OAuth

Guide to Authentication

Table Of Contents

Disclaimer **2**

Introduction **2**

Authentication vs. Authorization 3

Types of Authentication Methods 6

Types of Authentication Protocols 7

OAuth in a Nutshell **13**

What is OAuth 13

How OAuth Works 13

Why is OAuth so Popular? 15

Is OAuth Safe? 15

What is OAuth Used For? 16

The Role of OAuth in Application Development 16

OAuth & APIs 17

Pros and Cons of OAuth 17

OAuth 1.0 vs. OAuth 2.0 18

OAuth 2.0 Authentication **18**

OAuth 2.0 Compatibility 19

OAuth Authentication Flow Grant Types 20

OAuth Token Storage **21**

OAuth Token Types 21

How Are OAuth Tokens Stored? 23

Best Practice for Storing and Retrieving OAuth Token 24

OAuth Token Expiration Best Practices 25

What Not to Do for Token Storage? 27

Further Reading **28**

Disclaimer

I'm not an expert in OAuth and anything related to it. All the content in this eBook is based on my own understanding and research that I have done. If there are any errors that need to be corrected in this eBook, do let me know at hello@alanmythoughts.com.

The scope of this eBook is targeted at people who are not very well-versed in web development and cybersecurity. I've tried to keep it as simple as possible so that everyone can understand what OAuth is and how it works.

Introduction

This eBook will be about OAuth, an HTTP authentication scheme used to grant authorization for a user to access resources on a remote server. I will be discussing the basics of OAuth, how it works, who uses it, and how it can be implemented. We will also explore some of the fundamental flaws in the system and ways that they can be addressed.

Whenever you log in to specific accounts, you may notice that you always need a login username or email and a password. But it's a different story when you need access to a particular account on another application.

Certain applications require access to another application to simply a workflow if that's what you are looking for. However, some applications require access to your social media account, for example. Let's say you are creating an app that allows the user to post pictures on Instagram, and the user can share their photographs elsewhere.

This is where OAuth comes in. OAuth allows the client application to access the resources of another application without having to ask for your password.

Anyway, we need to understand how both authentication and authorization work in online accounts. Authentication and authorization are closely related concepts that are unique in each other as well as in their meanings.

Authentication vs. Authorization

While both terms seem similar, they actually serve different purposes. Let's look at the explanations of both terms below.

Authentication

Authentication is the process of confirming that a user is who they claim to be. It is how you log into your account on a website or application.

It is used to validate a user's identity and is generally initiated by providing credentials, i.e., user name, email, password, PINs, TOTP apps, or biometrics. I will explain all the types of authentication below. But all in all, the process of authentication verifies the user's identity by confirming that they are indeed who they are claiming to be, or in other words, that they can prove their identity with a pre-established set of credentials, as mentioned earlier.

For example, when you need to log in to a certain application with your online accounts, such as Facebook, Twitter, or Gmail, you will be asked for your login ID and password.

Authorization

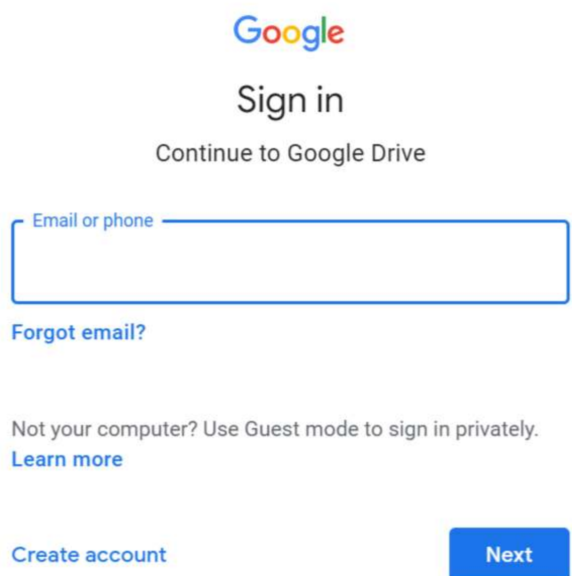
Authorization means you authorize an application to act on your behalf by providing credentials to access your account information. Authorization confirms access levels for particular resources on a remote server based on specified criteria. For example, let's say you want to access the Instagram account to post a picture. You access the Instagram website and are asked to provide your login credentials, such as your username and password. You enter these accounts because you need access to this user's account.

Once you have entered the correct username and password, you will be asked if the access is desired for yourself or the entire account. The authorization request access will request whether you want this application to be able to use your information for posting pictures or not.

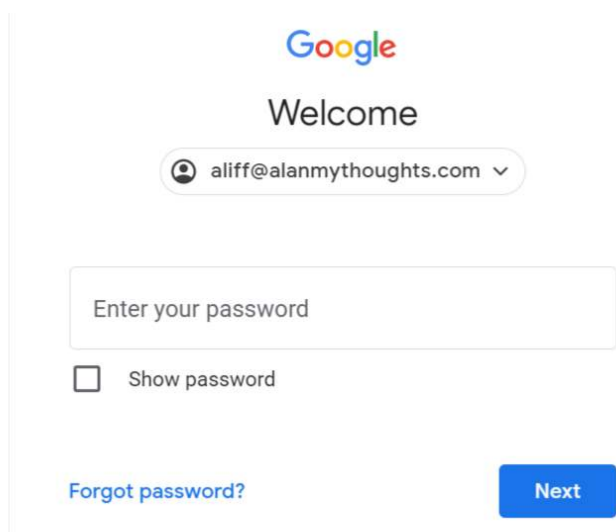
Still unclear?

Let's put this into a real-life example. I'm sure you may have encountered this too. For this example, I'm using my Google account to log in and authorize permissions from a third-party application called Frase.

