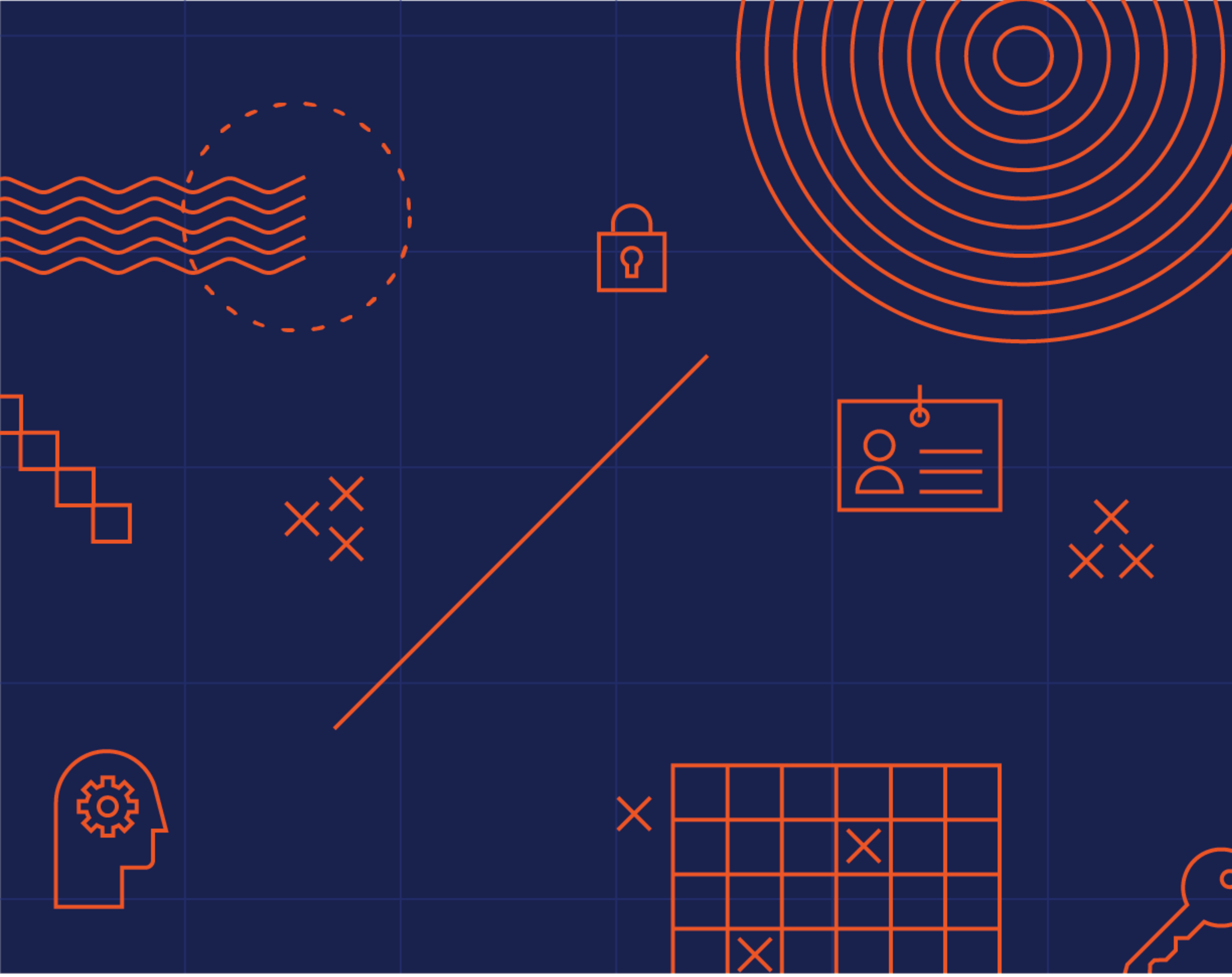

AUTHENTICATION SURVIVAL GUIDE

By João Angelo

Auth0

Authentication Survival Guide

João Angelo, Auth0 Inc.

