1 Introduction

A microservices-based architecture in applications and service mesh application infrastructure that provides various security services through service proxies has emerged as the widespread application environment for cloud-native applications. With the disappearance of the network perimeter due to the need for ubiquitous access to these applications from multiple remote locations using different types of devices, it is necessary to build the concept of zero trust [1] into this application environment. Zero trust assumes that there is no implicit trust granted to assets or user accounts based solely on their physical or network location (i.e., local area networks versus the internet) or based on asset ownership (enterprise or personally owned) [1]. Zero trust focuses on protecting resources (e.g., assets, services, workflows, and network accounts) rather than network segments since the network location is no longer seen as the prime component to the security posture of the resource. Furthermore, the loosely coupled nature of the components (i.e., microservices) of these cloud-native applications facilitates independent design, development, and agile deployment (e.g., Continuous Integration/ Continuous Delivery or Deployment (CI/CD) [2]) of the constituent microservices, enabling paradigms such as DevSecOps [3] (representing Development, Security and Operations) to be used.

The security requirements for microservices-based applications are discussed extensively in [4] and summarized here to provide context for this discussion. They are:

- Multiple, loosely coupled microservices communicate through network calls, and these communication links must be protected. In the case of monolithic applications, these communications take place through procedure calls.
- The entire network and all microservices are untrusted. Therefore, mutual authentication and secure communication channels between microservices through mechanisms such as mutual Transport Layer Security (TLS) (mTLS) are required.
- The logging data that pertains to each microservice must be consolidated to obtain a security profile in order for forensics, audits, and analytics to assess the overall health of the application.

Operating in multiple security domains and multiple clouds, cloud-native applications require a secure authentication and authorization framework. When implemented within the service mesh, the critical requirements of this framework are:

- The code that is part of this framework should be verifiable and non-bypassable (always invoked), thus satisfying the requirements of a security kernel.
- The framework should provide authentication and authorization services at both the service level and end-user level.
- The framework should be able to support a diverse set of authorization policies.

The operational assurances are required for meeting the above requirements, and others are provided by the service mesh. The specific features in the service mesh that enable these are given in the next section.

1.1 Service Mesh Capabilities

A service mesh provides a framework for building a set of operational assurances for an application through its various features and the set of controls they enable. That framework includes an authenticatable runtime identity for services, the ability to authenticate credentials of individual users of a service, encryption of communication in transit between services, a Policy Enforcement Point (PEP) separately deployable and controllable from the application (e.g., service mesh's side-car proxy), and logs and metrics for monitoring policy enforcement. Using these mesh features, a set of controls can be built for all applications that are part of the mesh (e.g., all traffic is encrypted, all traffic to an application goes through the side-car proxy [PEP]).

A significant advantage of the service mesh architecture is that the key piece that allows for these controls to be built – the sidecar proxy deployed next to every application – has more security benefits than the traditional approach of building these operational assurances into the application code. First, the life cycle of the sidecar is independent of the application, making it easier to manage across a fleet (e.g., push updates, ensure a consistent version is deployed everywhere). Second, modern implementations (e.g., Istio) allow for dynamic configuration. It is easy to update policies, and updates take effect nearly immediately and without having to redeploy applications. Finally, the mesh's centralized control allows security teams to build policies that apply to the entire organization so that application developers who build business value are secure by default.

A service mesh provides the ability to authenticate end user credentials attached to the request, like a JSON (JavaScript Object Notation) Web Token (JWT). Many service meshes (e.g., Istio) go further and provide the ability for the mesh's side-car to call external authentication and authorization systems on behalf of the application. This grants the ability to move request-level policy enforcement out of the application code, trusting instead on the mesh's assurance that requests that reach the service have been authenticated and authorized for the action that the request is attempting. The mesh can even be configured to pass proof of this to the application. This, coupled with the service mesh's centralized control, means it is possible for a central team to mandate and manage application-level security across the entire organization, delegating to individual application teams only to specify what permissions are required for each application's actions.

Using the service mesh architecture also means that authentication and authorization systems can be deployed as services in the mesh. Like any other service in the mesh, they benefit from the operational assurances the mesh provides: encryption in transit, identity, a PEP, authentication, and authorization for end user identity. This makes it cheaper to operate an organization's authentication and authorization systems securely and reliably.

In addition to the service mesh features, the capabilities of the access control model play an important role in the authentication and authorization framework. Attribute-based access control (ABAC) has emerged as a promising approach for supporting multiple authorization policies (third requirement above). As per [5], ABAC is defined as "an access control method where subject requests to perform operations on objects are granted or denied based on assigned attributes of the subject, assigned attributes of the object, (optionally) environmental conditions, and a set of policies that are specified in terms of those attributes and conditions." The main focus of this document is to provide guidance on an authentication and authorization framework, the latter using ABAC to secure microservices-based applications using service mesh.

1.2 Candidate Applications

The service mesh is most widely used today with containerized applications but can be extended into other environments, such as stateful applications.

1.3 Scope and Approach

This document focuses on providing guidance for building a secure authentication and authorization framework using components of a service mesh for securing services in microservice-based applications. The framework was developed by highlighting the set of features provided by the service mesh, illustrating how those features can be used to provide operational assurances for applications in the mesh, and making a set of recommendations for settings/defaults for those features that provide the best security posture for applications running on that mesh. The example of running the access control system – one of the most important systems in the organization – on the mesh leveraging those operational assurances makes the system more secure and cheaper to operate than it might be otherwise.

A reference application orchestration and resource management platform and a reference service mesh platform have been used as examples to illustrate these recommendations in the context of real-world application artifacts (e.g., containers and virtual machines (VMs)). The chosen reference application orchestration and resource management platform is the open-source Kubernetes (although other application platforms can be chosen for reference), and the chosen reference service mesh platform is Istio. Application infrastructure components in the service mesh that provide other services like network routing, network resilience, and monitoring are outside of the scope of this document.

1.4 Target Audience

The target audience of this guidance document for developing an authentication and authorization framework for microservices-based applications using the service mesh includes:

- Security solutions architects who want to protect the application workloads in microservices-based applications
- Platform architects who want to incorporate a service mesh into the platform offered by their organization to its developers
- Developers who want to develop authentication and authorization plug-ins in this application environment

1.5 Relationship to Other NIST Guidance Documents

This guidance focuses on building an authentication and authorization framework within the service mesh used for securing microservices-based applications. The following publications provide background information for the contents of this document:

- Special Publication (SP) 800-204, *Security Strategies for Microservices-based Application Systems* [4], discusses the characteristics of microservices-based applications and the overall security requirements and strategies for addressing those requirements.
- Special Publication (SP) 800-204A, *Building Secure Microservices-based Applications Using Service-Mesh Architecture* [6], provides deployment guidance for various security services (e.g., authentication and authorization, and security monitoring) for a microservices-based application using a dedicated infrastructure (i.e., a service mesh).
- Special Publication (SP) 800-162, *Guide to Attribute Based Access Control (ABAC) Definitions and Considerations* [17], provides a definition of attribute based access control (ABAC) as a logical access control methodology where authorization to perform a set of operations is determined by evaluating attributes associated with the subject, object, requested operations, and, in some cases, environment conditions against policy, rules, or relationships that describe the allowable operations for a given set of attributes.

