect external developers, and various frameworks that such developers use in turn don't provide real security or can't be used securely without strict discipline and caution.

This article in turn only covers some issues that I found rampant. Some of these issues are also the result of developers copying extremely poor practices they see others using alongside OAuth, practices which aren't even mandated by any OAuth specification.

Developers for both OAuth-based services and the consumers thereof need to read and understand all of the linked documentation for implementing and using OAuth-based platforms. One also needs to fully comprehend the 50 classes of attacks listed, the multitude of attacks in each class, and note that the implementation material and security guidelines are not exhaustive. It should also be noted that this article only touches the surface of OAuth security problems, even if it does cover some issues not documented in the official material. Compounded with this, any changes made to the official OAuth proposals introduce a whole new set of security challenges, and these kinds of changes are unfortunately common. One in turn also needs to understand other security-related fields such as random number

generation and security verification techniques to achieve any reasonable level of security with OAuth.

If you're looking for real security, I recommend you look elsewhere. I'll cover some alternatives to OAuth in the final section.

Usability Related

Pull the plug architecture

OAuth *as popularly implemented*, requires that every integration done with an OAuth-based service require a set of application specific credentials to simply exist. These credentials aren't actually used to manage any particular set of user or organization accounts.

This design allows EVS at any time to pull the plug (revoke the application credentials) on any application they no longer wish to service. This is often seen as an advantage to OAuth. However, this exact design is also telling of the intentions behind EVS's service, namely that they want full control over all use of EVS, and have the right to refuse service via APIs on a whim.

If the company behind EVS decides to go into their own camcorder business, and wants to reduce competition, they can pull the plug on competing products that integrate with EVS. Imagine if desktop applications from their inception required approval from the Operating System vendor in order to function. When Microsoft decided to create Excel, they could have pulled the plug on Lotus 1-2-3. Nowadays they would be able to just prevent anyone from using Open Office or Libre Office, and force everyone who wants some desktop application office suite to purchase Microsoft Office.

Besides being an evil way to practice business, this also hurts adoption of the service itself. Why would camcorder vendors want to waste money developing for EVS if any application they design can be shut down on a whim remotely? Even if they did offer some application with EVS compatibility, they certainly wouldn't want to advertise it for fear of being sued for false advertising and premature termination of service if EVS ever pulls the plug on them. Would you be comfortable providing a service subscription to someone, knowing that some third-party could

come and completely disable the service at any time? Would you really expect that a client (especially a paying client) would accept an apology that says “sorry, EVS shut our service down, there’s nothing we can do”? Do you really think that such clients would let it go without seeking refunds and damages from you?

As a manager making buying decisions and choosing a video hosting solution to purchase and integrate with AFCP, why would they choose EVS over other vendors, when EVS can quit working for their use-case at any time? When a vendor offers some service which does not use OAuth, they typically include some sort of API guarantee. Meaning their contract or terms of service include API usage specifically for the service package being purchased, and the package itself guarantees access to APIs. OAuth-based services, on the other hand, only guarantee access to the service directly, and tell prospective clients to review their OAuth APIs and apply for a completely separate set of developer access accounts if they wish to have API access. Due to this, the intelligent and discerning buyer will choose the non-OAuth-based service over an OAuth-based one every single time.

A malicious user can also make use of the security issue above - *Masquerading as an OAuth-using service*, to acquire application specific credentials to impersonate it and do something which violates the terms of service for the APIs in question. This way attackers can ensure your application gets its plug pulled. In the case of our example, a competing camcorder vendor need only to obtain one copy of the camcorder to EVS application, extract the *client_id* and *client_secret* from the software, and then they can ensure that their competition fails.

Incorrect accounting

On top of a pull the plug architecture, OAuth-based services *as popularly implemented* will apply limits to how and when a particular application can make use of the service in question. For example, EVS might enforce that any specific application which uses EVS can only upload 100 videos in a 24-hour period. They may even charge the application vendor automatically for usage above that amount. As a result, if a camcorder bundles software to upload videos to EVS, their entire client-base can only upload 100 videos in a 24-hour period, even though each client has their own account with EVS. If one camcorder owner is particularly active

one day and uploads a lot of videos to EVS, all other camcorder owners are now either locked out till the next day or will cause the camcorder vendor to be subject to additional fees. Worse, in the latter case, since the application performing the uploads is stand-alone without any intrinsic communication with any of the other upload applications, there is no way for the camcorder vendor to ensure that their limit is adhered to without significant overhead and a centralized tracking framework. Finally, many OAuth-based systems which do not change for overuse will simply pull the plug on third-party services which they see repeatedly going over their limits, holding the third-party service responsible for their users' actions instead of terminating or charging the overactive users directly. This entire architecture is a scalability nightmare for third-party service providers, as every camcorder the vendor sells now brings them closer to the point where EVS might shut them down and destroy their business.

