DaaS

Distributed App Access & Security

ensurity

info@ensurity.com

Table of Contents

1. What problem are we trying to solve?

Cloud adoption is rapidly gaining traction, even among enterprises & government clients who are looking to benefit from the superior efficiencies of scale and better analytics. Initial growth in cloud was more monolithic in nature, but the current spurt in growth is characterized by (a) move to multi-cloud & (b) containers & microservices taking the processing to the edges & ease of application rollout. The result is the cloud becoming more distributed & existing identity & security solutions are not sufficient by themselves.

The distributed nature of cloud calls for distributed identity, authentication & authorization architecture. Ensurity is trying to solve this problem by offering inter-service authentication using patented non-linear secret sharing.

Following are the key issues that need addressing:

1.1 Secrets management:

In monolithic cloud architecture, secrets are generated & secured inside HSM type hardware, further protected by perimeter-based security solutions. As the cloud sprawl increases & microservices are processed near the edge, centralized secrets management introduces cost & performance inefficiencies.

1.2 IP based security limitations:

Monolithic cloud, like on-prem, relies heavily on IP and port addresses for communication, authentication & authorization. IP addresses are short-lived in a distributed cloud environment & would be less effective in security. IAM (Integrated Access Management) based security model is better suited in this scenario. Dynamic identity generation models at the application level would result in better security & lower latencies.

1.3 Zero Trust Security:

Zero Trust is a security concept centered on the belief that organizations should not automatically trust anything inside or outside its perimeters and instead must verify anything and everything trying to connect to its systems before granting access.

With Inside network is not same as automatic access & free lateral movement. Privileged access should lead way to minimal, fine grained access. Every action must be explicitly authorized. Authorization should be context aware. IP based network access & privileged access management results in coarse access security. Solutions based on micro-authentication & access architecture defined by fine grained object identity management systems will help in achieving Zero Trust Security.

1.4 Service-to-service authentication:

Peer-to-peer services authentication & authorization works better in distributed cloud environments. However, such authentication needs to be tied to centralized IAM & policy directives. Stateless token-based models are used in transmitting stateless authentication & authorization.

1.5 Scalability:

Services need to authenticate & authorize to each other at ultra-scale. Rising sprawl & distribution of services further increases the complexity. JWT tokens using OAuth2/OIDC are good for distributed, stateless authentication, but there are a few key issues:

a) **Loss of JWT tokens:** There is no second defense against loss of JWT tokens – they are bearer tokens, hence a strong second line of defense is required in case of a loss. Currently, JWT tokens are made short-lived to minimize the risk of loss. Short-duration tokens though (i) limit flexibility by not allowing long duration tokens that may be needed in some cases (ii) require additional hops for regeneration of tokens (iii) need two tier token structures with intermediation by API gateways & (iv) does not fully eliminate the risk.

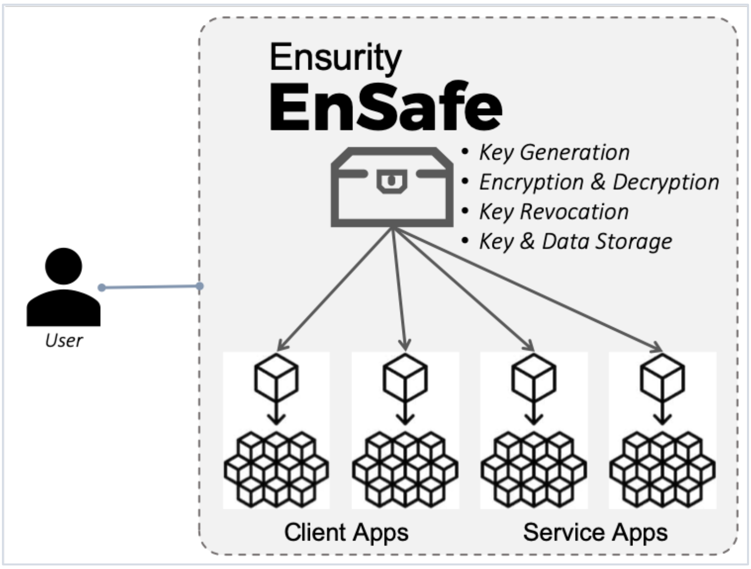
b) **Lack of strong encryption:** JWT tokens are just Base64 encoded. Adding stronger encryption currently requires additional overheads & secrets management.

