Enterprise Integration Platform

***API Security Policies***

***Guidelines and Patterns***

***Table of Contents***

[1. Introduction 3](#_Toc74326464)

[1.1 Purpose 3](#_Toc74326465)

[1.2 Intended Audiences 3](#_Toc74326466)

[2. Context 3](#_Toc74326467)

[2.1 What is API security? 3](#_Toc74326468)

[2.2 Why is API security important? 3](#_Toc74326469)

[2.3 Why Do Applications Need Security? 4](#_Toc74326470)

[2.4 REST API Security Essentials 4](#_Toc74326471)

[2.5 Security Design Principles 4](#_Toc74326472)

[2.6 What is web API security? REST API security vs. SOAP API security. 4](#_Toc74326473)

[2.7 Best Practices to Secure REST APIs 5](#_Toc74326474)

[2.8 Managing API Security 6](#_Toc74326475)

[2.9 Security in MuleSoft 6](#_Toc74326476)

[3. Policies Overview 7](#_Toc74326477)

[3.1 Policies 7](#_Toc74326478)

[3.2 Policy Management 7](#_Toc74326479)

[3.3 Architecture 8](#_Toc74326480)

[3.4 Policy Injection and Enforcement 8](#_Toc74326481)

[3.5 Policy Scope 8](#_Toc74326482)

[3.6 Policy Size 8](#_Toc74326483)

[3.7 Policies and SLAs 9](#_Toc74326484)

[3.8 MuleSoft Provided Policies 9](#_Toc74326485)

[3.8.1 Apply a Policy 9](#_Toc74326486)

[3.8.2 Policies applied 9](#_Toc74326487)

[3.8.3 More Policies 10](#_Toc74326488)

[4.Understand Downstream Security requirements 11](#_Toc74326489)

[4.1 FIS Policies 11](#_Toc74326490)

[4.1.1 Process to determine encryption fields 11](#_Toc74326491)

# 1. Introduction

## 1.1 Purpose

This document sets the context and provides guidelines as to the security considerations that need to be taken into account when client systems integrate with internal and external systems. The content of this document is meant to be used for designing the security of the APIs.

## 1.2 Intended Audiences

The primary audience for these guidelines are:

* Enterprise Architects
* Solution Architects
* Application Designers, Developers and Team leads
* Operations Team

# 2. Context

This section sets the context as to what an API security is, why API security is important and security considerations to take into account when building APIs. The context will help the reader understand the rationale for the proposed guidelines in the subsequent sections. The material presented below does not set out to change what an API is from the industry material, but to align it with Fisher's view of technology.

## 2.1 What is API security?

API security is the protection of the integrity of APIs—both the ones you own and the ones you use. What does that mean?

Well, you’ve probably heard of the Internet of Things (IoT), where computing power is embedded in everyday objects. The IoT makes it possible to connect your phone to your fridge, so that when you stop at the grocery store on the way home you know exactly what you need for that impromptu dinner party in an hour. Or maybe you’re part of a DevOps team, using microservices and containers to build and deploy legacy and cloud-native apps in a fast-paced, iterative way. APIs are one of the most common ways that microservices and containers communicate, just like systems and apps. As integration and interconnectivity become more important, so do APIs.

## 2.2 Why is API security important?

Businesses use APIs to connect services and to transfer data. Broken, exposed, or hacked APIs are behind major data breaches that expose sensitive medical, financial, and personal data for public consumption. That said, not all data is the same nor should be protected in the same way. How you approach API security will depend on what kind of data is being transferred.

If your API connects to a third party application, understand how that app is funneling information back to the internet. To use the example above, maybe you don’t care if someone finds out what’s in your fridge, but if they use that same API to track your location you might be more concerned.

## 2.3 Why Do Applications Need Security?

Businesses must ensure that the valuable information they store and make available through software applications and Web services is protected from unauthorized users and malicious attackers. However, protected resources — such as credit card information or Social Security numbers — must still be accessible to authorized legitimate users and systems in order to conduct business transactions.

To provide secure access to information, applications and services can apply a variety of security measures. The suite of security features in Anypoint Enterprise Security enables developers to protect applications according to security requirements, prevent security breaches and facilitate authorized access to data.