Most non-OAuth-based services usually have normal accounting in place where limits of the service are applied to each individual user or organization of users (depending on whether the service is for personal-use or enterprise-level, respectively). OAuth *as popularly implemented*, however, groups together completely unrelated users and organizations, and makes all of them subject to the whims of one.

A "simple" workaround for incorrect accounting is to require each user or organization who wants to use some application with EVS to go apply for their own developer account. This, though, has the unfortunate consequence of forcing every camcorder buyer to figure out for themselves how to obtain a developer account, a process which is often far from simple. Then the user will have to figure out how to set up the camcorder software to use that account, increasing the possibility of human error.

I took the above described approach with an application I wrote about a year ago which integrates with a Google service, and every first-time user of it has to log into the Google Developer Console and perform 15-20 steps before they can use the application. Besides being annoying, I found I couldn't even give clear guidelines on how to perform these steps, as Google has completely restructured their developer console and renamed things half a dozen times in the past year. Even the first time I

wrote a manual for navigating the console and selecting the right options, the manual became obsolete the very day after I published it. I can't point to any documentation on Google's side either, because they have no A to Z documentation specifically for obtaining the specific kind of OAuth credentials necessary for the application to work with the service in question.

The sane approach as used by non-OAuth-based services would be to allow every user or organization account manager (again, depending on whether the service is enterprise-level or not) to have a quick and easy to use control panel which can generate API access tokens for any service which is part of the account, with whatever permissions are needed. However OAuth *as popularly implemented* never seems to do this. Some OAuth-based services I've seen even require manual approval every time a user applies for a developer account. All this means using OAuth *as popularly implemented* creates considerable extra burden on the support staff for EVS, the camcorder vendor, and all of their users.

URI lock-in / Application incompatibility

OAuth *as popularly implemented* requires that a URI be provided up front to EVS when registering for a developer account or application credentials to be used as part of the authentication process in the new application. Remember, this URI will be used **every time** a user in the third-party service authenticates with the OAuth-based service. After the user logs in, the EVS special login pages will direct the user's browser to the specified URI with the tokens required to access the APIs. Some OAuth-based services will only allow for a single URI up front, while some other will allow several URIs to be entered. In the latter case, when the third-party service initially directs the user's browser to EVS's special OAuth login page, it can also specify which of previously enumerated return URIs to use.

If only one URI is allowed, or the amount of allowed URIs is not enough, often developers will find themselves being forced into opening themselves up to attacks discussed above under *Cross-site request forgeries*. But this is only the beginning of the issues this requirement causes.

OAuth *as popularly implemented* also tends to think that every use of it is by a web application built using the software as a service (SaaS) model. If some product, say

AFCP, is installed on-premise and deployed on each individual client's personal domain, what URIs do the vendors of AFCP provide when registering AFCP for EVS? This is even more ridiculous when the application using EVS is a desktop application like the camcorder software. Such software has no associated website at all! How are these applications supposed to work with these OAuth-based APIs?

It should be noted that getting a plain HTTP client, such as cURL, is readily available in every major programming language, and can be used in any kind of application. Web browsers with modern features cannot be used in non-graphical clients, such as command-line/terminal applications, and are only available in some programming languages. This means that OAuth-based APIs are severely limited in what kinds of applications they can be used with, making many desired use-cases impossible. It should be noted that many of the popular OAuth based services also require JavaScript on either their special OAuth login pages, or as part of the subsequent redirect process, further complicating simple usage in an application.

Even for SaaS products, if it's deployed on regionally diverse domains (mysaas.us, mysaas.co.uk, etc...) and EVS does not allow for entering enough URIs, then what? What about situations where every client gets their own sub-directory or sub-domain when using the service, how does one pre-register these or deal with the case when the amount of URIs required exceeds the amount EVS allows?