1.6 Organization of This Document

The organization of this document is as follows:

- Section 2 provides an overview of a microservices-based application, its security requirements, a brief description of the overall architecture of reference platform for orchestration and resource management of microservices-based applications and the reference service mesh platform. The latter two are used as examples to illustrate the building blocks involved in the deployment recommendations.
- Section 3 outlines the advantages of ABAC for the application environment and describes the functional architecture for two of the standard ABAC representations.
- Section 4 discusses the building blocks of the authentication and authorization framework, the requirements and recommendations for configuration of policies that are required in the reference orchestration and resource management platform and in the reference service mesh platform for implementing the framework. The recommendations span mechanisms for supporting both end user and service level authentication and authorization policies. The minimal set of policy elements needed in authorization policies are also outlined.
- Section 5 discusses some architectural features of the framework such as functionality of a reference monitor, supporting infrastructure, advantages as an ABAC implementation, and enforcement alternatives.
- Section 6 provides summary and conclusions.

2 Reference Platform for Microservices-based Application and Service Mesh

The objective of this document is to offer recommendations for the deployment of an authentication and authorization framework for microservices-based applications within a service mesh that provides the infrastructure for various services, including critical security services. A reference platform for hosting microservices-based applications and the service mesh is included to provide clarity and context for concepts and recommendations in real-world application environments. A brief description of these reference platforms is also provided in terms of their overall architecture and salient building blocks.

2.1 Reference Platform for Orchestration and Resource Management of a Microservices-based Application

Kubernetes is an orchestration and resource management system widely used for microservices-based applications. In a large application, there will be several microservices, each of which is implemented as a container. Scalable, automated means are required for deployments, operations, upgrading services, and monitoring the health of these containers. The Kubernetes platform provides the building blocks to achieve these goals.

The fundamental building blocks in a Kubernetes platform are: pod, node, cluster, and control plane components. A pod is the smallest object deployed, represents a set of running containers, and is identified by a label that is a name/value pair. A node is a physical or virtual machine that houses a set of pods as well as the interfaces necessary to run the housed pods. The set of nodes is called a cluster.

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new pod when workload increases). Some key control plane components include: Kubernetes API (application programming interface) server, a key/value store, a scheduler, and a set of controllers. The node controller is one such controller that manages various aspects of nodes.

To ensure adequate performance and continued availability, it is imperative to have some cluster-level mechanisms for the clusters that are configured using the hosts of the application components (i.e., microservices). Considering a scenario where the host is a node of a Kubernetes platform cluster and the application components are running inside of a container with a pod (i.e., a group of containers) as a deployment artifact, the following cluster-level mechanisms are required.

For example, one of the most well-known features of Kubernetes is pod-level *horizontal scaling*. This means that when services receive more traffic, more instances will be generated across machines that grow or shrink on demand. Kubernetes supports auto-vertical scaling on the pod level. Thus, a cluster could be configured to scale the machine on which a pod runs up or down to more accurately fit the anticipated power needs of any microservice. For example, if certain subsets of nodes saw spikes in traffic at key times, with the right usage analysis, one could potentially reschedule those jobs across machines in order to save costs and optimize performance [7].

Similarly, Kubernetes offers features to monitor the health of the microservices (e.g., check the status and readiness). The data to perform these functions is configured in declarative deployment documents, known as Yet Another Markup Language (YAML), that describe the ports that a pod's containers are listening on. One can specify what to do when services do not start, do not perform as normal, or exit unexpectedly.

2.1.1 Limitations of Reference Orchestration and Resource Management Platform for Security

Microservices-based applications require several application infrastructure and security services, such as authentication, authorization, monitoring, logging, auditing, traffic control, caching, secure ingress, service-to-service, and egress communication. The reference hosting platform with interfaces for efficiently scheduling and running the multiple services of an application lacks the interfaces to run a set of application infrastructure services in a coordinated and consistent manner. Moreover, the following advantages of API architecture are not fully leveraged in the reference platform [8]:

- A unified way to apply cross-cutting concernsOut-of-the-box plugins to quickly address cross-cutting concerns
- A framework for building custom plugins
- Managing security in a single plane
- Reduced operation complexity
- Easy governance of third-party developers and integrators
- Saving the cost of development and operations

By default, communication between Kubernetes containers is insecure, and there is no easy way to enforce TLS between pods since this would result in individually maintaining hundreds of TLS certificates. Pods that communicate do not apply identity and access management between themselves. Though there are tools to define a Kubernetes network policy that implements a firewall between pods, they provide a layer 3 solution rather than a layer 7 solution, which is what most modern firewalls do (a few tools for defining more sophisticated network policies are available). This means that while one can know the source of traffic, one cannot peek into the data packets to understand what they contain. It does not allow for making vital metadata-driven decisions, such as routing on a new version of a pod based on an Hypertext Transfer Protocol (HTTP) header.

There are Kubernetes ingress objects that provide a reverse proxy based on layer 7, but they do not offer anything more than simple traffic routing.[1] Kubernetes offers different ways of

[1] Custom development can extend Kubernetes ingress controllers and objects to provide more advanced capabilities. In fact, some service mesh implementations in some kubernetes environments leverage this as a way to route traffic. Further, the network configuration for the orchestration platform (Kubernetes) is not an integral part of the service mesh. Since microservices are implemented as containers, the network configuration is dependent on the associated Container Network Interface (CNI) implementation. Hence, the ability to have advanced networking has to do with the software that the operator

deploying pods that do some form of A/B testing or canary deployments, but they are done at the connection level and provide no fine-grained control or fast failback. For example, if a developer wants to deploy a new version of a microservice and pass 10 % of traffic through it, they will have to scale at least 10 containers – nine for the old version and one for the new version. Furthermore, Kubernetes cannot split the traffic intelligently and instead balances loads between pods in a round-robin fashion. Every Kubernetes container within a pod has a separate log, and hence a custom solution over Kubernetes must be implemented to capture and consolidate them.

Although the Kubernetes dashboard offers features like monitoring pods and checking their health, it does not expose metrics that describe how application components interact with each other, how much traffic flows through each of the pods, or what chains of containers make up the application. Since traffic flow cannot be traced through Kubernetes pods out of the box, it is unclear where the failure for that application request occurred on the chain.

A service mesh addresses these limitations [9]. This document will first consider the service mesh architecture, followed by implementation of service mesh capabilities in the context of the reference platform (Kubernetes).

2.2 Service Mesh Reference Platform – Conceptual Architecture

A service mesh is the network of microservices that provide the various application services and control the interactions between them. It helps to manage microservices-based applications using two major components:

1. **Data Plane.** This is the component that performs the actual routing or communication of messages between microservices. It also gathers telemetry data, which helps to monitor the health and state of the services. The traffic that flows through the data plane is the application-related (business) data.
2. **Control Plane.** This is the component that provides an API to define policies. This API is often independent of the platform on which the microservices application runs. The control plane also helps the administrator populate the data plane component with a configuration that determines how to route traffic. The control plane is the brain of a service mesh. The traffic that flows through the control plane consists of messages of interaction between service mesh components.