Ensurity offers JWT tokens that can be managed dynamically without being constrained by the limitations of security, scope & latency thus resolving these key issues and offering a scalable solution.

2. Ensurity Solutions

2.1 Secret Management Service

Ensurity EnSafe is a secure dynamic infrastructure for managing secrets by leveraging trusted identities across distributed cloud infrastructure. As applications & microservices sprawl across clouds & move closer to edge, centralized secrets management based on HSM type of hardware is not effective. EnSafe is a software-based secrets management solution that can be containerized & managed on enterprise's own hardware infrastructure, closer to the applications.

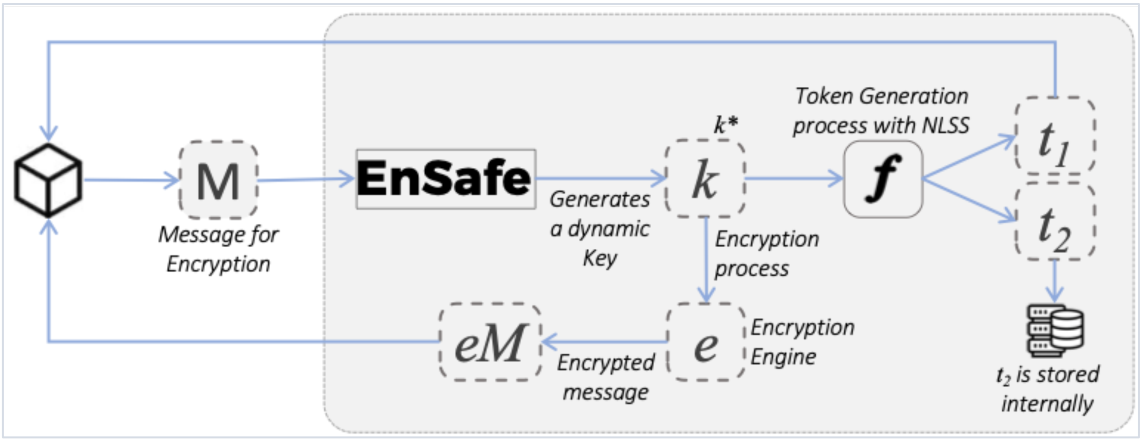


EnSafe is a ‘Secrets Management Service’ that stores, accesses and distributes dynamic secrets such as tokens, passwords, certificates, and encryption keys, to client & service applications. EnSafe can also be used to store customer data such as credit cards.

EnSafe assures security & privacy as none of the secrets are stored completely on the EnSafe databases. Only partial elements of the secrets are stored on EnSafe servers. A breach of EnSafe does not compromise client secrets. Similarly, applications are also secure since they do not store full secrets.

2.1.1 Encryption Process

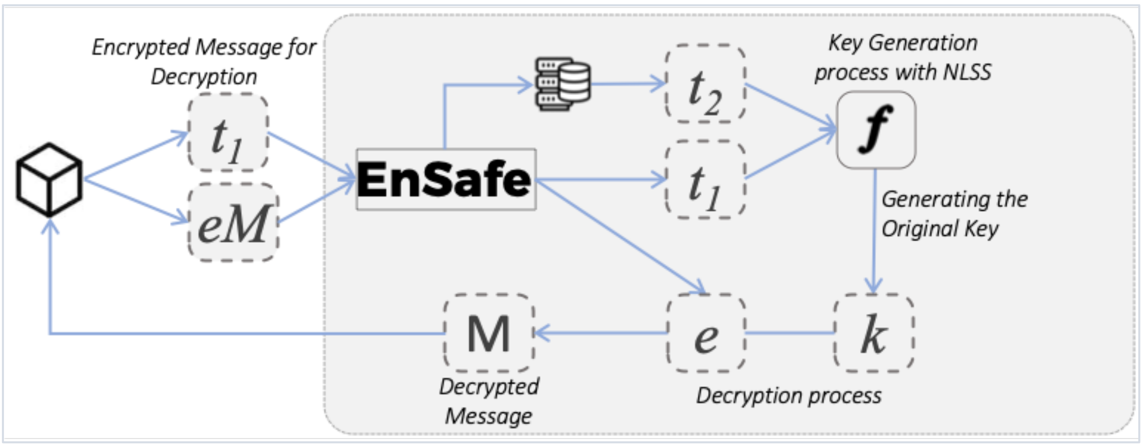
Let us look at a scenario where an application is trying to send an encrypted message/secret to another application.