## 2.4 REST API Security Essentials

Security isn’t an afterthought. It has to be an integral part of any development project and also for REST APIs. There are multiple ways to secure a RESTful API e.g. basic auth, OAuth etc. but one thing is sure that RESTful APIs should be stateless – so request authentication/authorization should not depend on cookies or sessions. Instead, each API request should come with some sort authentication credentials which must be validated on the server for each and every request.

## 2.5 Security Design Principles

In the context of this document, APIs serve the purpose to make separate applications work together to produce a unified set of functionalities. The following list describes the various security design principles:

* **Least Privilege** - An entity should only have the required set of permissions to perform the actions for which they are authorized, and no more. Permissions can be added as needed and should be revoked when no longer in use.
* **Fail-Safe Defaults** - A user’s default access level to any resource in the system should be “denied” unless they’ve been granted a “permit” explicitly.
* **Economy of Mechanism** - The design should be as simple as possible. All of the component interfaces and the interactions between them should be simple enough to understand.
* **Complete Mediation** – A system should validate access rights to all its resources to ensure that they’re allowed and should not rely on the cached permission matrix. If the access level to a given resource is being revoked, but that isn’t reflected in the permission matrix, it would violate the security.
* **Open Design** - This principle highlights the importance of building a system in an open manner—with no secret, confidential algorithms.
* **Separation of Privilege** - Granting permissions to an entity should not be purely based on a single condition, a combination of conditions based on the type of resource is a better idea.
* **Least Common Mechanism** - It concerns the risk of sharing state among different components. If one can corrupt the shared state, it can then corrupt all the other components that depend on it.
* **Psychological Acceptability** - It states that security mechanisms should not make the resource more difficult to access than if the security mechanisms were not present. In short, security should not make the user experience worse.

## 2.6 What is web API security? REST API security vs. SOAP API security.

Web API security is concerned with the transfer of data through APIs that are connected to the internet. OAuth (Open Authorization) is the open standard for access delegation. It enables users to give third-party access to web resources without having to share passwords. OAuth is the technology standard that lets you share that Corgi belly flop compilation video onto your social networks with a single "share" button.

Most API implementations are either REST (Representational State Transfer) or SOAP (Simple Object Access Protocol).

REST APIs use HTTP and support Transport Layer Security (TLS) encryption. TLS is a standard that keeps an internet connection private and checks that the data sent between two systems (a server and a server, or a server and a client) is encrypted and unmodified. This means that a hacker trying to expose your credit card information from a shopping website can neither read your data nor modify it. You know if a website is protected with TLS if the URL begins with "HTTPS" (HyperText Transfer Protocol Secure).

REST APIs also use JavaScript Object Notation (JSON), which is a file format that makes it easier to transfer data over web browsers. By using HTTP and JSON, REST APIs don’t need to store or repackage data, making them much faster than SOAP APIs.

SOAP APIs use built-in protocols known as Web Services Security (WS Security). These protocols define a rules set that is guided by confidentiality and authentication. SOAP APIs support standards set by the two major international standards bodies, the Organization for the Advancement of Structured Information Standards (OASIS) and the World Wide Web Consortium (W3C). They use a combination of XML encryption, XML signatures, and SAML tokens to verify authentication and authorization. In general, SOAP APIs are praised for having more comprehensive security measures, but they also need more management. For these reasons, SOAP APIs are recommended for organizations handling sensitive data.

## 2.7 Best Practices to Secure REST APIs

The following lists some best practices for securing APIs:

* **Authentication** - Determining the identity of an end user. In a REST API, basic authentication can be implemented using the TLS protocol, but OAuth 2 and OpenID Connect are more secure alternatives.
* **Authorization** - Determining the resources an identified user can access. An API should be built and tested to prevent users from accessing API functions or operations outside their predefined role. For example, a read-only API client shouldn’t be allowed to access an endpoint providing admin functionality.
* **Always Use HTTPS** - By always using SSL, the authentication credentials can be simplified to a randomly generated access token that is delivered in the username field of HTTP Basic Auth. It’s relatively simple to use, with many security features.
* **Use Password Hash** - Passwords must always be hashed to protect the system (or minimize the damage) even if it is compromised in some hacking attempts. There are many such hashing algorithms which can prove really effective for password security e.g. PBKDF2, bcrypt and scrypt algorithms.
* **Consider OAuth** - Though basic auth is good enough for most of the APIs and if implemented correctly, it’s secure as well – yet you may want to consider OAuth as well. The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.
* **Use tokens** - Establish trusted identities and then control access to services and resources by using tokens assigned to those identities.
* **Use encryption** - Encrypt your data using a method like TLS (see above). Require signatures to ensure that the right users are decrypting and modifying your data, and no one else.
* **Identify vulnerabilities** - Keep up with your operating system, network, drivers, and API components. Know how everything works together and identify weak spots that could be used to break into your APIs. Use sniffers to detect security issues and track data leaks.
* **Use quotas and throttling** - Place quotas on how often your API can be called and track its use over history. More calls on an API may indicate that it is being abused. It could also be a programming mistake such as calling the API in an endless loop. Make rules for throttling to protect your APIs from spikes and Denial-of-Service attacks.
* **Use an API gateway** - API gateways act as the major point of enforcement for API traffic. A good gateway will allow you to authenticate traffic as well as control and analyze how your APIs are used.