Version 1.0, 2017

Content

Special Thanks **4**

Introduction **5**

Concepts **7**

OAuth 2.0 **8**

OpenID Connect **12**

Enterprise protocols **14**

SAML 15

WS-Federation 16

Brief look into the past (OpenID and OAuth 1.0a) **18**

Token overload **19**

Formats 20

Types 22

Storage 27

But I love my cookies 31

Troubleshooting **33**

It's just HTTPS 33

Tools 36

Pitfalls **38**

Glossary **40**

Special Thanks

In no special order: **Kunal Batra** (for starting the discussion that lead me into writing this work), **Amaan Cheval** (for reviewing it), **Bárbara Mercedes Muñoz Cruzado** (for making everything look nice), **Alejo Fernández** and **Víctor Fernández** (for doing the frontend and backend work to distribute this content).

Introduction

Authentication is a complex topic, if you think otherwise you will probably get burnt down the road. There's a myriad of technologies, protocols, concepts and do's and don'ts. On top of that it's difficult to keep track of all the evolving best practices and outdated/overloaded terminology; the last one is especially troubling if you're just getting started on the subject and start to see different things being called by the same name or the other way around.

This guide aims to be an accessible starting point that will provide you enough food for thought to allow you to hold yourself together in a discussion involving authentication. It does this by providing a concise review of all the key protocols, formats, concepts and terminology that you'll likely come across when doing authentication related work.

It's particularly useful for developers that find themselves in the front-line of an implementation effort to create or redesign authentication systems and feel a bit overwhelmed with all the possibilities out there.

On the other hand, this doesn't try to be a definitive reference on all things authentication. There will be subjects that won't get discussed and others that will only be briefly mentioned and not fully explained. In summary this will give you a jump start on most common scenarios and things to watch out for, but then it's on you to continue the journey. Use this to find out what you need to know more about and what you can safely ignore.

Concepts

Authentication

Process by which an application verifies the identity of who's interacting with it. The outcome will generally be a set of attributes that are proven to be associated to the user accessing the application.

Authorization

Process by which an application determines if the user accessing it has the necessary permissions to access a resource or perform a given operation. Generally dependent of a preceding authentication step that ascertains the identity of the user.

Identity provider

An entity that creates, maintains and manages identity information. Provides an authoritative source of user identities and associated attributes.

Digital Signature

Technique used to ensure that a given message was created by a well known entity and that the message was not altered in any way since it's creation.

OAuth 2.0

OAuth 2.0 (OAuth2) is an authorization framework that enables an application to obtain access to HTTP based resources that are usually within the domain of an unrelated application and that also may be associated with a given user within that domain. It solves the problem of delegated access without requiring credentials sharing. Think your custom application accessing Facebook resources on behalf of a given Facebook user and it does this without ever having access to the user Facebook password.

The major players when it comes to OAuth2 are the following:

- **Authorization server:** the authority that is able to grant access to resources.
- **Client application:** the application wanting access to resources.
- **Resource owner:** usually an end user; the authorization server confirms with them if it's okay to give a client application access to their resources.
- **Resource server:** the application providing the actual resources.

The important part here is that this was not something originally aimed at user authentication. However, given that as part of granting access to user resources one of the necessary steps is to ensure who the user is, this framework/protocol ended up being used as a way to also achieve user authentication.

One way this could be used for authentication would be the following, again using Facebook as an example:

1. Application X is accessed by an anonymous user
2. Application X asks to access Facebook resources on behalf of the user
3. User is navigated to Facebook where they need to authenticate themselves
4. Upon successful authentication, Facebook asks the user if they want to grant Application X access to their resources
5. User grant the access and Facebook navigates the user back to Application X
6. Application X receives something (yes, it's a token) that can be used to prove that the user has indeed an account with Facebook with a specific set of attributes (claims)

As you can see, Application X only asked for access to resources, the authentication part was just a necessary intermediate step. However, the token it received would either contain data that directly identified who the user is or could be used to get to this data by performing additional queries to Facebook. Many applications use this flow purely for authentication purposes, that is, application X would not even be interested in accessing any user resource on Facebook and was just interested in uniquely identifying who the user was. Note that at least one request would have to be made in order to prove the token is valid according to Facebook terms.

Next time, an anonymous user accesses application X, the set of steps would be repeated and if an already known Facebook user identifier was returned to the application then it would know it was being accessed by a recurring user.

Although it can and it still is used for authentication is important that you are aware that this isn't for what it was designed to do and the most common applicable scenario OAuth2 was designed to solve was granting third-party applications access to HTTP – REST – based API's.

