WHAT'S THE RIGHT RECIPE TO SECURE YOUR APIS?

APIs are everywhere today. They allow exchanges of information both within information systems, and with partners and customers who need to access them. But what does good security practice look like?

AUTHOR

BERTRAND CARLIER
bertrand.carlier@wavestone.com

This publication has been produced with the contributions of Samantha MARECAUX and Parfait NANGMO.

Today, what are commonly called APIs, or Application Programming Interfaces, group together a raft of inter-application communication methods ranging from web services (REST or SOAP) to local or remote calls between processes (RPCs). These types of web services, while not the only ones to use APIs, have spread like wildfire in recent years, and are now a widely used and essential communication mechanism for all companies that have embraced digital transformation. These days, they can be found in an increasing number of use cases: public, personal, and sensitive data – mobile applications, exchanges between partners, the IoT, so-called " client-side" applications, and so on.

But they are not a new concept. The introduction and definition of the concept of REST architecture, in 2000, saw the emergence of the first APIs. The pioneers were eBay (in particular) and Flickr; then Facebook and Twitter, made them the core to their products, something on which third-party developers could build their own services. And, ever since the emergence of the concept, the question of how to secure access to this new type of web service has been in the air.

Experience tells us that securing APIs is a recipe based on four ingredients, all of which must be carefully measured out.

THE *SECURITY AS USUAL* BASELINE

In a **Wavestone benchmarking exercise on web-application security**[1], of the 128 applications we audited, **serious flaws were observed in 60%.** The situation on the ground for APIs is very similar.

The answer is simple but often difficult to implement – the **usual recommendations for web security** – for example, those for **OWASP**[2], must be taken into account in just the same way.

A number of security measures and good development practices are available to developers and operations teams when it comes to covering the vulnerable areas traditionally targeted by attackers:

Applications web & APIs – *Security as usual*

SESSION MANAGEMENT

Authentication and maintenance of sessions
Client side vs. server side
Non-guessable session identifier
Reauthenticate for criticalactions

SENSITIVE DATA

Separation of environments
Storage and management of secret information
Make use of proven security mechanisms

1
2
3
4
5
6
The risk "zones"

ACCESS CONTROL

Management of profiles and prvileges
Cases of competition
Separation of user spaces

EXCEPTION MANAGEMENT

Management of errors
Creation of logs
Capture all errors and address them

INPUT/OUTPUT MANAGEMENT

Encryption of data beyond responding

MEMORY MANAGEMENT

Memory allocation
Initialization of objects and variables
Monitoring of memory use

DO NOT FORGET THE BASIC RECOMMENDATIONS OF WEB SECURITY...

A PINCH OF OAUTH

Once these basics have been properly mastered and applied, the question of proper access management for the application comes up. This is a matter of determining the means of authentication for accessing an API (to authenticate both the user and the calling application), and agreeing on a common protocol between third parties.

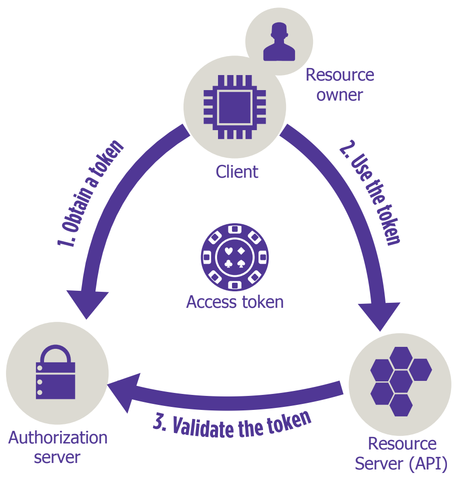
OAuth2 is now the best suited, and most widely, used standard for REST APIs. It consists of an authorization delegation standard that allows an application to obtain **authorization to access a resource (API) on behalf of a user.**

OAuth2 is designed to cover a wide range of use cases (web applications, mobile, access [or not] via a browser, server-to-server access, etc.); and, to this end, it offers:

/ **Four main process steps to obtain a token (RFC 6749)**

/ **Three mechanisms for using this token (RFC 6750)**

/ **Full documentation detailing the threat model and the right questions to ask when implementing OAuth2 within an architecture**



And lastly, a dedicated authentication overlay, which rounds off these initial set of steps: **OpenID Connect**[3]**.** This standard makes it possible to control the characteristics of user authentication more precisely (the means of authentication, Single Sign-On, transmission of identity attributes in a standard format, forced reauthentication, etc.)