## 2.8 Managing API Security

API security involves securing data end to end, from a request originating at the client, passing through networks, reaching the server/backend, the response being prepared and sent by the server/backend, the response being communicated across networks, and finally, reaching the client. Therefore, API security has been broadly categorized into below sections:

* Data in Transit/Data in Motion Security
* Securing Data in Motion between client & API gateway
* Securing Data in Motion between API gateway & Backend Services
* Authentication& Authorization: Reliably identify end user information using OAuth 2.0
* Data Confidentiality & Masking Personally Identifiable Information (PII)

## 2.9 Security in MuleSoft

Out of the box, Mule provides several tools to ensure the security of applications:

* **Mule Secure Token Service (STS) Oauth 2.0 Provider** - Mule can apply Oauth 2.0 security to a REST Web service provider or consumer. OAuth uses tokens to ensure that a resource owner never has to share credentials, such as a username or password, with a 3rd-party Web service.
  + For example, Facebook uses OAuth to connect to Gmail to collect a user’s Gmail contacts. When a user initiates the request to connect, Facebook opens a new browser window for the user to enter a Gmail address and password. If authenticated, Gmail provides all the user’s contacts to Facebook, but does not share the user’s Gmail login credentials. OAuth ensures that the 3rd-party Facebook application never has access to the user’s protected Gmail credentials.
* **Mule Credentials Vault** - Mule can encrypt properties in a properties file. The properties file in Mule stores data as key-value pairs. Mule flows may access this data — usernames, first and last names, credit card information — as the flow processes messages. In the context of Anypoint Enterprise Security, Mule refers to the encrypted properties file as the Mule Credentials Vault. Much more detailed information about using encrypted properties files is available in the Mule Credentials Vault documentation.
* **Mule Message Encryption Processor** - Mule can encrypt an entire payload or several fields of data within a message. Where sensitive information must move between users, yet remain hidden from them, a developer can encrypt message content to prevent unauthorized access. Typically, you may need to encrypt data such as a password, credit card number or social security number (SSN).
* **Mule Digital Signature Processor** - Mule uses digital signatures to ensure that messages maintain integrity and authenticity. Mule can verify that an incoming Web service request originates from a valid source, and can sign an outgoing Web service response to ensure its contents. Digital signatures ensure that a sender is valid, that a message is not modified in transit between Web services, and that no unauthorized user has tampered with a message.
* **Mule Filter Processor** - Mule can filter messages it receives to avoid processing invalid ones. With a filter processor in place, Mule discards any message it receives that does not match the filter’s parameters, for example, a message from outside a set range of IP addresses.

# 3. Policies Overview

Policies differ based on several different factors, such as category, purpose, version, and configuration options. Carefully consider these factors before you implement them in your environment. Based on the Mule runtime engine (Mule), a policy Exchange asset is consists of:

* A deployable JAR file that contains the policy business logic
* A YAML configuration file, in which the policy parameters and metadata are defined

## 3.1 Policies

Policies enable you to enforce regulations to help manage security, control traffic, and improve adaptability of your APIs. For example, a policy can control authentication, access, allotted consumption, and service level access (SLA).

You can implement all these regulations with no modification to the code implementation. Mulesoft provides ready-to-use default policies that are shipped with the product. Additionally, you can create custom policies based on your specific business requirements.

