https://doi.org/10.52326/ic-ecco.2021/SEC.03

Information Security in Microservices Architectures

Bazeniuc Ivan [1], Zgureanu Aureliu [2]
[1] ASEM, 61 Banulescu-Bodoni str., MD-2005, Chişinău, bazeniuc.i@gmail.com
[2] ASEM, 61 Banulescu-Bodoni str., MD-2005, Chişinău, zgureanu.aureliu@ase.md

***Abstract* — In this paper we investigate how can be provided security of an information system, which uses a microservice architecture. So, using of microservice architecture means that information system can be easily developed, deployed, and tested, but, on the other hand it means that it should be protected differently than the information system using monolith architecture. Firstly, a software architect should decide if each service should be protected separately or should be better to protect the system on the whole. Choosing the right way of protecting is very important, because, in some cases, protection of each service separately is not the best idea, as this could lead to code duplication. This means that, in case of necessity of changes a piece of code or fixing a bug it should be done everywhere this code appears. To avoid this, there are developed some services - so called gateways, which, also, very often have implemented the function of user/client authentication and authorization using protocol OAuth 2.0. Anyway, at each stage of development and implementation of a software product, it is necessary to solve many security related problems, and if it will not be done properly, then the company may incur enormous material losses or even may be closed.**

Keywords — microservice; OAuth 2.0 protocol; cloud infrastructure; information system security; gateway; software architecture

I. Introduction

Information is a product that can be bought, sold, or exchanged, and very often the cost of the data that is stored in the system exceeds the cost of the information system itself. In addition, information systems can store information with personal data of people, the leakage of which can negatively affect not only the company's image and its economic performance, but also ordinary people, data about which were obtained by unauthorized client. It is difficult to predict how such data might be used. In the best case, the data will not be used at all or will be used, for example, for targeted advertising. In the worst case, a person may be subject to blackmail or lose money from their accounts. Anyway, the data must be carefully protected.

Protecting information systems using a microservice architecture differs from protecting a monolithic architecture. A feature of the microservice architecture is that such systems are divided into tens or hundreds of small services, each of which must perform its function and, respectively, must be protected [1]. In addition, each of these services can be used by different clients: a browser application, mobile applications (with different operating systems), as well as applications written by third-party developers, etc. Often different clients need different data. Mobile app can reflect less data than desktop user application, and some functions may be hidden altogether.

Firstly, will be identified which issues exist on direct access to the service. After it will be described the characteristics of API gateways, its advantages and disadvantages, and an example of how to implement such edge functions as authentication and authorization using the OAuth 2.0 protocol.

II. Issues of Directly Accessing Services

As it was mentioned, each of the services can be used by different clients. One of the options for designing an information system is that clients access services directly through the Internet (or, less commonly, through a local area network), as is shown in the Figure 1.

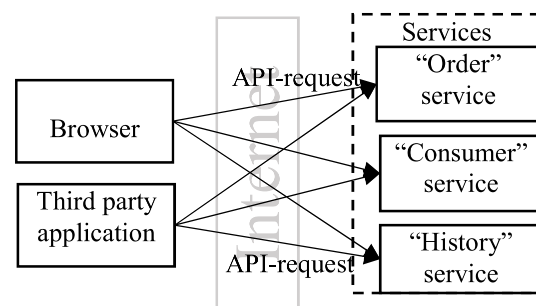

Figure 1. An example of a scheme of requests to services directly from different clients.

https://doi.org/10.52326/ic-ecco.2021/SEC.03

At first glance, this sounds pretty straightforward - after all, this is how clients call services in monolithic applications. However, this approach is rarely used in a microservice architecture because it has the following disadvantages:

1. To retrieve the desired data using finely divided services, clients will need to make several requests, which is not efficient and convenient. Too much communication between the application and services can adversely affect the responsiveness of the application, especially if it passes over the Internet. Perhaps, in some cases, requests can be executed in parallel and, in this case, the total time of all requests will not be more than in the case of one request. But sometimes the requests have to be executed sequentially, which reduces the usability of the client. Also, developers of client applications have to write rather complex code to combine services, which, moreover, may not work well on weak devices, but also distracts the developer from their main task - from creating convenient user interfaces. If the user uses it from a mobile device, then for each of his network requests is consumed electricity, which drains the battery faster.