- Application 1 submits message M to EnSafe for encryption.
- EnSafe generates a Dynamic Key [k]. EnSafe encrypts the message M with the key k. The encrypted message eM is sent back to the Application 1.
- EnSafe splits the key k into token shares using the Non-Linear Secret Share (NLSS) methods. Following this, the key k is destroyed.
- During the Token Generation, as per the configuration, [t_1] and [t_2] will be created.
- EnSafe transmits [t_1] Application 1; and securely stores [t_2] internally in a database.

2.1.2 Decryption Process

Services send a message to EnSafe for decryption.


Encrypted Message for Decryption
t_1
eM
EnSafe
t_2
t_1
Key Generation process with NLSS
f
Generating the Original Key
M
Decrypted Message
e
k
Decryption process

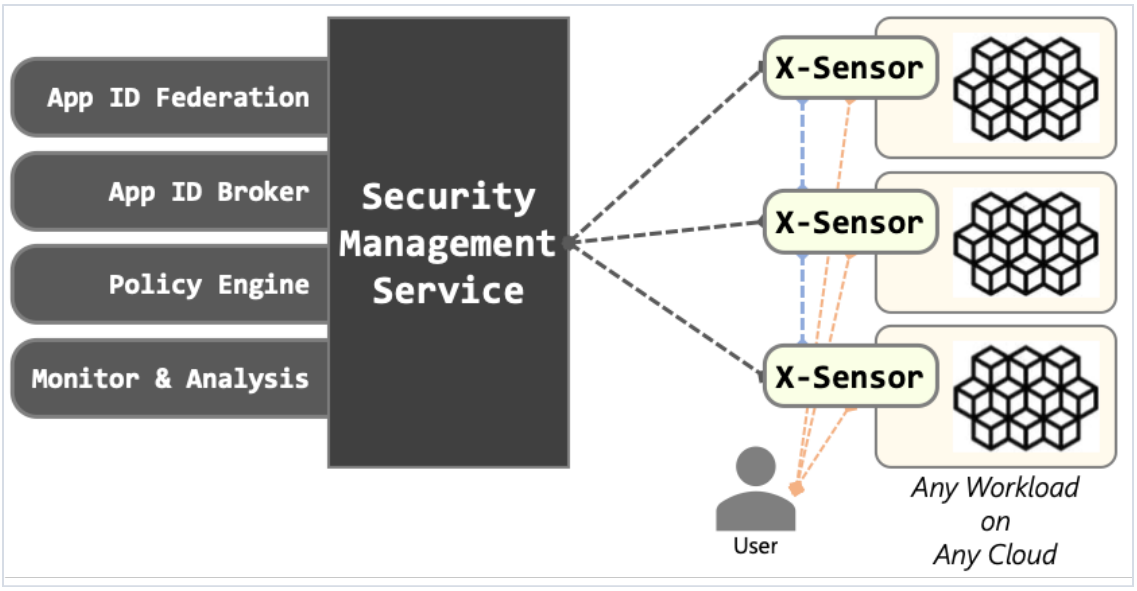
- Application 1 sends the encrypted message eM & t_1 to Application 2.
- Application 2 submits t_1 & eM to EnSafe.
- To generate the original Key [k], EnSafe extracts the [t_2] secure share from the internal database and recreates the original key [k] using NLSS.
- Once the original Key [k] is generated, using the configured encryption engine [e], the ciphered message will be decrypted and pushed back to the Application 2.