When you want to log in to your Google account, it will look like this:

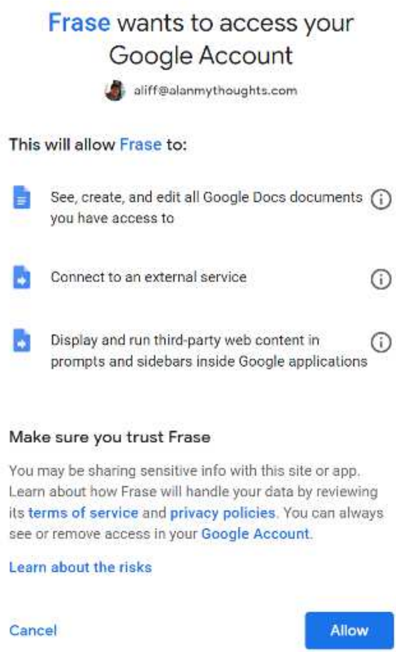


Then you enter your password:



This is called "authentication."

Now, when you want to access an app (in this case, Frase), you would like to allow Frase access to your Google account. Frase will list out the permissions needed, and then you can choose to allow it or otherwise.



This one is called "authorization."

Is this clear now?

Types of Authentication Methods

There are many ways to authenticate users. Following are some of the commonly used methods:

Password-based Authentication

It is also known as one-factor authentication, and it is based on a password in combination with a user ID. The user ID, however, is not considered a factor in this type of authentication. Password-based authentication is the most common type of authentication. Many online applications use this method. For example, when you try to log in to a website using your Facebook or Google account, the application will use your username and password to log in.

Two-factor Authentication

It is a particular type of authentication that depends on two factors. It includes the use of a form of an authentication method and another factor, like authentication via an app or device. For example, you use a password to log in to a website and then use a second factor, like your fingerprint, to log in again. The app or device is used to verify the user's identity. With two-factor authentication, an attacker is unlikely to be able to access your account even if he has access to your password.

Multi-factor Authentication

It's similar to two-factor authentication. It includes using two different sources of authentication to log in to a new session or service. If, in this case, it is multi-factor, it means it will be two or more forms involved to authenticate. This type of authentication is more secure than single-factor authentication. Without authenticating at least two forms of authentication, it is not possible to log in to the system.

For instance, a strong password will be used as the first form of authentication. The second form would be a hardware token, usually a USB drive that holds digital information related to the user's identity. If a user wants to add more level of security, consider adding more layers of authentication, such as biometric fingerprint or real-time location detection.

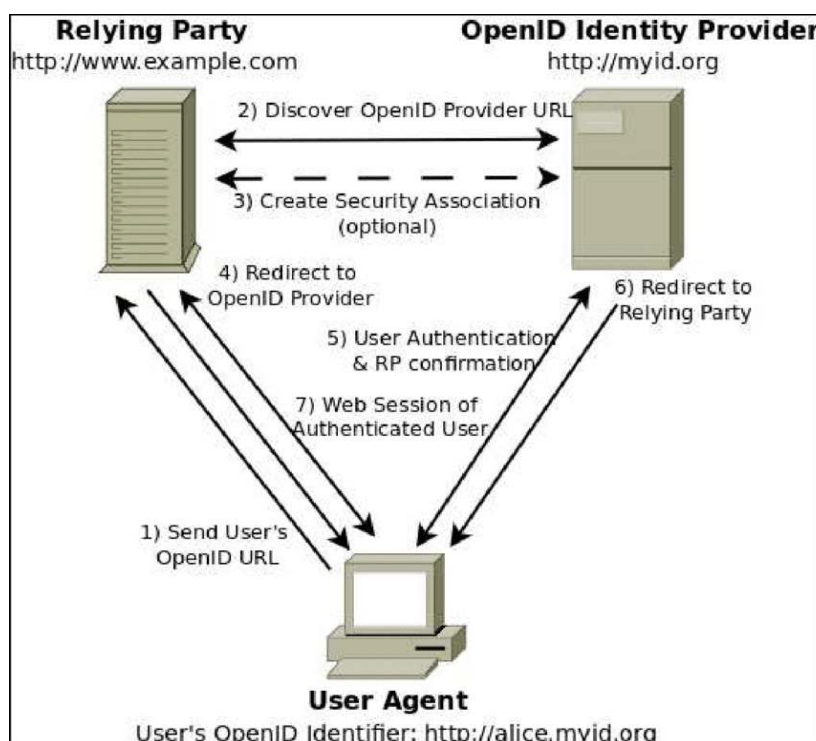
In short, the more factors required for authentication, the more secure it becomes. Typically, it utilizes these methods to authenticate online accounts:

- Something you know, like a username and password.
- Something you have, like a bank card, USB stick, or any other device that you carry.
- Something you are, like your fingerprint or voice.

Types of Authentication Protocols

Authentication protocols are used to make authentication smoother and more secure than it would be by using just a password. This is especially the case when you need to access an application from another app. Here are a few protocols that are utilized for authentication.