The control plane may consist of multiple modules, and the distribution of functionality among these modules may be different in various service mesh offerings. However, they all provide the following core functions:

a. A module that parses the policy rules defined in the control plane and converts them into configuration parameters in the data plane module (i.e., the sidecar proxy). These policies may pertain to various functions, such as authentication, authorization, service discovery, traffic management (including load balancing), intelligent routing, blue-green deployments,

chooses to use as the cluster's ingress controller.

and canary rollouts. It may also include configuration parameters related to resiliency in the service mesh, such as timeout, retry, and circuit-breaking capabilities.

b. A module that provides all of the infrastructure functionality for authentication, authorization, and establishing a secure, encrypted session while two microservices communicate. These functions include user authentication, credential management, digital certificate management, and traffic encryption.

2.2.1 Service Mesh Functions for Reference Orchestration and Resource Management Platform

In order to describe the generic service mesh functions in the context of the reference platform – which, in this case, is Kubernetes – the deployment details of both the microservices application and service mesh components in that platform must be considered. Since authentication and authorization functions are the focus of this document, discussions for those functions on the Kubernetes platform will be confined to the functions in the service mesh.

Since the sidecar proxy code "implemented as a container" is hosted in the same pod as the microservice container, they share the same network namespace and are presented in the same node (e.g., VM or a physical machine). Both containers have the same Internet Protocol (IP) address and share the same IP Table rules. That allows the proxy to take complete control over the pod's network and handle all traffic that passes through it [10].

Taking the example of establishing a mutual TLS session, the proxy will interact with the module in the control plane of the service mesh to check whether it needs to encrypt traffic through the chain and establish mutual TLS with the backend pod. Enabling this functionality using mutual TLS requires every pod to have a certificate (i.e., a valid credential), and, since a good-sized microservice application may be hosted in hundreds of pods, this may involve managing hundreds of short-lived certificates. This, in turn, requires the service mesh to have a robust identity, access manager, certificate store, and certificate validation. In addition, mechanisms for identifying and authenticating the two communicating pods are required for supporting authentication policies.

A service mesh not only provides various application services during runtime but also supports the DevSecOps development and maintenance paradigm. The development team can concentrate their efforts on efficient development paradigms, such as application architecture (code modularity and structuring) and secure deployment (including interaction with sidecars and data flows through PEPs), without worrying about the implementation (development and deployment) of all infrastructure services, including many aspects of security. The service mesh is reference platform-aware and thus automatically injects sidecar containers into the pods. Once the service mesh inserts the sidecar containers, the combined team of developers, operations, and security teams can define policies, deploy them, and monitor their enforcements during runtime. These teams can also configure monitoring of the microservices applications without interfering with the functioning of the applications.

3 Attribute-based Access Control (ABAC) – Background

Attribute-based access control (ABAC) is an authorization framework or engine that computes decisions for user access requests based on attributes (properties) of users (subjects), application objects (resources), and the environment and policies expressed in terms of the attributes [5]. The advantages of ABAC for microservices-based applications using service mesh include:

1. Cloud-native applications span different domains and require increased precision in specifying policy by considering a large set of variables (i.e., multiple attributes and their associated values). Because of its scalability with respect to attribute-value stores and associated policies, ABAC can meet this requirement. Attributes and their values associated with subjects are assigned by system administrators, while those associated with application objects and environment are independently assigned by developers.

2. A policy is a logical expression that involves the attributes of the subject, object, and environment with an allowed/prohibited verdict attached to it. Policies based on the attributes do not create a tight relationship between a particular instance of a subject and an instance of an object, since attributes associated with them can take on any values and may change over time. The access decision to allow or disallow the request will be entirely dictated by the values of attributes associated with the instances at the time of the request.

3. Policies are expressed in terms of attributes without prior knowledge of potentially numerous users and resources that are or will be governed under those policies, and users and resources are independently assigned attribute values without knowledge of policy details. This dual feature enables access decisions on user requests to be based on centralized, enterprise-wide policies while also supporting the DevSecOps approach that provides autonomy to each microservice development team to make all decisions regarding their subset of applications (i.e., microservice), including the assignment of attribute values to their application objects.

Due to the features described above, the ABAC authorization framework is a natural fit for the class of cloud-native applications whose design is based on microservices (with each being developed and deployed by independent teams).

The ABAC framework has two standardized, representational structures.

1. One uses a platform-neutral text-based language called eXtensible Access Control Markup Language (XACML) Version 3.0, which has been standardized by Open Artwork System Interchange Standard (OASIS).
2. The other is Next Generation Access Control (NGAC), whose data structure and operations have been standardized under InterNational Committee for Information Technology (INCITS) 565-2020 [11]. This standardization includes the APIs of functional components (i.e., PEP, PDP, and Resource Access Point (RAP)), allowing for the interoperability of these components from different sources. Furthermore, the PEP interface is common for enforcing policies over both application requests and policy administration requests. The biggest advantage of NGAC is the use of linear time algorithms for computing access control

decisions and performing policy reviews (i.e., determining the set of resources that a user can access, determining the set of users who can access a resource) [12,13].

The functional architectures for these two representational structures are given in Figures 3.1 and 3.2, respectively. It must be mentioned that there are certain open-source access control engines, such as Open Policy Agent (OPA) and Casbin, that have the capability to define ABAC policies and enforce them just like XACML implementations. A brief description of the modules in these two functional architectures is as follows:

1. Policy Decision Point (PDP) – This is the core module of the ABAC functional architecture that computes decisions to permit or deny user access requests for performing actions on resources (application objects). Requests are received from and responses are sent to a module called Policy Enforcement Point (PEP) in both representations.

2. Policy Enforcement Point (PEP) – This module is part of the application's platform and is tightly integrated with the application. It is designed to intercept all access requests and either perform deny/permit actions on its own or consult an external PDP module.

3. Policy Information Point (PIP)

In the XACML representation, this is a module that contains the database of attributes and their associated values for all application-relevant objects or resources. The information here is used to extract the attributes and associated values for users and resources found in the access request. The extracted attribute values are then matched to the target clause, which in turn is used to find the applicable target policies in the Policy Retrieval Point (PRP) (described below).

The NGAC representation is a repository of association relations of the form (u-ai, op-i, o-ai) for a given pc-i, where u-ai and o-ai are attribute values associated with a user and object (resource), respectively[11]. op-i denotes a set of allowed operations, and pc-i is an instance of the governing policy class. An association relation can be generically defined as a *configured relation* that defines an allocation of *access rights* (e.g., read, write) among *policy elements* (attribute values of user and object) that enables certain modes of access.

To minimize the set of association relations in the authorization database (e.g., having triples to represent every user and object in the application), containment relations of the form (U < u-ai) are used to show the members of the user group and object group represented in the association relations. Instead of using the individual policy elements (e.g., u-ai, o-ai), the containment relation will use the powerset (2^{PE}) of the set of all policy elements PE, which contain all possible subsets of the set PE, in formulating association relations. In addition, the same set of containment relations is used to denote the applicable policies for each object as well (O < pc-i). Once the applicable policies for an object O are known, the association relations for that pc-i and all o-ai (attribute values of that object) will provide the allowable set of users and operations for the Object O.

