**Policies in Mule 4**

**Client ID Enforcement**

The Client ID Enforcement policy restricts access to a protected resource by allowing requests only from registered client applications. The policy ensures that the client credentials sent on each request have been approved to consume the API.

When a client application is registered in Anypoint Platform, a pair of credentials consisting of a client ID and client secret is generated. When the client application requests access to an API, a contract is created between the application and that API. An API that is protected with a Client ID Enforcement policy is accessible only to applications that have an approved contract.

**Basic Authentication: Simple**

The Simple Authentication policy protects an API by forcing applications to provide a username and password when making requests

When you apply the Simple Authentication policy to an API, a request to that API must contain the following header:

Authorization: Basic <username: password>

**Rate limiting –SLA based**

The Rate Limiting policies based on a service level access (SLA) are client ID-based policies that use the client ID as a reference to impose limits on the number of requests that each application can make within a period of time. To use these policies, we need to create at least one SLA tier to define request limits with either manual or automatic approval

**Basic Authentication: LDAP Policy**

The Lightweight Directory Access Protocol (LDAP) authentication policy specifies how to restrict access to an API using LDAP authentication mechanism.

LDAP is an inverted tree, and each leaf has a username-password pair and associated metadata. Each level is like a tree branch

The Basic Authentication - LDAP policy intercepts the request to the protected resource and looks for the Authorization HTTP header.

The policy then extracts the username and password encoded in Base64 and then requests the configured LDAP instance to determine if the user credentials are correct in the provided LDAP context.

**Tokenization Policy**

Tokenization is the process of masking a value or piece of information that can be considered sensitive data into a token that can be mapped back to its original value using detokenization concepts. If your API contains sensitive data, the tokenization policy is a highly effective way to protect it.

Tokenization minimizes the risk of matching the sensitive data to the original value, in case a third party accesses the data. For example, you can use this policy to hide credit card numbers, personally identifiable information (PII), protected health information (PHI), and similar sensitive data.

**Detokenization Policy**

Detokenization is the process of returning the previously masked sensitive data back into its original value to reduce the risk of compromising sensitive information. You can tokenization your data to replace sensitive information with tokens to reduce the risk of compromising sensitive information.

For example, an application can require the bank account number or the credit card number for generating monthly credit card statements. Detokenized sensitive data must always be read under strict security controls.

The Detokenization policy returns tokenized information within the payload to its original values. To define which data in the payload is detokenized, you must provide a selector expression.

**Cross origin resource sharing**

CORS is a mechanism by which a web application can access resources that are defined in another domain. Using CORS, applications invoking JavaScript XMLHttpRequest (XHR) calls from a web page can interact with resources from non-origin domains.

Typically, web browsers implement this standard for allowing cross-origin requests. Your API is categorized as a public resource by default when you apply this policy. If you want to share only specific resources of your API, you can configure the policy as a restricted resource.

**JWT**

JSON Web Token (JWT) is a URL-secure method of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS), or as a JSON web encryption (JWE) structure in plain text. This enables the claims to be digitally signed and integrity protected with a message authentication code (MAC). Because the token is signed, we can trust the information and its source.

The JWT Validation policy validates the signature of the token and asserts the values of the claims of all incoming requests by using a JWT with JWS format. The policy does not validate JWT that uses JWE.

**OpenID Connect OAuth 2.0 Token Enforcement**

The OpenID Connect OAuth 2.0 Token Enforcement Policy restricts access to a protected resource, by only allowing HTTP requests if the token provided in such request is a valid one and, optionally, the required OAuth scopes are fulfilled. The policy validates the token, by connecting to a OpenID Connect authorization server. The token is obtained specifying the credentials of an [authorized client application](https://docs.mulesoft.com/api-manager/2.x/api-contracts-landing-page)

When you apply the policy, you may optionally define a space separated list of [OAuth 2.0 scopes](https://tools.ietf.org/html/rfc6749#page-23) to be enforced by it. OAuth 2.0 scopes are a way to further limit access to a resource protected by OAuth. You may define words like READ, WRITE, or some others that make sense in the context of your organization (eg. CONTRACTOR, PUBLIC, EMPLOYEES\_ONLY, etc).

**Xml Threat Protection**

Applications processing XML requests are susceptible to attacks characterized by unusual inflation of elements, attributes, and nesting levels. Attackers use recursive techniques to consume memory resources. Dramatic increases in the size of the application data often signal a security problem. The XML Threat Protection policy helps protect your applications from such intrusions.

**Json threat policy**

Json threat policy help protect the application against malicious Json in API Request

**Http Caching**

This policy provides a way to store HTTP responses from an API implementation or an API proxy for later reuse. This policy avoids performing multiple calls to the backend when the response of a service does not change often and to optimize against computationally expensive processing. This policy uses the concept of a cache which stores data so future requests for that data can be served faster.