Decentralized Authentication in Microservice Architectures with SSI and DID in Blockchain

Biagio Boi
Dept. of Computer Science
University of Salerno
Fisciano, Salerno,Italy
bboi@unisa.it

Christian Esposito
Dept. of Computer Science
University of Salerno
Fisciano, Salerno, Italy
esposito@unisa.it

***Abstract*—Microservice architectures aim at high modularity, reuse, and efficiency of code by structuring applications as a collection of services that are independently deployable, loosely coupled, and organized around business capabilities. As they are starting to be used in sensitive applications, security has started to be a priority, where authentication is one of the first protection means to be offered to developers by those products supporting microservice development. However, the available authentication solutions in these products are highly centralized, leveraging JSON Web Token (JWT) or related standards. This poses a serious issue in meeting the recent privacy legal obligations. In this paper, we propose a solution for integrating a decentralized blockchain-based authentication solution within the context of Istio, which is a service mesh supporting microservice developments. The usage of a Smart Contract, in combination with Decentralized Identifiers (DIDs) and JWT, paves the way for a concrete and fully decentralized revocation system without adding overhead or modification to existing microservices.**

***Index Terms*—Security, Microservices, Authentication, Digital Identities, Blockchain**

I. Introduction

The transition from monolithic to microservice architectures involves numerous advantages [1], such as the increase in maintainability, scalability, innovation, and a reduced time-to-market. The ideals of this architecture aim at one modular independence of the individual microservices, which have to implement exactly one feature. However, the main goal of microservices collides with what are some challenges that this step entails, such as security, observability, or traffic management; which, to be addressed, require the insertion of additional code to the microservice [2], increasing the coupling.

One of the solutions to these problems is represented by the Service Meshes [3], [4], which allows to management of features such as traffic management and balancing, observability, resilience, and security in a transparent manner. The developer does not need to explicitly design and reimplement the logic to realize these services and cope with these QoS but can leverage third-party implementations in a modular and integrated manner. By looking into the implementation of these solutions, a trivial approach to authentication is typically designed. Entities authenticate themselves with a username and password scheme, or with the mediation of third-party identity managers. This implies a possible security and privacy violation due to the centralization of the authentication decision and the simplicity of violating these systems due to the naive scheme being implemented. It would be preferable to have a more decentralized and robust approach, so as to have a stronger authentication scheme.

The goal of the paper is to provide a solution for managing decentralized identities through Decentralized Identifiers (DIDs) [5]; which make up a new model of digital identity management, represented by the Self-Sovereign Identity (SSI) paradigm [6], where the user is the only owner of his own digital identity and of all the associated data, without the need for an identity provider. This paradigm is one of the core applications of Web3, which promises a decentralized online experience, giving users back control over their data and information. More in detail, the research focuses on the analysis and use of a Service Mesh such as Istio [7] in managing authorization in microservices, integrating the DID solution for secure and decentralized identity management, implementable through the usage of the blockchain. Finally, we focus on smart policies and the use of smart contracts to ensure transparency about access policies. Obtaining this objective will ensure maximum privacy for the users themselves.

The document is structured into six sections:

- In the second section a state-of-the-art of the existing tools will be introduced in order to have a clear picture of the context;
- The third section introduces the functionalities offered by Service Mesh and Istio;
- The fourth section discusses the proposed architecture for solving the problem of security leveraging Service Mesh and Istio;
- The fifth section presents a real use case, within an organization and a discussion of the proposed system;
- The last section concludes the document giving insights about future works.

II. Background on Authentication in Microservices

The security aspects within a microservices-based application require a different approach from what are normal monolithic applications, due to the multiple independent services, each adapted to a specific predefined function. Authentication within microservices is handled in three layers [8]:

- Authentication of end-users who need to access the application.
- Authentication of microservices that need to connect to other microservices.
- Authentication of external services that need to connect to microservices.

These considerations ensure that only specified microservices and authorized users can access the appropriate one. In a microservices architecture, each microservice implements a specific function or piece of business logic. Each request to access the microservice must be authenticated and approved, which creates several problems, such as:

- **Centralized Dependency**: Authentication and authorization logic must be managed separately by each microservice. It is possible to use the same code in all microservices, but this requires that all of them support a specific language or framework.
- **Violation of the "do one thing well" principle**: Microservices are supposed to perform only one function. Adding global authentication and authorization logic to microservices, which now serve an additional function, makes them less reliable and more difficult to manage; violating the so-called decoupling principle.
- **Complexity**: Authentication and authorization in microservices can lead to very complex scenarios due to many users, the high number of microservices, and third-party systems accessing each microservice. This complexity can make implementation and maintenance difficult.

If we consider Edge-Level Authorization, in the simplest scenarios, authorization is done only at the edge of the network, often using an API Gateway; which applies authentication and access control for each microservice. Unfortunately, such an approach poses several maintainability and security issues, as the gateway represents a single point of failure.