4. Policy Retrieval Point (PRP) – In the XACML representation, this module is the repository for authorization policies expressed as logical formulas involving predicates on attribute values. The policy representation also contains the target resources that are covered by the policy. The resources requested in the access request are matched to these targets to retrieve the applicable policies by the PDP when computing decisions for those requests. This module is not part of the functional architecture in the NGAC representation.

5. Attribute Administration Point (AAP) – This is the interface for administering attributes stored in PIP in the XACML representation. This module is not necessary in NGAC representation since its association relations express the access rights on objects instantiated using attribute values.

6. Policy Administration Point (PAP) – This is the interface for administering policies stored in PRP.

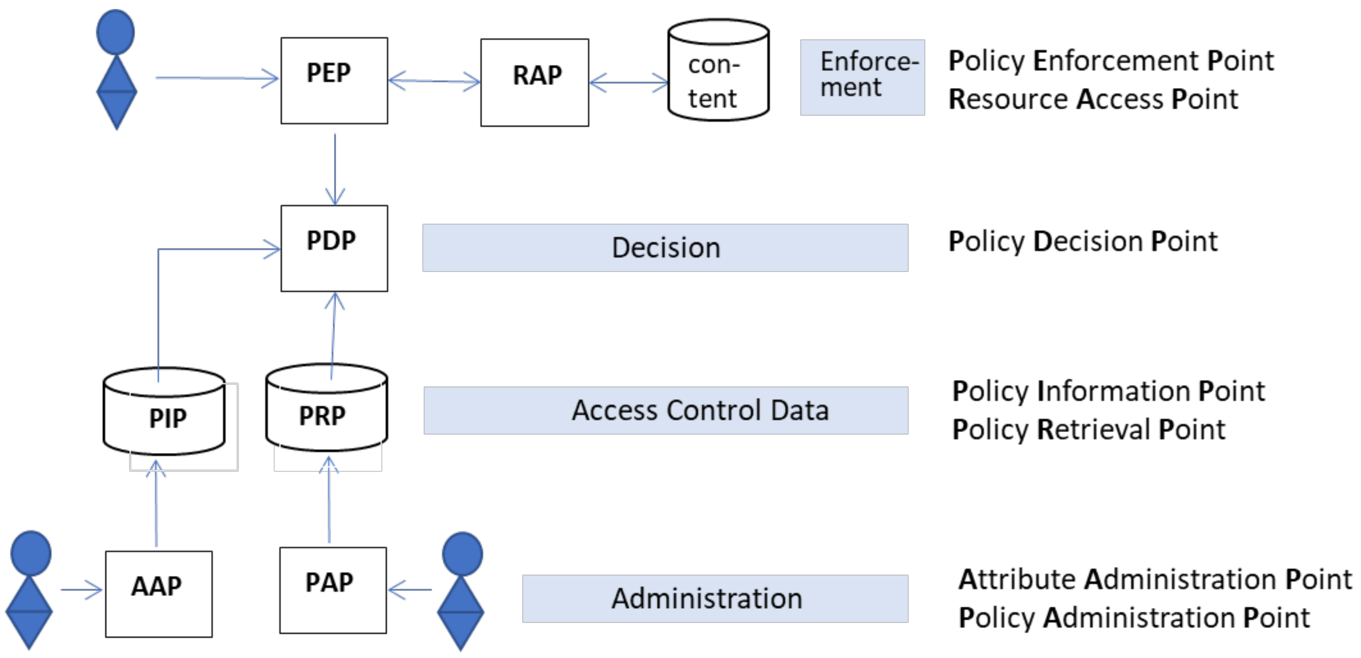

Figure 3.1 ABAC Functional Architecture based on XACML Representation

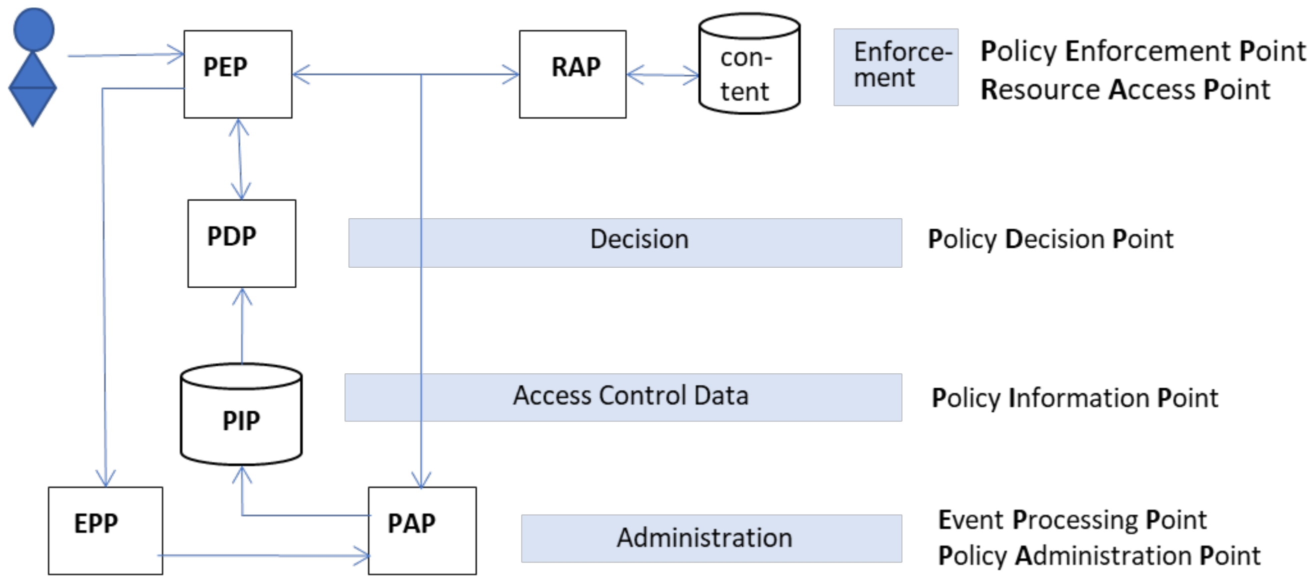

Figure 3.2 ABAC Functional Architecture based on NGAC Representation

3.1 ABAC Deployment for Microservices-based Applications Using Service Mesh

In the context of a microservices-based application using service mesh, an ABAC deployment can take the following forms:

- The proxies (e.g., Ingress, sidecar, and Egress) play the role of PEPs since they intercept all requests that emanate from each client, user, service, or external service.
- The PEPs enforce access control policies by performing the allow/deny actions based on the verdict provided by PDP.
- The enforcement function can be provided either natively (using local configuration structures, such as Access Control Lists [ACLs]) or using proxy extensions that call an external authorization server to obtain one or more of the data in the previous bullet.
- The assurance mechanisms in the service mesh (e.g., certificate-based authentication, secure session, non-bypassability, execution isolation) can be leveraged to deploy a high assurance authorization framework. Execution isolation is running each software component in a contained environment or sandbox-leveraging hardware virtualization that limits the damage of malware, limits the speed and propagation of worms and viruses, and efficiently detects attacks.

