Fine-Grained Access Control for Microservices

Antonio Nehme(✉), Vitor Jesus(✉), Khaled Mahbub(✉), and Ali Abdallah(✉)

School of Computing and Digital Technologies, Birmingham City University, Birmingham, UK
{antonio.nehme, vitor.jesus, khaled.mahbub, ali.abdallah}@bcu.ac.uk

Abstract. Microservices-based applications are considered to be a promising paradigm for building large-scale digital systems due to their flexibility, scalability, and agility of development. To achieve the adoption of digital services, applications holding personal data must be secure while giving end-users as much control as possible. On the other hand, for software developers, the adoption of a security solution for microservices requires it to be easily adaptable to the application context and requirements while fully exploiting reusability of security components. This paper proposes a solution that targets key security challenges of microservice-based applications. Our approach relies on a coordination of security components, and offers a fine-grained access control in order to minimise the risks of token theft, session manipulation, and a malicious insider; it also renders the system resilient against confused deputy attacks. This solution is based on a combination of OAuth 2 and XACML open standards, and achieved through reusable security components integrated with microservices.

Keywords: Microservices · Security · Confused deputy attack · Gateways · Access control

1 Introduction

Enterprise applications nowadays require using multiple, distributed and multi-owner components. While Service Oriented Architecture has been adopted for over a decade, its underlying model is now proving complex to manage given its tendency to a small number of large and complex components, referred to as monolithic applications ("monoliths") [4]. To ensure better software maintainability, faster development and deployment, and a more efficient scalability, microservice architecture is gaining popularity [22]. With Microservices, monolithic applications are replaced by a large number of loosely coupled components, yet each small and easy to maintain. By definition, microservices need to have a small role and should be designed to communicate with other services over a network [4] in a distributed fashion. Compared to monoliths, the considerable number of independent services renders enforcing security solutions and verifying every request's authenticity much more challenging.

Moreover, Microservices do introduce coordination complexity which, in turn, creates new security risks. This brings forward trust challenges as, effectively, every

N. Zincir-Heywood et al. (Eds.): FPS 2018, LNCS 11358, pp. 285–300, 2019.
https://doi.org/10.1007/978-3-030-18419-3_19

microservice is an independent party that, in the extreme case, cannot be trusted [22]. In particular, distributed architectures create access control problems such as the so-called *confused deputy* attacks and the use of *powerful tokens.* A *confused deputy*, referred to as the 'vulnerability du jour' [7], is a privilege escalation attack in which a microservice that is trusted by other microservices is compromised; this results in the trustees responding to the compromised microservice requests, not knowing that it is acting on behalf of the attacker [13]. *Powerful tokens*, in turn, result from the fact that, typically, one valid authorisation token is enough to have access to every microservice since requests pass through a gateway (the orchestrator) that can access all the system services with that access token. These are normally Open Authorization (OAuth) tokens that are created through one OAuth client, and their theft leads to data exposure at the level of every microservice [1].

The context of this paper is user-centred services that are multi-party and inter-domain. In particular, we consider scenarios where multiple parties are requesting access to personal data or assets; the data exchange process should be transparent to and controlled by the data owners. One example of such systems is a digital government portal: multiple administrations, that are nevertheless independent and segregated, have to coordinate to provide services for citizens, and citizens need to ensure that their data is safe, while being aware of how this data is being used, and what is being processed; on the other hand, each administration is responsible for protecting the citizens' data, and of correctly performing its role. Our solution for these requirements applies to microservice-based systems with access to sensitive data in different administrative domains.

This paper proposes a security solution for Microservices that enables fine-grained access-control policies to be deployed, thus mitigating several problems while giving the user control over their requests. Beyond globally validating a token at the entrance (the Gateway interfacing the user or another external application), we propose that each service has its own local Gateway that validates highly-descriptive and fine-grained tokens. These tokens are centrally generated, short-lived and have a narrow access scope. Additionally, these gateways include security checks that reveal and mitigate potential malicious activities, like data theft from government departments or tampering with government digital services, through a compromised microservice in one department. Furthermore, to enable scalability and reusability, we propose that these gateways are configurable and reuse security components that get added to microservices templates outside their core functionalities, and can scale with them when needed. Our solution is based on OAuth 2 and eXtensible Access Control Markup Language (XACML) open standards. To summarize, our architecture requires a user to explicitly allow actions from the multiple services engaged and belonging to different parties, while confining permissions of the services with pre-defined policies that all parties agree on.

The structure of this paper is as follows. The next section reviews related work and Sect. 3 describes the problem. In Sect. 4 we describe our solution, followed by an analysis in Sect. 5. Section 6 discusses our implementation, Sect. 6.1 shows the experimental results, and we conclude the paper in Sect. 7 with some future directions for our work.