You can apply policies to any HTTP-based APIs, such as:

* An APIkit project.
  + For example, deploy the APIkit project to Anypoint Platform using API Autodiscovery, and then apply a policy.
* An API running on CloudHub.
* Design an API on Anypoint Platform, configure a proxy for Cloudhub, and apply a policy.
* An API deployed to a private or cloud-based Mule runtime engine (Mule) 3.8.x or later.

You can apply a policy to any API implemented in Anypoint Platform, as long as the API is exposed through an HTTP listener. You can also apply a policy to APIs not implemented in Anypoint Platform by deploying a proxy application to control how and when a received request is forwarded to its implementation endpoint.

* Anypoint API Manager supports RAML, HTTP, or SOAP-based proxies.

When applying a policy with SLA, you can set an API alert to notify you when an API request violates that policy. By default, a policy applies to the entire API, filtering traffic requests to every resource and method. You can configure this to provide resource-level granularity if needed.

Policies applied to APIs are the same as those in the earlier release with a few exceptions.

* Classloader isolation exists between the application, the runtime and connectors, and policies.
  + To fulfill the isolation requirement, policies are self-contained and packaged to include libraries needed for execution, similar to an application
* Policies are non-blocking. You can order all policies except CORS which takes precedence over all other policies.

## 3.2 Policy Management

Classloader isolation exists between application, runtime, connectors and policies.

Other changes to policies are:

* All policies are non-blocking, which is described in Mule 4 documentation.
* All policies except CORS, which is executed first, can be ordered.
* Resource level policy support, which was restricted to RAML-based APIs, is extended to any HTTP API.
* You can distribute policies outside of the runtime, which simplifies upgrades.

## 3.3 Architecture

Policies are implemented through coordinated communication between the following components:

* API Manager
* One or more API gateway runtimes
* One or more API proxies or API applications

Using API Manager, you can configure and apply policies to an API instance.

* An API version can have one or more API instances.
* A specific API instance can also be tied to a specific API implementation endpoint, which can then auto-generate an API proxy application.
* You can deploy an API proxy to a Mule runtime engine.

Each API proxy application receives requests on HTTP or HTTPS URLs specified by the API. Normally requests are forwarded to the corresponding API implementation, and then the response travels through the API proxy application back to the requesting client application.

If a query parameter that functions as a “faux path parameter” is not present when needed, or is not valid, the API should return a 404 status. This follows from the purpose of the parameter: it identifies a resource, and if the resource can’t be found– whether it’s identified by a path parameter or a query parameter– 404 is returned.

## 3.4 Policy Injection and Enforcement

When you deploy an API proxy application for a specific API version to Mule, any applied policies in the platform or offline policies in the runtime are injected into the API proxy. This changes the behavior of the application. When the proxy application receives a new request, all injected policies are applied to decide if, and how, the request is forwarded to the API implementation.

In this way, the actual policy enforcement occurs inside the proxy application itself. This minimizes the cross talk between the API proxy, which is processing the received request, the Anypoint Platform agent, and online API Manager. The API proxy application does not require any communication with the Anypoint Platform agent running in the runtime, nor does it require any communication with API Manager.

However, the Anypoint Platform agent remains connected to API Manager. After policies are reconfigured or removed from API Manager, those policies are downloaded to any connected API gateway or Mule runtime engines, which updates each runtime or policies folder. The policy changes are then again injected into each API proxy application. This allows policies to be dynamically changed without having to redeploy the API proxy application, and without having to restart the API gateway or Mule runtime engines.

## 3.5 Policy Scope

Policies are isolated from the application and other policies. Some policies, such as the Token Enforcement policy, allow propagation of headers, including the information related to the token. This token is used by the implementation service or by the API itself.

## 3.6 Policy Size

The size of a policy varies based on its dependencies. The first time Mule polls for policies, it takes the longest to fetch. After policies are retrieved, they are cached on the file system. This ensures offline availability and also reduces latency in the future.

## 3.7 Policies and SLAs

Rate Limiting policy can be configured to adhere to a predefined SLA. A service level access (SLA) tier is a category of user access that you define for an API. The tier definition combined with an SLA-based policy determines whether or not the access to the API is granted.

For example, tiers of access can be created depending on one of the following SLA tiers applied to the application:

* A tier that limits requests to three per minute
  + No approval required
* A tier that limits requests to five per minute
  + Requires API Version owner approval of the application that needs to access the API

Having SLAs will limit access to only one API resource, however, access to other resources is unlimited. When an application attempts to consume the protected resource, the policy is enforced. The request must include the expected username and password. Repeated calls within SLA limits from the application to the API succeed.