If you have or are planning to have an HTTP API where the protected resources are tied to individual users and you intend to make their data accessible to third-party applications, then OAuth2 is most likely the right tool for the job; at least is the one with more adoption. They say nobody ever got fired for choosing IBM and for this particular scenario going with OAuth2 should also allow you to feel pretty safe.

Do have in mind though it's not the only tool for the job and for simpler scenarios almost for sure there will be simpler options. For example, going with API keys will be exponentially simpler and meet the goal, if your only requirement is to know who's calling your API so you can know who to address the bill. Always research the possible options and weigh their pros and cons against your specific scenario.

Another thing you should be aware of OAuth2 is that the specification describes itself as an authorization framework, it doesn't call itself a protocol and it does this intentionally. Although the specification covers the minimum set of requirements for each implementation it does so while leaving many aspects implementation specific.

This means that if you are approaching OAuth2 from a client application developer you need to do two things:

- Check the specification (RFC 6749)[1] to know about the baseline that you can expect to find when integrating with a given provider.
- Read the specific provider documentation in order to know what they did beyond what's mandated in the specification.

If you're approaching it from the perspective of creating your own authorization server implementation, my best advice would be to try to use something that already exists. If you can't go that route, I won't even try to come with a definitive list of resources, but at least also go through OAuth 2.0 Threat Model and Security Considerations (RFC 6819) [2] - it may help change your mind and go with my initial advice.

---

[1] **https://tools.ietf.org/html/rfc6749**

[2] **https://tools.ietf.org/html/rfc6819**

OpenID Connect

OAuth2 proved so popular that people started using it for scenarios outside of its original intent - user authentication and identification. However, it became apparent that there were too many decisions left to the interpretation of the reader when this framework was applied solely to solve the authentication problems.

This lead to implementations approaching the same underlying issue with different solutions which then had to be reflected in specialized client libraries for authenticating with each specific implementation. Additionally, the lack of definitive guidelines also meant that it was more difficult for client application developers to ensure proper security.

OpenID Connect can be described as an extension to OAuth2 that provides clear guidance on how to achieve a functional and secure authentication system.

By focusing on solving the specific issue of user authentication the protocol formalized the gray areas of OAuth2 that were causing disparate implementations. For example, where OAuth2 does not impose any format or specific requirements on the tokens used, the OpenID Connect states that an ID token uses the JWT format and it will always have a set of standard claims available.

Additionally, the protocol also formalized - by means of the UserInfo endpoint - the way that an identity provider can make available additional information about the identity of a user. This was somethings that many providers already supported, however, by putting it on paper implementation now have a baseline that can be used to achieve interoperability.

Being the new kid on the block when compared to the other authentication related protocols it's somewhat comprehensible that support for OpenID Connect both at a library and identity provider level is still evolving, particularly when you venture out of the core specification and move to a related specification that is optional to implement.

Nonetheless, this is the protocol you need to learn and use to solve the authentication problem in your applications.

Enterprise protocols

Before the consumer web was faced with the proliferation of independent user credentials for each application/service the business world had already gone through that experience. It was not uncommon for each business application to implement their own notion of a user store and do authentication in a custom way.

This was an issue for IT departments, but also for the users themselves because they had to manage multiple set of credentials and their workflows would be constantly disrupted whenever switching from one application to another was required. The first attempts to solve these problems worked satisfactory in homogeneous environments like the ones we would find within corporate networks; think Windows Active Directory and Windows Integrated Authentication. This and similar techniques achieved what's known as single sign-on (SSO) - a user would perform authentication once in a central server and then all the application would rely on the identity provided by this centralized service.

However as soon as an application was outside the restricted corporate network boundary we were back to square one so in the early 2000s new protocols started to emerge with the goal of addressing the need for SSO solutions across security boundaries. We'll have a look at two of them, SAML and WS-Federation, which are still in wide use today.

SAML

The Security Assertion Markup Language (SAML) covers a lot of ground in terms of application security scenarios, however, the most common usage of this protocol is to achieve web browser-based SSO across different domains.

This standard defines a set of structured messages and how these messages can be used by an application to perform authentication requests. The outcome of the authentication request will be a token that can be sent across security domains while still allowing the requesting application to validate its authenticity and thus allowing it to identify a user.

In the SAML world, both the application messages and the tokens themselves are defined in terms of XML which should be no surprise based on the time this protocol started to appear. This translates into a significant amount of overhead on the binary representation of the messages and tokens required by the protocol which among other reasons make this protocol a less than ideal candidate for usage in modern applications.

