# OAuth 2 IN ACTION

Justin Richer
Antonio Sanso

FOREWORD BY Ian Glazer

MANNING

# *OAuth 2 in Action*

by Justin Richer and Antonio Sanso

**Chapter 1**

# *brief contents*

# Part 1

# First steps

In this section, you'll get a thorough overview of the OAuth 2.0 protocol, how it works, and why it works the way that it does. We'll start with an overview of what OAuth is and how people used to solve the delegation problem before OAuth was invented. We'll also take a look at the boundaries of what OAuth is not and how it fits into the larger web security ecosystem. We'll then take a deep look at the authorization code grant type, the most canonical and complete grant type available in OAuth 2.0 today. These topics will provide a solid basis for understanding the rest of the book.

# 1 What is OAuth 2.0 and why should you care?

***This chapter covers***

- What OAuth 2.0 is
- What developers do without OAuth
- How OAuth works
- What OAuth 2.0 is not

If you're a software developer on the web today, chances are you've heard of OAuth. It is a security protocol used to protect a large (and growing) number of web APIs all over the world, from large-scale providers such as Facebook and Google to small one-off APIs at startups and inside enterprises of all sizes. It's used to connect websites to one another and it powers native and mobile applications connecting to cloud services. It's being used as the security layer for a growing number of standard protocols in a variety of domains, from healthcare to identity, from energy to the social web. OAuth is far and away the dominant security method on the web today, and its ubiquity has leveled the playing field for developers wanting to secure their applications.

But what is it, how does it work, and why do we need it?

## 1.1 What is OAuth 2.0?

OAuth 2.0 is a delegation protocol, a means of letting someone who controls a resource allow a software application to access that resource on their behalf without impersonating them. The application requests authorization from the owner of the resource and receives *tokens* that it can use to access the resource. This all happens without the application needing to impersonate the person who controls the resource, since the token explicitly represents a delegated right of access. In many ways, you can think of the OAuth token as a "valet key" for the web. Not all cars have a valet key, but for those that do, the valet key provides additional security beyond simply handing over the regular key. The valet key of a car allows the owner of the car to give limited access to someone, the valet, without handing over full control in the form of the owner's key. Simple valet keys limit the valet to accessing the ignition and doors but not the trunk or glove box. More complex valet keys can limit the upper speed of the car and even shut the car off if it travels more than a set distance from its starting point, sending an alert to the owner. In much the same way, OAuth tokens can limit the client's access to only the actions that the resource owner has delegated.

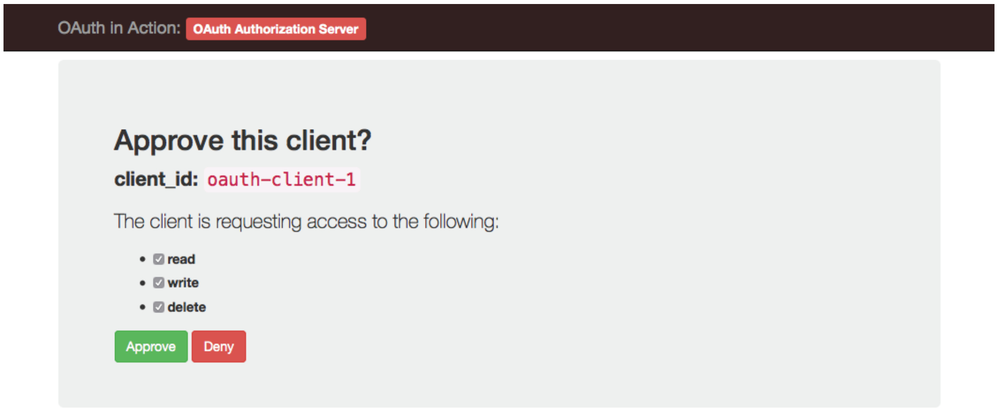
For example, let's say that you have a cloud photo-storage service and a photo-printing service, and you want to be able to print the photos that you have stored in your storage service. Luckily, your cloud-printing service can communicate with your cloud-storage service using an API. This is great, except that the two services are run by different companies, which means that your account with the storage service has no connection to your account with the printing service. We can use OAuth to solve this problem by letting you delegate access to your photos across the different services, all without giving your password away to the photo printer.

Although OAuth is largely indifferent to what kind of resource it is protecting, it does fit nicely with today's RESTful web services, and it works well for both web and native client applications. It can be scaled from a small single-user application up to a multimillion-user internet API. It's as much at home on the untamed wilds of the web, where it grew up and is used to protect user-facing APIs of all types, as it is inside the controlled and monitored boundaries of an enterprise, where it's being used to manage access to a new generation of internal business APIs and systems.

And that's not all: if you've used mobile or web technology in the past five years, chances are even higher that you've used OAuth to delegate your authority to an application. In fact, if you've ever seen a web page like the one shown in figure 1.1, then you've used OAuth, whether you realize it or not.

In many instances, the use of the OAuth protocol is completely transparent, such as in Steam's and Spotify's desktop applications. Unless an end user is actively looking for the telltale marks of an OAuth transaction, they would never know it's being used.[1] This is a good thing, since a good security system should be nearly invisible when all is functioning properly.

[1] The good news is that by the end of this book, you should be able to pick up on all of these telltale signs yourself.



Approve this client?

client_id: oauth-client-1

The client is requesting access to the following:

- ☑ read
- ☑ write
- ☑ delete

**Figure 1.1 An OAuth authorization dialog from the exercise framework for this book**

We know that OAuth is a security protocol, but what exactly does it do? Since you're holding a book that's purportedly about OAuth 2.0, that's a fair question. According to the specification that defines it:[2]

> *The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.*

Let's unpack that a bit: as an *authorization framework*, OAuth is all about getting the right of access from one component of a system to another. In particular, in the OAuth world, a client application wants to gain access to a protected resource on behalf of a resource owner (usually an end user). These are the components that we have so far:

- The *resource owner* has access to an API and can delegate access to that API. The resource owner is usually a person and is generally assumed to have access to a web browser. Consequently, this book's diagrams represent this party as a person sitting with a web browser.
- The *protected resource* is the component that the resource owner has access to. This can take many different forms, but for the most part it's a web API of some kind. Even though the name "resource" makes it sound as though this is something to be downloaded, these APIs can allow read, write, and other operations just as well. This book's diagrams show protected resources as a rack of servers with a lock icon.
- The *client* is the piece of software that accesses the protected resource on behalf of the resource owner. If you're a web developer, the name "client" might make you think this is the web browser, but that's not how the term is used here. If

[2] RFC 6749 https://tools.ietf.org/html/rfc6749

you're a business application developer, you might think of the "client" as the person who's paying for your services, but that's not what we're talking about, either. In OAuth, the client is whatever software consumes the API that makes up the protected resource. Whenever you see "client" in this book, we're almost certainly talking about this OAuth-specific definition. This book's diagrams depict clients as a computer screen with gears. This is partially in deference to the fact that there are many different forms of client applications, as we'll see in chapter 6, so no one icon will universally suffice.