Even if EVS is a service which allows post-editing of the URI white-list, and allows for a nearly unlimited amount of URIs to be entered, how does the list get updated exactly? I have not seen a single OAuth-based service which provides any meta-APIs for modifying the URI white-list. In today's day and age, most SaaS deployments are or should be automated. When a client registers for a service, credit cards can be automatically processed, servers automatically deployed, sub-domains automatically added to DNS, initial credentials automatically e-mailed to the administrator, and so on. But how do the new URIs get added for any OAuth-based services used by the just deployed SaaS instance? Manually entry is utterly unacceptable and completely unscalable.

The workaround most suggest is to have some website set up which handles redirection for all clients and users. However, besides the aforementioned security

issues when not done carefully, this can get expensive fast. In order to have all requests processed by a fixed location, enough load balanced infrastructure needs to be set up to handle the load, even though it could otherwise have been handled by the same server or servers which handle the rest of the service for the given client. This also binds on-premise and desktop applications to this extra website, and will be unavailable if the extra website goes down for some reason. This has now become an annoying and costly dependency solely to manage a misfeature.

Another workaround suggested for desktop applications that don't want to be bound to a website is to build in an HTTP server into the application, and set the return URI to *http://localhost/*. It's bad enough that a web browser - often a full featured one with JavaScript support - needs to be built into the desktop application just to handle the login process, now they recommend a web server should be built-in too! I should point out, though, that whitelisting *http://localhost/* is not enough, as that URL implies the server is running on port 80. Not every application can grab that port, nor is the port always available, as something else may be using it. To make a robust application that can handle every eventuality as best as possible, one also needs to whitelist *http://localhost:1/*, *http://localhost:2/*, ..., *http://localhost:65534/*, *http://localhost:65535/*. This probably sounds ridiculous to you, and that's because it is. This also doesn't even account for the fact that some security software may get cranky or scare users if it finds an application is launching a server.

Being faced with this and some of the previous issues and finding only poor solutions offered, I came up with a completely different solution which I use with some OAuth-based services. I created a small web application that I can throw up on any server which asks in a web-based form to be populated with a *client_id*, *client_secret*, and anything else the OAuth-based service requires from what it thinks is an application. After submitting the details, the application redirects the user to the special OAuth login page, and after redirected back, the application displays whatever tokens it just got. The user can then manually paste these tokens into the real application they want to use which does not need to be tied down to any site nor requires browsers, servers, and graphical interfaces be used by the real application.

My above solution first requires users register for their own application or developer account, whitelist whatever the URI is to the small web application, paste their data into the small web application, log in, get the tokens, and paste those into the real application. It works well, circumvents any needs for a web browser and *redirect_uri*, and it's only a mere 20-25 steps for every user to perform. This would be much simpler for the users if the OAuth-based service itself showed them the courtesy of just supplying a page in their account where they can generate tokens with whatever permissions they wanted. Instead, these services would rather force third party developers to provide such finery, and make their users lose their minds.

Open-source unfriendly

A minor point to the above, OAuth is also unfriendly to open source applications, because every user making use of it is required to register their own developer account in addition to whatever user account they already have with a service. It would be unsafe to include credentials in the source or within binary packages which can be stolen, see *Masquerading as an OAuth-using service* above. This is on top of all the previous remarks about not knowing what URI to use, incorrect accounting, and all the other limitations.

Low availability

Another major flaw with OAuth *as popularly implemented* is its low availability (which is the opposite of high availability). OAuth-based systems differ as to the exact details, but they usually expire the tokens they provide that are required to be used with the actual API calls. Some of them allow a way to refresh them, but usually only for a limited amount of time. These systems require that the user of an account be present to re-authenticate with the service like clockwork. Besides being annoying, requiring re-authentication in middle of an operation and being able to continue the process after doing so may cause the design of the platform to fail in ways described above under the issue *Cross-site request forgeries*.

Now in the case of the camcorder software, where the user was using the provided software to upload to EVS for a while, what happens when the tokens expire? Say the user queued up a bunch of videos and then went camping for the weekend.

They come home to find the first few videos uploaded, and then the rest failed because they were not around to re-authenticate. This user will be extremely annoyed, especially if they promised someone the videos would be up by a certain time before they left.

