IEEE Internet Computing

www.computer.org/internet computing

OAuth Web Authorization Protocol

Barry Leiba

Vol. 16, No. 1
January/February, 2012

Editor: Barry Leiba • barryleiba@computer.org

OAuth Web Authorization Protocol

Barry Leiba • *Huawei Technologies*

Allowing one Web service to act on our behalf with another has become increasingly important as social Internet services such as blogs, photo sharing, and social networks have become widely popular. OAuth, a new protocol for establishing identity management standards across services, provides an alternative to sharing our usernames and passwords, and exposing ourselves to attacks on our online data and identities.

Over the previous two magazine issues, this department has looked at identity management. In the September/October issue, ISOC's Lucy Lynch gave an overview of the topic and the work standards organizations, the open source community, and others are doing to address identity management concerns. In the November/December issue, Jeremy Grant from the US National Institute of Standards and Technology described NIST's National Strategy for Trusted Identities in Cyberspace (NSTIC) initiative. In this third identity management installment, I'll look closely at the OAuth Web authorization protocol, which Lucy mentioned in her column.

Internet identity management is an umbrella that covers several related problems, all of which stem from our use of multiple Internet services that come from different providers and reside in different trust domains. For each domain, we have a separate identity and use separate authentication. Where NSTIC seeks to consolidate these identities through central management, and software such as password managers tries to make it easier to manage authentication credentials for our various identities, OAuth takes aim at a different piece of the puzzle.

OAuth began as a community effort among numerous companies that provide Internet services. These organizations recognized the need to solve a specific type of identity management problem and developed the first version of a mechanism for doing so. The group approached the IETF in late 2008. After a birds-of-a-feather session in November, subsequent mailing-list discussion, and a second session the following March, the IETF chartered the OAuth Working Group in May 2009 (see www.ietf.org/dyn/wg/charter/oauth-charter). Since then, the working group has been developing an Internet standard version of OAuth[1] that's just about completed at this writing.

The OAuth Use Case

We often need one service to interact with another on our behalf. Consider two scenarios:

1. Alice has a Gmail account with hundreds of contacts in her contact list. She joins Facebook and would like to see which of her Gmail contacts she can befriend in the social network. She can search Facebook for each one individually, but allowing Facebook to read her Gmail contacts directly would be much easier.
2. Bob has created a private photo album on Picasa with photos from a family function, and he would like to use a photo-printing service to print all the photos and mail the hardcopies to his grandparents. Bob could print each separately, of course; better, though, would be to direct the print service to his Picasa album and have the photos printed directly.

Services have supported such functions in the past by asking us to turn our authentication

Published by the IEEE Computer Society
1089-7801/12/$31.00 © 2012 IEEE

credentials for the target service — our login username and password — over to the service we want to use. Bob might, in example 2, give his Picasa password to the printing service so that it can log on as Bob and access his private album.

But turning over authentication credentials creates a major problem: it gives the service unrestricted access to the account we've given the credentials for. That service can now not only perform the action we've asked for but can also do anything with our information. A service given access to Alice's Gmail account could not only read her contacts — it could also change or delete them, add new ones, read her email, send email on her behalf, and so on. Worse, because we often reuse passwords from one service to another, it could guess that Alice might use the same username and password on other sites — perhaps banking and credit-card sites.

OAuth addresses this exposure by providing an alternative mechanism through which we can authorize specific actions, and only those specific actions, without giving unrestricted or permanent access. It has the target service create an access token that we can give out that allows only the limited access we've authorized, perhaps for a limited time or on a one-time basis.