On the other side, Service-Level Authorization allows for direct authentication and authorization for each microservice. The benefit is that each microservice has more control to enforce its own access control policies. External Entity Identity Propagation involves making authorization decisions based on user context, whether based on user IDs, roles or groups, the user's location, time of day, or other parameters. In order to authenticate, in the latter case, it is necessary to receive information about the end user and propagate it to the microservices. An easy way to achieve this is to take and parse an access token received at the edge and push it to individual microservices. This strategy provides the most granular control over microservices authentication. However, the content of the token is shared with all microservices and as a result, malicious users can compromise it. One possible solution is to sign the tokens with an issuer that the microservice trusts.

Among the different techniques available are the JWT [9] and the Single Sign-On [10]. JSON Web Token (JWT) is an open standard (RFC 7519) for secure transmission between different entities using JSON objects. A JSON Web Token is a very compact and machine-readable representation of a series of claims (affirmations), together with a signature to verify their authenticity. Basically, for authentication using a JWT, the identity provider (IdP) or authentication server generates a JWT that certifies the user's identity, while the resource server decrypts and verifies the authenticity of the token using the public key of who released the same. Referring to a possible generic use case, we have the following steps:

1) User logs in using username and password or using a third-party authentication provider;
2) The authentication provider verifies credentials and issues a JWT signed using a private key;
3) The user's client uses the JWT to access protected resources by passing the JWT in the HTTP header in the Authorization field.
4) The resource server then verifies the authenticity of the token using the authentication provider's public key.

Single Sign-On (SSO) is an authentication process that allows a user to access multiple applications with just one set of login credentials. Through it, the user logs in only once gaining access to multiple resources simultaneously without the need to re-enter the credentials for each application. It makes access very convenient for end users, who instead of requesting new credentials to access the application, use only one. In addition, identity and access management (IAM) solutions [11] can be used to configure a user database and define permissions for microservices that target them. This way, microservices can redirect users to the IAM system for authentication, receive an encrypted SSO token, and then use it to log users in on subsequent attempts. Microservices can also use the IAM system for authorization, and the SSO token can specify which resources the user can access. On the other hand, this kind of methodology is centralized.

A new and completely revolutionary model of digital identity management is represented by the Self-Sovereign Identity (SSI) paradigm. The concept of SSI is completely "centric" towards the user, who is the sole and independent owner of his own digital identity and all the data associated with it. SSI is a model that differs from the previous ones and which aims to ensure that the user remains the only owner of their data (from the term Self-Sovereign), thanks to the use of protocols based on Blockchain. The models previously described use a "centralized name system" as a database for storing the various information relating to identities. On the contrary, SSI is based on the blockchain, creating what is referred to as a "decentralized name system", and on asymmetric cryptography. It should also be noted that thanks to the essential characteristics of a blockchain, such as immutability and resilience (no downtime), the Self-Sovereign Identity model boasts greater security for all the actors involved. The SSI architecture envisages three actors: who supply the user's credentials and attributes (Issuer), the Holder, which is the owner of credentials, and the relying party (Verifier), which is the entity that wants to verify the identity of the users who is submitting the credentials. When the issuer attests credentials to a user, he signs them with his private key. After receiving these credentials, the user can exhibit them, after having signed them (certificate of possession), to a relying party, in a relationship of trust with the issuer; which will validate the signatures, verifying everything through the use of public keys that are available to anyone on the blockchain. If all goes well, the relying party grants access to the user.

In order to guarantee identification, SSI uses Decentralized Identifiers (DIDs). DIDs are a new type of identifier that allows a digital identity to be verifiable and decentralized. A DID refers to any entity (for example, a person, an organization, a thing, a data model, an abstract entity, and so on), but, unlike the typical federated model, DIDs are designed in a way that they can be decoupled from databases centralized, identity providers and certification authorities. This design allows the owner of a DID to demonstrate control, without requiring the intervention of other parties involved. A DID is a URI made up of three parts: the DID scheme, a method identifier, and a unique identifier specific to the DID method. A DID URL extends the syntax of a basic DID to incorporate other standard URI components, such as path, query, and fragment, in order to locate a particular resource, such as a cryptographic public key within a DID document or an external resource to the DID document.

III. Service Meshes and Istio

A service mesh lets the user control the interaction between components of an application and the way in which they communicate and share data. Unlike other systems for managing communication between services, a service mesh constitutes an infrastructural layer integrated directly into the application, in a transparent manner. The positioning of the microservices in the mesh allows for a common level, shared by all the components within it, for the management and implementation of complex network logic. This simplifies development and reduces the risk of errors, allows monitoring and telemetry, lets the user obtain a detailed view of the performance of microservices, and detects any anomalies or latency issues. This integration is able to enhance reliability, flexibility, and scalability while simplifying the management of the architecture components by allowing developers to focus on business functions, leaving aside infrastructural ones. In fact, applications of all types of architecture have always needed rules that specify how requests move from one point to another.