When we consider our usage scenario with AFCP, this requirement to re-authenticate gets even worse. Normally when an administrator assigns someone new to a department, AFCP runs through the list of all integrated users in that department, and adds permission for this new user to access the EVS videos of the other users in the same department. However, if one of those previously existing users needs to re-authenticate, AFCP cannot automatically assign permissions for accessing videos to the newly added user. If the videos of the particular user who needs to re-authenticate is crucial for training a new employee, and the user is on some extended leave (vacation, maternity, etc...), the new employee cannot even be trained.

This expiratory handicap can make sense in various usage scenarios, but completely destroys what EVS integration with AFCP is trying to accomplish. If your usage scenario is AFCP, you really need to look for a service to use which does not use expiring tokens, which OAuth *as popularly implemented* unfortunately does. However, I did come up with a workaround which I use with some of my applications.

If a platform needs to re-authenticate on behalf of a user when they're not around, one would need their credentials to do so. I've exploited the flaw described above - *Stealing credentials / gaining elevated Access*, in which I prompt users for their credentials when they log in, store them (encrypted of course), and reuse their credentials automatically when tokens expire. This technique works as long as the user does not change their password and additional authentication factors are not in play.

I created a SaaS application which exploits this flaw over two years ago. Several thousands of users have since willingly entered their credentials into the application, and not a single one of them has ever complained or noticed anything suspicious. Better yet, the vendor behind the OAuth-based service I built upon heard

about my application from some of their users, and was pleased as to how they gained access to a larger user base. They contacted my employer and asked if we could provide them with some accounts on our platform for their sales department to show off to prospective clients, as well as a few other key personnel. We did so, and have since captured the credentials of several of their high ranking employees, including an executive responsible for managing their OAuth-based service. Not one of them noticed anything wrong with what we were doing, which goes to show that this is a viable workaround - however unorthodox.

While the above technique can turn a useless OAuth service into something profitable, it'd be better if it was achievable without needing to undermine security, defeat the few benefits of OAuth, and rely on exploitable flaws. I'll discuss below what really should be done instead of trying to always limit usage to the point that a service becomes unusable.

Not enterprise-ready

All the above usability issue sections show that OAuth *as popularly implemented* is not ready for the enterprise market. Contracts with OAuth based service providers do not actually cover the services you want, accounting is incorrect for organizations making them subject to unrelated third parties, there's extra setup annoyance, it’s not scalable, and the services repeatedly fail to work without manual user intervention from specific users. All this on top of not necessarily being able to conform with an organization’s security requirements.

However a real problem with OAuth-based services is that their APIs always seem to be geared towards only managing things for a single user as themselves. When someone personally gets an account for a service, they obviously do not want others to be able to take control of their account. But when an organization at the enterprise level purchases many accounts for their employees company-wide, they expect to have full control over all accounts, there is no room for individuals. In enterprise scenarios, no AFCP user should even be able to opt-out of the EVS usage, as the higher-ups want all resources within a department to be accessible to the entire department. Therefore, in order to be feasible for an enterprise, whoever or whatever is in charge of administrating all these accounts needs to be able to

perform any action on behalf of any user. If this requirement is not met, the enterprise company will need to find a different service which does make this possible.

I know of several OAuth-based services which in their front-ends for non-API usage allows for companies to set up authentication to their services using SAML. With SAML, any login request signed by a particular private key is able to gain entry to a platform as the user who is identified in the signed request. Whoever is in charge of managing the SAML identity provider and its private keys, usually the IT staff at the companies, can masquerade as any user they please and perform needed activities on their behalf.

What companies really want when they purchase a package with a service is to use an automated system to perform needed activities as required on behalf of their personnel. The APIs of the OAuth-based services, however, don't provide a way for any user - even at the administrator level - to perform activities on behalf of other users, nor do they provide a way to acquire tokens representing those lower-level users to use with the APIs on their behalf. An administrator cannot assign permissions for any videos in EVS except what they themselves have uploaded. When I asked developers of these services why not, they responded that providing such a feature would weaken the security of their platform. Yet these services which support SAML **already** have their security “weakened” in this way, though I fail to see how allowing administrators to effectively administrate their users, a situation which is usually considered desirable, can be called a “weakness”.