How the OAuth Protocol Works

Let's look at a particular case and examine its dataflow. First, we'll need some terminology:

- *Client* — the service asking for authorization. In example 1, Facebook is the client; in example 2, it's the photo-printing service. Note that this is the OAuth transaction's client, not the end user's client program.
- *Resource owner* — the entity that owns the information the client needs to access. In both examples, this is the end user (Alice or Bob).
- *Resource server* — the service that provides access to the information requested. In example 1, that's Gmail, and in example 2, it's Picasa.
- *Authorization server* — the service that verifies the resource owner's credentials and performs the authorization checks. This is often the same as the resource server (as it is in both examples), and is always in its trust domain.
- *Access token* — a piece of data the authorization server creates that lets the client request access from the resource server. This is the authorization credential the client will use in place of the resource owner's own credentials.
- *Authentication code* — a piece of data that the authorization server can check, used during the transaction's request stage.

Now, let's consider example 1, in which Alice wants to import her Gmail contacts into Facebook. This is how such a transaction might work using OAuth:

1. Alice (the resource owner) logs into her Facebook account and selects an option on the Facebook website to import contacts from Gmail.
2. Facebook (the OAuth client) sends a response to Alice's Web request. The response goes to her Web browser, redirects the browser to Gmail (the authorization server), and passes the request to it.
3. Gmail prompts Alice to accept the requested authorization. She sees this authorization prompt in the browser; it's the first apparent response to her original request. Such a prompt might say, "Facebook is requesting access to read your contact list in Gmail." The prompt will come from Gmail and will ask Alice to log in to Gmail to approve the request. The client (Facebook) isn't involved in this step.
4. Alice accepts the authorization request. She might have to log in first, or she might already be logged into Gmail in the browser. Typically, she would click a button that says "Accept," "Authorize," "OK," or the like.
5. Gmail sends a response to Alice's Web browser that contains an authorization code, and the response redirects the Web browser back to Facebook (Alice doesn't see this).
6. Facebook sends the authorization code, along with other information, directly to the authorization server, behind the scenes and not apparent to Alice. Gmail responds to Facebook with an access token, if everything checks out.
7. Facebook uses the access token behind the scenes to contact the resource server (Gmail) and perform the service for Alice — in this case, retrieve her Gmail contact list and import the contacts into her Facebook account. Facebook will give Alice's browser a response, and the browser will show her a visual indication that the action is in progress.

Figure 1 summarizes this data flow graphically.

From Alice's viewpoint, the interaction has been very simple — one benefit of OAuth. She's made a request from Facebook to access Gmail, she has seen a prompt from Gmail asking her to authorize it, and she's seen Facebook acknowledge that authorization was received and that the request is being handled. But behind that simplicity was a reasonable amount of behind-the-scenes complexity. As the user, Alice need not know about nor understand any of that.