We'll cover these all in greater depth in chapter 2 when we look at "The OAuth Dance" in detail. But for now, we need to realize that we've got one goal in this whole setup: getting the client to access the protected resource for the resource owner (see figure 1.2).

In the printing example, let's say you've uploaded your vacation photos to the photo-storage site, and now you want to have them printed. The storage site's API is the resource, and the printing service is the client of that API. You, as the resource owner, need to be able to delegate part of your authority to the printer so that it can read your photos. You probably don't want the printer to be able to read all of your photos, nor do you want the printer to be able to delete photos or upload new ones of its own. Ultimately, what you're interested in is getting certain photos printed, and if you're like most users, you're not going to be thinking about the security architectures of the systems you're using to get that done.

Thankfully, because you're reading this book, chances are that you're not like most users and you do care about security architectures. In the next section, we'll see how this problem could be solved imperfectly without OAuth, and then we'll look at how OAuth can solve it in a better way.

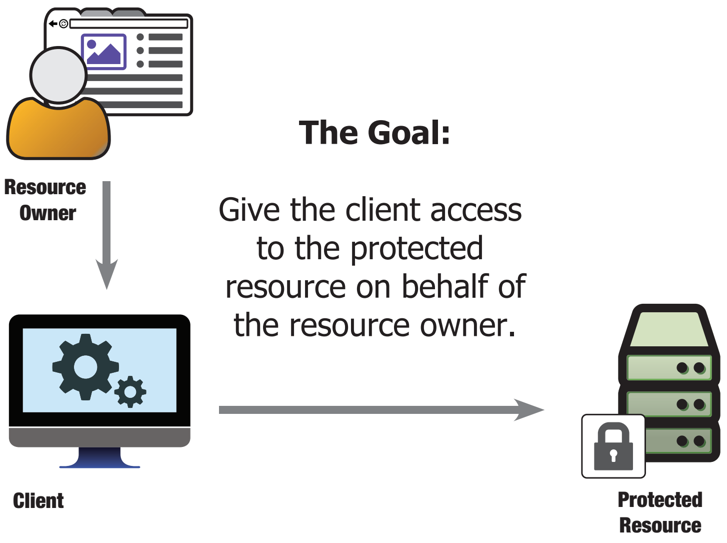

Resource Owner
The Goal:
Give the client access to the protected resource on behalf of the resource owner.
Client
Protected Resource

**Figure 1.2 Connecting the client on behalf of the resource owner**

## 1.2 The bad old days: credential sharing (and credential theft)

The problem of wanting to connect multiple disparate services is hardly new, and we could make a compelling argument that it's been around from the moment there was more than one network-connected service in the world.

One approach, popular in the enterprise space, is to *copy the user's credentials and replay them on another service* (see figure 1.3). In this case, the photo printer assumes that the user is using the same credentials at the printer that they're using at the storage site. When the user logs in to the printer, the printer replays the user's username and password at the storage site in order to gain access to the user's account over there, pretending to be the user.

In this scenario, the user needs to authenticate to the client using some kind of credential, usually something that's centrally controlled and agreed on by both the client and the protected resource. The client then takes that credential, such as a username and password or a domain session cookie, and replays it to the protected resource, pretending to be the user. The protected resource acts as if the user had authenticated directly, which does in fact make the connection between the client and protected resource, as required previously.

This approach requires that the user have the same credentials at the client application and the protected resource, which limits the effectiveness of this credential-theft technique to a single security domain. For instance, this could occur if a single company controls the client, authorization server, and protected resources, and all of these run inside the same policy and network control. If the printing service is offered by the same company that provided the storage service, this technique might work as the user would have the same account credentials on both services.

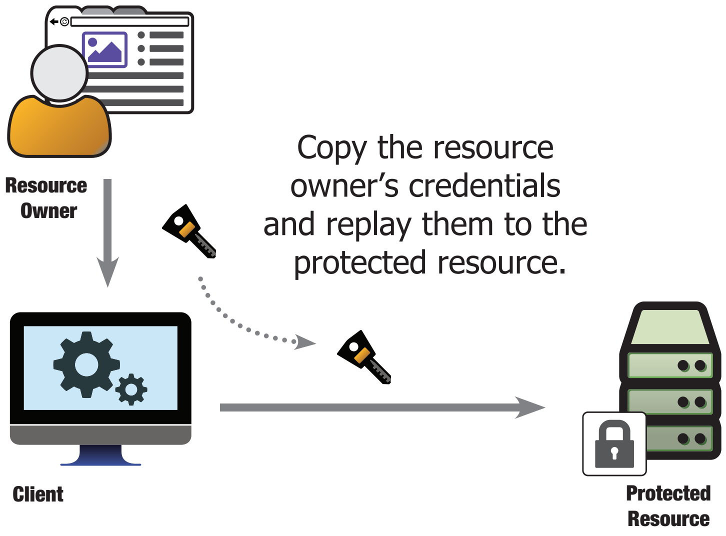

Resource Owner
Copy the resource owner's credentials and replay them to the protected resource.
Client
Protected Resource

**Figure 1.3 Copy the resource owner's credentials without asking**

This technique also exposes the user's password to the client application, though inside a single security domain using a single set of credentials, this is likely to be happening anyway. However, the client is *impersonating* the user, and the protected resource has no way to tell the difference between the resource owner and the impersonating client because they're using the same username and password in the same way.