2 Related Work

Many approaches found in the literature rely on *powerful tokens* strategy, i.e. one access token giving access to all the system's components, for access control. This results from using one OAuth client for a microservice-based application: [12] is an example of an implementation where powerful tokens are being used, and [3, 5, 6, 15] also point out to using similar approaches in their systems. OAuth token theft has been approached in literature. Ahmad et al. [2] used ID and OAuth tokens to minimise the possibility of token theft; however, the combination only reduces the chances of a successful attack and does not protect against powerful tokens theft in the service-to-service communication. Security architectures, [18] for example, recommend using standard mechanisms like OAuth 2 and XACML for API protection. XACML and OAuth 2 are discussed separately in [9, 14], and Suzic [17] mentioned the possibility of combining the two standards; however, the combination was not detailed or applied by any of them. Zhang et al. [20] based their implementation on this combination; however, their solution targets a specific use case that is not applicable to microservices. The confused deputy is another possible attack. Härtig et al. [7] call for tools to detect this attack; our work directly addresses that. Finally, work on a new OAuth grant type, Token Exchange [10], still in progress, tackles a similar problem as this paper. It is equally tailored for microservices in which the authorization server is in charge of policy decisions based on the identity of users, calling and called services, predefined action and access rules.

In short, to the best of our knowledge, this is the first attempt for designing a reusable and user-centric Identity and Access Management (IAM) security solution for primitive (only implementing functional requirements) microservices that mitigates token theft and the confused deputy problem. The reusability and configurability of our solution render it scalable and adaptable in agile Microservice Architecture (MSA) systems.

3 Problem Statement

This section presents a scenario to illustrate our security requirements. We then show our threat model for a Microservice-based system, give an overview of the principles that we are abiding by, the inadequacy of most used approaches and their common vulnerabilities, and a rationale for our design decisions.

To illustrate, we consider a digital government scenario of applying for a passport at the Department of State. The applicant needs to be a citizen to be eligible to apply for the passport service. The user logs in to a central portal, and selects the passport service; by logging in, the portal fetches the required information for access control: the citizenship status in this example. The passport service asks for further identity information required from the Department of Interior Affairs, and other data attributes from the Department of Justice to show a clean record; these attributes are already agreed on between the departments. The user needs to approve on the personal data attributes that will be shared between departments, and an access token will be produced for each consent. Each token only serves to access one specific service of one department.

3.1 Threat Model

In this threat model, we assume that traditional inter-domain security mechanisms, including intrusion detection and prevention systems, firewalls, input validation, mutual TLS authentication and encryption are placed between different security domains. We trust these security mechanisms, and that the authentication and authorisation servers are not compromised, but not the application microservices.

These microservices, and the Virtual machines (containers) which they run on, can be under the control of an attacker, or even abused by a privileged insider. This gives the adversary the ability to intercept requests and responses, steal and manipulate tokens by replacing a token belonging to a user with another, and send requests from the compromised microservice. A compromised microservice cannot generate a new access token without the user's consent on the list of scopes; this happens with a redirection to the OAuth server. Access Token theft can happen at the level of any compromised microservice, or by an insider monitoring local traffic.

3.2 Security Requirements

Considering the scenario detailed in Sect. 3, our approach uses the following as requirements:

- R1: Access policies are needed to control which services a user can access.
- R2: Every personal data attribute at each department needs user consent to be shared with another department.
- R3: Departments only share data following pre-defined and verifiable agreements with service consumers.
- R4: An access token should only serve to access the assets of a user exposed by a single service in one department.

Where the corresponding security goals are:

- R1 requires fine-grained access policies, that must relate to the (micro-)service itself
- R2 separates control between user and service providers by allowing administrative policies on a per-service basis
- R3 verifies the authenticity of consumers and limits insiders malicious activities
- R4 protects against Powerful Token and Confused Deputy attacks.

3.3 Decoupling Security from Functional Requirements

A further requirement is to decouple the control of the microservice from the service itself. We approach this by designing our architecture using reusable and configurable gateways at the level of each microservice. These components can be added to secure primitive services, and modified to meet different policies. Figure 1 shows a primitive Resource Microservice (RMS) protected by a local Gateway (GW). In order for a request to reach the RMS, security policies enforced by GW have to be met by the requesting service or party (the consumer microservice); note that the consumer microservice should have another gateway to enforce access control policies. The Resource Microservice (RMS), which encapsulates only the primitive functionality, is

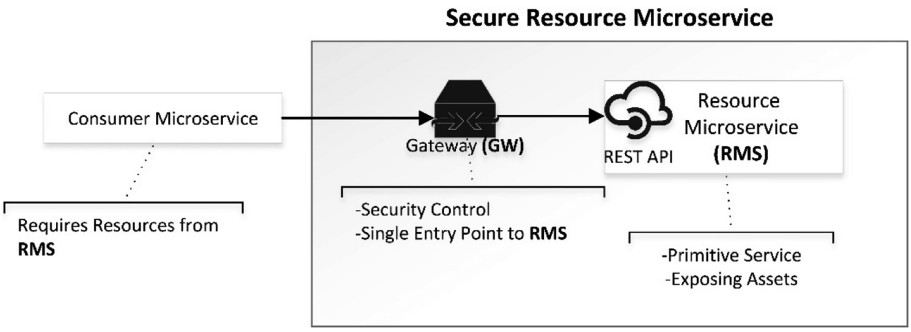

Fig. 1. Gateway to secure primitive services

thus released from the verification logic and only manages the assets themselves (such as personal data).