2. Large applications with many users can be deployed on several servers in order to reduce the waiting time for a service response. In this case, there will be several threads of execution that the client should know about, and moreover, the client should know which thread to send the request to.

3. Each non-public service must have a security filter that will check whether the user has the right to make a request and what information the user can receive in response to the request. Consequently, the need for any change in one of these filters often leads to the fact that such a change must be made in each service the number of which, for one information system, can reach tens or hundreds.

4. As the information system evolves, service developers sometimes change the API and their endpoints, disrupting the work of existing clients. This is due to what is known as insufficient encapsulation. Developers can add new services or split/merge existing ones, and if information about services or endpoints is embedded in the client application, changing them can be difficult. Unlike updating services, deploying a new version of a client application can take hours or even days. For example, an update for a mobile app must first be approved by a corporation such as Apple or Google, depending on the app store, and made available for download. At the same time, no one guarantees that users will download it immediately (or even ever).

5. A separate problem is the fact that many organizations provide their services to third-party developers to integrate third-party information systems or applications with the organization's information system. Based on this, third-party developers need a stable interface. When a new version of a service comes out, it is necessary to get third-party developers to use it, but very few organizations succeed. If an app's API becomes unstable, third-party customers may stop supporting it and move on to competitors. This means that the development of APIs that are used by other organizations must be carefully considered. This usually requires maintaining old versions for a long time, or even keeping them forever, which is a huge burden on the organization.

Instead of giving customers direct access to services, organizations often have a separate public API that a separate team develops. Further will be described the public architectural API component, better known as the API-gateway.

III. API GATEWAY

A. API Gateway Characteristic

An API gateway is a service that serves as an entry point to an application from the outside world. This means that the integration of all APIs over the Internet is no longer at the client level, but at the service or back-end level. It is responsible for routing requests and for some third-party functionality such as authentication. Thanks to the API gateway, the client does not need to make many requests to services, but only needs to make one request to the service, which serves as a single-entry point for API requests in the application. An example of a scheme of requests through the API gateway is shown in the Figure 2.

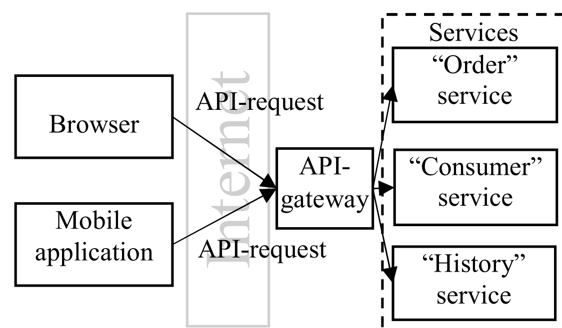

Figure 2. An example of a scheme of requests to services through an API gateway from different clients.

So, all requests made by external clients first go to the API gateway, which routes them to the appropriate services. The API gateway uses API aggregation to process other requests, accessing different services and aggregating the results.

Perhaps a few words should be said about another key function of the API gateway - request routing. Some API calls are implemented by directing requests to appropriate services. When the API gateway receives a request, it checks the routing map to determine which service to route the request to.

Of course, the main responsibilities of an API gateway are routing and API bundling, but it can also take over the implementation of edge functions. An edge function, as

https://doi.org/10.52326/ic-ecco.2021/SEC.03

the name implies, is a request processing operation at the application boundary. Examples include:

- authorization - checking that the client is allowed to perform a certain operation;
- authentication - checking the authenticity of the client making the request;
- limiting the frequency of requests - control over how many requests per second a certain client and/or all clients together can execute;
- caching - responses to reduce the number of requests to services;
- query logging - writing queries to the log;

metrics collection - collection of API usage metrics for analysis.