But what if the two services occupied different security domains, a likely scenario for our photo-printing example? We can't copy the password the user gave us to log into our application any longer, because it won't work on the remote site. Faced with this challenge, these would-be credential thieves could employ an age-old method for stealing something: *ask the user* (figure 1.4).

If the printing service wants to get the user's photos, it can prompt the user for their username and password on the photo-storage site. As it did previously, the printer replays these credentials on the protected resource and impersonates the user. In this scenario, the credentials that the user uses to log into the client can be different from those used at the protected resource. However, the client gets around this by asking the user to provide a username and password for the protected resource. *Many users will in fact do this*, especially when promised a useful service involving the protected resource. Consequently, this remains one of the most common approaches to mobile applications accessing a back end service through a user account today: the mobile application prompts the user for their credentials and then replays those credentials directly to the back end API over the network. To keep accessing the API, the client application will store the user's credentials so that they can be replayed as needed. This is an extremely dangerous practice, since the compromise of any client in use will lead to a full compromise of that user's account across all systems.

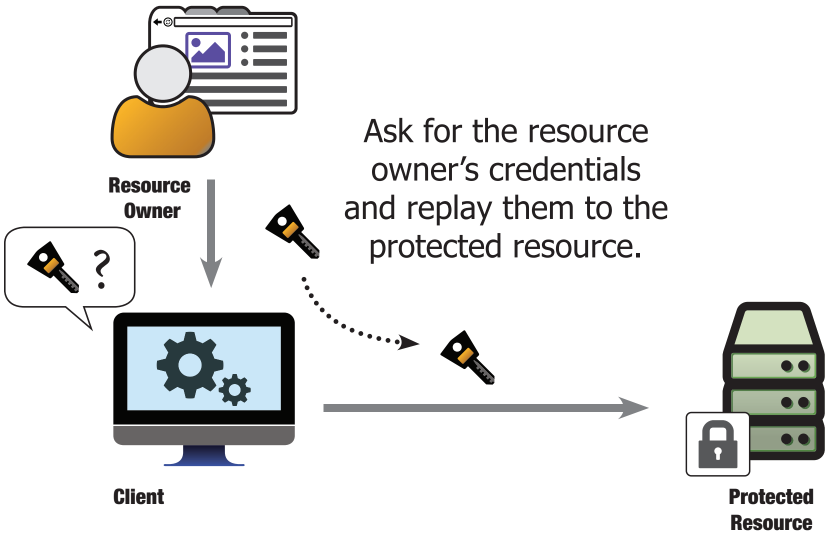

Ask for the resource owner's credentials and replay them to the protected resource.
Resource Owner
?
Client
Protected Resource

**Figure 1.4 Ask for the resource owner's credentials, and replay them**

This approach still works only in a limited set of circumstances: the client needs to have access to the user's credentials directly, and those credentials need to be able to be replayed against a service outside of the user's presence. This rules out a large variety of ways that the user can log in, including nearly all federated, many multifactor, and most higher-security login systems.

**Lightweight Directory Access Protocol (LDAP) authentication**

Interestingly, this pattern is exactly how password-vault authentication technologies such as LDAP function. When using LDAP for authentication, a client application collects credentials directly from the user and then replays these credentials to the LDAP server to see whether they're valid. The client system must have access to the plaintext password of the user during the transaction; otherwise, it has no way of verifying it with the LDAP server. In a very real sense, this method is a form of man-in-the-middle attack on the user, although one that's generally benevolent in nature.

For those situations in which it does work, it exposes the user's primary credentials to a potentially untrustworthy application, the client. To continue to act as the user, the client has to store the user's password in a replayable fashion (often in plaintext or a reversible encryption mechanism) for later use at the protected resource. If the client application is ever compromised, the attacker gains access not only to the client but also to the protected resource, as well as any other service where the end user may have used the same password.

Furthermore, in both of these approaches, the client application is *impersonating* the resource owner, and the protected resource has no way of distinguishing a call directly from the resource owner from a call being directed through a client. Why is that undesirable? Let's return to the printing service example. Many of the approaches will work, in limited circumstances, but consider that you don't want the printing service to be able to upload or delete photos from the storage service. You want the service to read only those photos you want printed. You also want it to be able to read only while you want the photos printed, and you'd like the ability to turn that access off at any time.

If the printing service needs to impersonate you to access your photos, the storage service has no way to tell whether it's the printer or you asking to do something. If the printing service surreptitiously copies your password in the background (even though it promised not to do so), it can pretend to be you and grab your photos whenever it wants. The only way to turn off the rogue printing service is to change your password at the storage service, invalidating its copy of your password in the process. Couple this with the fact that many users reuse passwords across different systems and you have yet another place where passwords can be stolen and accounts correlated with each other. Quite frankly, in solving this connection problem, we made things worse.

By now you've seen that replaying user passwords is bad. What if, instead, we gave the printing service universal access to all photos on the storage service on behalf of

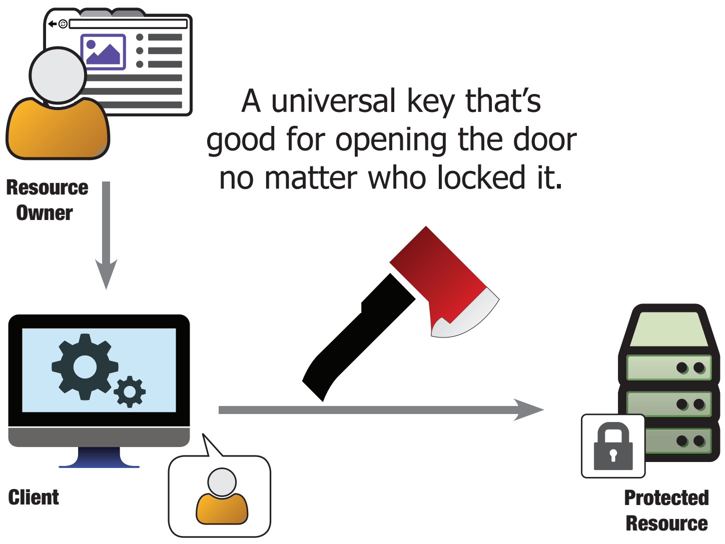

A universal key that's good for opening the door no matter who locked it.
Resource Owner
Client
Protected Resource

**Figure 1.5 Use a universal developer key, and identify the user on whose behalf you're (allegedly) acting**

anyone it chose? Another common approach is to use a developer key (figure 1.5) issued to the client, which uses this to call the protected resource directly.