A reusable security solution placed around services provides better consistency, simplicity, and portability [21]; adaptability and flexibility are essential requirements to follow. For different scenarios, a variety of attributes have to be considered when designing security solutions, and a trade-off has to be made between multiple variables including performance, security tightness, user-friendliness, and ease and flexibility of management.

3.4 Limitations of Current Practices

Open Authorization 2 (OAuth 2) is one of the most commonly used mechanisms in a microservice architecture for access delegation. OAuth 2 access scopes are used to define the token holder's access rights. However, the standard only gives the ability to define static, normally coarse-grained scopes, and does not provide any support for auditing and flexible policy enforcement [16, 17]. OpenID Connect, built on top of OAuth 2, is commonly used for authentication with MSA [12]; it is an enabler for identity federation by producing an ID token with end-user information, and a practice of the separation of concerns principle. Nevertheless, these approaches are not particularly suitable for MSA due to their large attack surface in such a fine-grained architecture [4]. These approaches normally rely on a single token that is used to access all parts of the system resulting in several problems, Powerful Token Theft being the most obvious [1, 2]. With this approach, any service having access to a session with a valid token can make requests to other components on behalf of the user [12].

On the other hand, we have the confused deputy problem. As explained, this consists of a component which has access to sensitive resources, and which can be manipulated by an adversary to have an indirect access to these resources [13]. In essence, the confused deputy attack arises from trusting a component based on mere identity information such as the component's IP address or an ID token [11]; in our scenario, presented in Sect. 3, the passport service is a potential confused deputy. The key point to prevent this is to have the resource services, the department of Justice and of Interior Affairs microservices in our scenario, verify that the calling microservice is

acting truthfully on behalf of the user. This requires, for example, tokens to be individual to each component, and have finer granularity reflecting users' consents on access rules.

4 An Access Control Solution for Microservices

Figure 2 represents our approach for access control between consumer and resource microservices. This solution is built on a combination of XACML for administrative and OAuth 2 for user-defined policies. The architecture involves an Access Control Server (ACS) acting as an OAuth 2 and XACML server, consumer microservices (CMS) containing OAuth 2 client credentials and requiring access to resources, Resource Microservices (RMS) hosting and exposing assets, and a Gateway (GW) to secure each microservice.

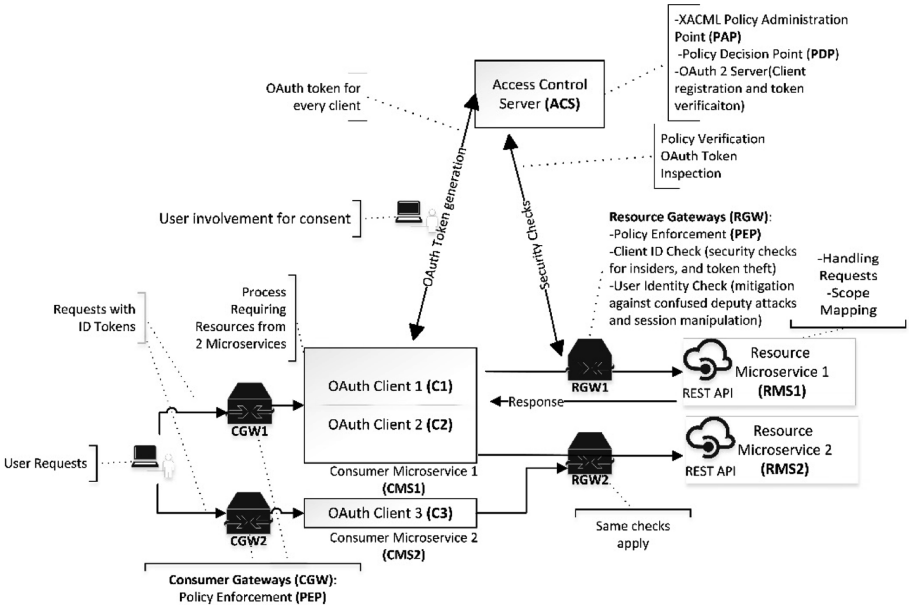

Fig. 2. Overview of our security architecture: Gateways for security enforcement, and an OAuth client per consumer-resource.

A request to CMS requires an ID token, generated by ACS when the user logs in, from the authentication session to verify the access rights of the user; to request resources from RMS, it also needs to generate an OAuth 2 token by having the user consent on the access scopes. As shown in Fig. 2, RGW1 and RGW2 are gateways to the resource microservices RMS1 and RMS2; consumer microservices also have a gateway each, CGW1 and CGW2, to enforce administrative access control policies. Typically, a central gateway in MSA sits in front of all services and can take different

roles ranging from a simple address forwarder to an orchestrator. In our architecture, each microservice has its own gateway that gathers security and control functions, and is minimally dependent of the microservice; this makes it reusable as a configurable component across different microservices. Note that a central gateway is still present as a single entry point to the administrative and security domain to provide conventional network security services such as intrusion detection and prevention, firewalls, input validation, mutual TLS authentication or encryption.