4 Authentication and Authorization Policy Configuration in Service Mesh

Fine-grained access control for microservices can be enforced through the configuration of authentication and access control policies. These policies are defined in the control plane of the service mesh, mapped into low-level configurations, and pushed into the sidecar proxies that form the data plane of the service mesh. The configurations enable the proxies to enforce the policies at application runtime (or request time), thus making the proxies act as Policy Enforcement Points (PEPs). As stated in the introduction, the objective of this document is to provide guidance for the deployment of an authentication and authorization framework that is external to the application and agnostic to the platform hosting the application and the service mesh product that implements the application infrastructure. However, Kubernetes is used as the reference application platform and Istio as the service mesh infrastructure platform to provide concrete examples of the concepts and to allow for recommendations with more clarity and specificity.

4.1 Application Orchestration and Resource Management Platform Configuration

The reference application orchestration and resource management platform configuration data for microservices-based applications using service mesh that are, at the minimum, needed for authentication and authorization policy configuration are:

- Metadata, like application service name and the sets of instances of that service hosted on the orchestration platform (such as Kubernetes)
- Runtime data, such as a service's protocols and ports
- Namespaces that provide logical isolation boundaries for sets of services
- Unique runtime identities for each service

In the instance of the reference platform Kubernetes, this is realized as:

- Service resource, which declares a service's name, protocol (e.g., TCP), and ports (e.g., 9080)
- Deployment resource, which declares deployment of pods that implement that service, including metadata such as labels and version
- Namespace construct and Role-based Access Control (RBAC) for managing how users are allowed to publish configuration into namespaces
- Service accounts, which are identities unique to each namespace bound to individual services

4.2 Service Mesh Configuration

The installation of any service mesh involves the following components:

- Ingress Gateway, which is the first point of entry into the microservices-based application. This gateway specification includes names, ports, and routes that the application client must take to access the application. In some implementations, an ingress gateway is optional as the sidecar itself can perform its functions.
- Egress gateway for the application to call outside services or applications. Just like an

ingress gateway, an egress gateway is optional as the sidecar itself can perform its functions.

- Injection of sidecar proxies (in the form of containers). The consequence of this is that each of the application's deployments in the platform will now have two containers – the original microservice container plus the mesh's sidecar proxy. These sidecar proxies enforce authentication and authorization policies during application runtime, thus acting as Policy Enforcement Points (PEPs). It must be mentioned that agents (e.g., OPA) can sometimes be used to play the role of PDPs for the authorization service. In addition, proxies should emit metrics and logs to enable continuous monitoring of the system, which can ensure that policies are in place and being enforced.
- A certificate authority (CA) module is needed to handle certificate requests from sidecar proxies, which need a runtime identity presented as an X.509 certificate. This CA generates, distributes, and manages keys and certificates used by the mesh and enables the mesh to perform automatic certificate rotation. The proof for the validity of the certificate is carried in the certificate itself by including a signed and dated "Online Certificate Status Protocol (OCSP)" server response as part of the certificate (called "OCSP stapling") via an X.509 extension.
- A control plane module in the service mesh that monitors configuration data in the hosting platform, encodes policies, and distributes those policies in the form of configuration to various proxies in the mesh (e.g., ingress, sidecar, and egress).

Having looked at the components in service mesh, we will now look at the minimal requirements for setting up mutual TLS authentication [14] for communication between Kubernetes workloads using these components as a pre-requisite for configuring and enforcing authentication and authorization policies. These requirements or recommendations come under initial service mesh configuration and are numbered using the acronym ISMC-SR-X, where ISMC stands for initial service mesh configuration, SR stands for security recommendation, and X is the sequence number. They include but are not limited to the following.

ISMC-SR-1: *If certificate-based authentication is used for authenticating service calls, the signing certificate used by the service mesh's CA module should be rooted in the organization's existing Public Key Infrastructure (PKI) to allow for auditability, rotation, and revocation.*

Some service meshes come with the ability to encrypt traffic using a self-signed certificate; such a certificate should not be used in secure deployments.

ISMC-SR-2: *Communication between the service mesh control plane and the application orchestration and resource management platform's configuration server must be authenticated and authorized.*

In this reference platform, authentication is typically achieved by the Kubernetes API server (the configuration server) with simple TLS. Authentication of the client is based on the pod's service account credential. Authorization for the client to receive platform information from the API server is enforced by Kubernetes RBAC.

4.3 Higher-level Security Configuration Parameters for Applications

Since the component microservices of the application are generally implemented as containers, the following higher-level security configuration parameters should be set. In the reference application orchestration and management platform Kubernetes, containers are implemented in pods, which contain a microservice container as well as a sidecar container. These higher-level security configurations are set through flags that come under the banner of pod security policies. The recommendations for these flag values are numbered using the acronym AHLC-SR-X, where AHLC stands for application higher-level configuration, SR stands for security recommendation, and X is the sequence number. They include but are not limited to the following [5]:

AHLC-SR-1: *Containers and applications should not be run as root (thus becoming privileged containers).*
In Kubernetes, the configuration setting for this is to set the value TRUE for "MustRunAsNonRoot" flag.

AHLC-SR-2: *Host path volumes should not be used, because they create tight coupling between the container and the node on which it is hosted, constraining the migration and flexible resource scheduling process.*

AHLC-SR-3: *Configure the container file system as read-only by default for all applications, overriding only when the underlying application (e.g., database) must write to disk.*
In Kubernetes, the configuration setting for this is to set the value of TRUE to "readOnlyRootFilesystem" flag.

AHLC-SR-4: *Explicitly prevent privilege escalation for containers*.
In Kubernetes, this is achieved by setting the value FALSE for the "allowPrivilegeEscalation" flag.

4.4 Authentication Policies

Authentication policies specify the process for validating identities. The integrity of this process and its strength determines the integrity of the authorization process since the latter depends on the strength of the authenticated identity. There are two types of identity needed in a microservices-based application:

1. Microservices or workload identity
2. End-user identity

Service (microservice) identity is critical for the following reasons:

- It enables the client to verify that the server with which it is communicating (server identity validated using the certificate it carries) is authorized to run the service. This assurance has to be provided by a secure naming service that maps the server identity to the service identity. In any orchestration platform (including Kubernetes), services can be moved around the nodes (server) for load balancing and service availability reasons. It is the responsibility of the control plane of the service mesh to refresh this mapping information

by interacting with the API that contains this configuration information (e.g., through API server in Kubernetes) and conveying it to the sidecar proxy in the data plane of the service mesh.
- The service identity is the basis for the target service to select and enforce applicable authorization policies.

4.4.1 Specifying Authentication Policies

Associated with these identities are the corresponding authentication processes that the service meshes have to support:

- Service-level authentication or peer authentication using service identity
- End user authentication or request authentication using end user credentials

It is assumed that the reference hosting platform has been configured with the high-level requirements outlined in Section 4.1. It is also assumed that the reference service mesh platform has been installed and configured with the initial requirements outlined in Section 4.2.