OpenID



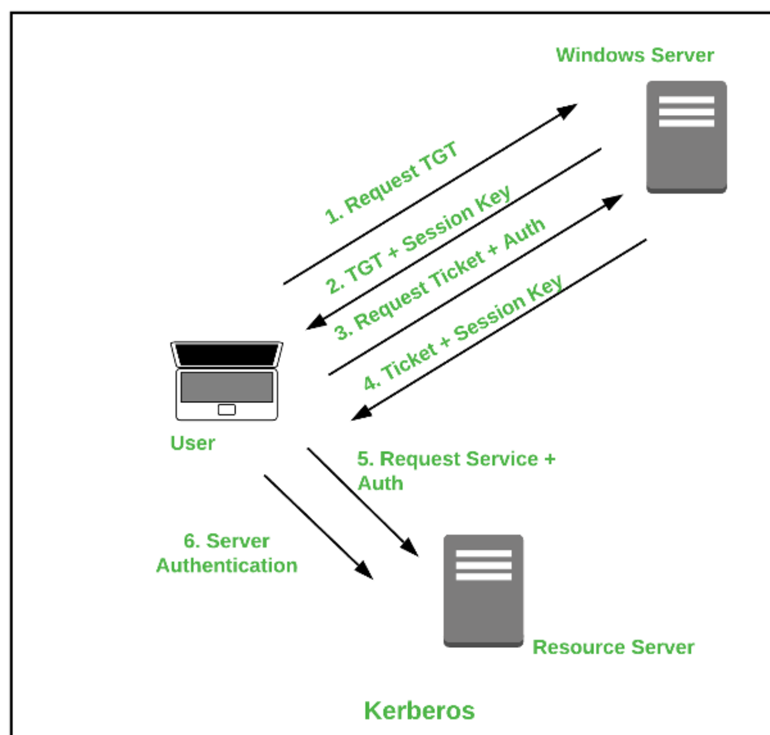
Source: ResearchGate

OpenID authentication is the use of an identifier in combination with another form of identification. It is the most commonly used protocol for authentication on the Internet. For instance, you can log into many websites by using your Facebook login or your Gmail ID. What makes OpenID identifiers so simple and easy to use is that many accounts and applications support them.

It allows for a more flexible authentication process. It consists of three steps:

1. The user provides their identity credentials to the OpenID provider's authorization server;
2. The authorization server authenticates the user, usually with a password, and extracts any other necessary information from the user's profile;
3. The OpenID provider receives this data from the Authorization Server and verifies it before allowing access to requested services by returning an authentication token that is sent back to the original website for verification.

Kerberos



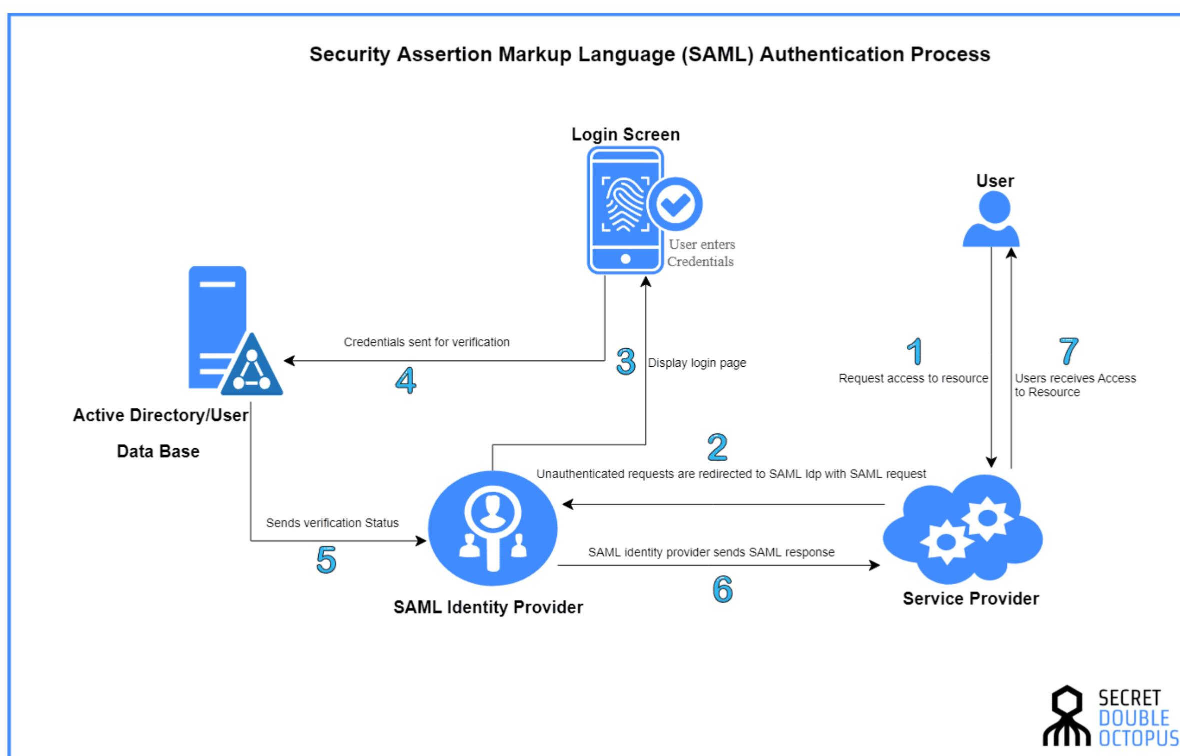
Source: GeeksforGeeks

Kerberos authentication is a computer network authentication protocol that was developed by MIT. It is commonly used for authenticating against Microsoft Windows networks. One of the main advantages of this method is that the whole process happens via a client-server mechanism and does not require any shared secret information to be shared between it and the user.

It also provides ease of scalability since it is more of a network authentication protocol and does not need to store secrets on any one computer. This protocol also allows for interoperability between the login process and other software programs or accounts. Kerberos authentication is supported by most operating systems, like macOS, Linux, iOS, and Android, making the authentication process with other apps and websites seamless.

When a user logs in to the system using this type of authentication method, they are authenticated with a ticket that contains the key. This part of the ticket is then sent from the client to the server. The server "knows" that it can trust this piece of information and can therefore use it to verify that the user is who he says he is.

