Security Design and Implementation Guidelines

GSMA Mobile Money API

This is a Non-binding Permanent Reference of the GSMA

Table of Contents

1 Introduction

1.1 Overview

This document provides guidelines for secure design and implementations of both API Client and API Gateway specified by the Mobile Money API specification.

1.2 Scope

This Mobile Money Security Design and Implementation Guidelines document focuses on the different authentication and data protection layers (confidentiality and integrity) that must be implemented by the backend server interfaces for the establishment of secure channels between the API Client and the API Gateway.

User authentication by a third party OIDC/OAuth 2.0 Identity Provider (IDP) and general best practices to allow a secure monitoring of the implemented Mobile Money APIs and to protect user privacy are also documented.

The rules and policies to be applied by the authorisation service as well as the application layers between the users and the client API and the implementation of the business functions of the mobile money platform are not within the scope of this document.

1.3 Intended audience

This security design is targeted for Mobile Money providers, platform vendors, third party service providers and other industry partners to guide them in implementing, setting up, and/or deploying a Mobile Money Platform compliant to the GSMA harmonized Mobile Money API.

1.4 Conventions

The key words “must”, “must not”, “required”, “shall”, “shall not”, “should”, “should not”, “recommended”, “may”, and “optional” in this document are to be interpreted as described in RFC2119 [24].

1.5 Objectives and Methodology

The Mobile Money environment is fragmented with each platform vendor offering their own API. For this reason, the GSMA has defined a RESTful harmonized Mobile Money API to standardize the connection between API Clients (e.g. Merchants, Aggregators, Utility Companies, Other Mobile Money platforms) and the Mobile Money Platforms.
With this harmonized Mobile Money API, the GSMA aims to provide easy and secure building blocks and rapid partner on-boarding and interoperability between multiple Mobile Money solutions.

This introduces additional intermediary logical or physical connector components in the existing Mobile Money ecosystem on both Client and Platform sides.

To minimize the increase of business and technical risks induced by those new components (i.e. new attack vectors) on the Mobile Money system, it is important that those components are developed, deployed and operated respecting good up-to-date secure coding, configuration and operation practices.

The main purpose of this document is to provide design and implementation guidance to developers and integrators of both API Client and API Gateway components, to minimize risks.

To achieve that, this document recalls state-of-the-art best practices for backend and RESTful API development and deployment, and focuses more on guidance for those following specific security services that should be implemented:

1. Client / Server authentication: TLS *handshake* and HTTP Basic or OAuth2.0 (e.g. secure channel between API Client and API Gateway, between API Gateway and a Mobile Money Platform, and, between a Mobile Money Platform and the API Gateway for callbacks)
2. End-user authentication: Open ID Connect and other relevant methodologies
3. Data protection services: TLS *application data* encryption and authenticity, and, JSON encryption (JWE) and signature (JWS)

Section 1 presents the purposes of this document and the high-level risk-based approach. In this sense, it recalls the main security properties, services and mechanisms that must be considered. It also recalls contextual information related to the business of mobile money and financial services (e.g. high business level threat modeling) as well as to the technologies implemented in Mobile Money APIs. It also gives a high-level overview of the Mobile Money APIs deployment architecture with main components and communication interfaces and channels.

Section 2 summarizes and ranks the security design options described later in sections 3-5 of this document. They are the only ones to be considered when implementing the Mobile Money API specification version 1.1.

Section 3 defines security guidelines for implementing the client / server authentication at different layers: TLS at the transport level and OAuth 2.0 [22] at the HTTP applicative level.

Section 4 details the security methods to be implemented for securely authenticating end-users. It captures various scenarios for authenticating end-users using industry standard protocols for authentication– OAuth 2.0 [22] and OIDC [27] along with custom authentication models to support existing username/MSISDN and PIN based credentials.

Section 5 provides state-of-the-art guidance to implement data encryption and data authenticity and integrity at both *TLS / application data* protocol level and, applicative RESTful API level (a.k.a. “Message Level Security”) with JOSE, JSON encryption (JWE) and JSON signature (JWS). These recommendations also relate to the state-of-the-art in terms of key management especially for secret and private keys, choice of algorithms and mode of operations. It provides a ranking of encryption algorithms according to three classification levels: banned, acceptable and recommended.