A service mesh extrapolates the logic that governs service-to-service communication from individual services and abstracts it into an infrastructure layer. This is why a service mesh is built through a series of proxies; which, within it, process the requests, directing them between the microservices. This is why the individual proxies that make up a service mesh are sometimes called "sidecars": they move alongside each service, rather than within them. These, decoupled from each service, together form a service mesh. With a proxy sidecar next to each instance, there is no need for extra workarounds and additions to your microservices code to handle circuit breaking, timeouts, service discovery, load balancing, and so on. Furthermore, the possibility of managing metrics, distributed tracing, and access control arises. Furthermore, since the traffic in the service mesh flows through these proxies, fine-grained control of the same is allowed, allowing the possibility of gradually releasing new versions, with canary updates, dark launches, and so on.

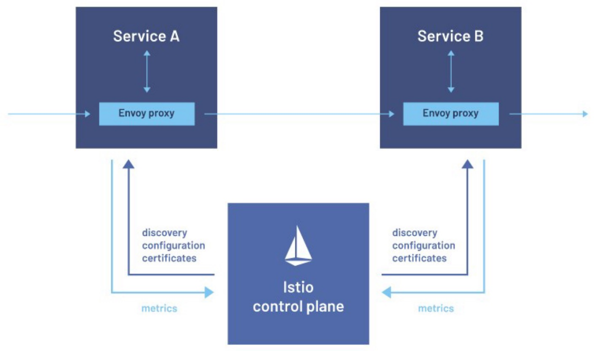


Fig. 1. An example of the architecture of a Service Mesh based on Istio.

Istio is an open-source service mesh that transparently complements existing applications; who must not know that they are part of it. Its capabilities help simplify running and managing a cloud-native service architecture by providing a uniform and efficient way to secure, connect, and monitor microservices [12]. Among the most important ones are:

- Secure service-to-service communications within a cluster with TLS encryption, advanced identity-based authentication, and authorization.
- Secure Automatic load balancing for HTTP, gRPC, WebSocket, and TCP traffic.
- Secure Granular traffic control, through routing, failover, and fault injection rules.
- Secure Metrics, logs, and tracing for all traffic within the cluster, including the traffic joining and leaving the cluster.

Istio is mainly composed of two components: the data plane and the control plane. The first represents communication between microservices. Without the latter, the mesh would not understand the traffic sent within it and could not make any type of decision-based on the type of traffic or who is communicating. Within Istio, the data plane is composed of several Envoy Proxies that act as sidecars, working alongside the microservices without weighing them down, through the use of containers. The control plane, however, allows us to communicate with the latter in a dynamic way, updating the rules, as depicted in Figure 1.

All these functionalities can be easily accessed through configuration files, which can react in response to situations within the environment, such as increased traffic or malfunctions. Considering that an essential requirement nowadays is safety, Istio is a good solution also from this point of view leveraging the concept of ”enabled by default”, since it controls each peer at the ends of the communication and can encrypt the traffic transparently. Istio is also able can manage the issuing, installation, and rotation of keys and certificates, in such a way as to allow the activation of mTLS for communication within the cluster. All without requiring the effort that would be necessary without the use of a service mesh. Furthermore, microservices also have special security needs:

- Traffic encryption to defend against man-in-the-middle attacks;
- mTLS and fine-grained access policies needed for Flexible access control to services;
- Auditing tools to determine who did what at what time. The objectives set are:
- Security by default able to not apply changes to the company code or infrastructure.
- Easy integration with multiple systems safety in order to increase the degree of robustness.
- Building of Zero-trust network security solutions for distributed systems.

Istio Security provides a comprehensive security solution to resolve these issues. Specifically, Istio security mitigates both internal and external threats against data, endpoints, communications, and platforms.

Last but not least, with Istio it is possible to implement organizational policies and limitations on speed and use. Very useful aspects when you need fine-grained rules regarding the use of SaaS belonging to different clouds (public and private). The components that implement these security measures, called Policy Enforcement Points (PEPs), are precisely the sidecars that support the microservices; while, the Policy Decision Point (PDP) is represented by the Istio control plane.

User authentication is critical for applications that store sensitive data. There are several authentication protocols for the latter; however, most of them rely on redirecting the user to an authentication server where, upon successful login, they are provided with a credential (stored as an HTTP cookie, or a JWT, and so on) that contains the user information, validated to provide any type of access. Istio provides two types of authentication. On the one hand, we find peer-to-peer authentication: used for service-to-service authentication to verify the client making the connection. Istio offers mutual TLS as a full-stack solution for traffic authentication, which can be enabled without requiring changes to the service code. On the other hand, we have request authentication: used for end-user authentication to verify the credentials attached to the request. Istio enables request-level authentication with JSON Web Token (JWT) validation and a simplified development experience using a custom authentication provider or any OpenID Connect provider, for example: Google Auth, Auth0, Keycloak, and so on. In both cases, Istio maintains information for access policies in a configuration file via a custom Kubernetes API. The control plane keeps them updated for each proxy. If the policy is changed, the new one is translated into the appropriate configuration indicating to the PEP how to perform the required authentication mechanisms. The control plane retrieves the public key and connects it to the configuration for JWT validation. Alternatively, Istio provides the path to the keys and certificates managed by the Istio system and installs them in the application pod for mutual TLS. Mesh administrators specify Istio authorization policies using .yaml files.