2.2 Distributed App Identity Architecture

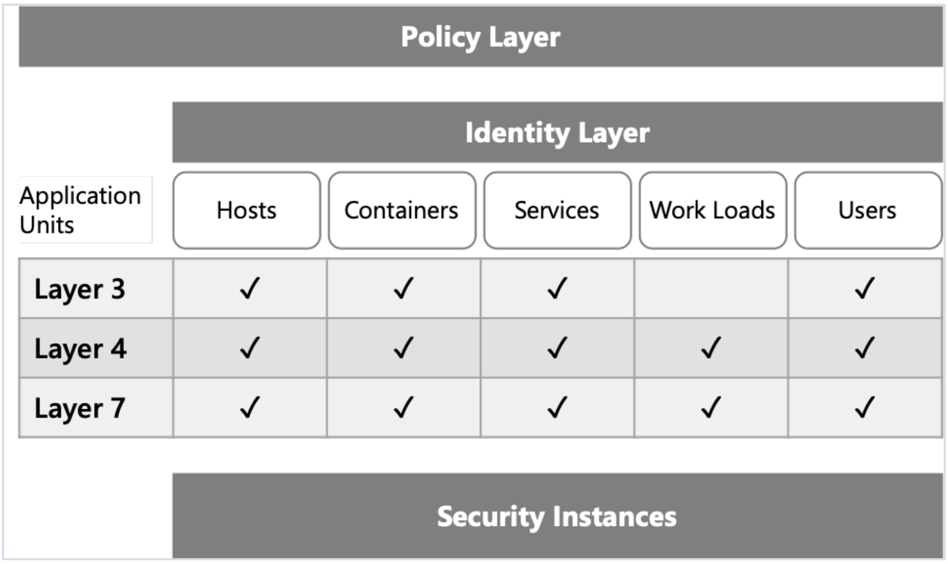
Zero Trust Security architectureAll applications (host applications, containers, services, functions & workloads) are given a unique, normalized Ensurity identity. Using a standard, distributed identity architecture is important for an identity driven security architecture.

A unique ID is created for every Host app. The UID is a signed JSON document, comprising of static & dynamic app metadata. Every instance or microservice created out of the Host app gets a derived ID that is a function of the Host UID as well as own static & dynamic metadata. The tiered ID architecture will make it easier to integrate distributed identity systems.

Ensurity Security Management Service (SMS) provides controlled access to tokens, passwords, certificates, encryption keys for protecting secrets and other sensitive data.



With normalized, Ensurity identities (a) all apps can authenticate & communicate across the cloud sprawl (b) can communicate over L3, L4 & L7 modes and (b) access control is fine grained, resulting in Zero Trust Security environment.



Policy Layer

Identity Layer

Application Units	Hosts	Containers	Services	Work Loads	Users
Layer 3	✓	✓	✓		✓
Layer 4	✓	✓	✓	✓	✓
Layer 7	✓	✓	✓	✓	✓

Security Instances

2.2.1 X-Sensor

X-Sensor is a proxy layer for Microservices to be able to communicate with CMS server.

The X-Sensor performs the following processes:

- Setup Functional Rules & Firewall settings for Microservices, as defined in the CMS
- Encryption engine (for L3, L4 and L7 layers)
- Obtain Key from CMS
- Layer-7 authentication
- Identity of X-Sensor
- Identity of Microservice Applications
- Identity of User

- Peer-to-peer authentication between Microservices

2.3 Services authentication & claims validation

Stateless tokens like JWT tokens are bearer tokens. Ensurity's solution addresses some clear gaps in the stateless (JWT) token architecture. With Ensurity's solution, JWT tokens can be managed dynamically without being constrained by the limitations of security, scope & latency.

2.3.1 Token Revalidation

Let us look at a scenario where Client-Application 'C_1' presents claims in the form of JWT token T to Service Application 'S_1'. A loss of the token T in transit will impede C_1 from accessing S_1. C_1 needs to revalidate identity with IdP server to gain an alternative token. Furthermore, if the token is stolen by a third app C_2, C_2 can successfully make claims to S_1. To verify that the token bearer is the originally intended app C_1, Ensurity's solution creates an extra sense of T which can be used dynamically between both the apps CA & SA without needing to check the state with a server. The process of how the extra secret sense is embedded into the JWT token is explained in the section 2.3.2.