By just looking at these four documents—, which already represent the equivalent of some 250 pages—we can understand why OAuth2 has a poor reputation for being a complex, heavy protocol that is liable to implementation errors.

This reputation isn't entirely undeserved: some major web players, such as Facebook and Twitter, have had their fingers burned, and have seen their users' personal data rendered accessible with no prior authentication.

It's important to understand that the root of the problem isn't the protocol itself: fortunately, it's quite possible to implement OAuth2 in a secure way—but the abundance of implementation options, which, if poorly assessed and

1- *https://www.wavestone.com/app/uploads/2016/10/Benchmark-Securite-Web-1.pdf*
2- *https://www.owasp.org/index.php/OWASP_Cheat_Sheet_Series*
3- *https://openid.net/specs/openid-connect-core-1_0.html*

selected, lead to critical flaws: the misuse of an application's identity, access to the personal data of a third-party user, theft of Facebook/Google cookies when logging in using social media accounts, or even the compromise of a user's account.

The recommandations below can be used to add an initial level of security to your implementation:

/ **Local storage of secret information:** The client application is provided with identifiers enabling it to authenticate itself with the OAuth server; so, don't put this secret information (the service identifier) in the mobile application; and, if you do, consider it compromised

/ **Redirected URIs:** Validate redirected URLs strictly with the application, without the use of wildcards

/ **Implicit:** Avoid OAuth2 "Implicit" flows, whose security is debatable (for example, tokens in the URLs can be present in the browser history), as well as the user experience it provides, for example during token expiry

/ **Authorization codes:** Validate authorization codes strictly: a code must be checked only once—and only by the client for whom it was intended.

/ **State and PKCE:** Use these protocol options to ensure the integrity of the entire series of process steps

/ **Authorization ≠ Authentication:** Use **OpenID Connect**[4] to authenticate, but OAuth to delegate access

LIMIT THE ADDITIVES

No sooner has this first pinch of OAuth been swallowed – a necessary step along the way – when questions begin to surface about very frequent use cases.

THE SINGLE SIGN-ON MOBILE... OR, HOW TO ALLOW MOBILE EMPLOYEES OR CLIENTS TO EASILY ACCESS MULTIPLE APPLICATIONS WITHOUT RE-AUTHENTICATING?

It might be a field agent in a customer-facing role, or making a series of interventions at different sites, all while using a good dozen applications every day; or it might be a client who's installed several applications on the public app store and needs to access them all, without having to reauthenticate on each...today, these are all very common scenarios. Since 2008, the techniques that make it possible have varied depending on the possibilities offered by the mobile OS (iOS's KeyChain, URL parameters, Mobile Device Management, etc.). Nevertheless, Apple and Google converged towards a common solution in 2015: the use of a browser system as an anchor point for an SSO session. This is now officially good practice, formalized in "Best Current Practice[4] - OAuth2 for native applications."

CONTEXTUAL AUTHENTICATION... OR, HOW TO ADAPT THE ACCESS LEVEL TO DATA ACCORDING TO ITS CRITICALITY

One of the many issues concerning authentication is to simplify, as much as possible, user access to data, while still guaranteeing satisfactory levels of security. Contextual authentication provides an answer to this issue, adapting the level of access to the nature of the transaction: its characteristics, user habits, context, and so on.

A mobile banking application, for example, allows the user to access their bank account, and see account balances, without having to reauthenticate each time these are accessed. However, the application will require authentication when performing a sensitive operation (transferring money between their own accounts, for example), and strong authentication when performing a very sensitive operation (adding an external recipient for a transfer, for example).

The market now offers solutions designed according to a logic where the application client is responsible for initiating the token request by specifying the level of authentication required. But the real need is to define and apply these data access policies in a single point within the authorization server. This is essential when there's a need to apply an authentication proportionate to the level of risk (geolocation, is it a known terminal or not, transaction habits, etc.). And it has to be said that the solutions available on the market today do not yet offer the required flexibility in this respect.

IDENTITY PROPAGATION... OR, HOW TO PASS AN ACCESS TOKEN BETWEEN TWO (OR MORE) APPLICATIONS.