All in all, we see that the typical service with APIs built upon the foundation of OAuth *as popularly implemented* is designed and maintained with myopia regarding what enterprise customers actually want to achieve with a system. These APIs generally don't allow for anything to be done which couldn't already be done the same way with their existing user interface. However if you can't build anything that wasn't directly achievable with the provided interface, why bother offering APIs at all? Providing APIs is not so that the exact same UI with the same limited feature-set can be recreated over and over in other contexts, since today with SSO technology to access a website (such as SAML), there's no real point in that. The point in providing APIs is so that something new can be created which can offer added value to a service. If it's not possible to do more with the APIs than what is

already built-in, then the APIs are of little value and no self-respecting enterprise is going to take the product seriously.

Don't get me wrong, with limited APIs in place, there are still those making third party applications. However these applications are generally just embedding a subset of a service's existing features into a different context with no real added value. If a third-party application is actually providing something with added value with these crippled APIs, they end up also requiring users to perform dozens of steps and tons of copying and pasting to accomplish even the simplest of tasks. In such a situation, odds are a discerning buyer is not going to consider buying that product for all personnel, especially when there are alternatives available which aren't such a pain to use. A good rule of thumb, bad integrations from otherwise competent developers are the result of bad APIs. Therefore, if all of the so called "integrations" being created for a service either lack added value or are extremely clunky, it's probably an indication that the APIs are lacking what enterprises need. If your product lacks what enterprises need, you can expect that most enterprises will not be purchasing your product.

It should be noted that the only major services which have thriving OAuth-based ecosystems are platforms designed specifically for individual user use, and are not geared for enterprise use-cases. Facebook and Twitter (both of whom were in part responsible for the formation and adoption of OAuth), have a huge repertoire of apps making use of them. Google is another interesting example. Several applications, platforms, and services exist which make use of Google's OAuth APIs, but all of these are noticeably geared towards individual Google accounts, despite the fact that Google has a large business offering as well. In contrast, Google Apps for Business services generally offer alternative systems and protocols which are more appropriate for the enterprise market than OAuth, as they recognize their OAuth-based APIs to be incapable of providing the needed functionality.

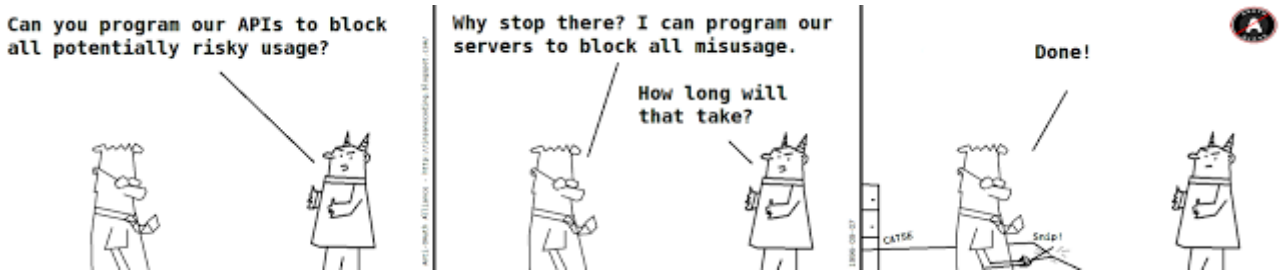
Section conclusion

While the security of OAuth *as popularly implemented* is not all that it's considered to be, the worst part about it is how crippling it is to actually work with. Many of the things OAuth sets out to accomplish are actually detrimental, especially in

enterprise settings, and at best, you'll need to find ridiculous workarounds to even get any use at all out of a service whose APIs are built upon OAuth.

The following is a mostly accurate depiction from one of my colleagues as to where the world with OAuth is heading:

> Allow me to present an alternative to OAuth which provides the same level of functionality and security but is far simpler to implement. We call it *NoAuth*. The protocol is as follows: No one may ever authenticate, period. Simple, effective, and we estimate that it will take developers a mere fraction of the implementation time of OAuth, cutting your costs significantly. *"NoAuth, because the future deserves absolute impenetrability."*



A key misdesign of OAuth *as popularly implemented* is thinking that all external services should be mistrusted equally. Let administrators define what should and shouldn't be done with a service, and what capabilities should be made available with various API tokens. Don't make this decision for them and limit what can be done with a service in its entirety just because there can be some rogue applications out there.

For applications I provide which make use of API services, I frequently get clients who ask me what exactly is my system doing that it needs certain permissions, and that's okay. Administrators are capable of deciding what they want or don't want to allow and what the ramifications may be.