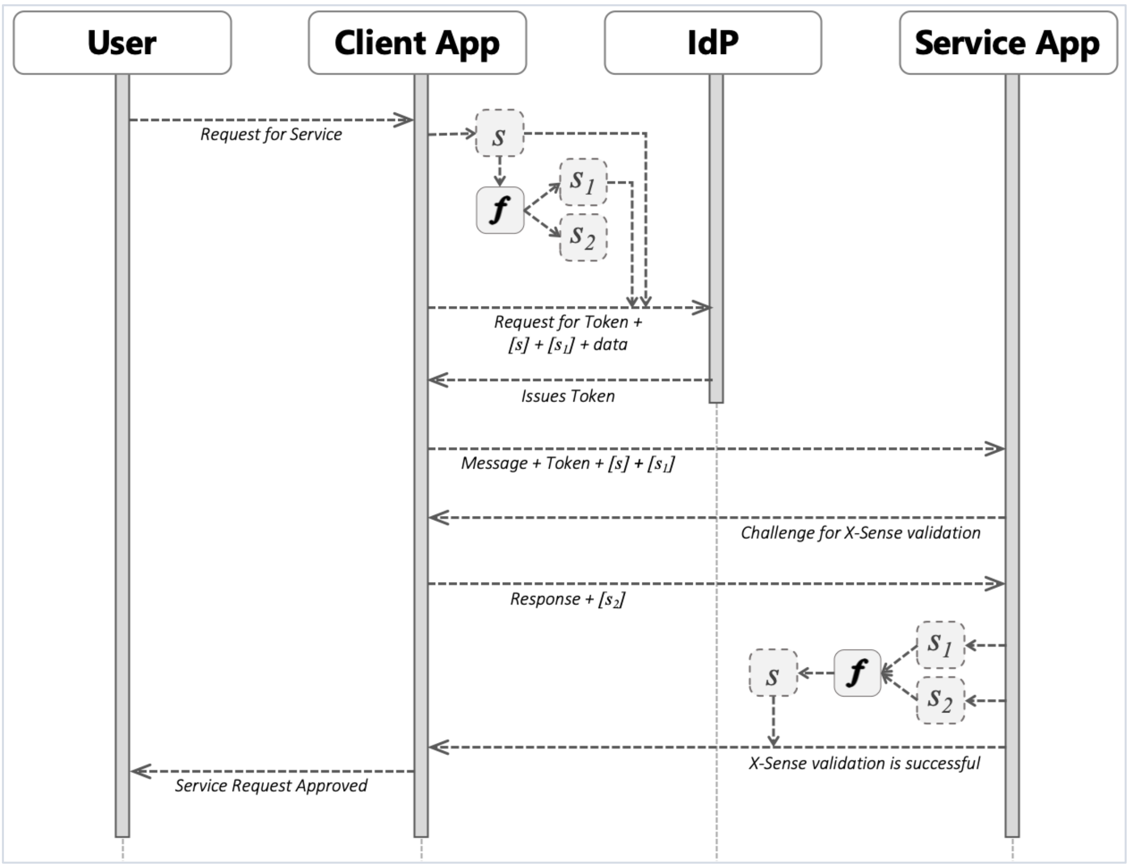
2.3.2 Token Management

On receipt of Service Request by the User, the Client App (CA) initiates a request for Token from the IdP server. During the request, the Client App generates a Secret [s] and generates additional secrets — [s_1] and [s_2] from [s]; and appends [s] and [s_1] to the request. After validating the identity of the Client App, the IdP server generates & issues a JWT Token. Before issuing the token, IdP server embeds [s] & [s_1] inside the token - [s] & [s_1] becomes part of the cryptographically signed stateless token.

In order to access the Service App, the Client App will furnish the Token to the Service App for validation and access. Upon receiving the Token, the Service App first validates the Token for properness & signatures. The Service App also parses the Token for [s] & [s_1]. The Service App then initiates a Challenge-Response schema with the Client App using [s] & [s_1]. The Client App responds to the challenge using the secret retained with itself [s_2]. Both the Service & Client Apps are able to complete the Challenge-Response process without needing to share [s_1] or [s_2] with each other at point of time.