What's more, the access token the client (Facebook) receives can be

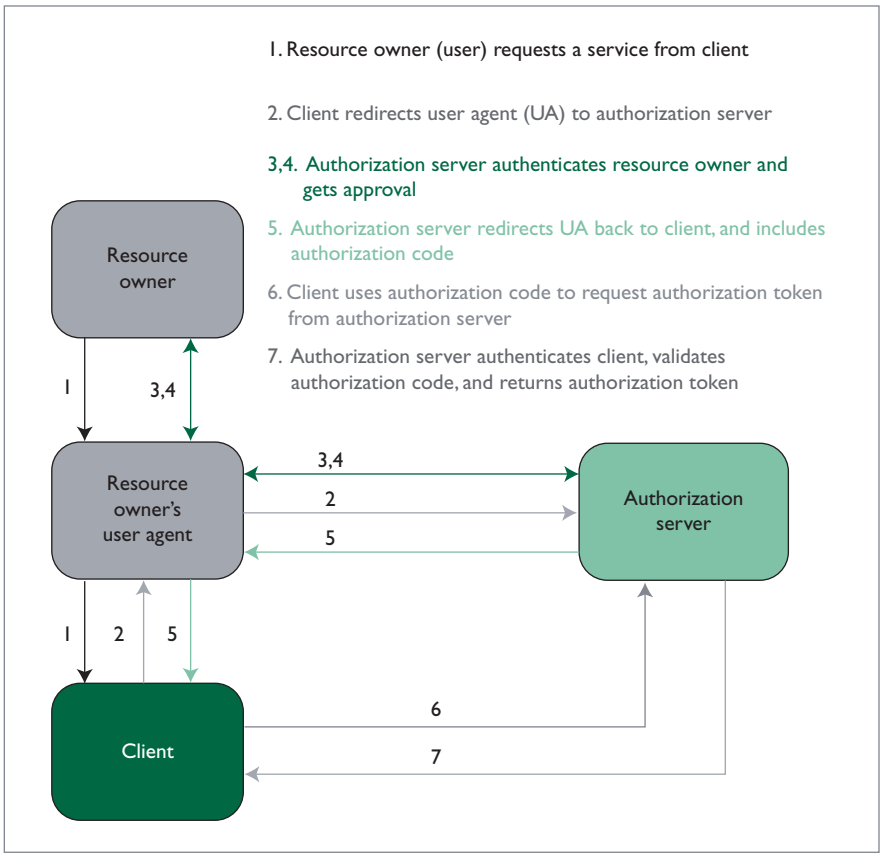

Figure 1. An OAuth transaction. Here, a Facebook user wants to import Gmail contacts into Facebook without giving Facebook her email username or password.

very specific in what it authorizes. Unlike Alice's normal Gmail login credentials, which would allow Facebook to do anything, the access token can authorize read-only access to the contact list without permitting additions or modifications, access to her email, or access to any other Gmail functions. It can also be limited in time as well as scope, permitting access only once, and for only five minutes, say. This lets Alice provide just the access needed and no more. Furthermore, it can hide from the client not only Alice's password but also her identity, making attempts at password guessing more difficult. All this makes OAuth an enormous improvement over giving Gmail login credentials directly to Facebook.

Not a Perfect Answer

Of course, even "an enormous improvement" isn't perfection, and OAuth still leaves some things exposed. The Web server redirections that are central to the mechanism provide places for attackers to target, and implementations that aren't careful to secure these points or don't adequately secure the tokens themselves are vulnerable. The OAuth Working Group is developing a set of token types[2,3] that will allow implementations to choose different security characteristics that might be appropriate for different use cases and operational environments. Another working group document describes the "Threat Model and Security Considerations" in some detail.[4]

But perhaps the most troublesome problem that OAuth doesn't solve is the need for users to understand what they're authorizing and to be relied on not to compromise their own security. When faced with security-related questions, many — perhaps most — users are simply used to clicking "OK" or whatever they need to click to get things to continue. A user visiting his or her bank's website will, if faced with a browser popup warning of an expired or otherwise suspicious security certificate, tell the browser to accept the certificate and continue. Most of the time, the bank has made an error, and accepting the certificate is the right thing to do. In any case, the user is unaware of the risks and wants to go to the bank's website; accepting the certificate is the only way to make that happen.

Such is the case with OAuth. If you refer back to the numbered list of steps in the OAuth transaction example, you'll see that Alice is presented with a message in step 3 that asks for authorization. To proceed to step 4, she must decide to authorize the request, which implies that she must understand what she's being asked. The user interface is critical at this point.

A user might need to understand the following questions, and know the answers to them.

Who is requesting the access? This can be a tricky point. The authorization server might know only the domain name, or even just the IP address of the client making the request. Alice might be able to make some sense of the domain name, but would really do better with a real, human-readable name that matches what she calls the service. But the authorization server has no cause to trust any human-readable string the client gives.

Who will be granting the access? It's easy to leave this out, with the idea that it should be self-evident. The problem is that if a client is requesting access beyond what it should be asking for, it might be asking the wrong entity as well. If Bob (from example 2) asks for a photo to be printed, the requested authorization shouldn't be to his email account.

What specific access is being requested? This might not always be obvious, depending on the request and on how the prompt is worded. "Print a photo for me" will likely translate into read access to the photo. For "Auto-adjust the contrast before printing, and save the adjusted version," the service will need access to update the photo or to save a new copy. "Send e-cards to my family on their birthdays" might translate into authority to send email on Bob's behalf, plus gain read access to his address book. The address book access is somewhat less evident and opens an avenue for abuse.