If you do come across this protocol, you'll probably do so when integrating with already existing systems. In case this occurs be aware that there are two significant versions of the protocol still in active use - the 1.1 and 2.0 versions.

Another interesting point about SAML that might trip you over is that SAML can be used both to refer to the protocol, but also to the token format that goes with it. This separation is actually important because as we'll soon find out, the token format itself can be used independently of the underlying protocol.

WS-Federation

Another standard that tackled the web browser-based SSO was WS-Federation. This was part of a big family of specifications that covered a lot of ground in the application integration field; from server-to-server interactions to rich-client applications interacting with server-side web services. The group of protocols was collectively addressed as WS-* protocols. To be honest these were significantly complex and do not invoke any kind of good memories from my part because the richness of the specified features carried a significant price tag due to the inherent implementation complexity.

Fortunately, for the purpose of this guide, WS-Federation was probably the saner one in the family and in a similar fashion to SAML, defined the necessary web interactions required between two systems in order to achieve SSO. I believe it's also safe to say that it did so in a much simpler way than SAML and maybe for that reason is still a protocol with a substantial usage.

However, to the contrary of the SAML protocol, WS-Federation does not define its own token format and instead makes this implementation specific. The end result is that in most scenarios you end up using the SAML - token format - alongside the WS-Federation protocol.

Brief look into the past (OpenID and OAuth 1.0a)

Both OAuth2 and OpenID Connect share the fact that they were designed to replace existing protocols that, at least judging by their names, seem to be somewhat related. However, the relation is almost only of the names as in both cases the most recent editions are not backward compatible and can and, in my opinion, should be seen as completely different standards.

I won't go into comparisons or claim that OAuth2 and OpenID Connect are better versions of their respective counterparts, because that does not change the reality; OpenID, OpenID 2.0 and OAuth 1.0a are protocols that despite still being used have no active development, are in the process of being phased out by many providers and for better or for worse have been replaced.

It's highly unlikely that you'll need to interact or develop against the earlier iterations of these standards so my recommendation is that even though you should be aware that these exists, it's not worthwhile for you to waste time on these unless you need to integrate with systems that provide you no other options.

Token overload

Although the hype around tokens is relatively new, particularly with JWT, the use of tokens for authentication is not new at all.

If you think about it, the traditional cookie-based authentication systems used in Web applications for ages is a good example of the use of tokens as a way to authenticate the user. Most implementations rely on an opaque string, which at least would be randomly generated and maybe even digitally signed, that was then used on the server-side to lookup additional information about the authenticated user.

This session identifier is nothing more than a token and the overall cookie-based authentication system, that we know so well, would be better described as a token authentication system that uses by-reference tokens instead of self-contained tokens and relies on cookies as the mechanism to store and transmit the token from the client application to the server application - resource server.

Having said that, nowadays, when you mention or see someone referring to the use of a token it's highly likely that it implies a self-contained/by-value token. I personally like the by-value terminology because it makes a good contrast with the by-reference tokens and it's also terminology that is widely used within programming languages to describe the exact same kind of implied semantics, more specifically, the notion of value types and reference types.

Formats

If there is one token format you need to absolutely know about then that format is JWT (JSON Web Token). It's a self-contained token which means that information about the entity the token target is contained within the token value itself. This makes it an ideal choice for scenarios where you want to move authenticated session data from your server to the client application and achieve the so-called stateless authentication systems.

The JWT token format is also relatively simpler when pitted against popular predecessors like SAML, however, this simplicity is relative and you should not assume JWT is so simple that you might as well use your own logic to process them.

The specifications related to JWT allow you some flexibility on how you end up using the tokens, for example, there's more than one way to represent tokens and you can optionally choose to use signing and encrypting methods.

The most common representation of a JWT uses base64url encoding to represent in an HTTP friendly way it's building blocks like the header, payload, and signature. It's actually uncommon to see encrypted tokens being used in common implementations of authentication systems using JWT's, mostly because both OAuth2 and OpenID Connect mandate that TLS must be used when transmitting tokens on the wire and the actual contents of the tokens do not include sensitive information.

Before JWT, the SAML token format would be the one you most typically encounter when working with authentication systems aimed at solving authentication issues that crossed domain boundaries like, for example, achieving single sign on.