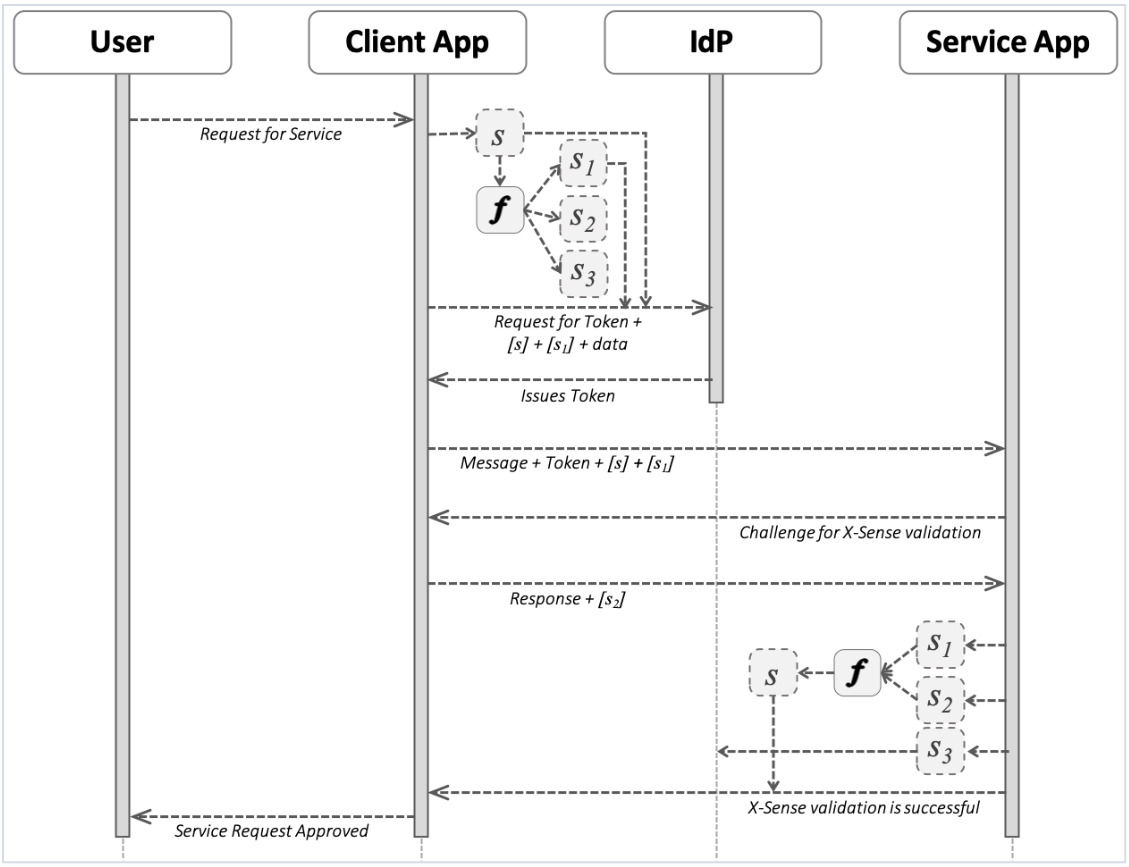
Let us look at a scenario wherein the JWT Token is lost. The Client App is able to request a replacement token from the IdP server by proving that it has the secret [s_1] using the same Challenge-Response mechanism.

In another scenario where the Token is stolen by an impersonating app CA_1. While CA_1 is able to provide proof of the bearer token, it is unable to offer the extra proof of the secret [s_1]. Thus, the loss of the bearer token does not result in the loss of the access. IdP servers can issue long-duration JWT Tokens without worrying about the loss of the tokens. Replacement of tokens is also relatively easy without needing to go through the IAM process again. This eliminates the need for an additional layer of refresh tokens & also API gateways validating tokens.