The key functionality of a gateway per microservice is becoming a single entry point to each microservice that, while being fairly agnostic to the service itself, is able to validate the authenticity of the incoming requests. Our implementation of these gateways include other mechanisms for security assurance, policy enforcement, token theft detection, auditing and incident reporting; these serve to minimise blind trust between services, and therefore limit the effect of a successful confused deputy attack. The details for these checks and the requests flow of requests are explained in the next parts of this section.

4.1 A Fine-Grained Access Control

XACML is used to create access control policies. These define whether a user can interface a particular microservice. Policies are directly enforced by the GWs, each acting as a Policy Enforcement Point (PEP). The PEP component of the GW checks the user's identifier by inspecting the user's ID token in the authentication session. The ACS is the Policy Decision Point (PDP) and determines if this user is authorized to access a microservice endpoint to make a particular request. In the case of resource microservices, the request goes through other security checks discussed in Sect. 4.2.

OAuth 2 is used for users to delegate access to part of their protected data, residing at a resource microservice, to a consumer microservice. We use OAuth 2 to produce a token that maps to access scopes; these scopes are indicators for what the token gives access to. Being part of the token, scopes are used by RMS to share only the data that the owner has given consent for. Our proposal includes creating an OAuth client for every pair of consumer-resource microservices, to allow the generation of verifiable tokens with access scopes tailored for the combination.

Consider Fig. 2 OAuth clients C1 and C2 are used to send requests from the consumer microservice CMS1 to two different resource services, RMS1 and RMS2 respectively, exposing user's data. Although our approach gives the flexibility of using one OAuth client for multiple microservices, we recommend one OAuth client per consumer-resource microservices to limit the power of access tokens. Also, a microservice can receive requests from more than one consumer service as shown with RMS2 receiving requests from both CMS1 and CMS2. OAuth client creation is always done at the ACS level following the OAuth 2 common practice. Scopes are defined during creation, and client credentials (a unique identifier and a password) are generated to be used by consumer microservices for access tokens production.

4.2 Proposed Security Checks

For each request from microservices to access resources of another, an OAuth access token needs to be provided, alongside the ID token in the authentication session, by the sender of the request. The ID token is inspected by the GW of every microservice (Consumer or Resource) to verify the eligibility of access of the user, and the OAuth token is to be inspected by the RGW of the resource microservice before the request gets through to the RMS. This gateway sends the access token to an endpoint of the ACS to verify its authenticity and retrieve the information mapped to it.

To illustrate with Fig. 2, if a service CMS1 needs some of the user's personal information from RMS1, CMS1 uses C1's client credentials to produce an OAuth access token following OAuth 2 common practice. The user is required to choose the access scopes and confirm access for the OAuth token to be produced for CMS1. An access request is sent from service CMS1 to RMS1. RMS1, through its gateway RGW1, uses the token inspection endpoint of ACS to verify the authenticity of the access token and to decode it. The token would have a reference to the OAuth client ID, token scopes, the subject (user) identifier, and an expiry date. Given that the token is authentic and valid, RGW1 would perform the following security checks:

1. 'User Identity Check' by verifying that the user in the ID Token (the user that authenticated to the portal) is the same as the subject of the token
2. 'Client ID Check', by checking C1's client ID against a set of authorized client IDs to access the service

The first check reveals tokens' theft and manipulation attempts, and the second diminishes a token's power, and limits blind trust between components. If these checks pass, the gateway forwards the token information to the microservice; otherwise, the request is denied and the incidence gets reported. If the gateway lets the request through to the resource microservice, the latter returns the attributes of the user mapped to the scopes of the access token.

4.3 Operational Flow

The sequence diagram in Fig. 3 shows a representative example of our proposal. This shows the dataflow of an operation between a CMS and RMS of Fig. 2; it also reflects an access request between the passport microservice and one of the resource microservices in the scenario presented in Sect. 3. One service, the CMS, is to retrieve resources from another service, the RMS. A central ACS is used as an OAuth 2 authorization server, as well as an XACML server with policy administration and decision points. The ACS can include or be linked to an authentication server that produces and keeps track of authentication sessions with ID tokens. Each gateway (CGW and RGW) functions as a PEP which inspects the ID token, and uses the ACS PDP to check the policy rules. Access rules can be defined as a set of URLs and actions mapped to a group of users (i.e. Role-based); however, more complex policies can be defined following any policy definition criteria.