In this approach, the developer key acts as a kind of universal key that allows the client to impersonate any user that it chooses, probably through an API parameter. This has the benefit of not exposing the user's credentials to the client, but at the cost of the client requiring a highly powerful credential. Our printing service could print any photos that it wanted to at any time, for any user, since the client effectively has free rein over the data on the protected resource. This can work to an extent, but only in instances in which the client can be fully known to and trusted by the protected resource. It is vanishingly unlikely that any such relationship would be built across two organizations, such as those in our photo-printing scenario. Additionally, the damage done to the protected resource if the client's credentials are stolen is potentially catastrophic, since all users of the storage service are affected by the breach whether they ever used the printer or not.

Another possible approach is to *give users a special password* (figure 1.6) that's only for sharing with third-party services. Users don't use this password to log in themselves, but paste it into applications that they want to work for them. This is starting to sound like that limited-use valet key you saw at the beginning of the chapter.

This is starting to get closer to a desirable system, as the user no longer has to share their real password with the client, nor does the protected resource need to implicitly trust the client to act properly on behalf of all users at all times. However, the usability of such a system is, on its own, not very good. This requires the user to generate, distribute, and manage these special credentials in addition to the primary passwords they already must curate. Since it's the user who must manage these credentials, there is also, generally speaking, no correlation between the client program and the credential itself. This makes it difficult to revoke access to a specific application.

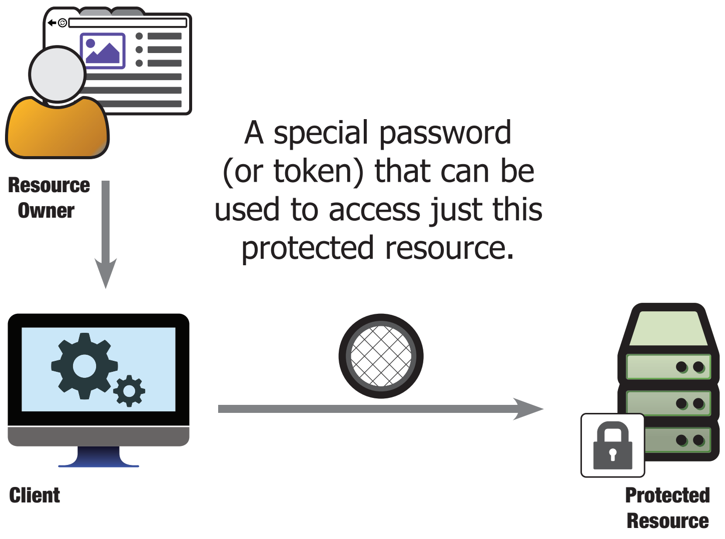

Resource Owner
A special password (or token) that can be used to access just this protected resource.
Client
Protected Resource

**Figure 1.6 A service-specific password that limits access**

Can't we do better than this?

What if we were able to have this kind of limited credential, issued separately for each client and each user combination, to be used at a protected resource? We could then tie limited rights to each of these limited credentials. What if there were a network-based protocol that allowed the generation and secure distribution of these limited credentials across security boundaries in a way that's both user-friendly and scalable to the internet as a whole? Now we're starting to talk about something interesting.

## 1.3 Delegating access

OAuth is a protocol designed to do exactly that: in OAuth, the end user *delegates* some part of their authority to access the protected resource to the client application to act on their behalf. To make that happen, OAuth introduces another component into the system: the *authorization server* (figure 1.7).

The authorization server (AS) is trusted by the protected resource to issue special-purpose security credentials—called OAuth access tokens—to clients. To acquire a token, the client first sends the resource owner to the authorization server in order to request that the resource owner authorize this client. The resource owner authenticates to the authorization server and is generally presented with a choice of whether to authorize the client making the request. The client is able to ask for a subset of functionality, or scopes, which the resource owner may be able to further diminish. Once the authorization grant has been made, the client can then request an access token from the authorization server. This access token can be used at the protected resource to access the API, as granted by the resource owner (see figure 1.8).

At no time in this process are the resource owner's credentials exposed to the client: the resource owner authenticates to the authorization server separately from anything

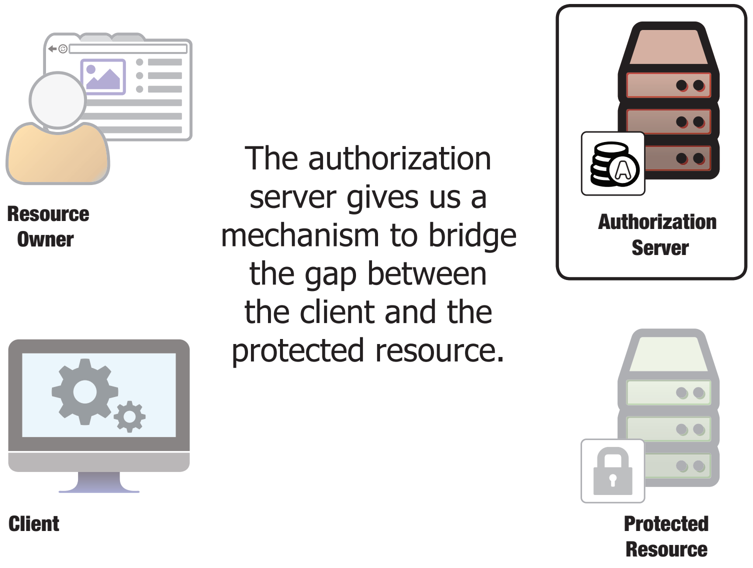

Resource Owner
The authorization server gives us a mechanism to bridge the gap between the client and the protected resource.
Authorization Server
Client
Protected Resource