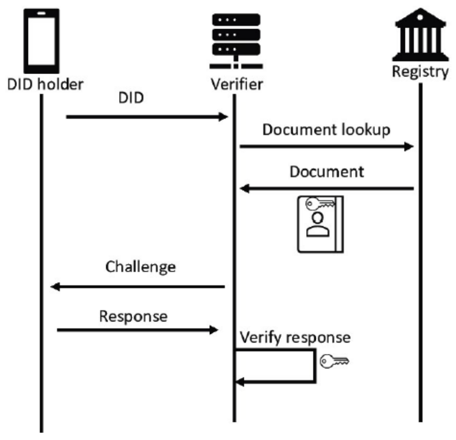


Fig. 2. DID Authentication Protocol.

IV. Proposed Solution

The solution aims to provide a user with the ability to authenticate to an authorization server while exhibiting control over their DID, previously claimed by the responsible department. The reference idea concerns the DID Auth protocol. As it is possible to see in Figure 2, a cryptographic exchange is prepared to certify ownership of the DID to the user with whom the authentication server interacted.

A. Protocol Definition

The implementation of this exchange involves sending a six-digit pseudorandom code from the authentication server to the user. The latter signs a message containing the code obtained previously with his private key and responds to the server. At this point FireFly comes into play, in fact, the server verifies the signature received by accessing the keys stored on the blockchain and releases a JWT token containing the claims of the properties associated with the user, present in the DID document. The latter is created in the registration phase with a simple client which involves the creation of a pair on the supernode (private key – Ethereum address). Starting from this information, the user will register his private key in the designated wallet, while the registration phase will claim the identity by entering a unique ID for the DID and

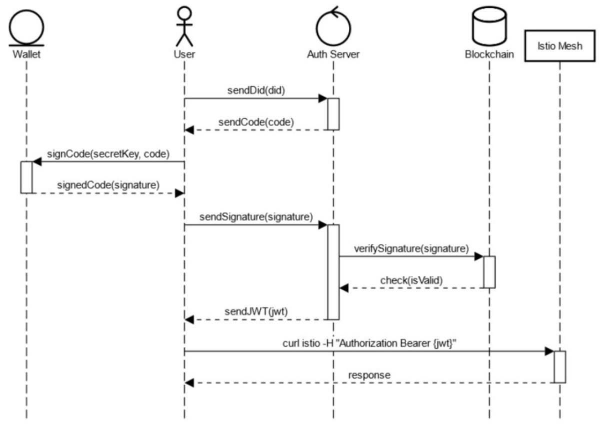


Fig. 3. Sequence diagram of the proposed authentication scheme.

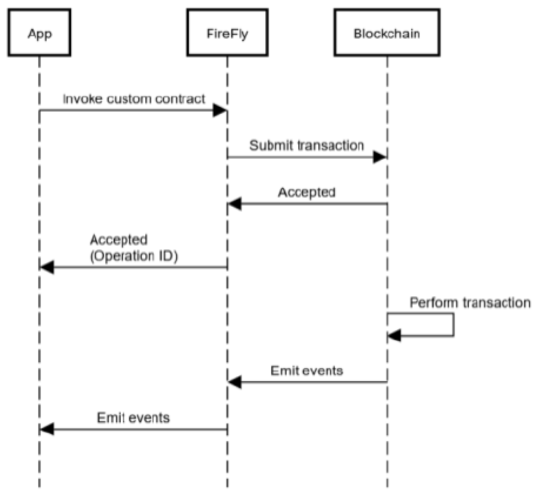


Fig. 4. Asynchronous flow for invoking a Smart Contract via FireFly.

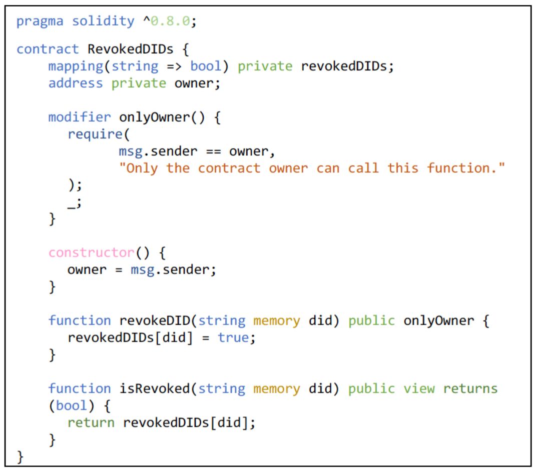

```
pragma solidity ^0.8.0;

contract RevokedDIDs {
    mapping(string => bool) private revokedDIDs;
    address private owner;

    modifier onlyOwner() {
      require(
              msg.sender == owner,
              "Only the contract owner can call this function."
      );
      _;
    }

    constructor() {
      owner = msg.sender;
    }

    function revokeDID(string memory did) public onlyOwner {
      revokedDIDs[did] = true;
    }

    function isRevoked(string memory did) public view returns
    (bool) {
      return revokedDIDs[did];
    }
}
```

Fig. 5. Revocation smart contract.

the properties associated with it. Regarding this step, attention must be paid to the question of privacy; this is because the DID document is public and, therefore accessible to everyone if the blockchain is also public. The idea, in this case, is that the identity properties are encrypted using the authentication server's public key. In this way, only the latter can decipher and view this information. By formalizing the entire operation, the result is divided into a series of steps, which can be seen in Figure 3.