Section 6 recalls the state-of-the-art generic security best practices when implementing RESTful APIs, providing guidance on other security controls that must be considered for such backend implementations.

1.6 High-Level Architecture

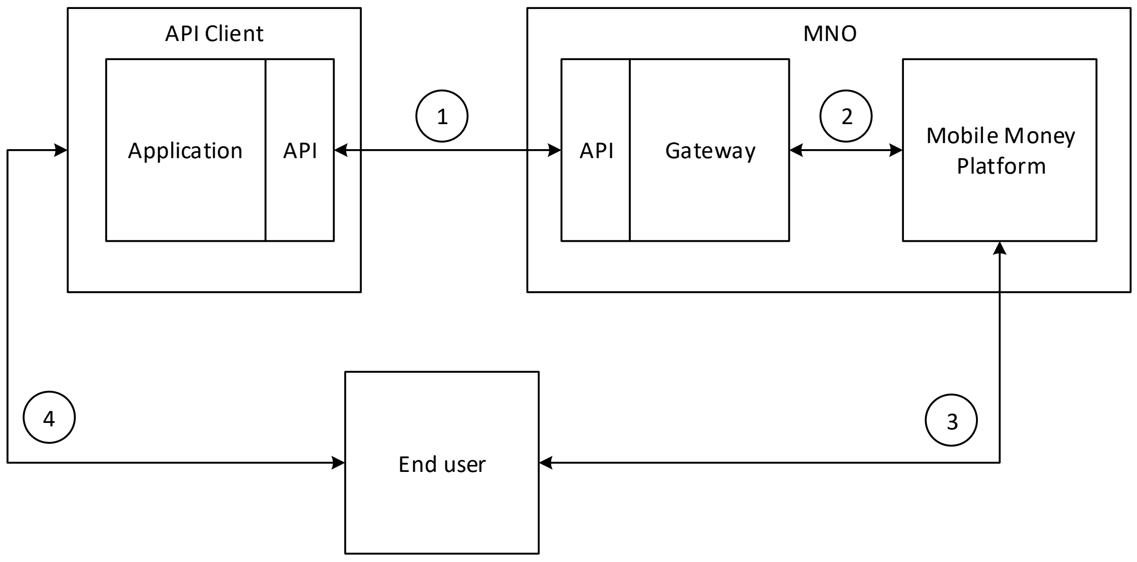

Figure 1: Solution overview including interfaces

Interface 1: The interface between the API Client and the API Gateway
This interface is the interface which is in scope in this security design and for which integrity and confidentiality are considered. The security design covers the transport layer security on this interface as well as application level authentication and authorization.

Interface 2: Connection from API Gateway to Mobile Money platform
This connection is the proprietary interface for the connection to the existing Mobile Money platform. This connection depends on the vendor chosen for the Mobile Money platform.

Interface 3: Interface between the Mobile Money platform and end-user
For authentication within some use cases the Mobile Money platform might request the end-user to provide an authentication. This interface could be IP based or be using protocols like USSD.

Interface 4: Interface between the end-user and the API Client
In some use-cases the end-user will communicate to the API Client. This interface can be digital, but it can also be different in the case of smaller shops.

1.7 Security concepts

The following table summarises the key security concepts and associated properties.

1.7.1 Security properties

Security Property	Description
Confidentiality	"*Property that information is not made available or disclosed to unauthorized individuals, entities, or processes*" [12]
Integrity	Property that information is not modified (created, deleted, or updated) by unauthorised individuals, entities, or processes
Authenticity	Property that information is originating from an authenticated source and hasn't been altered.
Availability	Property of being able to be used for a service or obtain for information by authorised individuals, entities, or processes
Non-repudiation	Property that issued actions or transactions cannot later be denied by their issuer
Auditability	Property of systems to be able to be audited or monitored by producing evidences on the processing of their critical functionalities
Traceability	Property for a data to be followed during its lifecycle to track all its access and changes

1.7.2 Security services

1.7.2.1 Data confidentiality protection

The adapted mean to protect the confidentiality of data from being disclosed to unauthorised individuals, entities or processes is Encryption.
It can be used to protect sensitive data in persistent storage (e.g. in a DB or in a hard disk file), in temporary storage (e.g. in RAM at runtime), as well as in transit (e.g. between network components or when exchanged inter processes).

The reverse operation of Encryption is Decryption.