My employer is commonly approached by third parties asking us to integrate some service with another service, telling us how much revenue we'll pull from sales if we partner with them. Every time we find out one of these services is using OAuth we groan, because almost always, it means the project is infeasible. Yet the third

party thinks there must be some magical way to do it because we've done some amazing things with other third parties which are not using OAuth. When we ask them to improve their APIs or drop this ridiculousness with OAuth, they claim they can't because *this is how it has to be done*.

More often than not, we find whatever is desired really is not feasible, and the workarounds are either not possible or too ridiculous, time-consuming, and annoying to the end-user for the case at hand. These third parties beg us to then *just do something with it*, as if that were a sane business practice. We're not going to create some product for which there's no business case and doesn't look like it'll break even, let alone make a profit.

I've seen a number of these projects come and go over the years. Usually the third party can't find anyone dumb enough to waste their time and effort on working with them, although on occasion they do. The outcome is always something which doesn't sell well, and the third party throws in the towel a couple of months later.

Even large enterprise organizations are coming up with some new product every year that they then try to force on the market, but time after time, these never seem to go anywhere. Despite the fact that the product has no real use, and other businesses can find no way to make it useful, they blame everything but a crucial issue at the heart of the matter. Instead of fixing the actual flaws preventing integration, the lifeblood of enterprise adoption and retention, they think it must something else, discontinue the product, and try releasing a different but similarly crippled product the next year.

Alternatives to OAuth *as popularly implemented*

What do proper OAuth-based designs look like?

Now that we've seen that OAuth *as popularly implemented* is utterly broken, when does OAuth actually work?

OAuth systems that adhere mostly to the earlier OAuth specifications and concepts are generally more secure and less broken than those based on later specifications. This is not to say that all OAuth 1.0 implementations are secure, but they're usually less problematic. These systems will normally follow one of two distinct approaches:

- By providing users the ability to directly generate access tokens in their account which they can dole out to software they wish to integrate, a lot of the silly requirements and insecure transfers and redirects are bypassed.
- By using a system where the *client_secret* does not function as a piece of data which is passed verbatim like a common password, but is instead itself a cryptographic shared secret used to generate authentication codes, the remainder of the token system, silly requirements, and insecure transfers and redirects are bypassed. In these systems, users generally do not even have their own credentials and only use SSO mechanisms.

Precious few OAuth-based systems are designed like this though, and these systems generally look nothing like OAuth as used everywhere else. Since these systems stick closer to the OAuth 1.0 specifications, which is officially deprecated, systems using this approach eventually get *"updated"* to be restructured with all sorts of OAuth 2.0 concepts and additions, thereby ruining their security and usability. This is a reason

I'm hesitant to condone anything OAuth based, because even if using an earlier, more operable style of OAuth, some manager is going to have the bright idea that the system will need *"improving"* and will break it. Better to use something else altogether than begging for problems.

Other options

Looking for other options, people often want to know what other *frameworks* are out there. However, one does not need some framework to achieve a well understood and secure design. As is, every service has their own take on what OAuth looks like, so there really is no exact approach on how authorization works anyway. Hunting for a framework is often over-complexifying what can be done simply. The only really difficult component that requires a good specification is how to sign variables to prevent tampering with key parameters used, and most OAuth-based implementations do nothing of the sort anyway.

The largest provider of web services on the market today is Amazon. They are the premier provider for enterprises the world over, and utterly dwarf everyone else with a whopping greater than 30% combined market share. Amazon's approach is to provide all their accounts and account administrators with access to a control panel where they can generate application credentials. These credentials can be specified as to what Amazon services they can work with, what actions they can perform in those services, and what permissions they have to work on. These credentials can be revoked by the account holder at any time if necessary.

Amazon's authentication and authorization techniques for their APIs do not require any redirection methods which are inherently limiting and potentially unsafe. Amazon's protocol never sends along any credentials directly, rather they're used for signing data, and can ensure parameters remain tamper-proof if there's any need to send them through a browser.

Amazon's design has proper usage accounting, includes API usage as part of every account, and all API authentication and authorization is initiated from the Amazon side with the creation of application credentials from **their** control panel. These credentials are then used directly in the API process without any sort of additional token exchange system. This design achieves all the real security objectives OAuth

as popularly implemented achieves, and it also avoids all the security and usability issues enumerated above.