## 3.8 MuleSoft Provided Policies

### 3.8.1 Apply a Policy

To apply a policy:

* After logging in to MuleSoft Anypoint Platform, from the home page, navigate to menu and select API Manager module.
* This module is also available on the home page under Management Center.
* Navigate to API Manager > API Administration and select the API instance from the details window.
* From the left navigation, select Policies > Apply New Policy.
* Expand the policy that you want to apply.
* Select the policy version and click Configure Policy.

### 3.8.2 Policies applied

API across the three layers (Experience, Process and System) for all the integration workflows has been applied.

**OAuth 2.0 access token enforcement using Mule OAuth provider Policy**

* Mule OAuth 2.0 Provider is an OAuth 2.0 provider alternative developed by MuleSoft that can be used in any MuleSoft API Platform organization, including the federated ones.
* This policy is applied on the Experience API which is exposed to the consumer with http(s) end point.

**Client ID Enforcement Policy**

* 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 only accessible to applications that have an approved contract.
* This policy is applied to Process & System APIs since these layers are not exposed to any consumer applications/systems. These APIs will be consumed by Experience API.

### 3.8.3 More Policies

Below are a few more policies available out of the box, depending on requirements:

**IP Whitelist Policy**

* The IP Whitelist policy allows a list or a range of specified IP addresses to access a configured API endpoint.
* The IP Whitelist policy allows access to a protected resource when a match is found between a source IP (specified when configuring the policy) and a list of individual IPs or range of IPs.

**JWT Validation Policy**

* 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, you 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.

**Basic Authentication: Simple Policy**

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

**Rate Limiting and Throttling**

* Selecting a limit in API Manager defines the quota per time window configuration for a rate limiting and throttling algorithm. The algorithm is created on demand, when the first request is received. This event fixes the time window.
* Each request consumes quota from the current window until the time expires. When quota is exhausted, the resulting action depends on the policy.
* Rate limiting rejects the request
* Throttling queues the request for retry
* When the time window closes, quota is reset and a new window of the same fixed size starts. The policy creates one algorithm for each limit with its quota per time window configuration.
* Therefore, when multiple limits are configured, every algorithm must have available quota in its current window for the request to be accepted.

**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 you need to create at least one SLA tier to define request limits.
  + - You can configure an SLA tier to require manual approval of requests from users who want to register an app to call your API.

# 4.Understand Downstream Security requirements

## 4.1 FIS Policies

FIS recommends that you carefully consider what data should be encrypted when using an API. In shot FIS recommends any PII data in URI or query parameters need to be send after encrypting it. In addition to that all rest API should be made using https protocol. FIS uses OAuth 2 and client credentials grand type for authentication of API call which it receives.

Use the following process to encrypt data as necessary when using IBS Open APIs.

### 4.1.1 Process to determine encryption fields

Task 1: Assess Your API Encryption Needs

1. For each API that your organization chooses to use, determine if specific data (fields) within the API should be encrypted.

2. Use the following information as guidance in making your encryption decisions:

a. FIS requires encryption for any API that contains a card number in the URI such as URI path variable or query parameter.

b. In addition to the encryption that HTTPS provides, FIS recommends encrypting:

* IBS Authorization header field (RACF ID and password)
* Any API that contains an account number in the URL such as URI path variable or query parameter
* Any API that requires NPI sensitive information in the URI. (Any individual or multiple components of an NPI data item.)

Task 2: Document the List of IBS Open APIs Requiring Encryption

1. Create a document that contains a table of REST APIs that require encryption.

2. Use the following format for your table. Example data is included for your reference.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Method | API | URI Parameters | Query Parameters | Encrypt RACF User ID/ Password | Secret Cipher | Data Cipher |
| GET | IBS-Cards/cards/{CBCrdNbr} | CBCrdNbr |  | No | RSA/ECB/PKCS1Padding | AES/CBC/PKCS5Padding |
| GET | IBS-Customer-Information/ customers/{CustNbr}/phone-numbers | CustNbr | RepoInd, RepoResnCde, RepoEffDte | Yes |  |  |
| GET | IBS-Deposits/ accounts/{AcctNbr} | AcctNbr |  | No |  |  |