SAML

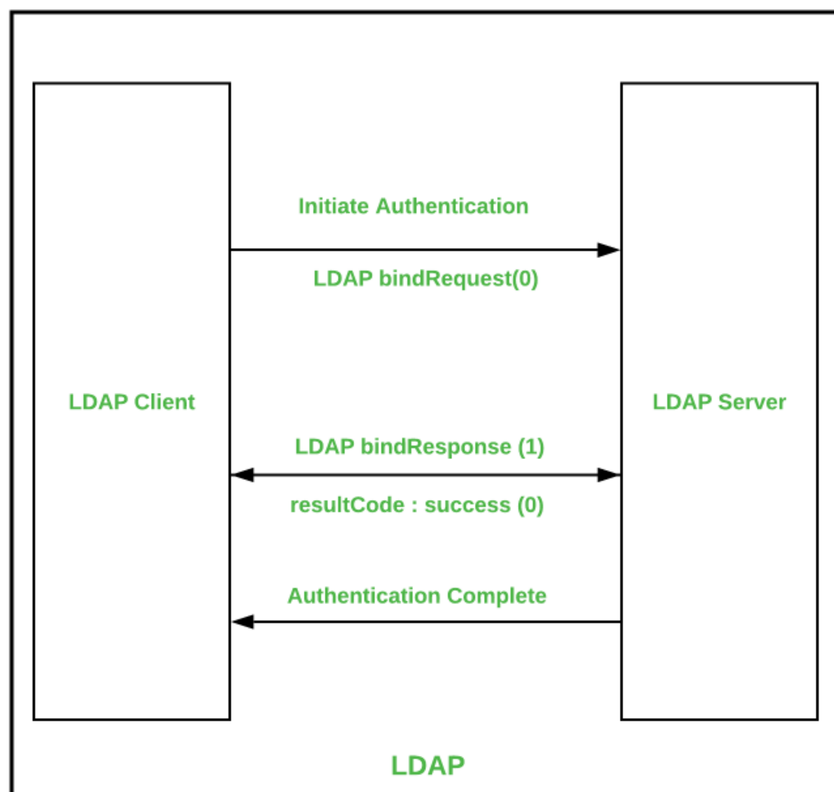


Source: Secret Double Octopus

SAML stands for Security Assertion Markup Language. It is an XML-based open-standard for exchanging authentication and authorization data between parties. SAML is based on the Username/Password combination of two parties exchanging messages via a requestor and responder.

There's also an additional requirement: how credentials are exchanged: the process must be mediated by a security token service (STS). The STS identifies the user and ensures that their identity matches what they're trying to access before granting them access to it or not.

LDAP

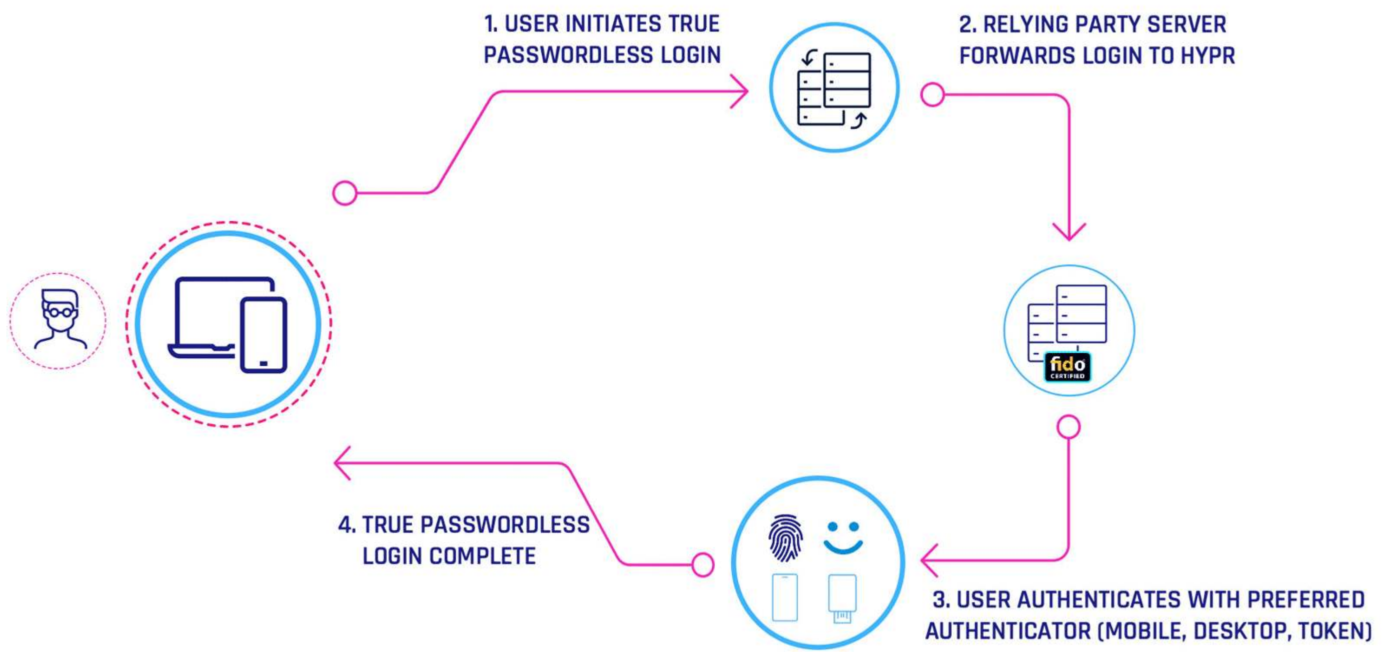


Source: GeeksforGeeks

LDAP stands for Lightweight Directory Access Protocol. It is considered a protocol that provides access to a directory service over the Internet. LDAP has an advantage because it only allows access to information that is stored in the directory, which means it doesn't require any type of authentication. The only way to be granted access is if you know how to get into the directory and find what you're looking for.