In respect with the Kerckhoffs's principle (*"A cryptosystem should be secure even if everything about the system, except the key, is public knowledge"*) modern encryption relies generally on public algorithms.
We can distinguish two families of encryption: symmetric encryption (e.g. AES) and asymmetric (e.g. RSA) encryption. Usually, primarily for performance reasons, symmetric encryption is used to encrypt data and sometimes keys while asymmetric encryption is limited to encrypt keys only.

Encryption / Decryption rely on the use of cryptographic keys (shared secret for symmetric and public / private for asymmetric) that have to be managed (i.e. be generated, stored, used, erased and exchanged if needed) with due care, especially both secret and private keys that have to be kept secret (i.e. not accessible and not predictable) from unauthorised individuals, entities and processes.
To protect those keys and their use, according to both the isolation and least privilege principles, access control mechanisms have to be implemented and processes that use them must be isolated as much as possible.

Notes

1. Particular attention must be paid to asymmetric encryption and key agreement that are, by construction, vulnerable to Man In The Middle attacks. For this reason, both require to be processed in a fully authenticated context.
2. To be unpredictable, the keys must be generated by processes using unbiased random generators and with state-of-the-art sizes to embed enough entropy.

1.7.2.2 Data integrity protection

Different means exist to protect the integrity of data. All consist in computing a datagram from the entirety of the correct data before its storage or sending which can be recomputed and checked at the time of its re-reading or its reception. This so as to detect any unintentional or intentional modification of the data during its storage or transfer.

Unintentional modification detection can rely on simple CRC or message digest (a.k.a. hash functions) mechanisms, while intentional malicious detection cannot, unless to encrypt the integrity datagrams or protect them against tampering. This because CRC and message digest algorithms are public.

Adapted means to protect against intentional malicious modification of data are Message Authentication Code (MAC) and Digital Signature that both assume the use of cryptographic keys. That way, individuals, entities, and processes that are not authorised to use the keys cannot compute nor guess any integrity datagram from the maliciously modified data.

Message Authentication Code are generally constructed from other cryptographic primitives such as Message Digest with HMAC and, Block Ciphers with specific mode of operations, e.g. CBC-MAC or GMAC.
Both the MAC computation and MAC verification processes always rely on the use of a same shared secret key that have to be managed (i.e. be generated, stored, used, erased and exchanged or agreed) with due care to be kept secret and not predictable from unauthorised individuals, entities and processes.

Digital Signature is based on the use of asymmetric cryptography generally combined with the use of a hash function to reduce the size of data to be signed.
The signature generation uses a private key that only the signer (individual, entity or process) knows or can use, while the signature verification uses the corresponding public key that everyone knows.

Notes: Particular attention must be paid to both Digital Signature and key agreement for MAC if used, being based on asymmetric cryptography they are both vulnerable to Man In The Middle attacks. For this reason, both require to be processed in a fully authenticated context.

1.7.2.3 Data authenticity protection

Means adapted to protect the authenticity of data are the same as the one used to protect its integrity from malicious intentional modification, i.e. Message Authentication Code and Digital Signature.
In both cases, the verification process ensures that the individual, entity or process that generated the authenticity datagram was authorised to access or use the secret or private key.

As these means require to be processed in a fully authenticated context, since the keys are not compromised, they provide in addition to data authenticity protection a service of authentication of its source.

1.7.2.4 Service availability protection

The availability of a system is measured in percentage of time or occurrence when it is up and accessible compare to when it is down or inaccessible. E.g. a system with 99.999 % of availability is supposed to be down less than 5 minutes and 16 seconds per year, in that case we talk of high availability.
Protecting the availability of a service is definitively one of the most difficult tasks in security. This requires the implementation of a large number of logical, physical and organizational security controls (e.g. making regular backups and defining recovery plans and rollback procedures plans, implementing system redundancy, defining key parameter indicators to be permanently monitored by operations, implementing permanent functional testing and heart bit, defining sanity verification criteria and procedure, running business continuity exercises, audit, regular vulnerability scanning, IDS/IPS, SIEM, etc…)

Service disruptions can occurs due to power outages, hardware failures, and system upgrades but also malicious attacks. Especially on a public API, these attacks can vary from an attacker planting malware in the system to a highly organized distributed denial of service (DDoS) attack. DDoS attacks are hard to eliminate fully, but with a careful design their impact can be minimized.
In most cases, DDoS attacks must be detected at the network perimeter level—so, the application code doesn't need to worry too much. But vulnerabilities in the application code can be exploited to bring a system down.