*Note: The Secret Cipher must be an asymmetric cipher, and the Data Cipher must be a symmetric cipher. A Secret Key will be generated for the Data Cipher, and the Secret Cipher will be used to encrypt the Secret Key. The Data Cipher will use the Secret Key to encrypt the data fields.*

Task 3: Transform Encrypted Information

Transformation of encrypted information is accomplished in two stages:

* Once per API call
* Once per field within the API call

Once Per API Call

Complete the following: (note: red text in the java samples is optional code to allow FIS to assist in debugging the encrypted values you generate)

1. Create a thumbprint of the FIS Cert **provided separately specifically for encryption**. (Sample A)

a. Create a SHA-256 Hash of the encoded form of the FIS Cert. (It is required to use SHA-256.)

b. Base64 encode the Hash to create the thumbprint.

c. Provide the Thumbprint in fis-ic-enc-x5ts256 Header.

2. Create the encoded secret key.

a. Generate a (symmetric) Secret Key. FIS recommends a 256-bit random generated key to be used for the symmetric Data Cipher. (For example, in Java this is a KeyGenerator instance of AES initialized with 256 bits.) (Sample B and Sample C)

b. Create a Cipher using the specified Secret Cipher algorithm. (It must be an algorithm that supports asymmetric keys. For example: RSA) (Sample D)

c. Initialize the Secret Cipher with the FIS public key (from the FIS encoded public Cert).

d. Encrypt the primary encoded format of the Secret Key using the Secret Cipher.

e. Base64 encode the encrypted secret key to create the Encoded Secret Key.

f. Provide the Encoded Secret Key in the fis-ic-enc-sk Header.

Once Per Field within the API Call

For each data value to encrypt, complete the following steps:

1. Using a secure Random Number Generator, generate an Initial Vector, which is the size of the number of bytes required for the Data Cipher Encryption Algorithm. (Sample E)

2. Base64URL encode the Initial Vector.

a. Save it for later as base64IV.

3. Create a Cipher using a Symmetric Data Cipher Encryption Algorithm.

4. Initialize the Data Cipher with the Secret Key and Initial Vector. (The cipher algorithm and secret key algorithms must match.)

5. Encrypt the data value (i.e. account number) using the Data Cipher.

6. Base64URL encode the encrypted data value.

a. Save it for later as base64DV.

7. Provide the Encrypted Data Value field as:

base64IV.base64DV

8. Repeat steps 1-7 as necessary for each data value that needs encrypting.

* 1. o The following two fields need to be Base64 encoded: "fis-ic-enc-sk" and "fis-ic-enc-x5ts256".
  2. o The remaining header, path and query fields (if encrypted) need to be Base64URL encoded.

Task 4: Submit the List of APIs Requiring Encryption to FIS

Send your document containing the list of APIs that require encryption to the FIS contact with whom you are working with for your IBS Open APIs implementation.

Task 5: FIS Configures Encryption for Your IBS Test Bank

If your organization has an IBS test bank, FIS then configures the encryption for your test bank.

Task 6: Test Your APIs with Encryption

After the APIs have been configured, FIS will notify you to begin testing your APIs with encryption. When you are ready take your integration live, FIS will configure the encryption for your production IBS bank.

Encryption Implementation Best Practices

Below are the Best Practices that will help in implementing encryption:

When applying encryption, pay close attention to Base64 encoded fields versus Base64URL encoded fields.

o The following two fields need to be Base64 encoded: "fis-ic-enc-sk" and "fis-ic-enc-x5ts256".

o The remaining header, path and query fields (if encrypted) need to be Base64URL encoded.

* The IBS-Authorization header - you need to take the RACFID plus a colon character plus the password and then Base64 encode this. Then encrypt the Base64 encoded value. Then Base64URL encode the encrypted value.
* The secret key that is generated must be the same secret key that is applied when encrypting the parameter values. Many developers generate the secret key, then RSA encrypt it and hold it for the “fis-ic-enc-sk” header field. They then create a separate instance/class to encrypt each parameter value and typically create a separate secret key in that instance. That does not work as the secret key needs to be the same key used for the “fis-ic-enc-sk” header field and the key used when encrypting the parameter values.