This protocol also provides ease of scalability since it is more of a network authentication protocol and does not need to store secrets on any one computer. On the other side, this type of protocol allows for interoperability between the login process with other software programs or accounts.

FIDO



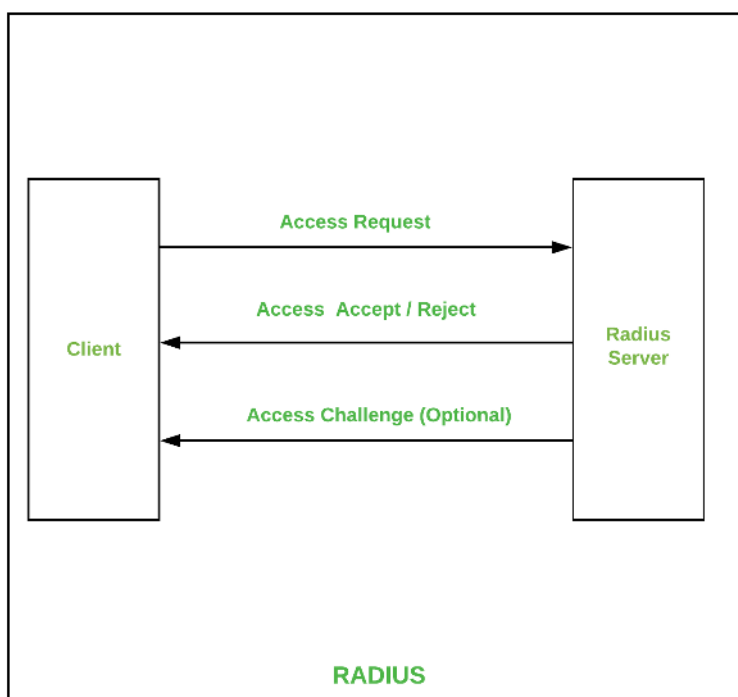
Source: HYPR

The FIDO protocol is based on public-key cryptography and peer-to-peer exchange of identity data. It provides more flexibility by only binding credentials to device hardware rather than relying on end-user devices like mobile phones or PCs.

It provides a secure and scalable solution for web authentication by using biometric or asymmetric cryptography. In this kind of protocol, users are authenticated via their devices. If the user loses access to their device, the account will be locked out until the user regains access to it.

It doesn't require any shared secret information to be shared between parties involved in authenticating; this makes it more secure and scalable than other authentication protocols that need shared secrets to be exchanged between parties.

RADIUS



Source: GeeksforGeeks

RADIUS, or Remote Authentication Dial-In User Service, is an application that's used to authenticate and validate users. It's a system that authenticates the user through a network service. RADIUS is based on two different protocols: Password Authentication Protocol (PAP) and Challenge Handshake Authentication Protocol (CHAP). This protocol provides authentication as well as validation of the user, making it more secure than other protocols.

The main idea behind this type of protocol is to provide identification by asking the user to enter their credentials before being granted access. The authentication process begins by validating the username and password combination with a remote server through either PAP or CHAP protocols; if it matches with what is stored on the system, it is considered to be an authentic user.

OAuth

As you know, OAuth is one of the most used authentication protocols. Since this eBook is dedicated to OAuth, I will explain further about it in the following few sections below. Keep reading!

OAuth in a Nutshell

What is OAuth

OAuth is a protocol that solves this problem by providing a way for the user to authorize access to your account without sharing passwords or usernames. An application and the user make the authorization through a series of steps.

OAuth provides authorization for users to access resources on a remote server without exchanging their credentials (username and password) with that remote server.

OAuth focuses primarily on the authentication of users, clients, and servers. It provides a standardized way of allowing requests for resources from third-party websites or applications to be made without requiring additional usernames and passwords. Furthermore, OAuth also enables the user to share specific data from one service to another securely.

How OAuth Works

OAuth allows for cross-application and service authentication without requiring additional username/password information. It is considered to be more secure than other protocols that require the use of shared secrets. There are three main steps in this process:

1. An application (the initiator) contacts the server to request authorization;

2. The server then sends an access token back to the application, which can now make a request for resources from the server;

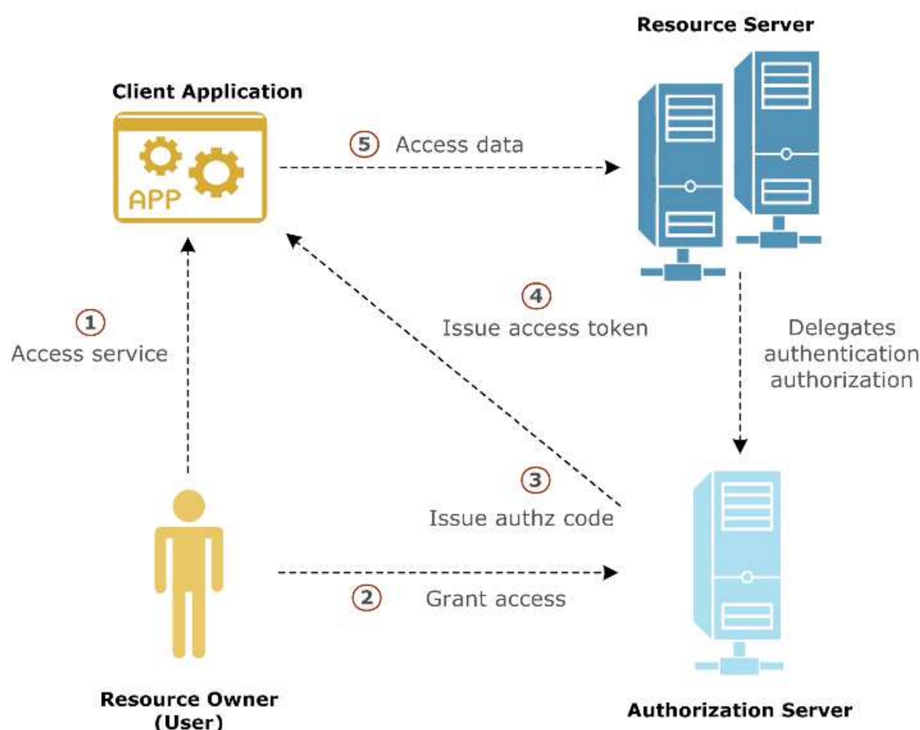
3. If the response from the resource server (considered a "resource") contains an expiration time, it must be renewed again before it expires for users/APIs being contacted to have continued access.