B. Advantages and Disadvantages of API Gateways

The API gateway pattern has many positive aspects, but it is clear that there are no perfect technologies, and therefore the disadvantages of a gateway should also be known.

The big advantage of using an API gateway is that it encapsulates the internal structure of an application. Instead of calling certain services, clients only need to communicate with the gateway. Each client receives a separate API, which in turn communicates with the services, which reduces the number of requests between the front-end and back-end. It also greatly simplifies the client code.

On the downside, the API gateway will be another component that needs to be developed, deployed, and administered. In addition, there is a risk that the API gateway will slow down the development of the information system. It should be updated with every new service deployed.

IV. OAuth 2.0 Protocol

A. OAuth 2.0 Characteristic

As discussed above, an API gateway can implement edge functionalities. Two of these functionalities - authentication and authorization - are directly related to information system data security and to user data security also. To implement them, we can use, for example, the OAuth 2.0 protocol.

So, the OAuth 2.0 protocol is an open authorization protocol (scheme) that allows a third party to provide limited access to a user's protected resources without the need to transfer to it (a third party) a login and password [3]. It works by delegating user authentication to the platform on which the user's account resides, allowing a third-party application to access the user's account.

For example, there is a certain information system that specializes in trading on the Internet and the management of which, in order to increase sales, decided to resort to the implementation of the OAuth 2.0 protocol. For the end user, as a result, this decision will mean:

- there is no longer necessity to register in this information system, and instead, you can use other services, such as, for example, Facebook, Twitter or Gmail, which may have a better protection degree;
- access to any data will be provided only after the user's consent, otherwise the information system will not be able to use the user's personal data.

This solution will bring impressive advantages for the information system itself, because:

- there is no longer necessity to save passwords or other personal data in the information system database if the user prefers authentication on a third-party service (Facebook, Twitter, Gmail, etc.);
- with the consent of the user, the information system will be able to access some of the user's personal data (name, surname, date of birth, etc.), on the basis of which it will be able to make recommendations for goods.

So, Protocol OAuth 2.0 defines four main roles:

- *Owner of the resource*. The owner of the resource is the user who authorizes the application to access his account. The application's access to the user account is limited to the “scope” of the authorization rights granted (for example, read or write access).
- *Resource server*. The resource server directly stores the protected data of user accounts and often also acts as an authorization server.
- *Authorization server*. The authorization server verifies the authenticity of the information provided by the user, and then creates authorization tokens for the application, through which the application will access the user data.
- *Client*. The client is the application that wants to access the user's account. Before being accessed, the application must be authorized by the user, and it must be approved by the API.

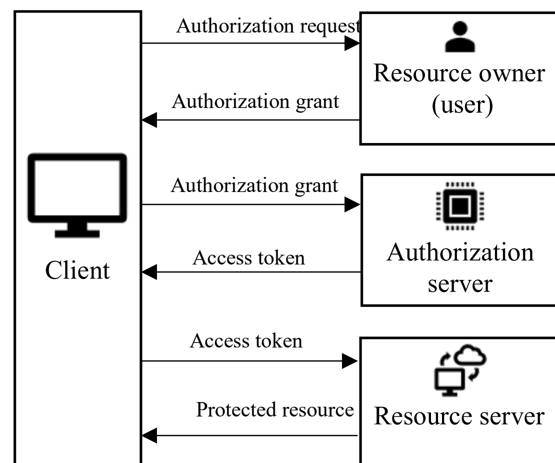

Figure 3. Client request scheme for granting access to a user to his protected resources.