One downside I have to say about Amazon is that their permission system is somewhat confusing and not all that user friendly. This happens to be true though of most control panels, and in any case, is a matter of user interface design and is not a mark against the authorization process itself. Also, Amazon's control panel can be navigated fairly quickly and even be used with their APIs, unlike, say, Google's, which, as far as I know, has no meta-APIs, and requires at least a dozen steps to do anything with it.

Amazon's authentication and authorization method is also copied by several other service providers on the market. Google themselves even allow for it in certain of their enterprise products. Google themselves also acknowledge that a pure OAuth design is poorly suited towards enterprise services, and for their enterprise services, they recommend the use of JSON Web Tokens.

JWT is a specification for allowing SSO or API usage between services. In many ways JWT is like SAML, although unlike SAML which is confusing, built upon XML Security (which is anything but secure), and not geared for API use, JWT achieves the primary SAML objectives in a simple and easy to use manner without all the headaches. If you have an HMAC implementation and know how to structure and parse JSON, then you can use JWT. For those of you who want something off the shelf, there are plenty of JWT libraries already available.

Google's use of JWT is more advanced than typical though, and instead of HMAC, they require the use of RSA digital signatures which is a more advanced, and less popular concept than HMAC in this area. Google's control panel allows account administrators to generate a new key-pair for their enterprise services, and download the private key to use for signing API log-ins. While this is more secure than HMAC, Google really over-complexifies the whole process, not to mention that they completely redesign their control panel frequently, making it confusing to repeatedly use. I recommend looking instead at what others are doing with JWT if you need examples.

Another technique being used is that of services allowing definition of what kind of permissions third parties need in some kind of XML or JSON file that they can post on a web site. Users can then visit a page in their account where they can paste the URL to this file (or paste the contents of it), and the service will display a list of what kind of permissions the external service or application wants to use and any descriptive information it may contain. If users want to approve this, they can generate credentials which they can then paste into the third party application or service they are using. Users can later revoke the credentials if they want to disable the third party product. This is also a very secure design which doesn't have any ridiculous burdens placed on developers, includes API services for all accounts, provides permissions, and initiates the flow with the service itself, not the third party.

All you really need from the service-side in order to manage authorization is some panel which allows users with the appropriate role (administrator or account owner) to generate API usage credentials which each have permissions and a possible expiration (if desired) assigned to them. These credentials can then be used over any secure authentication system you choose, such as something simple like HTTP Basic Authentication over HTTPS, which is available with practically every HTTP library available, HTTP Digest Authentication, which is more secure and supported by most high-grade libraries, or something else based on authentication codes utilizing a cryptographic technology which does not require any credentials ever being passed over the network, such as HMAC, RSA, or Elliptic Curves. HMAC in particular is already in use by nearly everyone implementing some form of authorization or authentication (including Amazon and even some OAuth implementations).

These various well established techniques decrease the burden of needing to study the effects of one framework’s interaction on others, such as those for anti-CSRF, in order to create a secure platform, and can generally be implemented in a modular manner which can simply be dropped into an existing architecture. They avoid the possibility for stealing the user's or the application's credentials. They also don't require any constant use of elaborate CSPRNGs. These kinds of systems existed long before OAuth and are popular today too. OAuth may have better security than some poorly designed systems which require users’ personal credentials and have other

weaknesses, but OAuth is not a substitute for the real designs that already existed. The problems that OAuth claims to solve do not actually exist in the existing well-designed systems, and OAuth *as popularly implemented* actually introduces many of the problems it claims to solve, along with several others which never existed in the first place. Despite the hype, OAuth does not inherently provide amazing security, and due to the numerous drawbacks and implementation difficulties, the other well-thought-out options are vastly better.

If you're going to be designing a service and provide API access, please, really think about what you're trying to achieve, and don't just copy what you see others do or buy into ridiculous hype. If you must copy someone, try to copy Amazon (the best), Rackspace, IBM SoftLayer, Linode, VULTR, Zoho, Zoom, or others who seem to currently have some inkling on how to structure a straight-forward and sound authentication system for APIs.

Written April 2016 by Insane Coder.

Comments: http://insanecoding.blogspot.com/2016/04/oauth-why-it-doesnt-work-and-how-to-zero-day-attack.html

Website: http://no-oauth.insanecoding.org/