This protocol aims to provide a secure and scalable solution for web authentication by using biometric or asymmetric cryptography. On the other side, OAuth also allows for sharing specific data from one service to another.

This protocol provides a safe and secure way for users to allow access to resources on a remote server without exchanging their credentials with that remote server.

This type of protocol is going through some changes because it's considered insecure. It's currently being replaced by Web Token. You wouldn't need passwords or usernames when authenticating or authorizing requests from third-party applications on your account, but instead, use information embedded in a URI token that will be assigned as soon as the authorization is made.

Long story short: The user goes to a third-party app or website and authorizes it with their account the first time it is accessed by that app/site. This grants the third-party application access through their account for that specific resource. However, any resource which needs "refreshed" by this authorization still requires a username and password exchange between both users and applications for continued access. This allows control over who has access to information on your account without compromising security measures.



Source: Axway

Why is OAuth so Popular?

OAuth is popular because it has solved the problem of users needing to share usernames and passwords across many different applications. It has also gone through some changes because it's considered insecure. Still, it is currently being replaced by Web Token, in which you would not need to use a password or username when authenticating or authorizing requests from third-party applications on your account.

It provides a standardized and secure way for websites/APIs to access resources from external sites. This allowed OAuth to become much more popular than other protocols because the burden of managing passwords, usernames, etc., was moved onto the application instead.

The OAuth protocol is a widely used authentication method for authorizing third-party applications to access information from an HTTP service provider. It is both highly secure and capable of being shared with applications without compromising the security of a user's account.

In short, OAuth was established to avoid having users constantly having to remember multiple usernames and passcodes for different applications.

Is OAuth Safe?

Yes, OAuth is safe because it only requires users to authorize their account with a third-party application once before the third-party application can access resources through that account again.

It's secure because the system requires two parties to authenticate each other first: applications and users. The applications can send requests for permission after they have been recognized and authorized by the user. OAuth also allows for the sharing of specific data from one service to another.

What is OAuth Used For?

OAuth is primarily used for facilitating authorization of third-party applications to quickly and securely access resources over the Internet. It's a standardized and secure way of allowing requests for resources from third-party websites or applications to be made without requiring additional usernames or passwords to be exchanged between both parties.

OAuth is primarily implemented for web-based applications, desktop applications, mobile applications, or any other application where it is needed to provide access to something which belongs to the end-users.

The main advantage of OAuth is that it allows access to services without requiring users to remember lengthy usernames and passwords. The user would have access to their account through a third-party application, avoiding the need for users to enter usernames and passcodes into different applications manually. This way, users would be able to share specific data from one service with another, such as the content on a website or social media, without compromising that user's security.

The Role of OAuth in Application Development

OAuth's primary role in application development is as a secure and scalable method of authorization. As mentioned earlier, it provides the ability to authorize an application for specific resources without having to remember usernames or passwords.

This way, user accounts are given out to multiple applications instead of having access to the user's username/password. This allows having various apps to integrate with the same account, such as social media sites, as long as each app requires individual authorization.

With this authorization method, users no longer need to remember different usernames and passwords for each application they want to use.

OAuth & APIs

OAuth is a protocol that was initially designed for web-based applications. This means that it can also be applied to APIs so you can authorize third-party applications to quickly and securely access your online resources.

Many businesses are recognizing how vital it is to provide integrations with more than one platform to grow their business and make things easier for their customers. This includes providing a way for users of one platform (like social media) to connect with another (e.g., your website).

With OAuth, you would be able to allow or restrict specific data from one service on a user's account without compromising the security of the information being shared.

Pros and Cons of OAuth

The pros of using the OAuth protocol are:

1. It allows for cross-application and service authentication without requiring additional username/password information.
2. It is considered to be more secure than other protocols that require the use of shared secrets.
3. Allows for sharing specific data from one service with another without compromising security measures
4. It provides a standardized and secure way to access resources from external sites which would not otherwise require you to exchange your credentials with the external site/API
5. Allows you to make an unlimited number of requests for one user.

The cons of using the OAuth protocol are:

1. It can be challenging to keep track of all the different applications using your account
2. Even if you don't use OAuth, it's still possible for one service to share your information with third-party applications
3. It's not the most secure method of providing access to resources
4. It requires a centralized server or blockchain and a trusted party that handles credentials which is vulnerable to attacks

OAuth 1.0 vs. OAuth 2.0

OAuth 1.0 is the original OAuth framework drafted in 2007 that described several authorization grant types, such as resource owner password credentials, client credentials, and implicit grants. It also detailed a mechanism for using these grant types, known as the Authorization Code Flow. With OAuth 1.0, one entity (usually a web client or web servers like a mobile app or a web browser) can access the resources hosted by another entity (usually some sort of API).

OAuth 2.0 followed this framework through a more detailed authorization code flow, a discovery process for finding and communicating with service providers, and new grant types like authorization code and resource owner credentials. With the OAuth 2.0 framework, one entity (usually a web client or web servers like a mobile app or a web browser) can access the resources hosted by another entity (usually some sort of API).