Before any attempt to access CMS, the user has to have an active session with an ID token. When the user sends a request, the PEP at CGW inspects the user's ID token

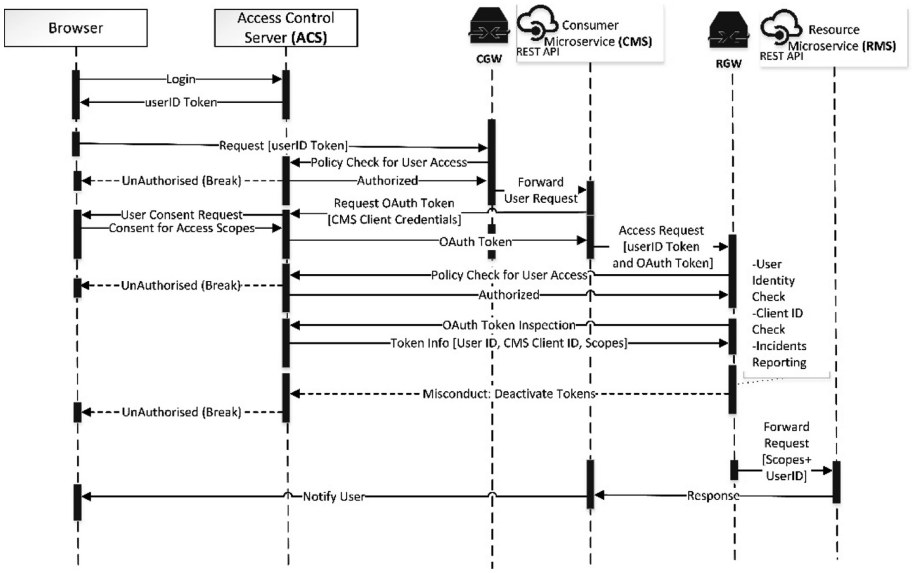

Fig. 3. Sequence diagram representing a service-to-service interaction.

with the Policy Decision Point of ACS, and if the action with CMS is allowed, this user is able to initiate a request with the service. When CMS requires data/service from an external resource (RMS), it first needs to request an OAuth access token. CMS uses its OAuth credentials, specific for RMS, to initiate the token production request with ACS. In turn, ACS requires the user to be authenticated and to choose the access scopes. At the level of ACS, an Intrusion Detection System can detect session manipulation attempts between the last two interactions with it. The produced access token is sent to CMS, and a request with the ID and OAuth tokens in the header is sent to RGW. RGW, protecting the resource microservice, checks if the user is authorized to access the service that it protects and, if so, the OAuth token is sent to the OAuth token inspection endpoint of ACS. This token gets verified, decoded, and sent back to RGW to perform the User Identifier and Client ID Checks described in Sect. 4.2. If any of the previous checks fails, an appropriate alert will be sent to the system administration and the user session and access token get deactivated. If all conditions are met, RGW sends the request with the user ID and the access scopes to RMS. This service has now the data attributes and/or methods mapped to the token scopes and the data of the user will now be sent to CMS.

5 Analysis

We now revisit the early requirements listed in Sect. 3 and discuss how our proposal addresses them.

5.1 Fine-Grained Access Control

With PEPs used at each microservice gateway level, access policies allow defining access roles for users to particular services (R1). Gateways here keep any unnecessary potential load off the microservices and act as a further defence layer. Since XACML allows to define complex policies, one can further add contextual access rules such as time and location.

Having multiple OAuth 2 clients helps to enforce transparency in the system by requiring users' consent for each access operation to their personal data, and giving them the option to choose what they want to share. Scopes are defined during OAuth 2 client creation following agreements between the resource and consumer microservices departments (R3), and having an OAuth client per consumer-resource microservice enables a fine-grained user-centred access control at the level of microservices (R2). Scope to resource mapping is done at the RMS level, and having scopes tailored to each service gives the transparency needed for systems in which privacy is key to users' trust.

5.2 Token Theft Mitigation

Having multiple OAuth 2 clients, for different consumer-resource combinations, limits the power of access tokens. With one OAuth 2 token per access task, a stolen token would only be a threat to the data of a particular person in one microservice only. These tokens can have a short lifespan since they are meant to be used once and for one particular request. Also, due to the User Identity Check at the gateway level, access to information from a stolen access token is not possible without access to the ID token of the same user. Any attempt from a conflicting user session would result in deactivating the tokens and reporting the incidence; even session hijacking can be rendered ineffective with a stolen token's short lifespan. Also, the Client ID Check diminishes the token's power, by limiting the services that accept the token. This partially fulfils the security goal of R4.

5.3 Confused Deputy Mitigation

Going back to Fig. 2, a token produced with C1, belonging to CMS1 and valid for RMS1, would not be valid for RMS2. This is also valid if service CMS1 is allowed to access both services RMS1 and RMS2, and even if RMS1 and RMS2 belong to the same department (R4). The combination of the User Identity Check, the Client ID Check, and requiring user consent for every service to service data access is a mitigation against the confused deputy attack. These security checks and practices minimise trust between services and give an assurance that a service is acting faithfully on behalf of the user. Therefore, this solution achieves the security goal of R4.

On a related note, our design mitigates some malicious insiders' activities. According to an IBM report in 2015, 60% of attacks are due to an insider [8]. If she or he manages to create an OAuth client on ACS to be used by a malicious node, the resource microservices would not accept any access token from this new client since its

ID is not in the list of trusted clients of any RGW. This approach minimises the possibility of having a service confused with a rogue/fake client (R3).