It's increasingly common that a call to an API triggers a cascade of calls to other APIs, in particular within a micro-service-type architecture setting. The transmission of the identity of the user must then be assured while still maintaining security:

/ Transmitting a single token right along the chain is highly risky: the token has far too many rights, and may be misappropriated at any point in the chain

/ Checking the user's identity only at the beginning of the chain, and then authenticating just the services when transmitting it, is also risky: a compromised service could misuse the identity of any user

Besides, the rights (i.e. scopes) contained in the initial token may not match the rights required at every level of the chain of service calls.

4- *https://tools.ietf.org/html/draft-ietf-oauth-native-apps*

It is to address this use case, and the issue of the traceability of identity misuse, that a new grant type is currently being offered: Token Exchange[5]

Each intermediate caller can exchange the token received from the upstream service (which contains the identity of the latter and that of the user) against a token that can be transmitted to a downstream service (always supplying the identity of the user, the identities of the services through which the chain passes, and the rights required to call the service).

A major benefit of this new series of process steps is that it makes it possible to centralize the calls policy between micro-services, as well as the application of this same policy, thereby, ensuring the traceability of calls.

PROTECTING AGAINST TOKEN THEFT... OR, HOW TO GUARD AGAINST THE THEFT OF A TOKEN BASE?

Ever since the OAuth2 protocol was designed, the token it uses has been considered sufficient for access to a resource. Token theft is therefore a permanent threat which must be protected against.

Two approaches are possible:

/ **Try** to prevent such theft (by playing a game of ”cat and mouse“),

/ Or, **make** this token necessary, but not sufficient, to access an API.

This second approach, set out in the draft ”OAuth2 Token Binding[6],“ requires the application client to authenticate itself using a cryptographic key pair when generating the token, and to use the same key pair when using that token. The token and key pair are linked, and a stolen token without the client’s private key is, therefore, unusable.

WRITE DOWN AND SHARE THE RECIPE

What’s the last ingredient of the recipe? The need to set out a reference architecture for OAuth in order to adapt it to the context of the company’s IS. To do this, the API framework must be defined, by:

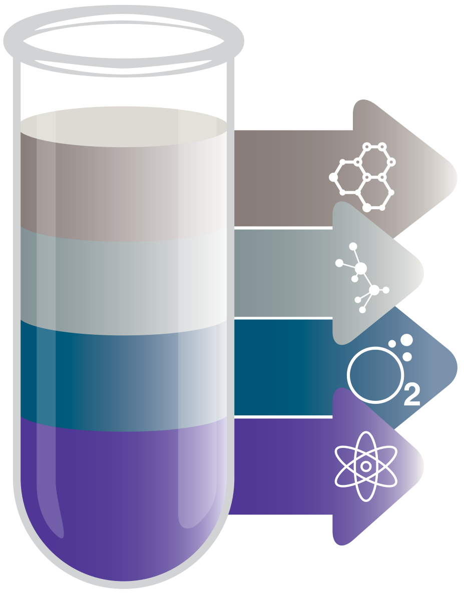
/ **Defining and sharing the security rules:** The authorized process steps and the application framework, the security checklists, and the reference architecture, must all be formalized.

/ **Training and equipping developers:** You’ll need to organize training sessions, and presentations on the principles to adopt. Project teams can be made autonomous in terms of their integration with the rest of the IS.

/ **Integrating security resources into agile sprints:** The resources that act as a ”security coach“ must be identified in order to support the application design, provide ready-to-use solutions, and serve as an accelerator.

/ In summary, rather like the recipe for a good broth, securing APIs requires a list of ingredients, ranging from the most basic to the most sophisticated, while keeping the needs and context firmly in mind. And above all, such work has to be a joint effort—towards a common goal!

The recipe for secured APIs



WRITE THE RECIPE

Use defined architecture and tailor it to the application context

LIMIT THE ADDITIVES

Ask yourself whether the ”typical measures“ are actually real needs

A PINCH OF OAUTH

Without falling into the potential traps its use can involve

ADOPT ”SECURITY AS USUAL“ AS THE BASELINE

After all... an API is a web application

WAVESTONE

www.wavestone.com

Wavestone is a consulting firm, created from the merger of Solucom and Kurt Salmon’s European Business (excluding retails and consumer goods outside of France). The firm is counted amongst the lead players in European independent consulting.

Wavestone’s mission is to enlighten and guide their clients in their most critical decisions, drawing on functional, sectoral and technological expertise.

5- *https://tools.ietf.org/html/draft-ietf-oauth-token-exchange*
6- *https://tools.ietf.org/html/draft-ietf-oauth-token-binding*