This token format also supports signing and encryption, however, you could claim that it does it with an increased implementation complexity when compared to JWT; hence the decline in its usage. The reason for this is mostly because while JWT is based on the JSON format, SAML relies on XML instead and this difference alone causes a significant increase in the overall complexity of any feature that the token format decides to support.

Additionally, SAML was born in a time that web services and SOAP were kings; a time where XML was everywhere. Nowadays, the focus is on making things simpler; HTTP over SOAP, JSON over XML and, to what authentication is concerned, JWT over SAML.

Types

The format of the token is just one small aspect of it, one important distinction to be made is the role of each token in your authentication system. Within the realms of OAuth2 and OpenID connect there are three types of token in common use:

- Access token
- Refresh token
- ID token

An access token is defined within the OAuth2 specification as credentials used by a client application to access a protected resource made available on a given resource server. Although the client application makes use of the token to perform this authorized request the access token itself should be considered opaque to the client; this purely means that a client application is granted the token after completing an authorization grant where the resource owner consents to delegate access to their resources and then treats it as a black-box.

It's an artifact that the client application knows it can use to perform authorized requests on a given resource server, but without any knowledge of the actual contents of the token. Access tokens will have an associated expiration date which is implementation specific, but it's usually fairly short ranging from minutes to hours. The reason for this short expiration is that since these access tokens are typically used as bearer tokens the short lifetime will reduce the impact of possible leaks.

There is no requirement in the format used for an access token; it can vary from a self-contained token like a JWT to randomly generated value that is used as a by-reference token. The most common approach it to go with self-contained tokens because the consumption of the tokens does not require a resource server to make an additional call to the authorization server in order to obtain the information associated with the token.

Having said that, there are still scenarios where by-reference tokens may provide better characteristics, for example, in cases where the authorization server and resource server roles are played by the same entity this type of token may prove more lightweight on the wire without the extra lookup to obtain the information causing significant overhead.

One final note on access tokens; ideally, they should not be used as means to authenticate users. The reason for this is that they are issued for use against an API and if you use them for authentication you may leave your application vulnerable, for example if an access token was issued for API X of vendor A and an application Y of vendor B decides to use the access token also for user authentication in the application itself, the application can't ensure the access token was issued in an authorization flow initiated by itself or if it was issued to a completely unrelated application under the control of an attacker which is now trying to impersonate a user using the access token.

Any token used for user authentication must have a way to ensure that the token was issued for authentication with a specific application so that the application can reject tokens with a different audience.

Refresh tokens are another type of token introduced with OAuth2 and they have a very specific purpose; they allow the client application to obtain additional access tokens when the original one expires. In contrary to what's the norm for access tokens, the lifetime of a refresh token is substantially greater in the range of years and in some cases they don't even expire.

This difference in lifetimes are a source of confusion for client application developers because if the application is issued a short-lived access token and a long-lived refresh token that can be used to obtain additional access tokens wouldn't it be simpler to just issue a long-lived access token?

Simpler, maybe; more secure, not really. Remember that an access token is issued by an authorization server and then used against a resource server. If this token is self-contained, the authorization server won't ever be called again. However, a refresh token is issued and consumed by authorization servers which means that token revocation policies for refresh tokens can be implemented by the authorization server because it knows for sure that client applications will need to call it again with the refresh token.

Additionally, the number of times a long-lived credential would need to be passed on the wire is also significantly reduced in the scenario you have the two different tokens. With only a long-lived access token each call to the resource server would include this long-lived credential increasing the impact of an eventual leak.

Another advantage of refresh tokens is that they can provide an immediate security boost to legacy applications that used to store user passwords in order to ensure that they could continue to function without the presence of the end-user. The application can now trade the password for a refresh token and store the token instead. It's true that both passwords and refresh tokens are long-lived credentials but storing a token that can only be used with a given authorization server versus storing a password that the user may reuse across a range of different services is a much better solution.

Even though they are better than storing passwords, client applications need to ensure proper storage of refresh tokens in order to mitigate against the possibility of leaks while the token is in storage. This is another point of confusion because not all client application types can ensure proper storage for long-lived credentials and as such the usage of refresh tokens must be avoided in those situations. The most common example of applications that fail to provide adequate means of storage is browser-based applications.

The last token type that we'll cover is the one most important when speaking of user authentication; the ID token. This token was introduced with the OpenID Connect specification and is, as the protocol, tailor-made to solve authentication problems.