Sample Encryption Code Snippets

Java Samples

Sample A

Create a thumbprint of the FIS Cert. (Create an SHA-256 Hash of the FIS Cert)

// Get FIS Public Cert from FIS provided java keystorebyte[] keystoreBytes = Files.readAllBytes(Paths.get(keyStore));

KeyStore pubkeystore = KeyStore.getInstance(keyStoreType);

char[] passwd = String(”keyStorePswd”).toCharArray();

pubkeystore.load(new ByteArrayInputStream(keystoreBytes), passwd);

alias = String(”keyStoreAlias”);

Certificate cert = pubkeystore.getCertificate(alias);

if (cert == null) throw exception (no cert for alias)

...

// create (hash) identifier of FIS supplied public cert

// hash must be SHA-256 (can't be SHA-1 or MD5

MessageDigest md = MessageDigest.getInstance("SHA-256");

byte[] der = ((X509Certificate)cert).getEncoded();

md.update(der);

byte[] digest = md.digest();

// String hexThumbprint = ConvertBytesToHex( digest ); // create an output string of the thumbprint

String thumbprint = Base64.getEncoder().encodeToString(digest);

...

Create encoded secret symmetric key for encrypting user data (optional method)

// create a random secret key and then make a hash for the bytes

// SHA-256=32 byte key, SHA-1=20 bytekey or MD5=16 byte key

MessageDigest md = MessageDigest.getInstance("SHA-256");

// byte array should be a random value and is not limited to 16 bytes

byte[] keyBytes = new byte[]{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};

md.update(keyBytes); // hash the encoded value of the certificate

byte[] digest = md.digest(); // creates a 32-byte hash value as a key

// Java requires the "AES" algorithm to match data cipher algorithm

String algorithm = "AES";

secretKey = new SecretKeySpec(digest, algorithm);

...

Sample D

Encrypt the Secret Key with the Secret Cipher and Base64 Encode

// Encrypt symmetric Secret Key with Asymmetric Secret Cipher

String secretCipher = "RSA/ECB/PKCS1Padding";

Cipher rsaCipher = Cipher.getInstance(secretCipher);

rsaCipher.init(Cipher.ENCRYPT\_MODE, cert.getPublicKey() );

byte[] byteCipherSecretKey = rsaCipher.doFinal(secretKey.getEncoded());

String encodedCipherSecretKey = Base64.getEncoder().encodeToString(byteCipherSecretKey);

...

Sample E

Encrypt each Data Field with the Data Cipher

// Create Random Initial Vector

byte[] iv = new byte[AES\_BLOCKSIZE / 8];

SecureRandom prng = new SecureRandom();

...

// Perform this block repeatedly to encrypt each cleartext field

// create a new initial vector

prng.nextBytes(iv);

// String hexIV = ConvertBytesToHex( iv ); // create an output string of the initial vector

String encodedIv = Base64.getUrlEncoder().encodeToString(iv);

String dataCipher = "AES/CBC/PKCS5Padding";

Cipher aesCipher = Cipher.getInstance(dataCipher);

// Encrypt user data with secret key and initial vector

aesCipher.init(Cipher.ENCRYPT\_MODE, secretKey, new IvParameterSpec(iv));

// Sample cleartext field to encrypt: ”USERID:Passw0rd”;

// Note: IBS-Authorization field must be Base64 encoded before encrypting

byte[] clearTextBytes = clearText.getBytes("utf-8");

byte[] byteCipherText = aesCipher.doFinal(clearTextBytes);

String encodedCipherText = Base64.getUrlEncoder().encodeToString(byteCipherText);

String encodedDataField = encodedIv + "." + encodedCipherText;

Additional Java Subroutines to allow FIS to assist in debugging Encryption

Convert Bytes to Hex String

/\*

// Additional Subroutines to convert a byte array to a hex string

private final static char[] hexArray = "0123456789ABCDEF".toCharArray();

private static String ConvertBytesToHex(byte[] bytes) {

char[] hexChars = new char[(bytes.length \* 2)+(bytes.length-1)];

for ( int j = 0; j < bytes.length; j++ ) {

if ( j > 0 )

hexChars[(j \* 3) - 1] = '-';

int v = bytes[j] & 0xFF;

hexChars[j \* 3] = hexArray[v >>> 4];

hexChars[j \* 3 + 1] = hexArray[v & 0x0F];

}

return new String(hexChars);

}

\*/