Regarding the revocation of an identity, the idea is to use a Smart Contract to implement a register that contains all the revoked identities within the organization; which will be consulted by the authentication server, in the case that the cryptographic exchange is successful, to deduce whether to release or minus the JWT. In this way, the user, after the registration phase, will have the possibility of accessing the service mesh in a completely decentralized manner, relying solely and exclusively on the possession of the private key which will be released during the registration phase.

B. Implementation

Java has been considered for the implementation of the proposed protocol, in combination with Apache Maven. Web3j library can be used to provide the necessary primitives such as signing and verifying messages with the ECDSA algorithm, wallet management, and so on. Leveraging on Spring it is possible to securely deploy an authentication server, while the DID revocations can be handled with a Smart Contract hosted in the Ethereum blockchain. FireFly can be taken into consideration during the user authentication phase to verify whether or not the DID provided belongs to the revocation register.

1) Deployment: As regards the configuration of FireFly, an Ethereum-type stack must be started with at least two members, a necessary condition for the deployment of the Smart Contract. In fact, FireFly provides support for using RESTful APIs in order to interact with the Smart Contracts distributed in the target blockchains and listen to events via websockets, as can be seen in Figure 4.

The Smart Contract that implements this revocation registry uses a map (string $\rightarrow$ bool) to keep track of revoked DIDs. Furthermore, it offers two methods: the first one for the revocation of the DID itself, while the second one checks whether the input DID is revoked, returning a boolean, as shown in Figure 5.

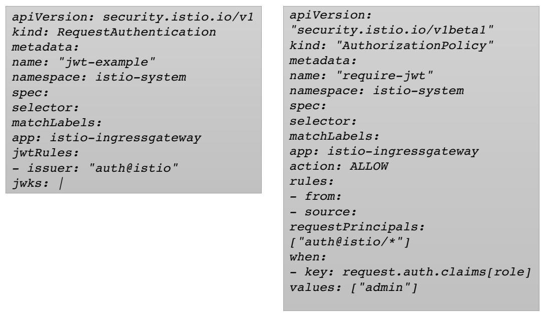
Once the contract has been deployed on the Ethereum blockchain, it is needed to deploy the FireFly interface. It can be obtained automatically by sending a request to the endpoint: api/v1/namespaces/default/contracts/interfaces/generate, attaching a body containing the abi of the previously generated Smart Contract. Subsequently, the contract interface will be broadcast as soon as it has been received in response to the previous request. The endpoint to contact is: api/v1/namespaces/default/contracts/interfaces, to which the body obtained in the response to the previous step must be sent.

The last step involves the creation of an HTTP API for this Smart Contract, able to make possible the interaction between the contract and the service. To perform this task it is necessary to recall the identifier obtained in the response of the previous step into the interface.id field in the body of the underlying request. Finally, in the location.address field, we are telling FireFly where an on-chain contract instance is deployed. Once the body of the request has been defined, it is possible to send a request to the endpoint: api/v1/namespaces/default/apis. With these steps, the Smart Contract has been correctly deployed and can be reached at the URL obtained as a response in the last step.

The logic of this implementation has already been described previously and involves the creation of a cryptographic exchange to certify the user's ownership of the DID he claims to own.

Once the server has been deployed, it accepts requests on two endpoints, which represent the two phases of the cryptographic exchange: /sendDid and /sendProof. The idea involves the user sending his DID when he contacts the first endpoint. In this circumstance, the server generates a code and places it in a pool of requests together with the DID, generating a pair (DID, code), before responding with the same to the user. When the latter receives the code in response, he signs it using his private key, contained within the wallet and issued during the registration phase. It then contacts the second endpoint with the signed message. In the last phase of the process, the server verifies the signature, comparing the Ethereum address associated with the DID (available on the blockchain) with that obtained from the signature received from the user. Subsequently, the revocation register is consulted to ascertain the validity of the DID. If these checks are successful, the JWT token is created, recovering the user properties associated with the DID document from the blockchain and inserting them as claims within the token.

2) Enrollment and Revocation: In this phase, a department responsible or an authority will be leveraged for managing corporate identities with the possibility of inserting or revoking an identity on the blockchain, associated with the company information that he is intended to include in the DID document. This phase initially involves the creation of an account via the FireFly shell, which will respond with a pair (Ethereum address, private key). The key must be provided to the user in a secure manner. The latter will insert it into their wallet the first time the DID Authentication application is run. The only problem in this case is the public exposure of sensitive information within the DID document. To overcome this problem, the authentication server exposes a final endpoint /getPublicKey, which provides the public key to be used to encrypt sensitive information and make it accessible only and exclusively on the authentication server. In this way, it is possible to use the RSA algorithm for encryption and decryption of sensitive information within the DID document. Regarding the revocation of DIDs, the endpoint used for the execution of the previously deployed Smart Contract is contacted, sending the DID that you want to revoke as the body of the request, which, from this moment on, will no longer be validated in the authentication actions and authorization by the



```
apiVersion: security.istio.io/v1
kind: RequestAuthentication
metadata:
name: "jwt-example"
namespace: istio-system
spec:
selector:
matchLabels:
app: istio-ingressgateway
jwtRules:
- issuer: "auth@istio"
jwks: |
```

```
apiVersion:
"security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
name: "require-jwt"
namespace: istio-system
spec:
selector:
matchLabels:
app: istio-ingressgateway
action: ALLOW
rules:
- from:
- source:
requestPrincipals:
["auth@istio/*"]
when:
- key: request.auth.claims[role]
values: ["admin"]
```

Fig. 6. Istio configuration.

server.