In terms of design and logical protections developers must consider scalability or elasticity and self-testing for critical functions (e.g. authentication), avoiding potential single point of failure, contingency management (capability to deal positively with whatever occurs using whatever resources are available), minimising the attack surface (interfaces exposed to untrusted components), performing negative and security testing on exposed entry points before going to production.

1.7.2.5 Identification service

Identification is the ability of a system to get information about an identity, an identifier (e.g. a login name). In other words, an identification service is a service which allows to determine an identity by an identifier.

1.7.2.6 Authentication service

An authentication service is a process by which a system determines that users are who they claim to be.

As a corollary, it consists for users to prove their identity to a system via one or more authentication factors.
The main factors of authentication (i.e. evidence that can prove the user identity) are the following:

- Something the user is the only one to know (e.g. a secret such that a password or a PIN)
- Something the user is the only one to possess (e.g. a cryptographic key on a smart device, a passport)
- Something inherent to the user (e.g. a biometric characteristic: a fingerprint, a behavioural measurement, iris or voice recognition)
- Something the user know how to do

Over http, a successful authentication can be carried by a session cookie or an identity token (e.g. ID Token in Open ID Connect protocol).

1.7.2.7 Authorisation service

Authorisation is a process that determines what resources, data and services, can and cannot be accessed by authenticated users.

In other words, authorisation happens after authentication and is for a system the ability to grant and control user rights to access resources according to their identity.

Over http, authorisation can be carried by an access token used by the system to control the access to resources (e.g. Access Token in OAuth 2.0 framework).

1.8 Threat Model considerations

The most important assets for the Mobile Money API are the following:

- transaction data (non-repudiation, authenticity)
- Mobile Money set of services defined by APIs (availability, integrity)
- users PII (privacy)
- API Client credentials (confidentiality)
- API Gateway private key (confidentiality, authenticity)
- Call back URLs (authenticity)

Main threats and feared events at the business level are:

- Credential interception (e.g. authentication asset e.g. login/password, API key or authorisation tokens)
- End-User / transaction data disclosure (privacy)
- API abuse (e.g. if any use cases vulnerable by design, or vulnerable technological bricks)
- Abuse call back URLs
- Malicious modification of transaction data

The main technical threats to be considered are as follows:

Threat	Mitigations
Intercepting API request and obtaining OAuth2 token	• TLS to encrypt token • Put short expiry time for OAuth2 token
POST /transactions API: Intercept request and amend transaction parties to move e-money to a fraudulent actor	Sign Payload (JOSE) making it impossible to tamper with payload
JSON deserialization issues	• Use JOSE to prevent tampering of payload • On server side restrict use of non-standard deserialization packages • Regularly scan vulnerabilities • Put in place a strong patch management
Capturing of sensitive information in the URL, e.g. /msisdn/+44xxxxxxxxxx or /email/user@example.com	Use encryption in URL
POST /links API: Intercept request and amend the URI part (e.g. /links/msisdn/+44xxxxxxxxxx) containing the target link to another account of a bad actor. This would enable the bad actor to 'pull' funds using the established link	Use encryption in URL
Intercept API and modify X-Callback-URL header to direct API response to a bad actor	TLS to encrypt request headers
Invalid curve attack on JWE encrypted payload to decipher data. (https://auth0.com/blog/critical-vulnerability-in-json-web-encryption/)	• Do not use a vulnerable library • Perform vulnerability scanning • Put in place a strong patch management

1.9 Mobile Money APIs Actors

Actor	Description
API Client	The backend system of the clients of the API. These will be systems from e.g. Merchants, Aggregators, Utility Companies.
API Gateway	The API Gateway is the entry point for API Clients to connect to the Mobile Money Platform. This API Gateway is the layer of harmonization standardized by the GSMA at this moment to provide a generic interface for API Clients across different Mobile Money Platform vendors.
Relying Party	A Relying Party can either be API Client and API Gateway and integrates with 3rd party IDP or OAuth 2.0 authorisation server for authenticating and authorising end-user. Some examples of 3rd party IDP are GSMA's Mobile Connect, Facebook Connect, Google IDP etc.
End-user	User performing mobile money transactions on the consumption device and who will be authenticated on the authentication device as per the proposed security models in this document.