5.4 Manageability and Reusability

To help manageability, categorising services into groups, according to their security requirements is likely necessary. These requirements are decided based on the functionality of the microservices, the criticality of assets, and the trust context. This is a common approach for large enterprise software to protect their resources [19]. In our scenario, we separated consumer from resource microservices and required different gateways for each; other microservices may require encrypting their data at rest and on exchange for example. Having reusable security components helps to define configurations with security functions to meet different requirements; this facilitates securing new primitive microservices by plugging in these predefined gateways. Security gateways are extensible and can include other security functionalities including, but not limited to, logging and auditing, cryptographic roles, and throttling. However, these are out of the scope of this paper.

6 Implementation

We implemented a proof of concept using ForgeRock[1] open source components. ForgeRock Access Management (AM[2]) is used as the central access control server (ACS) for its ability to manage authentication, OAuth access delegation and XACML policies. As for microservices local gateways, ForgeRock Identity-Gateway (IG[3]) is used due to its Policy Enforcement and OAuth 2 token validation filters, and the flexibility that it provides to extend its functionality. This solution is feasible using any technological stack implementing OAuth 2 and XACML; a gateway can be written with any programming language that supports XACML, HTTPS calls, and the implementation of our proposed security checks. For the sake of clarity of this demonstration, we have used Postman[4] to play the role of a consumer microservice with an ID token, accessed by the authentication cookie, and an OAuth 2 token, sending an access request to an RMS protected behind an IG. This shows the same behaviour of a consumer-to-resource microservice call, with the resource microservice protected by RGW.

Figure 4 shows the response of an RGW on a failed User Identification Check. This is one approach to detect session hijacking and OAuth token theft. Both tokens would be deactivated in this case.

Figure 5 shows a request sent to RMS from an unauthorized OAuth client; this reflects the response of using an access token for a different consumer-resource

[1] https://www.forgerock.com/platform/.

[2] https://www.forgerock.com/platform/access-management.

[3] https://www.forgerock.com/platform/identity-gateway.

[4] https://www.getpostman.com/.

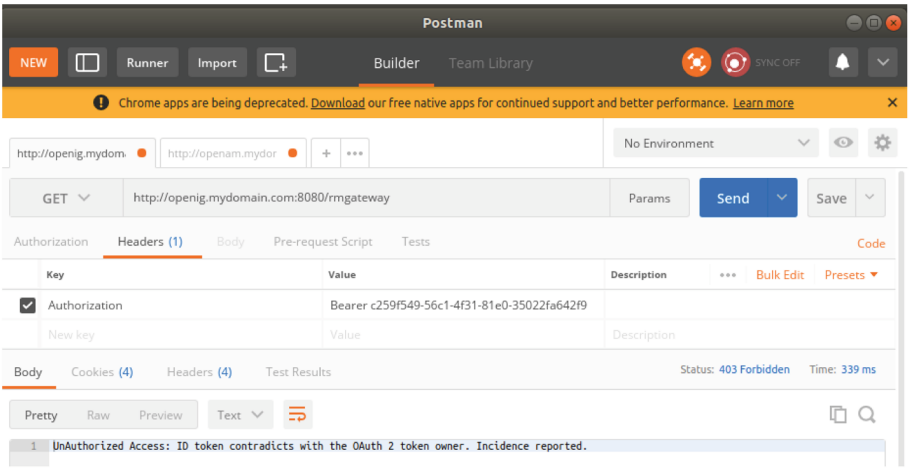

Fig. 4. Token theft detection

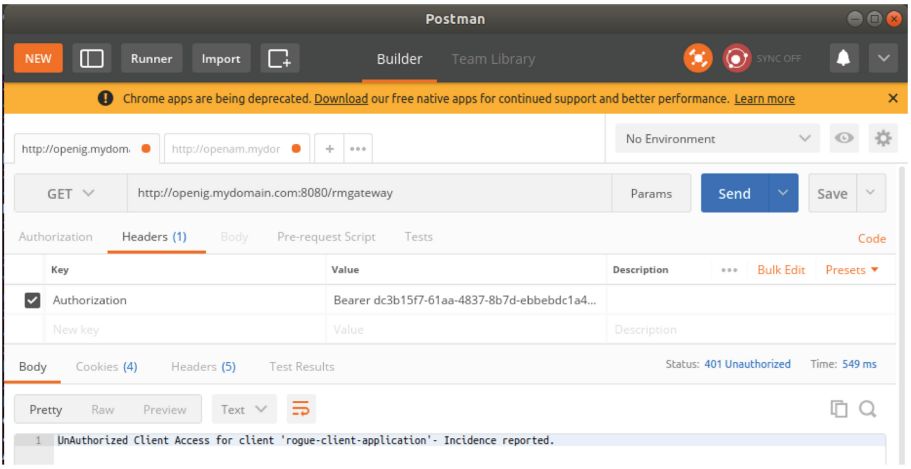

Fig. 5. Unauthorized client detection

combination, even if this resource and RMS are part of the same department. Client ID Check weakens the power of tokens, limits the trust between services to minimize the effect of a successful confused deputy attack, and mitigates creating fake OAuth clients by an insider.