**Figure 1.7 The OAuth authorization server automates the service-specific password process**

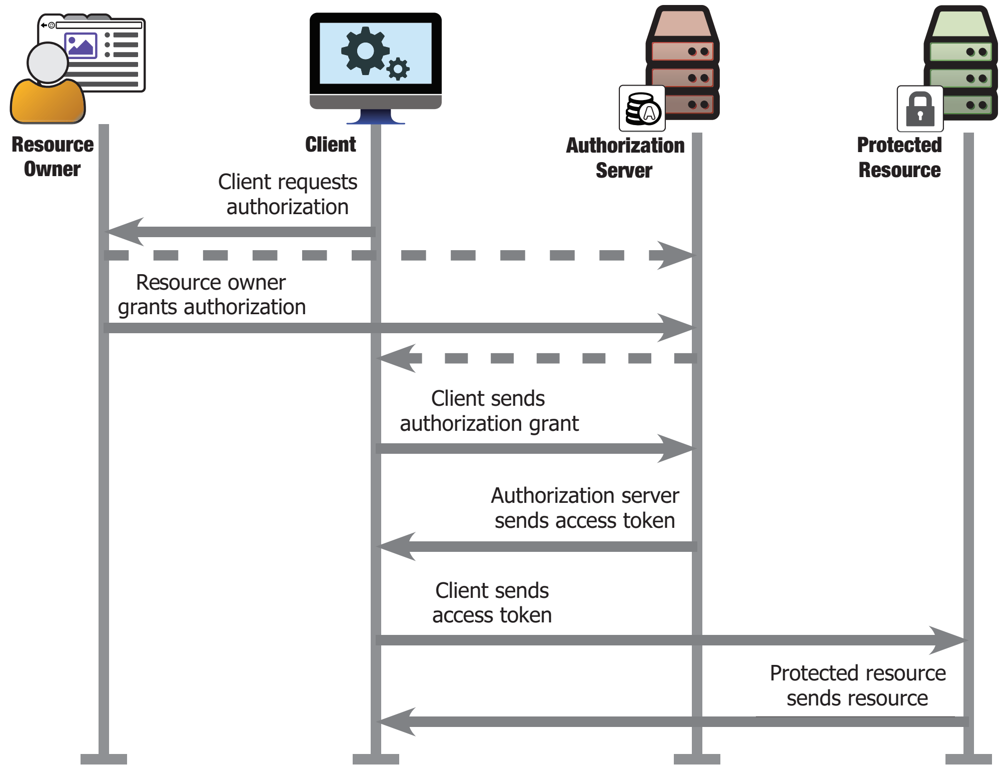

Resource Owner
Client
Authorization Server
Protected Resource
Client requests authorization
Resource owner grants authorization
Client sends authorization grant
Authorization server sends access token
Client sends access token
Protected resource sends resource

**Figure 1.8 The OAuth process, at a high level**

used to communicate with the client. Neither does the client have a high-powered developer key: the client is unable to access anything on its own and instead must be authorized by a valid resource owner before it can access any protected resources. This is true even though most OAuth clients have a means of authenticating themselves to the authorization server.

The user generally never has to see or deal with the access token directly. Instead of requiring the user to generate tokens and paste them into clients, the OAuth protocol facilitates this process and makes it relatively simple for the client to request a token and the user to authorize the client. Clients can then manage the tokens, and users can manage the client applications.

This is a general overview of how the OAuth protocol works, but in fact there are several ways to get an access token using OAuth. We'll discuss the details of this process in chapter 2 by looking in more detail at the authorization code grant type of OAuth 2.0. We'll cover other methods of getting access tokens in chapter 6.

### 1.3.1 Beyond HTTP Basic and the password-sharing antipattern

Many of the more "traditional" approaches listed in the previous section are examples of the password antipattern, in which a shared secret (the password) directly represents the party in question (the user). By sharing this secret password with applications, the user enables applications to access protected APIs. However, as we've shown, this is fraught with real-world problems. Passwords can be stolen or guessed, a password from one service is likely to be used verbatim on another service by the same user, and storage of passwords for future API access makes them even more susceptible to theft.

How did HTTP APIs become password-protected in the first place? The history of the HTTP protocol and its security methods is enlightening. The HTTP protocol defines a mechanism whereby a user in a browser is able to authenticate to a web page using a username and password over a protocol known as HTTP Basic Auth. There is also a slightly more secure version of this, known as HTTP Digest Auth, but for our purposes they are interchangeable as both assume the presence of a user and effectively require the presentation of a username and password to the HTTP server. Additionally, because HTTP is a stateless protocol, it's assumed that these credentials will be presented again on every single transaction.

This all makes sense in light of HTTP's origins as a document access protocol, but the web has grown significantly in both scope and breadth of use since those early days. HTTP as a protocol makes no distinction between transactions with a browser in which the user is present and transactions with another piece of software without an intermediary browser. This fundamental flexibility has been key to the unfathomable success and adoption of the HTTP protocol. But as a consequence, when HTTP started to be used for direct-access APIs in addition to user-facing services, its existing security mechanisms were quickly adopted for this new use case. This simple technological decision has contributed to the long-running misuse of continuously-presented passwords for both APIs and user-facing pages. Whereas browsers have cookies and other session-management techniques at their disposal, the types of HTTP clients that generally access a web API do not.

OAuth was designed from the outset as a protocol for use with APIs, wherein the main interaction is outside of the browser. It usually has an end user in a browser to start the process, and indeed this is where the flexibility and power in the delegation model comes from, but the final steps of receiving the token and using it at a protected resource lie outside the view of the user. In fact, some of the key use cases of OAuth occur when the user is no longer present at the client, yet the client is still able to act on the user's behalf. Using OAuth allows us to move past the notions and assumptions of the HTTP Basic protocol in a way that's powerful, secure, and designed to work with today's API-based economy.

### 1.3.2 Authorization delegation: why it matters and how it's used

Fundamental to the power of OAuth is the notion of delegation. Although OAuth is often called an authorization protocol (and this is the name given to it in the RFC which defines it), it is a delegation protocol. Generally, a subset of a user's authorization is delegated, but OAuth itself doesn't carry or convey the authorizations. Instead, it provides a means by which a client can request that a user delegate some of their authority to it. The user can then approve this request, and the client can then act on it with the results of that approval.

In our printing example, the photo-printing service can ask the user, "Do you have any of your photos stored on this storage site? If so, we can totally print that." The user is then sent to the photo-storage service, which asks, "This printing service is asking to get some of your photos; do you want that to happen?" The user can then decide whether they want that to happen, deciding whether to delegate access to the printing service.