3) Verification: As described at the beginning of this section, the verification starts by sending the DID to the authentication server, which will respond with a code. The latter must be signed with the private key, obtained previously, during the registration phase. Notice that the first time the service is run, the key will be requested to be stored in the wallet. After signing the code, the client sends the signature to the authentication server, which will be validated by verifying the correspondence of the associated address and the validity of the DID itself. After this phase, the server will respond with the JWT containing the claims with the user's information present on the blockchain. At this point, the user can use the token to access the Istio service mesh.

V. DEMO

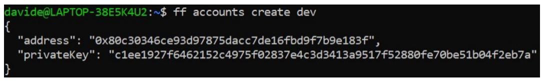
In this section, a running example of the proposed architecture is presented together with the configuration and execution phase.

A. Configuration

The first resource needed for the Istio configuration is the RequestAuthentication, whose main purpose is to validate the JWT, extract the token claims statements, and store them in the request metadata. The latter is used by the AuthorizationPolicy resource, to make a decision, based on one or more attributes. This is shown on the left side of Figure 6. The resource in our case indicates the use of JWTs for authentication, specifying the issuer and the JWKS (omitted to improve readability). The latter represents a set of public keys that will be used to validate the token. This is shown on the right side of Figure 6.

B. Execution

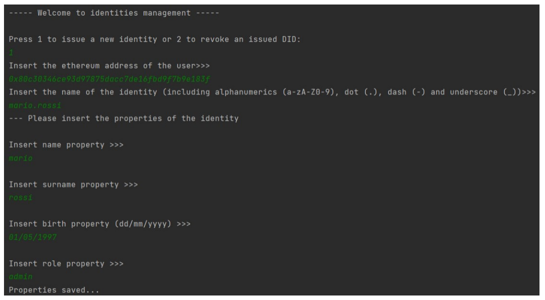
After the configuration phases, it is possible to execute and view the result of this implementation activity, which is divided into several steps. First of all, we start an Ethereum stack with the appropriate command: ff start ¡stack_name¿; possibly, to create the stack (with at least two members), run ff init Ethereum. In the latter case, it will be necessary to deploy the Smart Contract for the revocation of DIDs as discussed in section IV-B2. Finally, we start the authentication server which will listen to the endpoints described previously. Subsequently, within the supernode shell, it is possible to execute a command for creating a new pair composed of *(address, privateKey)*. The latter must refer to the user to whom the identity is being issued for inclusion in the designated wallet. Additionally, the ID of the organization is needed for issuing the identity, which can be obtained via a GET call to the URL: api/v1/status.



```
davide@LAPTOP-38E5K4U2:~$ ff accounts create dev
{
  "address": "0x80c30346ce93d97875dacc7de16fbd9f7b9e183f",
  "privateKey": "c1ee1927f6462152c4975f02837e4c3d3413a9517f52880fe70be51b04f2eb7a"
}
```

Fig. 7. Creation of a new identity.

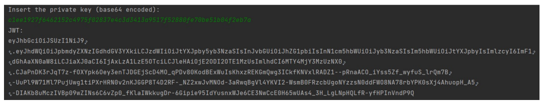
Now it will be up to the department responsible for identities to launch the IdentityManagement client, which allows the user to claim and/or revoke a DID. In this step, all the information about the user's identity will therefore be entered, as in Figure 8.



```
----- Welcome to identities management -----

Press 1 to issue a new identity or 2 to revoke an issued DID:
1
Insert the ethereum address of the user>>>
0x80c30346ce93d97875dacc7de16fbd9f7b9e183f
Insert the name of the identity (including alphanumerics (a-zA-Z0-9), dot (.), dash (-) and underscore (_))>>>
mario.rossi
--- Please insert the properties of the identity

Insert name property >>>
mario

Insert surname property >>>
rossi

Insert birth property (dd/mm/yyyy) >>>
01/05/1997

Insert role property >>>
admin
Properties saved...
```

Fig. 8. Claiming a new identity.