In contrary to what happens with the two previously mentioned token types which can have an implementation-specific representation, the OpenID Connect protocol mandates that compliant implementations use JWT as the means to represent this token type. This makes sense because as we've seen before one of the requirements of any token used for user authentication is to provide means to allow the client application to know for sure that the token issued to its use.

Additionally, since information to uniquely identify a user is also required, the use of a token format that is self-contained is the right choice for a user authentication token.

Incorrect or insufficient validation of an ID token will allow an attacker to wreak havoc within your application and you may be surprised, but unfortunately it's not that uncommon to encounter situations where applications fail to properly validate tokens. In order to be sure you don't fall victim to this possibility, you must ensure that you strictly follow the rules for ID token validation included in section 3.1.3.7 of the OpenID Connect specification.

Storage

As mentioned before the proper storage of tokens is a must have requirement in token-based authentication systems - particularly on ones that use them as bearer tokens. This is also something that needs to be addressed depending on the role and type of application that requires token storage.

For native mobile client applications the requirement is that you must use a storage that ensures that other applications on the device cannot access the tokens. For example, in Android the SharedPreferences feature would accomplish this requirement.

The situation is a bit trickier with browser-based applications; you do have a few options to consider as possible storage, but all come with their specific shortcomings. Choosing the storage for a browser-based application it's a pick your poison kind of game where the best choice highly depends on your exact scenario. In order to help you choose, I'll go through some available options next while presenting some of the associated advantages and drawbacks of each.

Web Storage (local Storage or session Storage)

The pros:

- The browser will not automatically include anything from Web storage into HTTP requests making it not vulnerable to CSRF
- Can only be accessed by Javascript running in the exact same domain that created the data
- Allows to use the most semantically correct approach to pass token authentication credentials in HTTP (the Authorization header with a Bearer scheme)
- It's very easy to cherry pick the requests that should contain authentication

The cons:

- Cannot be accessed by Javascript running in a sub-domain of the one that created the data (a value written by example.com cannot be read by sub.example.com)
- Is vulnerable to XSS
- In order to perform authenticated requests you can only use browser/library API's that allow for you to customize the request (pass the token in the Authorization header)

HTTP-only cookie

The pros:

- It's not vulnerable to XSS
- The browser automatically includes the token in any request that meets the cookie specification (domain, path and lifetime)
- The cookie can be created at a top-level domain and used in requests performed by sub-domains

The cons:

- It's vulnerable to CSRF
- You need to be aware and always consider the possible usage of the cookies in sub-domains
- Cherry picking the requests that should include the cookie is doable but messier
- You may (still) hit some issues with small differences in how browsers deal with cookies
- If you're not careful you may implement a CSRF mitigation strategy that is vulnerable to XSS
- The server-side needs to validate a cookie for authentication instead of the more appropriate Authorization header

Javascript accessible cookie (ignored by server-side)

The pros:

- It's not vulnerable to CSRF (because it's ignored by the server)
- The cookie can be created at a top-level domain and used in requests performed by sub-domains
- Allows to use the most semantically correct approach to pass token authentication credentials in HTTP (the Authorization header with a Bearer scheme)
- It's easy to cherry pick the requests that should contain authentication

The cons:

- It's vulnerable to XSS
- If you're not careful with the path where you set the cookie then the cookie is included automatically by the browser in requests which will add unnecessary overhead
- In order to perform authenticated requests you can only use browser/library API's that allow for you to customize the request (pass the token in the Authorization header)

This may seem a weird option, but it does has the nice benefit that you can have storage available to a top-level domain and all sub-domains which is something Web storage won't give you. However, it's more complex to implement.

My recommendation for most common scenarios would be to go with the Web storage option, mostly because:

- If you create a Web application you need to deal with XSS; always, independently of where you store your tokens
- If you don't use cookie-based authentication CSRF should not even pop up on your radar so it's one less thing to worry about

None of the cookie-based options mention it, but use of HTTPS is mandatory of course, which would mean cookies should be created appropriately to take that in consideration.

A final word regarding storage, this is something not specific to client applications, if you implement or maintain an authorization server then you may also need to store - by-value - tokens generated by you so that you can later ascertain their validity. In these occasions, you'll be concerned in ensuring that tokens provided by a third-party match ones you previously created. You can accomplish this even if you don't store the tokens as plain text and as such it's highly recommended that you only store hashes of the generated tokens.