4.4.2 Service-level Authentication

Service-level authentication is the mutual authentication of the communicating services and setup of a secure TLS session. Enabling this requires the capability to define a policy object, which should meet the following requirements. These requirements are enablers for service-level authentication and are numbered using the acronym SAUN-SR-X, where SAUN stands for service-level authentication, SR stands for security recommendation, and X is the sequence number.:

SAUN-SR-1: *A policy object relating to service-level authentication should be defined that requires mTLS be used for communication. The policy object should be expressive enough to be defined at various levels (given below) with features for overrides at the lower levels or inheritance of the requirement specified at the higher levels.*

The following are the minimum required levels [6]:
- *Global level or the service mesh level*
- *Namespace level*
- *Workload or microservices level, used for applying authentication and authorization policies for a subset of traffic to a subset of resources (e.g., particular microservices, hosts or ports)*
- *Port level, taking into account that certain traffic is designed for communicating through designated ports*

This form of authentication also requires assigning a strong identity to each service, authenticating that identity by mapping it to the server identity (where the service is hosted) and establishing the authenticity of the mapping using a digital signature. One way of implementing this is through a special digital authentication certificate (Secure Production Identity Framework for Everyone (SPIFFE)). To provide assurance that the server whose identity is found in the SPIFFE certificate is the one that is authorized to run the target service, the following requirements (also specified in SP 800-204A) are needed.

SAUN-SR-2: *If the certificate used for mTLS carries server identity, then the service mesh should provide a secure naming service that maps the server identity to the microservice name that is provided by the secure discovery service or DNS. This requirement is needed to ensure that the server is the authorized location for the microservices and to protect against network hijacking.*

The information for mapping the server identity to a service is obtained by the control plane of the service mesh by accessing the configuration information from the platform that is hosting the microservices-based application. In Kubernetes, the control plane of the service mesh obtains the mapping information through the API server module of the Kubernetes platform and populates that information in the secure naming service. Thus, the mutual certificate validation not only enables validation of the associated service identities of both the client and target services but also enables creation of a secure mTLS session. In Istio, the policy object for this type of authentication is called "peer authentication."

4.4.3 End User Authentication

For the mesh to authenticate end user credentials (EUC), the application must participate in some way. Client services that make the request should acquire and attach an appropriate credential to each request (e.g., a JWT) in the request header. End user authentication, or request authentication, is the process of validating the credentials of the end user making a request by extracting them from the request's metadata and authenticating them (locally or against an external server). For example, a common flow at many organizations is to exchange an external EUC, like an Oauth bearer token, at ingress for an internal credential that is encoded within a JWT. The JWT can be created by a custom authentication provider or standards-based OpenID Connect provider.

The minimal information requirement in an end user authentication policy is designated using the acronym EAUN-SR-X, where EAUN stands for end user authentication, SR stands for security recommendation, and X is the sequence number.:

EAUN-SR-1: *A request authentication policy must, at the minimum, provide the following information and must be enforced by the sidecar proxy:*

- *Instructions for extracting the credential from the request*
- *Instructions for validating the credential*

For a JWT, this might include:

- Location (header name) of the JWT token that contains the user's claims
- How to extract the subject, claims, and issuers from the JWT
- Public keys or the location for the key used for validating the JWT

4.5 Authorization Policies

Authorization policies, just like their authentication counterparts, can be specified at the service level as well as the end user level. In addition, authorization policies are expressed based on constructs of an access control model and may vary based on the nature of the application and enterprise-level directives. Further, the location of the access control data may vary depending on the identity and access management infrastructure in the enterprise. These variations result in the following variables:

- Two authorization levels – service level and end user level
- Access control model used to express authorization policies
- Location of the access control data in a centralized or external authorization server or carried as header data

The supported access control in the service mesh uses abstraction to group one or more policy components (described below in Section 4.5.1) for specifying either service-level or end user-level authorization policies. Since microservices-based applications are implemented as APIs (e.g., Representational State Transfer (REST)ful API), authorization policy components described using key/value pairs will have attributes pertaining to an API, including the associated network protocols. The types of authorization policies are:

- Service-level authorization policies
- End user-level authorization policies
- Model-based authorization policies

4.5.1 Service-level Authorization Policies

Service-level authorization policies are defined using a policy object that provides positive or negative permission (authorization) with the following policy components:

a. The scope of the policy can span all applications at the service mesh level, namespace level, or one or more designated applications (microservice level).
b. The permissions or operations can be restricted to one or more designated methods of a given service (e.g., an "HTTP GET method on the '/details' path of an application named PRODUCT-CATALOG") or to designated ports through which an application can be accessed.
c. Conditions under which access can take place (e.g., possession of a token) are specified.
d. Sources allowed access are specified at the namespace or a particular service level (in terms of the service's runtime identity).

The requirements for a policy object for enabling service-level authorization policies are numbered using the acronym SAUZ-SR-X, where SAUZ stands for service-level authorization, SR stands for security recommendation, and X is the sequence number:

SAUZ-SR-1: *"A policy object describing service-to-service access should be in place for all services in the mesh. At a minimum, these policies should* ***restrict*** *access* ***to*** *the namespace level (e.g., "services in namespace A can call services in namespace B"). Ideally, policies should* *restrict access to individual services (e.g., "service Foo in namespace A can call service Bar in namespace B")."*

Policies should describe the minimum access required for application functionality (e.g., "service 'foo' in namespace A can perform 'GET /bar' on service 'bar' in namespace B").

4.5.2 End-user Level Authorization Policies

Given an authentication policy like Section 4.4.3, a sidecar in the mesh can extract a principal from the request to perform authorization on. Further, the sidecar typically has additional context about the request, including the resource being accessed (e.g., the path in an HTTP/REST API) and the action being taken (e.g., the HTTP verbs GET and PUT in the request to that API). This gives the sidecar enough information to act as a PEP and call a policy decision point.

This is the most common case, especially for organizations with traditional Identity and Access Management (IAM) systems that exist as an external service, often called by an Software Development Kit (SDK). To handle this case, a service mesh's sidecar proxy will typically support calling external services to render an authentication and authorization verdict. For example, the reference implementation Istio supports this via Envoy's (i.e., the sidecar proxy) external authorization service [15].

The requirements for enabling end-user level authorization policies are numbered using the acronym EAUZ-SR-X, where EAUZ stands for end-user level authorization, SR stands for security recommendation, and X is the sequence number:

EUAZ-SR-1: *When a sidecar communicates with an authentication or authorization system, that communication must be secured with either the mesh's built-in service-to-service authentication and authorization capabilities or using an existing enterprise Identity and Access Management (IAM) that is not part of the service mesh.*

An example is the external Software as a Service (SaaS) IAM, such as AuthO.

EUAZ-SR-2: *The sidecar should generate logs for every service request to ensure that authentication and authorization policies are enforced and relay telemetry data for the generation of metrics to ensure no degradation of service that will impact availability.*

End user authorization is not applied to the decision endpoint of the external authorization (PDP) service since the service is the principal making the call. It also avoids needing a default policy that allows all users to call the decision endpoint of the PDP. End user authorization should be applied to the PAP and other administrative endpoints of the authorization system, and applying that authorization can be facilitated by the mesh.

However, there is another case that is common enough to address in which an external authorization system is not required. Making a network call to an authorization service for every hop in a service chain can be expensive and cause centralized failures. To mitigate these problems, many organizations will exchange end user credentials at ingress for an internal, trusted, authenticatable credential that conveys not just the user's principal but also that user's capabilities in the system. A JWT is frequently used for this because it is locally authenticatable and conveys the user's principal (the JWT's subject), the issuer of the JWT (issuer), and arbitrary claims that the organization can control (e.g., to use for access control).