In Fig. 6, we show a successful malicious request caused by the absence of our security checks. In this case, an unauthorized client, potentially created by an insider, is used to send the request, and the resource microservice responded with the data. Due to the lack of our Client ID Check, a malicious microservice with a fake OAuth client can be a threat, leading to data exfiltration from RMS. Having an OAuth client per

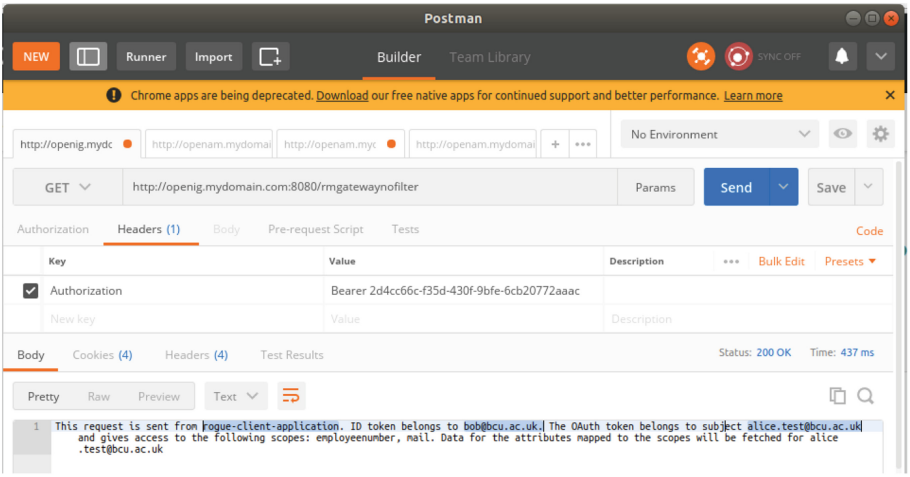

Fig. 6. Malicious request without our security checks.

consumer-resource combination, alongside the Client ID Check mitigates this threat. It also minimizes the trust between microservices by only allowing essential communications between them, and requiring the access control server involvement for token production and verification rather than blindly trusting a microservice or its domain. This practice minimises the impact of a confused deputy attack by limiting what can be done with a potentially compromised microservice.

Moreover, the user of the session and the OAuth token subject are not the same, which suggests using a stolen token for the request. Without our User Identity Check, token theft would not be detected. This gives this malicious user the ability to apply to services using another user's information. Our User Identity Check mitigates these attacks.

6.1 Performance

In this section, we show the overhead resulting from our proposed solution. We have conducted this experiment on an Ubuntu 17.10 running on a machine with 2.6 GHz Core i7 processor and 12 GB of RAM; we show the overhead caused by adding gateways configured for consumer microservices (CGW) and resource microservices (RGW). The line chart in Fig. 7 visualises the response time of 250 service calls for the same microservice without any gateways, with CGW, and with RGW. ACS is placed in a separate Linux container, on the same machine, to isolate the effect of data propagation over the internet. The lines show that the response time is the highest for microservices protected by RGW; the numbers confirm that, on average, an overhead of 23% results from adding a CGW, and of 32% occurs from adding RGW to a microservice. This means that a mixture of gateways protecting consumer and resource microservices should lead to an overhead of less than 32% on average. The overhead of User Identity Check and Client ID Check is minimal, given that it is all done within the GW program with no data propagation to the ACS.

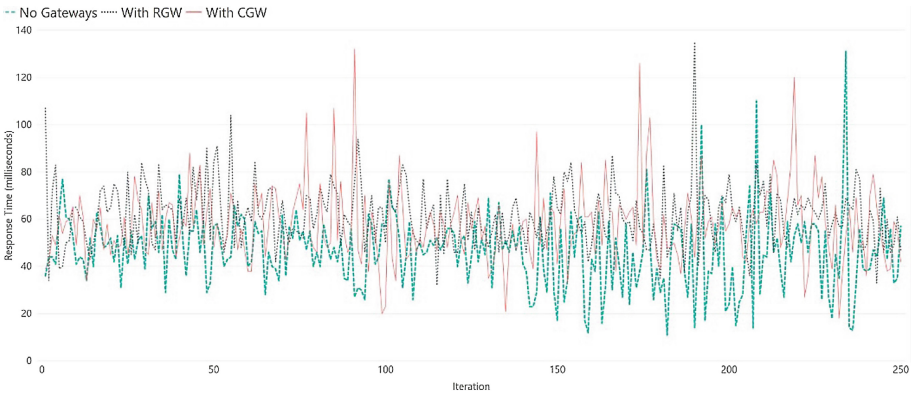

Fig. 7. Line chart showing our experimental results

As for ACS, the load factor is mostly affected by the number of exposed resource microservices due to the extra checks of OAuth 2 tokens; this is relatively easy to overcome with the cheap cloud elastic scaling.