What is the access's scope? If Bob wants to print a single photo, then read access to just that photo will do. If he wants to print all photos in an album, he'll need to grant read access to the whole album.

What is the access's duration? If Bob just wants to print a photo, then one-time read access to the photo should be enough. If he wants the service to automatically print all his new photos every week, persistent, long-term read access to a "new uploads" photo album might work.

Because end users are accepting or rejecting the authorization that the client service is requesting, their understanding of what they're being asked is important to the system's overall security. Because such users often know nothing about computer security, the way these various points are presented to them is a critical piece of the security design — that is, we must consider the prompts and users' understanding of and response to them as part of the security model.

This is especially important because a user thinks in terms of a task, whereas the authorization system works in terms of what accesses it needs for that task. The mapping between the two often isn't clear to the user, and his or her trust of the service requesting access (the client) might be tenuous.

We must avoid asking users questions they're not prepared or qualified to answer. Unfortunately, most security-related questions fall into this category. The more we can put the request into plain language, and the better we can explain in clear, simple terms what's being asked and what the ramifications are, the more likely it is that we'll be working with informed consent and will be able to fend off attacks on the system.

A prompt such as

> Give printpix.example r/w access to http://photoshare.example/usr213554/

will likely be unintelligible to most users. One that says

> The Print My Pix service (printpix.example) is asking PhotoShare for access to all your photo albums. Granting access will allow Print My Pix to read, alter, and delete your photos. Access will be allowed permanently. For a more detailed explanation of what this means, [click here].

might seem excessive, but it conveys the scope of what's being asked and makes it evident that Print My Pix is probably asking for more than it needs. Specific warnings might also be added for such atypical access requests, ones that seem to be overstepping. Even so, it might well be a lost cause: expecting end users to understand and respond correctly to any security-related question is probably asking too much.

So, although OAuth removes the need for users to give away their login credentials in the use cases it supports, it still leaves an avenue for unethical or outright malicious services to fool users into authorizing nearly anything.

The OAuth Working Group has recently approved the OAuth 2.0 protocol specification, which is moving through the IETF's process. We'll likely see it published as a proposed standard in the first quarter of 2012. The first two token specifications won't be far behind. Several OAuth 1.0 implementations — the pre-standard version, published as an informational specification[5] — have been updated to be compatible with OAuth 2.0. Google, Yahoo, Facebook, Twitter, and other services already use OAuth, and we expect it to see even broader deployment after the proposed standard version is published.

At this writing, the working group is about to begin discussing re-chartering and deciding what to work on next. The discussions should be finished and the new charter in force by the time this column is published, so see the current OAuth Working Group charter (www.ietf.org/dyn/wg/charter/oauth-charter) for the results.

References

1. *The OAuth 2.0 Authorization Protocol*, IETF OAuth Working Group draft, work in progress, Sept. 2011.
2. *The OAuth 2.0 Authorization Protocol: Bearer Tokens*, IETF OAuth Working Group draft, work in progress, Oct. 2011.
3. *HTTP Authentication: MAC Access Authentication*, IETF OAuth Working Group draft, work in progress, May 2011.
4. *OAuth 2.0 Threat Model and Security Considerations*, IETF OAuth Working Group draft, work in progress, July 2011.
5. *The OAuth 1.0 Protocol*, IETF RFC 5849, Apr. 2010; http://tools.ietf.org/html/rfc5849.

Barry Leiba is a standards manager at Huawei Technologies. He currently focuses on the Internet of Things, messaging and collaboration on mobile platforms, security and privacy of Internet applications, and Internet standards development and deployment. Leiba has been active in the IETF for roughly 15 years, is an author of several current and pending proposed standards, chairs numerous working groups (including OAuth), and served on the Internet Architecture Board from 2007 to 2009. He edits this column, and can be reached at barryleiba@computer.org.