But I love my cookies

Although tokens, particularly self-contained formats like JWT are all the rage right now, cookies are not obsolete. This is not surprising from the perspective that cookies are just a mechanism for state management within HTTP and as we've seen they can be even used as client-side storage of tokens.

However, even from the perspective of cookies as an authentication mechanism, there are still valid use cases for them. For example, in a web application, you can establish a cookie-based session upon the initial interaction with the user where an ID token is presented as user credentials.

If you think cookies help simplify your use case, don't reject them purely because they aren't the new shiny thing available or because someone throws the stateless argument into the mix. Remember, you can also push session state to the client with cookies, just remember to do it right; signing and/or encrypt them. The big downside with cookies is that since they are automatically included in requests made by user agents you need to ensure proper mitigation for CSRF attacks.

Troubleshooting

In software development things go wrong, it's almost unavoidable giving the number of things outside of your application control. On top of that, there's also the human error coupled with the operating system stubbornness on doing exactly what the written code says to do instead of doing what we - developers - were thinking it should do when we wrote it. Whoever solves this one will be pretty rich…

One of the big benefits of modern authentication systems is that they are applied to solve authentication problems for application systems that rely heavily on HTTP communications. This is a godsend for troubleshooting because HTTP has been around for ages with an enormous tooling support and it's also easy on the eyes.

You could argue that the downside would be that you require a solid understanding of HTTP in order to be successful in implementing and debugging modern authentication schemes, which is absolutely true, but this is much better than other alternatives; I bet no one would prefer to troubleshoot WS-* over SOAP.

It's just HTTPS

As we'se seen most, if not all, of the interactions required to complete an authentication process will happen over HTTP and will also rely heavily on HTTP redirects so that a user is navigated across all the entities taking part in the authentication process. My recommendation would be for you to spend some time on learning the details around the following areas:

- HTTP request/response semantics
- HTTP redirects
- HTTP authentication (the Authorization header)
- HTTP state management (cookies)

With this knowledge on your hands and a couple of tools on your toolbelt you'll find it very easy to analyse and troubleshoot modern authentication flows. Let's look at a general authentication flow example so that you know what to expect:

- User accesses a protected resource on Application Y (this application delegates authentication to another external entity)
- Upon detecting an anonymous user request to a protected resource Application Y issues an HTTP redirect to the external identity provider authentication endpoint
- The identity provider also detects that the user is unauthenticated and prompts for credentials (this and the next step could be skipped if the user had previously authenticated and started a session on the identity provider; our famous SSO)
- The user provides the necessary credentials and the identity provider validates them

- The identity provider issues an HTTP redirect to Application Y with sufficient information that can be used to prove the user identity (probably a token digitally signed by the entity provider)
- Application Y processes the user identity information and establishes an authenticated session with the user in order to not require interaction with the identity provider for every subsequent request

This flow represents the most common set of steps used by more than one authentication protocol. There are of course variations, but in the general case debugging these flows consists of checking at each step the issued HTTP request and response and validate if the contained payload (query parameters, headers, cookies and request/response body) matches what you would expect to find according to the specific protocol requirements.

Another important point to make is that by relying on the HTTP Archive file format (HAR) you can easily troubleshoot the problem after the fact and/or in another machine. This is also ideal for hard to reproduce issues that may occur in a very specific environment. This format is supported by major browsers which can export HTTP network traces which you can then inspect using HAR viewers.

Be advised that capturing the HTTP requests associated with an authentication transaction will contain sensitive information, including user passwords so be sure to have this in mind if you ask your users to provide this type of troubleshooting information. The best thing to do is to be explicit and request them to at least ensure that no passwords are included; the HAR file is a JSON based text file so any editor can be used to remove this type of sensitive information.

Tools

Unless you're only targeting native platforms it isn't surprising to say that you're going to spend a lot of your time in a browser. The good news is that all major browser come with awesome developer tools that will prove indispensable when you want to inspect what's happening at the wire. One major benefit of working directly with browser tools is that you'll be dealing with plain old HTTP instead of encrypted HTTPS without any extra configuration needed. Additionally, since most authentication flows involve redirects the first thing you should do is to ensure your browser is configured to preserve the network trace upon navigation as most time the root cause of the issue is a few request down the line in relation to where your browser ultimately gets redirected to.

Another must-have tool when it comes to HTTP debugging is Fiddler, now Telerik Fiddler. This is a standalone tool which acts as a proxy and allows for the analysis of the HTTP traffic that passes through it. Depending on your browser of choice you may need to configure it in order to work with the Fiddler proxy. Additionally, by default you won't be able to see decrypted HTTPS traffic unless you explicitly enable that functionality.