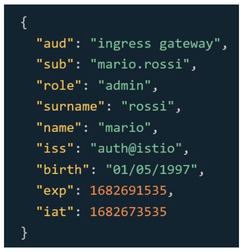
Subsequently, the user who received the private key must insert it into his wallet; this only happens during the first execution of the entire workflow, where if a wallet is not already present, one is created for the persistence of the private key. This client communicates with the authentication server, in order to obtain the JWT, visible in Figure 9, for access to Istio.



```
Insert the private key (base64 encoded):
c1ee1927f6462152c4975f02837e4c3d3413a9517f52880fe70be51b04f2eb7a
JWT:
eyJhbGciOiJSUzI1NiJ9
.eyJhdWQiOiJpbmdyZXNzIGdhdGV3YXkiLCJzdWIiOiJtYXJpby5yb3NzaSIsInJvbGUiOiJhZG1pbiIsInN1cm5hbWUiOiJyb3NzaSIsIm5hbWUiOiJtYXJpbyIsImlzcyI6ImF1
dGhAaXN0aW8iLCJiaXJ0aCI6IjAxLzA1LzE5OTciLCJleHAiOjE2ODI2OTE1MzUsImlhdCI6MTY4MjY3MzUzNX0
.CJaPnDK3rJqT7z-fOXYpk6Oey3enTJDGEjScD4MO_qPDv80KodBExWuIsKhxzREKGmQwg3ICkfKNVxlRADZ1--pRnaACO_iYss5Zf_wyfuS_lrQm7B
-UuPl9W71Ml7PujUwg1tiPXrHRN0v2nKJGGP8T4D2RF-_NZ2xwJvMNOd-3aRwqBgVl4YKVI2-WsmB0FRzcbUgoNYzzsN0ddFWO8NA78rbYPK0sXj4AhuopH_A5
-DIAKb8uMczIVBp09wZINs6C6vZp0_fKlaIWkkugDr-6Gipie95IdYusnxWJe6CE3NwCcE0H65wUAs4_3H_LgLNpHQLfR-yfHPInVndP9Q
```

Fig. 9. JWT token obtained for accessing Istio.

By decrypting the JWT it is possible to notice the token claims, as depicted in Figure 10.



```
{
  "aud": "ingress gateway",
  "sub": "mario.rossi",
  "role": "admin",
  "surname": "rossi",
  "name": "mario",
  "iss": "auth@istio",
  "birth": "01/05/1997",
  "exp": 1682691535,
  "iat": 1682673535
}
```

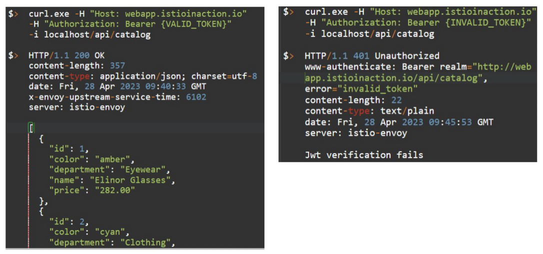
Fig. 10. Structure of an issued JWT.

After obtaining the token, it is possible to access an Istio service mesh configured as explained in the previous paragraphs. To create the service mesh, the microservices made available by some tutorials available online will be used. The commands for creating the mesh are:

```
$ kubectl config set-context
$ kubectl config current-context
--namespace=istioinaction
$ kubectl delete virtualservice,deployment,service,
destinationrule,gateway,
peerauthentication,authorizationpolicy --all
$ kubectl delete peerauthentication,
authorizationpolicy -n istio-system --all
$ kubectl apply -f services/catalog/../catalog.yaml
$ kubectl apply -f services/webapp/../webapp.yaml
```

After creating the mesh, it is necessary to apply the two resources, described in the previous paragraph, used for authorization via JWT.

In fact, we can see how making a request with a valid token provides access to the microservices, while a request with an invalid token does not allow it, as shown in Figure 11.

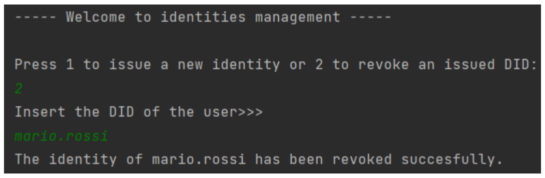


```
$>  curl.exe -H "Host: webapp.istioinaction.io"
    -H "Authorization: Bearer {VALID_TOKEN}"
    -i localhost/api/catalog

$>  HTTP/1.1 200 OK
    content-length: 357
    content-type: application/json; charset=utf-8
    date: Fri, 28 Apr 2023 09:40:33 GMT
    x-envoy-upstream-service-time: 6102
    server: istio-envoy

    [
      {
        "id": 1,
        "color": "amber",
        "department": "Eyewear",
        "name": "Elinor Glasses",
        "price": "282.00"
      },
      {
        "id": 2,
        "color": "cyan",
        "department": "Clothing",
```

```
$>  curl.exe -H "Host: webapp.istioinaction.io"
    -H "Authorization: Bearer {INVALID_TOKEN}"
    -i localhost/api/catalog

$>  HTTP/1.1 401 Unauthorized
    www-authenticate: Bearer realm="http://web
    app.istioinaction.io/api/catalog",
    error="invalid_token"
    content-length: 22
    content-type: text/plain
    date: Fri, 28 Apr 2023 09:45:53 GMT
    server: istio-envoy

    Jwt verification fails
```

Fig. 11. Examples of accessing Istio with a valid and an invalid token.