The distinction between a delegation and an authorization protocol is important here because the authorizations being carried by the OAuth token are opaque to most of the system. Only the protected resource needs to know the authorization, and as long as it's able to find out from the token and its presentation context (either by looking at the token directly or by using a service of some type to obtain this information), it can serve the API as required.

**Connecting the online world**

Many of the concepts in OAuth are far from novel, and even their execution owes much to previous generations of security systems. However, OAuth is a protocol designed for the world of web APIs, accessed by client software. The OAuth 2.0 framework in particular provides a set of tools for connecting such applications and APIs across a wide variety of use cases. As we'll see in later chapters, the same core concepts and protocols can be used to connect in browser applications, web services, native and mobile applications, and even (with some extension) small-scale devices in the internet of things. Throughout all of this, OAuth depends on the presence of an online and connected world and enables new things to be built on that stratum.

### *1.3.3 User-driven security and user choice*

Since the OAuth delegation process involves the resource owner, it presents a possibility not found in many other security models: important security decisions can be driven by end user choice. Traditionally, security decisions have been the purview of centralized authorities. These authorities determine who can use a service, with which client software, and for what purpose. OAuth allows these authorities to push some of that decision-making power into the hands of the users who will ultimately be using the software.

OAuth systems often follow the principle of TOFU: Trust On First Use. In a TOFU model, the first time a security decision needs to be made at runtime, and there is no existing context or configuration under which the decision can be made, the user is prompted. This can be as simple as "Connect a new application?" although many implementations allow for greater control during this step. Whatever the user experience here, the user with appropriate authority is allowed to make a security decision. The system offers to remember this decision for later use. In other words, the first time an authorization context is met, the system can be directed to trust the user's decision for later processing: Trust On First Use.

**Do I *have* to eat my TOFU?**

The Trust On First Use (TOFU) method of managing security decisions is not required by OAuth implementations, but it's especially common to find these two technologies together. Why is that? The TOFU method strikes a good balance between the flexibility of asking end users to make security decisions in context and the fatigue of asking them to make these decisions constantly. Without the "Trust" portion of TOFU, users would have no say in how these delegations are made. Without the "On First Use" portion of TOFU, users would quickly become numb to an unending barrage of access requests. This kind of security system fatigue breeds workarounds that are usually more insecure than the practices that the security system is attempting to address.

This approach also presents the user's decision in terms of functionality, not security: "Do you want this client to do what it's asking to do?" This is an important distinction from more traditional security models wherein decision makers are asked ahead of time to demarcate what isn't permissible. Such security decisions are often overwhelming for the average user, and in any event the user cares more about what they're trying to accomplish instead of what they're trying to prevent.

Now this isn't to say that the TOFU method must be used for all transactions or decisions. In practice, a three-layer listing mechanism offers powerful flexibility for security architects (figure 1.9).

The whitelist determines known-good and trusted applications, and the blacklist determines known-bad applications or other negative actors. These are decisions that can easily be taken out of the hands of end users and decided a priori by system policy.

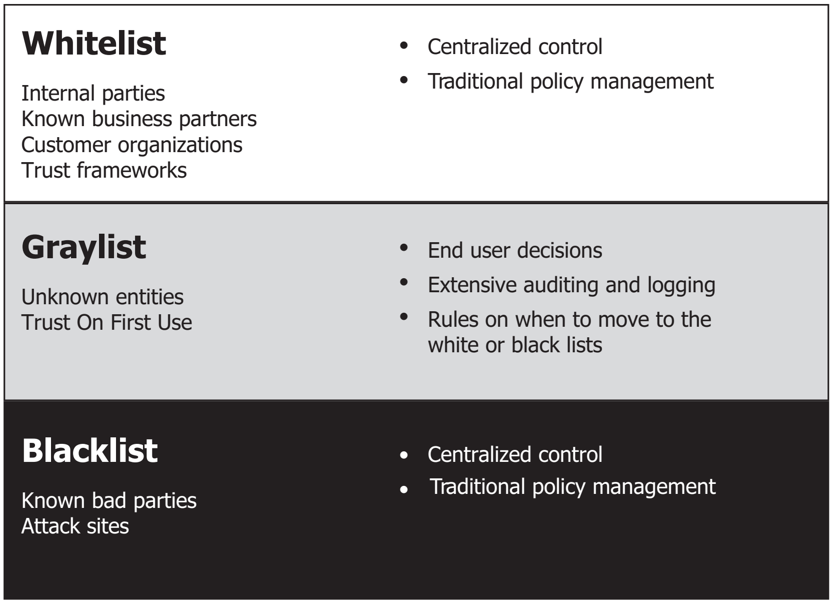


**Figure 1.9 Different levels of trust, working in parallel**

In a traditional security model, the discussion would stop here, since everything not on the whitelist is automatically on the blacklist by default. However, with the addition of the TOFU method, we can allow a graylist in the middle of these two, an unknown area in which user-based runtime trust decisions can take precedence. These decisions can be logged and audited, and the risk of breach minimized by policies. By offering the graylist capability, a system can greatly expand the ways it can be used without sacrificing security.

## 1.4 OAuth 2.0: the good, the bad, and the ugly

OAuth 2.0 is very good at capturing a user delegation decision and expressing that across the network. It allows for multiple different parties to be involved in the security decision process, most notably the end user at runtime. It's a protocol made up of many different moving parts, but in many ways it's far simpler and more secure than the alternatives.