User
Client App
IdP
Service App
Request for Service
s
f
s1
s2
Request for Token + [s] + [s1] + data
Issues Token
Message + Token + [s] + [s1]
Challenge for X-Sense validation
Response + [s2]
s1
s2
f
s
X-Sense validation is successful
Service Request Approved

2.3.3 Revocation of Tokens

Section 2.3.2 explained how peer-to-peer validation is made possible for JWT tokens. The Tokens sometimes need to be revoked too by a centralized authority. For revocation scenario, secret [s] is split into three secrets [s_1], [s_2] & [s_3] (as opposed to two secrets explained in 2.3.2). While the rest of the process remains the same as in 2.3.2, the additional secret [s_3] can be stored at the IdP server or any other trusted server. To check if tokens are revoked or not, the Service App will revalidate also with the trusted server using a similar Challenge-Response schema as detailed in 2.3.2.


User
Client App
IdP
Service App
Request for Service
s
f
s_1
s_2
s_3
Request for Token + [s] + [s_1] + data
Issues Token
Message + Token + [s] + [s_1]
Challenge for X-Sense validation
Response + [s_2]
X-Sense validation is successful
Service Request Approved

2.3.4 License Management

Another tricky issue is management of licenses (how many instances to be run or how many times an instance can be accessed). This is typically seen during trials or PoCs of software. With Ensurity solution, one access (JWT) token can be used to create multiple licenses using the NLSS secret shares. For example, one token T_1 can be used to create multiple pairs of split shares T_1, T_2; T_3, T_4; T_5, T_6; …

Once the client application runs out of the tags, license automatically expires – there is no need to create multiple JWT tokens for the same. All pairs are orthogonally different.

Similarly, same JWT token can be split into multiple secret share pairs. Different client applications can use different pairs to access the service application. This is particularly useful in scenarios where a service application gives same access rights to multiple client applications (well defined access). Instead of creating multiple JWT tokens for each client application, same token can be used by many client applications.

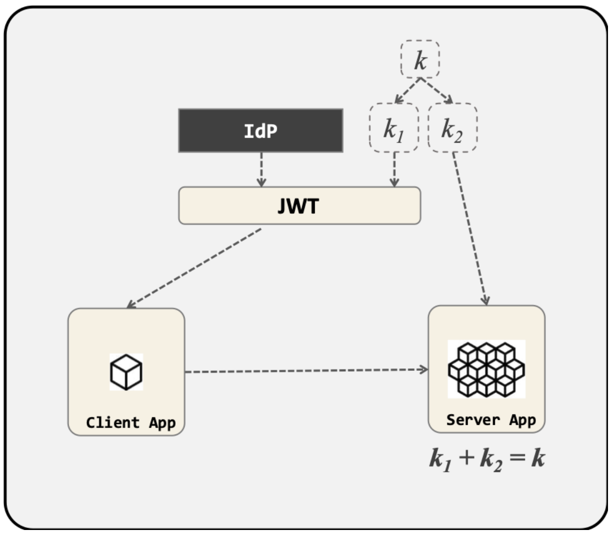
2.3.5 Peer-to-Peer Authentication using HMAC signed JWT tokens

Secrets management is a challenge in services authentication. JWT tokens can be signed either by HMAC or RSA/ECDSA signatures. HMAC signatures are cheaper & easier to scale in a distributed cloud environment. However, HMAC is not commonly used due to key drawbacks (a) secure sharing of HMAC symmetric keys & (b) securing the password secrets that derive HMAC keys. To resolve these issues, IdP server signs the JWT tokens with HMAC based on the secret.

2.3.5.1 Key transmission

HMAC keys are transmitted over the network as split secret shares. Since the NLSS algorithm is keyless in nature, HMAC keys can be shared over the network to the service application without the need for additional key management.

The IdP server splits the key [k] into two secrets [k_1] & [k_2]. [k_1] is embedded into the JWT token. [k_2] is transmitted to the Service App. The IdP server deletes the key subsequently (the server does not necessarily need to store the HMAC secrets anymore). Upon receiving the JWT token from Client app, the Service App parses the token for [k_1] & combines with [k_2] to recreate the original key [k]. This process ensures that the key [k] is securely transmitted. Moreover, the decryption happens only with the presentation of the token.



k
IdP
k_1
k_2
JWT
Client App
Server App
$k_1 + k_2 = k$