Fiddler also supports importing HAR files so you capture traffic on the browser and then analyze it on Fiddler if you want to use some of the extensive features that the tool provides.

Another possibility for you to review HAR files that does not require installation of separate tools is to use the online viewer provided by Google:
https://toolbox.googleapps.com/apps/har_analyzer/.

Finally two other good additions for your toolbelt when it comes with working with the tokens themselves are the online tools:

- **http://samltool.io/** - Lets you decode SAML tokens and quickly review the most significant attributes
- **https://jwt.io/** - Lets you quickly decode and verify signatures for JWTs using HS256 and RS256 signature algorithms. In the former case you can even edit the payload.

Pitfalls

There is already a lot of awareness on this one and both the OAuth2 and the OpenID Connect specification extensively call your attention to this, but it's something so essential that it's never too much to repeat; the usage of these heavily based HTTP protocols imply that the communication between the involved parties happens through TLS (HTTPS). This is not optional as not doing this severely compromises the security of the system.

Delegating the authentication of your application users to an external system means that your application will not have to care about certain functionality like password resets, brute force detection and password policies. However, there are aspects that do get more complex.

One of the things where delegated authentication increases the complexity is the functionality to logout a user. Depending on the type of client application you may be tempted to do one of the following:

- For web applications that create an application specific session you remove this session, clear any applicable cookies and that's it; exactly the same thing as if the user authenticated with your application.
- For client application that talk with Web API's you delete the access token client-side and since the server is stateless in terms of session data you don't need to do anything else.

These approaches are incomplete because with delegated authentication your users actually authenticated at two independent systems, although from their perspective, that authentication was seamlessly so they tend to not think about it.

The implication is that if you only clear your local artifacts the user may still have an active session at the identity provider level which means trying to login again will not require inputting credentials again. This is important because if your user authenticated in Google just for the purposes of using your application and he did so on a public computer or multi-user workstation and then your application provides him a logout functionality that only clears the local session you're basically leaving their Google session active which is something many non-technical users will not understand.

Glossary

Access token

A string accepted as credentials to access protected resources. It imposes no requirement on the format, structure or process in which it's used.

Authentication

Process by which an application verifies the identity of who's interacting with it. The outcome will generally be a set of attributes that are proven to be associated to the user accessing the application.

Authorization

Process by which an application determines if the user accessing it has the necessary permissions to access a resource or perform a given operation. Generally dependent of a preceding authentication step that ascertains the identity of the user.

Authorization server

Server capable of issuing access tokens for use in associated resource servers. The process to issue an access tokens requires authenticating the resource owner and obtaining their consent.

Bearer token

A more specific type of access token that implies that the only mechanism used to validate the access to a protected resource is the token itself. The nam ing stems from the fact that anyone in possession of the token - the bearer - can access the protected resource as long as the token remains valid.

Client application

Any type of application that issues requests to access protected resources on behalf of the resource owner.

Identity provider

An entity that creates, maintains and manages identity information. Provides an authoritative source of user identities and associated attributes.

Refresh token

A specialized token used to obtain or refresh expired access tokens; while access tokens are issued by authorization servers and consumed by resource servers, the refresh token is both issued and consumed solely by an authorization server. A refresh token represents all the resources to which the resource owner granted access and can be used to obtain access tokens that target either all or just a subset of those resources.

Relying party

An entity that provides resources/services to authenticated entities, but does so while relying on security tokens issued by an identity provider in order to ensure the calling entity authentication. It's mainly used within WS-Federation and is equivalent to the SAML service provider.

Resource server

Server hosting protected resources. In the context of OAuth2 it has the additional capability of accepting access tokens as a way to allow access to the protected resources.

Resource owner

The entity who has the authority to grant access to a protected resource. In the situation where this entity is a person it can be referred to as an end-user.

Service provider

An entity that provides services to authenticated entities. Delegates the actual authentication of entities that try to use the service to an external identity provider. This terminology is used within SAML protocol and is equivalent to the WS-Federation relying party.

Digital signature

Technique used to ensure that a given message was created by a well known entity and that the message was not altered in any way since it's creation.

For more identity and security related terminology check the Auth0 Identity Glossary (**https://auth0.com/identity-glossary**).