One key assumption in the design of OAuth 2.0 was that there would always be several orders of magnitude more clients in the wild than there would be authorization servers or protected resource servers (figure 1.10). This makes sense, as a single authorization server can easily protect multiple resource servers, and there are likely to be many different kinds of clients wanting to consume any given API. An authorization server can even have several different classes of clients that are trusted at different levels, but we'll cover that in more depth in chapter 12. As a consequence of this architectural decision, wherever possible, complexity is shifted away from clients and onto servers. This is good for client developers, as the client becomes the simplest

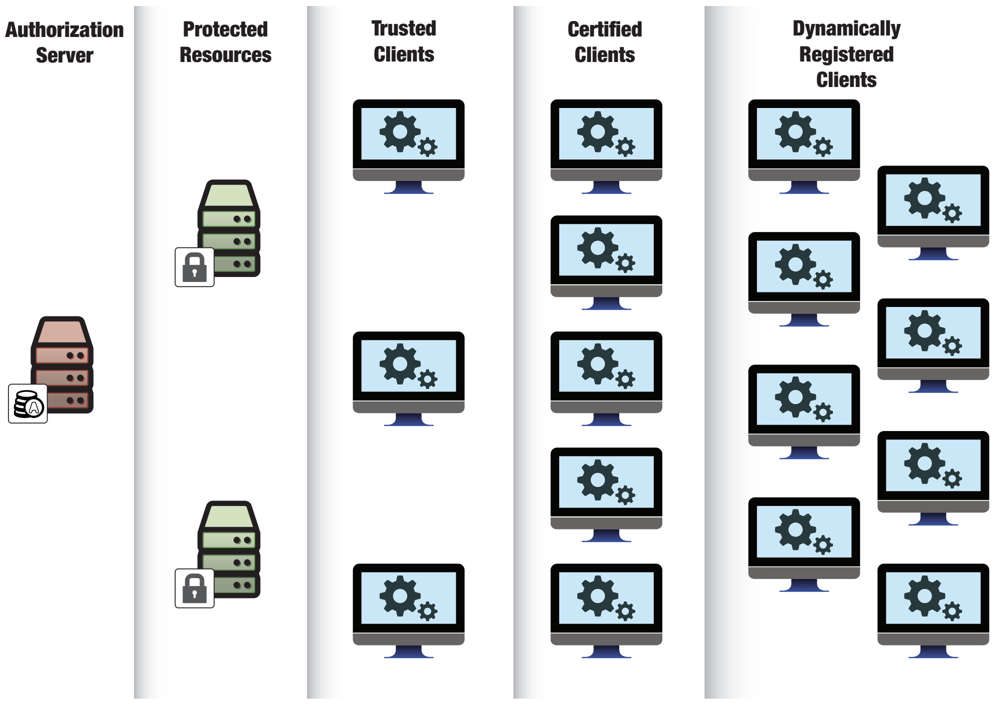


**Figure 1.10 Notional relative numbers of components in an OAuth ecosystem**

piece of software in the system. Client developers no longer have to deal with signature normalizations or parsing complicated security policy documents, as they would have in previous security protocols, and they no longer have to worry about handling sensitive user credentials. OAuth tokens provide a mechanism that's only slightly more complex than passwords but significantly more secure when used properly.

The flip side is that authorization servers and protected resources are now responsible for more of the complexity and security. A client needs to manage securing only its own client credentials and the user's tokens, and the breach of a single client would be bad but limited in its damage to the users of that client. Breaching the client also doesn't expose the resource owner's credentials, since the client never sees them in the first place. An authorization server, on the other hand, needs to manage and secure the credentials and tokens for all clients and all users on a system. Although this does make it more of a target for attack, it's significantly easier to make a single authorization server highly secure than it is to make a thousand clients written by independent developers just as secure.

The extensibility and modularity of OAuth 2.0 form one of its greatest assets, since it allows the protocol to be used in a wide variety of environments. However, this same flexibility leads to basic incompatibility problems between implementations. OAuth leaves many pieces optional, which can confuse developers who are trying to implement it between two systems.

Even worse, some of the available options in OAuth can be taken in the wrong context or not enforced properly, leading to insecure implementations. These kinds of vulnerabilities are discussed at length in the OAuth Threat Model Document[3] and the vulnerabilities section of this book (chapters 7, 8, 9, and 10). Suffice it to say, the fact that a system implements OAuth, and even implements it correctly according to the spec, doesn't mean that this system is secure in practice.

Ultimately, OAuth 2.0 is a good protocol, but it's far from perfect. We will see its replacement at some point in the future, as with all things in technology, but no real contender has yet emerged as of the writing of this book. It's just as likely that OAuth 2.0's replacement will end up being a profile or extension of OAuth 2.0 itself.

## 1.5 What OAuth 2.0 isn't

OAuth is used for many different kinds of APIs and applications, connecting the online world in ways never before possible. Even though it's approaching ubiquity, there are many things that OAuth is *not*, and it's important to understand these boundaries when understanding the protocol itself.

Since OAuth is defined as a framework, there has historically been some confusion regarding what "counts" as OAuth and what does not. For the purposes of this discussion, and truly for the purposes of this book, we're taking OAuth to mean the protocol defined by the core OAuth specification,[4] which details several ways of getting an access token. We're also including the use of bearer tokens as defined in the attendant specification,[5] which dictates how to *use* this particular style of token. These two actions—how to get a token and how to use a token—are the fundamental parts of OAuth. As we'll see in this section, there are a number of other technologies in the wider OAuth ecosystem that work together with the core of OAuth to provide greater functionality than what is available from OAuth itself. We contend that this ecosystem is evidence of a healthy protocol and shouldn't be conflated with the protocol itself.

*OAuth isn't defined outside of the HTTP protocol.* Since OAuth 2.0 with bearer tokens provides no message signatures, it is not meant to be used outside of HTTPS (HTTP over TLS). Sensitive secrets and information are passed over the wire, and OAuth requires a transport layer mechanism such as TLS to protect these secrets. A standard exists for presenting OAuth tokens over Simple Authentication and Security Layer (SASL)–protected protocols,[6] there are new efforts to define OAuth over Constrained Application Protocol (CoAP),[7] and future efforts could make parts of the OAuth process usable over non-TLS links (such as some discussed in chapter 15). But even in these cases, there needs to be a clear mapping from the HTTPS transactions into other protocols and systems.

[3] RFC 6819 https://tools.ietf.org/html/rfc6819
[4] RFC 6749 https://tools.ietf.org/html/rfc6749
[5] RFC 6750 https://tools.ietf.org/html/rfc6750
[6] RFC 7628 https://tools.ietf.org/html/rfc7628
[7] https://tools.ietf.org/html/draft-ietf-ace-oauth-authz

*OAuth isn't an authentication protocol*, even though it can be used to build one. As we'll cover in greater depth in chapter 13, an OAuth transaction on its own tells you nothing about who the user is, or even if they're there. Think of our photo-printing example: the photo printer doesn't need to know who the user is, only that *somebody* said it was OK to download some photos. OAuth is, in essence, an ingredient that can be used in a larger recipe to provide other capabilities. Additionally, OAuth uses authentication in several places, particularly authentication of the resource owner and client software to the authorization server. This embedded authentication does not itself make OAuth an authentication protocol.