Performing end user authorization based on a JWT is built directly into Envoy, the sidecar proxy of the reference mesh Istio. Envoy can be configured with a filter [16] that will process requests in two steps:

1. JWT token verification involves extracting the token from the request header, verifying whether issuers and audiences are allowed, fetching the public key, and verifying the digital signature on the token.
2. Match the resources in the request to the claims in the token to determine whether the end user should be allowed access to the requested resources or denied.

Envoy's JWT filter acts as the PDP, making the access decision entirely locally. This requires that policy documents be small enough to reside on an individual sidecar proxy. Although a full ABAC is ideal for handling resource-level policies, the JWT filter is valuable as a steppingstone from a traditional system that only performs access control on the edge to a zero-trust system that performs authentication and authorization at each service. The limitation of this approach is that it becomes hard to revoke a user's access once a JWT is issued. Community standards, such as Internet Engineering Task Force (IETF) Request for Comments (RFC) for OAuth2 Introspection, can be used to address this.

An example of an authorization policy that will enable the sidecar itself to make an access control decision based on claims in the JWT and enforce it, without calling on an external authorization system, is given below:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
 name: backend
 namespace: product
spec:
 action: ALLOW
 rules:
 - from:
   - source:
       principals: ["cluster.local/ns/product/sa/frontend"]
   to:
   - operation:
       methods: ["GET"]
       paths: ["/info*"]
   - operation:
       methods: ["POST"]
       paths: ["/data"]
   when:
   - key: request.auth.claims[iss]
     values: ["accounts.google.com"]
```

Figure 4.1 – An example Istio authorization policy

This allows the front end to call specific methods on the backend only if the request has an EUC attached issued by "accounts.google.com."

EUAZ-SR-3: *All application traffic should carry end user credentials, and there should be a policy in the mesh enforcing that credentials are present.*

This is recommended even if the application is enforcing authentication and authorization independently of the mesh because these organization-wide controls allow functionalities like audit to be built on top of the mesh at a lower cost to central teams responsible for compliance and controls.

4.5.3 Model-based Authorization Policies

The service-level authorization policies and a use case of end-user authorization policies that uses JWT are natively implemented in the proxies. Since these cannot be used for resource-level authorization policies, support for model-based authorization policies is needed as well. As already alluded to in Section 4.5.2, this requires a call from the proxy to an external authorization server, which holds the model-based authorization engine to obtain an access decision.

The service principals in these model-based policies are identities (e.g., Service Account) provided by the underlying application orchestration platform (e.g., Kubernetes) and are the same as those used by authorization policies natively supported in the proxies. The user principals are usually obtained from the JWT. The popular access control models in the external authorization servers are RBAC and ABAC.

4.6 Authorization Policy Elements

The requirements that stipulate the minimal set of policy elements that should be present in authorization policies are given below and are numbered using the acronym APE-SR-X, where APE stands for authorization policy elements, SR stands for security recommendation, and X is the sequence number:

APE-SR-1: *The authorization policy should, at the minimum, contain the following policy elements*:

- *Policy types – Positive (ALLOW) or Negative (DENY)*
- *Policy target or authorization scope – the namespace, a particular service (application name), and version*
- *Policy sources – covers the set of authorized services*
- *Policy operations – specifies the operations on the target resources that are covered under the policy*
- *Policy conditions – the metadata associated with the request that must be met for the application or invocation of the policy*

4.6.1 Policy Types

Positive and negative policies are specified in order to set precedence relationships (e.g., DENY overrides and ALLOW). They are also used for situations that allow one type of policy for all services under a group and to specify exceptions (e.g., have an ALLOW policy for all services in a namespace but a DENY policy for a specified service)

4.6.2 Policy Target or Authorization Scope

This refers to the target resources in terms of a set of services, versions, and the namespaces under which the services are located. The service can be specified in the following ways:

- Using path: The location of the target resource is specified using paths (e.g., for resources accessed using HTTP or Google Remote Procedure Call (gRPC) protocols)). The list of paths to be included in the authorization policy scope and paths that need to be excluded can be defined. Both of these sub-elements of the policy target component (i.e., the list of paths to be included and the list of paths to be excluded) are optional.
- Using host name: In some instances, the target resources are specified using the host sub-element. The list of hosts to be included in the authorization policy scope as well as those hosts that need to be excluded can be defined. Both of these sub-elements of the policy target component (i.e., list of hosts to be included and the list of hosts to be excluded) are optional.
- Using network ports: The network port through which the target resource (the service) is accessed is often specified using the port sub-element. The list of ports to be included in the authorization policy scope as well as those ports that need to be excluded can be defined. Both of these sub-elements of the policy target component (i.e., list of ports to be included and the list of ports to be excluded) are optional.

4.6.3 Policy Sources

The policy sources are the set of services that are authorized to operate on the set of resources specified under the policy target (specified using name, path, host name, and ports). The policy sources are usually specified using a service account or name (called principal), all services in a particular logical group (e.g., namespace), or all services that are accessed from a group of network locations (e.g., IP blocks). Both included and excluded principals, namespaces, and IP blocks can be specified in some implementations.

4.6.4 Policy Operations

APE-SR-2: *The policy should cover all of the operations that are part of the application type. For example, if the application is implemented as a REST API, all of the operations (also called HTTP verbs or HTTP methods) that are part of the REST API must be included:*

POST: This is equivalent to creating a resource.

GET: This is equivalent to reading the contents of the resource.

PUT: This is equivalent to updating the resource by replacing.

PATCH: This is equivalent to updating the resource by modifying.

DELETE: This is equivalent to deleting the resource.

If the resource is accessed using gRPC instead of a RESTful protocol, there is only one operation or method: "POST." The authorization policy definition may also have a feature to specify the list of operations (methods) to be excluded. Both policy sub-elements – one to specify the operations to be included in the authorization policy scope and the other to be excluded – are optional.

4.6.5 Policy Conditions

Policy conditions specify the constraints in the form of a key-value pair for the metadata associated with the request. This metadata may cover the following:

- *Metadata associated with the source*: Some of the metadata (e.g., service account name, namespace, and IP blocks) are specified as part of the policy source specification itself. In addition, it is possible to list IP addresses in Classless Inter-Domain Routing (CIDR) format of the policy sources.
- *Metadata associated with the request*: In this type of metadata, the parameters or attributes that pertain to a specific request can be specified. These parameters can include an audience that can present the authentication information expressed in the form of a Uniform Resource Locator (URL) (only applicable to HTTP protocol-based requests), a specific end user identifier associated with the audience that can present the authentication credentials, or the claim name that is carried in the token presented by the presenter. In addition,

parameters that pertain to the user-agent (e.g., browser name) can also be specified for HTTP protocol-based requests.
- *Metadata associated with the destination*: The range of allowable IP addresses can be specified in CIDR format as well as the associated list of ports.

4.6.6 Default Authorization Policy

APE-SR-3: *A default policy should be authored in the system that rejects all requests that are unauthenticated, mandates that service and end-user credentials be present on every request, restricts all communication to services within the application's own namespace, and allows service communication across namespaces only through an explicit policy*.

5 ABAC Deployment for Service Mesh

The last section introduced three different types of authorization policies, including two use cases for end-user level authorization policies. This section will leverage those architectural choices to describe an ABAC-based authorization framework in the service mesh:

- Security assurance for authorization framework enforcement
- Supporting infrastructure for authorization requests
- Advantages of an ABAC authorization framework for service mesh
- Enforcement alternatives in proxies

5.1 Reference Monitor Concept in Authorization Framework

The authorization policy enforcement mechanism implemented in the service mesh for a microservices-based application must satisfy the three requirements of a reference monitor concept. It must be 1) non-bypassable, 2) protected from modification, and 3) verified and tested to be correct. These three requirements can be ensured by the following:

- Every request from a client to the microservices-based application, from one service to another (inter-services call), and from a microservice to an external application is intercepted by the ingress gateway, sidecar proxy, and egress proxy, respectively. These PEPs are non-bypassable.
- The policy enforcement modules are independent executables that are decoupled from the application logic and cannot be modified.
- Their outcome can be independently verified and tested through both shadow operations and live production requests.

In short, a proxy running in the data plane of the service mesh is the reference monitor with respect to authorization enforcement. The authorization policy engine (e.g., NGAC-based ABAC policy engine) implemented as a container executing either natively in the proxy memory space or callable from a corresponding filter module in the proxy runs as a separate process that does not share any memory space with the calling application. Hence, it satisfies the requirement of a security kernel.

5.2 Supporting Infrastructure for ABAC Authorization Framework

Now consider the basic building blocks of the supporting infrastructure for service-to-service and end user + service-to-service requests.

5.2.1 Service-to-Service Request (SVC-SVC) – Supporting Infrastructure

The policy object used for authorizing this type of request was described in Section 4.5.1. Service-to-service requests must be authorized based on the identity of the calling and called services. The trusted document that carries the identity of the service is an X.509 certificate issued by one of the control plane components of the service mesh after verifying whether the requested identity is valid for the microservice by consulting an identity registry. The proxy communicates with this control plane component through a local agent, obtains a certificate, and sends it to the proxy, which then performs the certificate validation process on behalf of the calling service or client during each service request. The identity is encoded as URI and carried in a certificate's SAN (subject alternate name) field. It must be mentioned that the certificates that carry service account identities are short-lived certificates (rotated every hour or few hours) rather than the conventional HTTPS TLS terminating certificates whose validity lasts for several months.

5.2.2 End User + Service-to-Service Request (EU+SVC-SVC) – Supporting Infrastructure

The policy object used for authorizing this type of request was described in Section 4.5.2. This request type requires the verification of two identities: the calling user identity and the service identity. As described in the previous section, the service mesh provides the feature to perform authorization based on service identities. Since this is a standard feature, no extra components need to be built in the service mesh infrastructure for this type of authorization. However, when end user identities are introduced for authorization, the authorization framework should be tightly integrated with the following components of the architecture:

- The service orchestration control plane for obtaining application object attributes as well as attributes of the registered application users (which includes user credentials), thus playing the role of PIP in ABAC-based authorization
- A service mesh control plane for obtaining tokens that encode the claims based on the authorization decision
- A service mesh data plane in the service proxy for making calls to the authorization engine (which is just another service), obtaining the authorization decision, enforcing the service-to-service authorization policies, making calls to the service mesh control plane for authorization tokens (e.g., JWT), and attaching the tokens to the service request

An advantage of an EU+SVC-SVC request processing scheme is that authorizations at a finer level of granularity than the method level can be specified, and conformant claims can be included in the authorization token. A disadvantage is that there is overhead involved in enforcing two layers of authorization – one layer based on policies specified for SVC-SVC requests and a second layer based on EU+SVC-SVC requests. Access control processing logic based on the second layer involves multiple calls by service proxy, such as (1) a call to the authorization engine service to obtain the access decision after obtaining the user attributes (including user credentials) and application object attributes from the orchestration system, (2) obtaining the authorization token from the service mesh control plane based on the access decision, and (3) including the authorization token along with the service request.

5.3 Advantages of ABAC Authorization Framework for Service Mesh

This section provides the justification for the various building blocks of the architecture for the authorization framework (e.g., the service mesh and NGAC-based ABAC model). It also highlights the scalability and flexibility of certain components such as proxy APIs and the NGAC authorization engine.

a. A service mesh is the right architecture for the enforcement of authorization policies since the components involved are moved out of the application and executed in a space where they can form a security kernel that can be vetted.

b. Both types of authorization requests (i.e., SVC-SVC and EU-SVC-SVC) can be handled by a runtime infrastructure that involves the coupling of an orchestration platform control plane, service mesh control plane, and mesh data plane to the access control engine.
c. The extensible API of the proxy can be used to integrate any authorization engine using the appropriate type of access control model. ABAC has been found to be one of the most flexible, scalable access control models because of its ability to incorporate any number and type of attributes associated with the subject, object, and environment.
d. Performance requirements for the authorization engine are met due to the linear time processing speed of the graph-based, NGAC-based ABAC model.
e. The flexibility outlined in (c) can be leveraged to incorporate models for both application and data protection by making data protection models as part of the authorization server.

5.4 Enforcement Alternatives in Proxies

Authorization can be enforced through a native structure (e.g., authorization policy) supported in the particular version of the service mesh or using calls to an external authorization server. The external authorization server can use any access control model and any representation of policy expressions (logical rules or acyclic graph representations), but the mediation of a request coming into the proxy can be performed in the following ways:

a. Each request is passed on to the external authorization server through the external authorization filter in the proxy, and the response from the authorization server is used for request mediation in the form of ALLOW or DENY.
b. Prestored ACLs can be used in the proxy itself, generated by calls to the authorization server. If the authorization server uses an enterprise-wide access control model, an administrative API may be needed that will perform the function of mapping the enterprise resources to resources, users, and groups pertaining to the service served by its proxy to generate ACLs that are customized for the service.

6 Summary and Conclusions

Deployment guidance has been provided for an ABAC-based authorization framework for securing microservices-based applications using a service mesh. Background information on a reference platform for orchestration and resource management of microservices-based applications in terms of constituent building blocks and functional capabilities is outlined in Section 2. In the same section, the architectural layers and features of a reference platform for service mesh are described. Section 3 provides background information on ABAC and discusses two representational structures and associated functional architecture.

Section 4 provides the building blocks of the authentication and authorization framework – the set of recommendations for settings/defaults of those features of the service mesh that provide the operational assurances for applications, specifically for implementing authentication and authorization policies. The recommendations span mechanisms for supporting both end user and service level authentication and authorization policies. The minimal set of policy elements needed in authorization policies is also outlined.

The presence of a reference monitor concept in the authorization framework, the description of the supporting infrastructure for implementing the framework, the advantages the service mesh provides for implementation of an ABAC authorization framework, and the enforcement alternatives in proxies form the content for Section 5.

The list of all recommendations for deployment of an ABAC-based authentication and authorization framework for microservices-based application using a service mesh is given in Appendix A.