Below it is described step-by-step the process presented in the Figure 3:

https://doi.org/10.52326/ic-ecco.2021/SEC.03

1. The application asks the user for authorization to access the resource server.
2. If the user authorizes the request, the application receives an authorization grant.
3. The application requests an authorization token from the authorization server by providing information about itself and authorization from the user.
4. If the application is authenticated and the authorization permission is valid, the authorization server generates an access token for the application. The authorization process is complete.
5. The application requests a resource from the resource server, while providing an access token for authentication.
6. If the token is valid or, for example, it has not expired yet, the resource server provides the requested resource to the application.

The actual order of the steps in the described process may differ depending on the type of authorization you are using, but the overall process will look like this. We will discuss in more detail about the different types of authorization permissions in section *B*.

So, the whole process described above means that every request sent to the API gateway must be accompanied by an access token. An example of an access token string is presented below: "*eyJpc3MiOiJodHRwOi8vZ2FsYXhpZXMuY29tIiwiZXhwIjoxMzAwODE5MzgwLCJzY29wZXMiOlsiZXhwbG9yZXIiLCJzb2xhci1oYXJ2ZXN0ZXIiXSwic3ViIjoic3RhbmxleUBhbmRyb21lZGEuY29tIn0*".

It would be difficult to counterfeit such a token. In addition to all these, the token is often subjected to symmetric encryption, and when it enters the API gateway, it is decrypted and checked to see if it is valid. If the token is valid, then the request is redirected to services.

B. *Authorization Permissions*

In the previous section, the first four steps in the process of granting user access to his protected resources deal with the issue of creating an authorization permission and an access token. The type of authorization permission depends on the method used by the application to request authorization, as well as what types of permission are supported on the server side. There are four different types, each of which is useful in specific situations:

- *Authorization Code*: is one of the most common type of authorization permission, because it perfectly fits for multilayered architecture applications, that is, where the application source code and client secret are not available to outsiders. This type is divided into two main parts, that is, user requests for authorization will be made to the authorization endpoint, and, after successful authorization, requests for the token endpoint will be made on the server side of the application. Thus, for information systems using a microservice architecture, this type is likely to be ideal;
- *Implicit*: used by applications where client secret confidentiality cannot be guaranteed. This type is most often used when the application does not have a backend. Due to the lack of a backend, all authorization requests will be made only to the authorization endpoint;
- *Resource Owner Password Credentials*: with this type of authorization permission, user provides the application with his authentication data in the service (login and password). The application, in turn, uses the received user credentials to obtain an access token from the service. This type of permission should only be used when the user has trust in the application or, for example, by applications that are part of the service itself.
- *Client Credentials*: are used when the application accesses the server, that is, user authentication data is not used at all. Unlike the previous types of authorization permission, in this type, the token that will be created will not contain any user data at all. It should be said that this type differs from others also in that it does not have an authorization endpoint, but only the token endpoint is used. This type is very convenient to use when we need to perform operations in which the user is not directly involved. An example of such an operation can be a listener in the information system of an organization, which checks daily if an employee was hired or fired (in other words, for example, if an account was created / deleted on the Outlook service). This type can be used in conjunction with other types of authorization permission.

CONCLUSIONS

Nowadays, when the world is closely associated with information technology and, information is a key resource, data security is becoming one of the key aspects in the activity of each enterprise.

Today, microservices architecture is one of the most popular architecture on developing an informational system, and more and more often big organizations uses this type of architecture. However, as mentioned earlier, there are no ideal technologies, so information systems using a microservice architecture must also be protected, that can be done by implementing an API component such as an API gateway.

In turn, an excellent decision when implementing an API gateway will be the implementation of such a edge function as authentication and authorization, which is directly related to the security of the information system. It is also necessary to assess the necesety to store some sensitive data in the information system database, or it is better to resort to implementing the OAuth 2.0 protocol.

REFERENCES

[1] Microservices documentation. https://microservices.io/index.html
[2] Chris Richardson, "Microservices Patterns", 1st Edition, ISBN: 978-1617294549, Publisher: Manning Publications, 520 p., 2018.
[3] OAuth Working Group Specifications (https://oauth.net/specs/