OAuth 2.0 Authentication

OAuth 2.0 is an authentication protocol that enables a service provider to achieve single sign-on for its users. It's the successor to OAuth 1.0, and part of the OAuth 2.0 specification is still being developed by the IETF at the time of writing. Still, it has been standardized as RFC 6749: "Uniform Resource Identifier (URI) Fragment."

OAuth 2.0 enables the central authentication of a user's identity while also giving the access provider and/or the third party an authorization token. This access token is then used to perform authorization checks when accessing a protected resource.

OAuth 2.0 Compatibility

If you are developing an API project, you'll have to know if your provider supports the correct version of OAuth for your project's security needs. Your OAuth provider should be able to support the following:

The Provider Should Support OAuth 2.0

Often, developers will compare OAuth 1.0 and 2.0 as they are both protocols used for authenticating users and gaining access to resources. However, there are differences between the two protocols that need to be considered when choosing which one is appropriate for your application's needs.

The provider should be able to support OAuth 2.0, which would be the newest and most secure version of the protocol. This is because OAuth 2.0 has many more benefits, including the ability to quickly and securely access protected resources without having to remember usernames or password information.

Compatibility Should Exist With a Wide Variety of APIs to Provide an Interface for Different Platforms

This is important because it will allow you to provide integration with more than one platform and make it easier for your customers. This way, users of one platform can connect with another platform without having to enter usernames and passwords into different platforms manually. Having the OAuth 2.0 protocol will allow you to provide access without compromising security.

Compatibility Should Also Be Present With the Device the Application Is Running On

The compatibility of your application with the different devices that it runs on is essential. This is because you want to make sure that the app will work smoothly on all devices. For instance, if you have an Android app and an iPhone app, both apps should be compatible with operating systems. If an application needed to access a database in the cloud that didn't have an OAuth server, then you would need to provide compatibility with the device.

OAuth Authentication Flow Grant Types

There are different flow grant types in OAuth 2.0, which I outlined below:

Client Credential

A client credential grant type should be used when the application already has credentials from the user. The data is typically obtained once by first getting the authorization code, and then from there on, requesting access tokens to use.

Authorization Code

The authorization code grant type can be used for a variety of purposes, such as to authenticate the user and authorize access. After the user has been authenticated, they will be granted a specific set of permissions on behalf of the client or service provider. After this, when the application needs to access resources on behalf of this particular authorization code or token ID, they would be able to do so without having to contact OAuth again.

Implicit Code

An implicit grant type should be used when a resource owner is not currently logged in to an account and doesn't have credentials that are associated with it. This would usually involve redirecting the user's browser to another site to request an authorization token and returning it to their application using your API.

Resource Owner Password Credential

A resource owner password credential can be used when the user has no credentials or is not currently logged in to their account. One of the most important things to keep in mind with this grant type is that your application may need a valid token ID, such as an Authorization Code or even an Implicit Grant. This means that your application will need to make requests to OAuth repeatedly until it receives one from them.

Device Code

A device code grant type deals with a user's access device, such as smartphones or computers. This grant type aims to authorize access to resources hosted on the end user's device without having to authenticate the user and get credentials for them. This would involve presenting an authorization code or token ID through your API and then implementing some kind of authorization check against it.

OAuth Token Storage

OAuth tokens are used for storing user credentials such as usernames and passwords in an API. The token can't be decoded, which means that the data it keeps is secure even if a third party intercepts it.

For instance, a company will generate OAuth tokens as compensation for the client requesting access to the service provider's API resources. The OAuth token stays valid until the user chooses to revoke access from your application.

OAuth Token Types

There are several types of tokens, which include:

Access Token

This grants access to a specific set of resources on behalf of an end-user, usually at the resource owner's request. It is used to access a user's protected resources.

Refresh Token

A refresh token grant type is one of the more unique ones because it does not authorize access on behalf of a particular authorization code or token ID. It's usually used for facilitating the creation or revocation of credentials.

With this token type, you specify that your application will be able to use a specific set of credentials created by OAuth and then generate a refresh token that can be exchanged for another set of credentials after the expiry date or the amount has been exceeded. This can be used to generate new access tokens if the original ones have expired or if the amount that has been generated has exceeded its limit.

User Token

This grants access to a single user's resources. It usually contains the same information that is in a resource owner's password credentials, which you would have to request from the resource owner.

Authorization Code Token

This type grants access to a single set of resources on behalf of a specific authorization code. The authorization code token ID must be requested from the user itself, and it represents all of the end user's permissions that were granted. The token ID is typically presented through some kind of UI or API interaction by your application.

Client Token

This grant type can be used for obtaining client credentials. But it is not recommended to replace OAuth tokens (which are often referred to as "Client Secrets").

Resource Owner Token

This grant type allows an end-user to obtain a set of access tokens that can be used to access protected resources in the form of an authorization code or some other kind of authentication. The access tokens could be obtained from the resource owner, who is usually the entity expected to host your application or API.

How Are OAuth Tokens Stored?

OAuth tokens can be stored in two ways: server-side or client-side.

Client-side Storage

Storing the token on the client-side (such as in a cookie) is considered to be a more secure method, especially if the client is using HTTPS. This means that it would still be stored securely even if a third party intercepted it. Storing these tokens client-side can involve embedding them into an HTML page or even in your JavaScript code.

OAuth tokens are stored by the client application in a cookie, which is made available to the service provider for accessing resources. The cookie is encrypted, which means that it can't be decoded. If a user chooses to revoke access from your application, the refresh token that is stored in the cookie will become invalid. Once this happens, you will need to generate another refresh token to use instead of the invalid one.

Server-side Storage

Storing the token on a server-side system is considered to be less secure, as it would be stored in the clear. If someone intercepts this token, then they will be able to use it to access protected resources. In comparison to storing the tokens on the client-side, this would be considered to be more secure.