7 Conclusions and Future Work

In this paper, we highlighted some security challenges that microservice-based applications are prone to in connection to access control and authorisation, both when the User is the trust anchor and when microservices work in conjunction. We presented a security design allowing fine-grained access control, while not compromising scalability, by proposing an access gateway at a per-microservice level. We have demonstrated the concept by implementing a prototype using XACML and OAuth 2, two leading open standards and readily available for microservices.

This work is part of a larger project that, on one hand, is looking into the chain of trust of distributed, multi-party many-component systems; on the other hand, we are developing solutions for digital governments where user control and trust are the central requirements. Several challenges are kept open. Our solution still largely depends on trusting key elements – for example, the Access Control Servers pose a risk and are able to compromise the whole system if they get compromised. On the other hand, from a user perspective, aspects such as repudiation and secure delegation of control are still open. Finally, we are also looking into the implications of having interdomain borders on which different (human) administrations sit. In other words, how to dynamically set very short-lived and on-on-the-fly trust boundaries, between multiple security administrations and environments.

References

1. Our approach to API authentication. https://gdstechnology.blog.gov.uk/2016/11/14/our-approach-to-authentication. Accessed 20 May 2018
2. Ahmad, A., Hassan, M.M., Aziz, A.: A multi-token authorization strategy for secure mobile cloud computing. In: 2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud), pp. 136–141. IEEE, April 2014
3. Yarygina, T., Bagge, A.H.: Overcoming security challenges in microservice architectures. In: 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE), pp. 11–20. IEEE, March 2018
4. Dragoni, N., et al.: Microservices: yesterday, today, and tomorrow. In: Mazzara, M., Meyer, B. (eds.) Present and Ulterior Software Engineering, pp. 195–216. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67425-4_12
5. Gao, X., Uehara, M.: Design of a sports mental cloud. In: 2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA), pp. 443–448. IEEE, March 2017
6. Geisriegler, M., Kolodiy, M., Stani, S., Singer, R.: Actor based business process modeling and execution: a reference implementation based on ontology models and microservices. In: 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 359–362. IEEE, August 2017
7. Härtig, H., Roitzsch, M., Weinhold, C., Lackorzynski, A.: Lateral thinking for trustworthy apps. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pp. 1890–1899. IEEE, June 2017. https://doi.org/10.1109/ICDCS.2017.29
8. IBM: An integrated approach to insider threat protection. https://www-05.ibm.com/services/europe/digital-whitepaper/security/growing_threats.html. Accessed 15 May 2018
9. Ilhan, Ö.M., Thatmann, D., Küpper, A.: A performance analysis of the XACML decision process and the impact of caching. In: 2015 11th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS), pp. 216–223. IEEE, November 2015
10. Jones, M., et al.: OAuth 2.0 token exchange draft-ietf-oauth-token-exchange-13. https://tools.ietf.org/html/draft-ietf-oauth-token-exchange-13
11. Newman, S.: Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Inc., Sebastopol (2015)
12. Patanjali, S., Truninger, B., Harsh, P., Bohnert, T.M.: Cyclops: a micro service based approach for dynamic rating, charging & billing for cloud. In: 2015 13th International Conference on Telecommunications (ConTEL), pp. 1–8. IEEE, July 2015
13. Rajani, V., Garg, D., Rezk, T.: On access control, capabilities, their equivalence, and confused deputy attacks. In: 2016 IEEE 29th Computer Security Foundations Symposium (CSF), pp. 150–163. IEEE, June 2016
14. Samlinson, E., Usha, M.: User-centric trust based identity as a service for federated cloud environment. In: 2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT), pp. 1–5. IEEE, July 2013
15. Suryotrisongko, H., Jayanto, D.P., Tjahyanto, A.: Design and development of backend application for public complaint systems using microservice spring boot. Procedia Comput. Sci. **124**, 736–743 (2017)
16. Suzic, B.: Securing integration of cloud services in cross-domain distributed environments. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, pp. 398–405. ACM, April 2016

17. Suzic, B.: User-centered security management of API-based data integration workflows. In: 2016 IEEE/IFIP Network Operations and Management Symposium (NOMS), pp. 1233–1238. IEEE, April 2016
18. Tang, L., Ouyang, L., Tsai, W.T.: Multi-factor web API security for securing mobile cloud. In: 2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), pp. 2163–2168. IEEE, August 2015
19. Yu, Y., Silveira, H., Sundaram, M.: A microservice based reference architecture model in the context of enterprise architecture. In: 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), pp. 1856–1860. IEEE, October 2016
20. Zhang, H., Li, Z., Wu, W.: Open social and XACML based group authorization framework. In: 2012 Second International Conference on Cloud and Green Computing (CGC), pp. 655–659. IEEE, November 2012
21. Linthicum, D.S.: Practical use of microservices in moving workloads to the cloud. IEEE Cloud Comput. **3**(5), 6–9 (2016)
22. Nehme, A., Jesus, V., Mahbub, K., Abdallah, A.: Securing microservices. IT Prof. **21**(1), 42–49 (2019). https://doi.org/10.1109/MITP.2018.2876987