*OAuth doesn't define a mechanism for user-to-user delegation,* even though it is fundamentally about delegation of a user to a piece of software. OAuth assumes that the resource owner is the one that's controlling the client. In order for the resource owner to authorize a different user, more than OAuth is needed. This kind of delegation is not an uncommon use case, and the User Managed Access protocol (discussed in chapter 14) uses OAuth to create a system capable of user-to-user delegation.

*OAuth doesn't define authorization-processing mechanisms.* OAuth provides a means to convey the fact that an authorization delegation has taken place, but it doesn't define the contents of that authorization. Instead, it is up to the service API definition to use OAuth's components, such as scopes and tokens, to define what actions a given token is applicable to.

*OAuth doesn't define a token format.* In fact, the OAuth protocol explicitly states that the content of the token is completely opaque to the client application. This is a departure from previous security protocols such as WS-*, Security Assertion Markup Language (SAML), or Kerberos, in which the client application needed to be able to parse and process the token. However, the token still needs to be understood by the authorization server that issues it and the protected resource that accepts it. Desire for interoperability at this level has led to the development of the JSON Web Token (JWT) format and the Token Introspection protocol, discussed in chapter 11. The token itself remains opaque to the client, but now other parties can understand its format.

*OAuth 2.0 defines no cryptographic methods*, unlike OAuth 1.0. Instead of defining a new set of cryptographic mechanisms specific to OAuth, the OAuth 2.0 protocol is built to allow the reuse of more general-purpose cryptographic mechanisms that can be used outside of OAuth. This deliberate omission has helped lead to the development of the JSON Object Signing and Encryption (JOSE) suite of specifications, which provides general-purpose cryptographic mechanisms that can be used alongside and even outside OAuth. We'll see more of the JOSE specifications in chapter 11 and apply them to a message-level cryptographic protocol using OAuth Proof of Possession (PoP) tokens in chapter 15.

*OAuth 2.0 is also not a single protocol.* As discussed previously, the specification is split into multiple definitions and flows, each of which has its own set of use cases. The core OAuth 2.0 specification has somewhat accurately been described as a security protocol generator, because it can be used to design the security architecture for many different use cases. As discussed in the previous section, these systems aren't necessarily compatible with each other.

**Code reuse between different OAuth flows**

In spite of their wide variety, the different applications of OAuth do allow for a large amount of code reuse between very different applications, and careful application of the OAuth protocol can allow for future growth and flexibility in unanticipated directions. For instance, assume that there are two back end systems that need to talk to each other securely without referencing a particular end user, perhaps doing a bulk data transfer. This could be handled in a traditional developer API key because both the client and resource are in the same trusted security domain. However, if the system uses the OAuth client credentials grant (discussed in chapter 6) instead, the system can limit the lifetime and access rights of tokens on the wire, and developers can use existing OAuth libraries and frameworks for both the client and protected resource instead of something completely custom. Since the protected resource is already set up to process requests protected by OAuth access tokens, at a future point when the protected resource wants to make its data available in a per-user delegated fashion, it can easily handle both kinds of access simultaneously. For instance, by using separate scopes for the bulk transfer and the user-specific data, the resource can easily differentiate between these calls with minimal code changes.

Instead of attempting to be a monolithic protocol that solves all aspects of a security system, OAuth focuses on one thing and leaves room for other components to play their parts where it makes more sense. Although there are many things that OAuth is not, OAuth does provide a solid basis that can be built on by other focused tools to create more comprehensive security architecture designs.

## 1.6 Summary

OAuth is a widely used security standard that enables secure access to protected resources in a fashion that's friendly to web APIs.

- OAuth is about *how to get a token* and *how to use a token*.
- OAuth is a delegation protocol that provides authorization across systems.
- OAuth replaces the password-sharing antipattern with a delegation protocol that's simultaneously more secure and more usable.
- OAuth is focused on solving a small set of problems and solving them well, which makes it a suitable component within larger security systems.

Ready to learn about how exactly OAuth accomplishes all of this on the wire? Read on for the details of The OAuth Dance.

WEB DEVELOPMENT/SECURITY

# OAuth 2 IN ACTION

Richer • Sanso

Think of OAuth 2 as the web version of a valet key. It is an HTTP-based security protocol that allows users of a service to enable applications to use that service on their behalf without handing over full control. And OAuth is used everywhere, from Facebook and Google, to startups and cloud services.

**OAuth 2 in Action** teaches you practical use and deployment of OAuth 2 from the perspectives of a client, an authorization server, and a resource server. You'll begin with an overview of OAuth and its components and interactions. Next, you'll get hands-on and build an OAuth client, an authorization server, and a protected resource. Then you'll dig into tokens, dynamic client registration, and more advanced topics. By the end, you'll be able to confidently and securely build and deploy OAuth on both the client and server sides.

## What's Inside

- Covers OAuth 2 protocol and design
- Authorization with OAuth 2
- OpenID Connect and User-Managed Access
- Implementation risks
- JOSE, introspection, revocation, and registration
- Protecting and accessing REST APIs

Readers need basic programming skills and knowledge of HTTP and JSON.

**Justin Richer** is a systems architect and software engineer. **Antonio Sanso** is a security software engineer and a security researcher. Both authors contribute to open standards and open source.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/oauth-2-in-action

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]

Free eBook
SEE INSERT

“Provides pragmatic guidance on what to do ... and what not to do.”
—From the Foreword by Ian Glazer, Salesforce

“Unmatched in both scope and depth. Code examples show how protocols work internally.”
—Thomas O'Rourke
Upstream Innovations

“A thorough treatment of OAuth 2 ... the authors really know this domain.”
—Travis Nelson
Software Technology Group

“A complex topic made easy.”
—Jorge Bo, 4Finance IT

ISBN-13: 978-1-61729-327-6
ISBN-10: 1-61729-327-X
54999
9 781617 293276