Finally, we show the revocation of a DID and the subsequent authentication attempt by the user. By starting the IdentityManagement client, we will be asked for the DID to be revoked, as shown in Figure 12.



```
----- Welcome to identities management -----

Press 1 to issue a new identity or 2 to revoke an issued DID:
2
Insert the DID of the user>>>
mario.rossi
The identity of mario.rossi has been revoked succesfully.
```

Fig. 12. Revocation of an identity.

After revocation, it is possible to switch to the DidAuthentication client. If the client tries to access using the previously revoked DID, then it will fail, confirming the enhanced robustness of the system.

C. Advantages

The solution implemented using Decentralized Identifiers (DIDs) within the SSI for authentication and authorization in service meshes represents a great step forward in ensuring security and access control. The integration of the latter into the Self-Sovereign Identity paradigm offers numerous advantages. First of all, it allows people to have full control of their digital identities, avoiding having to depend on centralized entities for credential management. This is essential to protect privacy and ensure that personal data remains in the hands of users. Additionally, permission management is simplified as DIDs act as unique identifiers for users within the system and allow you to verify permissions without having to have multiple accounts and rely on external or centralized entities. This reduces complexity and the risk of malicious attacks. Another important feature of DIDs is their interoperability capability. They are designed to work across multiple platforms and networks, enabling secure communication and access to resources within a service mesh. This flexibility is essential to adapt to the needs of a distributed and scalable infrastructure.

D. Challenges

To avoid the authentication and authorization process becoming a bottleneck for the entire network, system scalability, and performance must be appropriately handled. Despite the fact that a decentralized approach to authentication can improve overall process performance, the authentication process is not simple and must be carefully developed.

Furthermore, it is essential to ensure the security of the DIDs themselves. Protecting the private keys associated with DIDs is crucial to preventing unauthorized access. Robust encryption and cybersecurity measures must be taken to preserve the integrity of DIDs and related credentials. A typical approach to this problem is the usage of a hardware-based key pairs generator or cryptographic wallet; able to prevent attackers from stealing private keys.

VI. Conclusions

The paper discussed a decentralized architecture for enhancing security within the microservices context. Service meshes put the user in the center of the microservices-oriented architecture, where Istio is an open-source solution. The centralized approach to the authentication within these microservices poses challenges to both performance and security, where the authentication server can be easily intended as a point of failure. The usage of SSI represents a very promising approach to improving the security and management of digital identities within this context, by changing the way in which the authentication process is managed. Their implementation requires attention to detail and proper management of the technical and security challenges involved. However, when adopted correctly, DIDs can offer users greater security, privacy, and control within service meshes. While a distributed ledger is able to act as a highly scalable solution, future works aim at evaluating the performance of the proposed approach and an extension of the current demo will be considered within a real scenario.

Acknowledgment

This work was partially supported by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU.

References

[1] Mishra, Ridhima, et al. "Transition from Monolithic to Microservices Architecture: Need and proposed pipeline." 2022 International Conference on Futuristic Technologies (INCOFT). IEEE, 2022.

[2] Hannousse, Abdelhakim, and Salima Yahiouche. "Securing microservices and microservice architectures: A systematic mapping study." Computer Science Review 41 (2021): 100415.

[3] Li, Wubin, et al. "Service mesh: Challenges, state of the art, and future research opportunities." 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE). IEEE, 2019.

[4] Hahn, Dalton A., Drew Davidson, and Alexandru G. Bardas. "Mismesh: Security issues and challenges in service meshes." Security and Privacy in Communication Networks: 16th EAI International Conference, SecureComm 2020, Washington, DC, USA, October 21-23, 2020, Proceedings, Part I 16. Springer International Publishing, 2020.

[5] Lux, Zoltán András, et al. "Distributed-ledger-based authentication with decentralized identifiers and verifiable credentials." 2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS). IEEE, 2020.

[6] Mühle, Alexander, et al. "A survey on essential components of a self-sovereign identity." Computer Science Review 30 (2018): 80-86.

[7] Calcote, Lee, and Zack Butcher. Istio: Up and running: Using a service mesh to connect, secure, control, and observe. O'Reilly Media, 2019.

[8] de Almeida, Murilo Góes, and Edna Dias Canedo. "Authentication and authorization in microservices architecture: A systematic literature review." Applied Sciences 12.6 (2022): 3023.

[9] Ahmed, Salman, and Qamar Mahmood. "An authentication based scheme for applications using JSON web token." 2019 22nd international multitopic conference (INMIC). IEEE, 2019.

[10] De Clercq, Jan. "Single sign-on architectures." International Conference on Infrastructure Security. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002.

[11] Mohammed, Ishaq Azhar. "Intelligent authentication for identity and access management: a review paper." International Journal of Managment, IT and Engineering (IJMIE) 3.1 (2013): 696-705.

[12] Ozair Sheikh, Serjik Dikaleh, Dharmesh Mistry, Darren Pape, and Chris Felix. 2018. Modernize digital applications with microservices management using the istio service mesh. In Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering (CASCON '18). IBM Corp., USA, 359–360.