Still, it would also be possible for a third party to intercept this token through a man-in-the-middle attack. This would result in a users' access being revoked when they log in with your application through one of its endpoints.

Best Practice for Storing and Retrieving OAuth Token

Here are some recommendations for storing OAuth tokens:

Client-side storage:

Store Tokens in the Browser's Memory

Storing tokens in the browser's memory is considered to be more secure because it would still be held securely even if it were intercepted by a third party. This means that a token won't be made available in plaintext on the user's computer, resulting in privacy issues.

Use Okta Instead of Local Storage

Okta's API uses JSON web tokens (JWT) instead of cookies. JWT allows you to store your access tokens on the server and use an API, and you can verify if a token is valid for use by notifying the client application when it has expired or exceeds its limit.

This allows you to scale because there are not many authorization calls for each user. It also reduces lock-in because they do not require any modification in your app or backend, and they work with the Okta SSO. This ensures that your data will be encrypted in a way that can't be decoded, which is better than an unencrypted cookie in the local storage.

Use Secure Cookies

Store tokens in a cookie that is encrypted and only accessible to your application. If a user chooses to revoke access to your application, the refresh token that is stored in the cookie will become invalid. Once this happens, you will need to generate another refresh token to use instead of the invalid one.

Server-side storage:

Encrypt the Database Values

Encrypting the value of each token is crucial for protecting it from being decrypted by a third party, which would prevent you from getting the access granted to your application. Regardless of what's stored, encryption should be used in order to avoid any risks with your data. If there's a vulnerability in the application, this can also affect the security of each token.

Enable User and Device Identity

This is also a way of achieving more security with your application if you're using token-based authentication. You can use this to verify the validity of the user and that they are logging in from a trusted device. It can help track the way your application works and the information that is shared with other parties. It also helps to ensure that a person's end-user experience is secured while being able to recognize who they are when using an application.

OAuth Token Expiration Best Practices

Limit the Number of Tokens That You Generate

To be safe, you should limit the number of tokens that you generate. The exact number may depend on how many different API endpoints or resources your application is accessing. This will limit the risk of a token being invalidated by one token expiring. When this happens, just a tiny portion of your users will likely experience an interruption in service and not all users.

If exceeded a certain amount of tokens are generated, then make sure that you're using client-side storage so they can't be intercepted by third parties and revoked from user devices when users log into your application through an endpoint.

Check for Token Expiration When Retrieving a Token From the Server

You'll need to implement a check for token expiration when retrieving a token from the server. This would require your application to make another request for authorization and then present the authorization code or other authentication information that you receive from the user on behalf of their account. If a failure response is received in this situation, it could mean that the access token has expired.

Implement a Graceful Error Handler

A graceful error handler would be helpful to prevent your application from crashing and killing a user's session or workflow. This can be implemented if your application crashes when a token has expired. This could happen if you're storing tokens on the client-side and something goes wrong with the user's connection. Such a scenario would cause an interruption in service, and not all of your users would experience this.

However, it's possible for an interruption to happen if any token expires, whether they're stored server-side or client-side. When this happens, your users will not be able to access one of the protected resources that they initially granted access to from your application. Typically, their username and password credentials mean that a graceful error handler should be implemented to prevent them from being kicked out of their session when this happens.

Use Rotating Tokens

Also known as refresh tokens, this would involve exchanging older tokens for newer ones and then refreshing them after they expire or when you have exceeded your desired amount of permissions (such as API requests). This way, not only does this eliminate the need for users to store their credentials, but it also provides some shortcuts in case of emergency, such as if a server is no longer available. OAuth will automatically give out a new token even if you don't provide a refresh token.

This kind of token rotation is also helpful for the authorization code grant type because you only have one-time use for each access code.

What Not to Do for Token Storage?

Here's what you should not do to store tokens.

Hardcore Tokens in Applications

Since applications store the hard tokens, this could lead to the authentication credentials being vulnerable to hacking. There's a high degree of risk and potential for a security breach. This would let an attacker get hold of sensitive information that their system is storing on behalf of other people who are using it.

Store Tokens in Source Codes Like GitHub

When storing credentials in source code, they run the risk of their application being compromised. This could be due to the fact that it would be easy for a third party to access this information and use it maliciously. When an attacker gets this information, they can impersonate somebody else and get access to protected resources on behalf of somebody who's using it without their consent and potentially cause serious harm.

Roll Your Own OAuth Authentication

OAuth is a commercially proven framework that has been designed specifically for this purpose. For the most part, it's not recommended to roll your own authentication. This is because the time that would be spent designing and implementing such a system could instead be better spent on improving other aspects of your application, such as its features or its design.

It's worth noting that OAuth can be used for more than just authentication and authorization purposes. OAuth was designed to provide an easy-to-use but secure API interface so developers can spend more time creating their product instead of developing the management of security credentials themselves.

Store Tokens in Local Storage

Caching tokens in local storage can be dangerous. This is because they are stored in the clear, which means that it would be possible for a third party to intercept your token and then use it to access sensitive information on behalf of your account without you knowing about it. Because of this risk, server-side storage is typically preferable over client-side storage, as long as you're using a system that encrypts the database values and makes sure that user and device identity is prevented from being enabled.

Send Tokens Over Cleartext

When you send your tokens over an unencrypted connection, you run the risk of a third party intercepting it and compromising your system. This means that they can impersonate your account without permission and use it to access protected resources on behalf of somebody who's using it without their knowledge. It is recommended to use HTTPS when sending tokens over an unencrypted connection.

Further Reading

- OAuth2 Introduction Through Flow Diagrams in 5-minutes
- The OAuth 2.0 Authorization Framework
- Auth0 Token Best Practices
- Auth0 Token Storage