Actor	Description
OIDC compliant Identity Provider	The entity providing the authentication and identity services for authenticating end-users, e.g. GSMA’s Mobile Connect, Facebook Connect, Google IDP.
Consumption Device	This is the device where the user is consuming the service from the SP. This can be any Internet connected device, e.g. a mobile device, a laptop, table, smart TV etc. The access network used by this device can be any if it can initiate an HTTP(S) interaction.
Authentication Device	This is the device where the end-user is authenticating or providing authorisation. This device is always a mobile device, connected to the mobile network.
Consent Device	The consent device is the logical device through which the end-user provides consent to the IDP system, e.g. providing consent to debit wallet account in case of P2P transfer.
Authenticator	Authenticators are the authentication mechanism used by 3rd party IDP to authenticate the user. Some examples of authenticators are USSD Authenticator, SIM Applet Authenticator, Smartphone App Authenticator.

1.10 Common OAuth 2.0 terms

Actor	Description
Resource owner	An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user. API Client plays this role in the case of GSMA Mobile Money API.
Resource server	The server hosting the protected resources, capable of accepting and responding to protected resource requests using OAuth access tokens. API Gateway is a resource server responsible for OAuth token validation to process API requests. API Gateway interacts with its authorisation server for OAuth token validation.
Authorisation server	The authorisation server is implemented in compliance with the OAuth 2.0 specification, and it is responsible for validating authorisation grants and issuance of access tokens that give the client access to the protected resources on the resource server. It should be possible to configure "token endpoints" on API Gateway, in which case the API Gateway takes on the role of authorisation server. Alternatively, the API Gateway can use a third-party OAuth 2.0 compliant authorisation server.
Client Credentials grant type	The client credentials grant type can be used as an authorisation grant when the authorisation scope is limited to the protected resources under the control of the client. Client credentials are used as an authorisation grant typically when the client is acting on its own behalf (the client is also the resource owner) or is requesting access to protected resources based on an authorisation previously arranged with the authorisation server. This is the recommended grant type for authenticating API Client to API Gateway.
Authorisation code grant type	Considered the most secure grant type. Before the authorisation server issues an access token, the RP must first receive an authorisation code from the resource server. In this flow, 3rd party app opens a browser to the resource server’s login page. On successful log in, the app will receive an authorisation code that it can use to negotiate an access token with the authorisation server. This grant type is considered highly secure because the

Actor	Description
	client app never handles or sees the user's username or password for the resource server. This grant type flow is also called "three-legged" OAuth. This is one of the recommended grant type for authenticating end-users to API Gateway.
Access token	Access tokens are credentials used to access protected resources. An access token is a string representing an authorisation issued to the client. The string is usually opaque to the client. Tokens represent specific scopes and durations of access, granted by the resource owner, and enforced by the resource server and authorisation server.
Protected resource	Data owned by the resource owner. In case of GSMA Mobile Money API, the protected resources are identified by API resources URL.
Access token scope	The access token endpoint allows the client to specify the scope of the access request using the “scope” request parameter. In turn, the authorisation server uses the “scope” response parameter to inform the client of the scope of the access token issued. The value of the scope parameter is expressed as a list of space-delimited, case-sensitive strings. The strings are defined by the authorisation server. This parameter can be used by API Gateway to control the access to different resources. It should be possible to group the API set into individual product set each identified by a “scope” value. The API Gateway can decide to assign these scope values to specific API Clients based on policy and licensing rules thereby enforcing authorisation of endpoints.

1.11 Glossary & Abbreviations

Abbreviation	Description
AES	Advanced Encryption Standard
AEAD	Authenticated Encryption with Associated Data
API	Application Programming Interface
BAM	Business Activity Monitoring
CA	Certificate Authority
CEK	Content Encryption Key
Consent	Agreement that SP can use the attributes they're requesting
Consent Device	The device through which the user provides consent for the sharing or validation of attributes
Consumption device	The device through which the user is accessing and consuming mobile money service
CRL	Certificate Revocation List
DDoS	Distributed Denial of Service
ECDHE	Elliptic Curve Diffie Hellman Ephemeral
ECDSA	Elliptic Curve Data Signature Algorithm
GCM	Galois Counter Mode
GSMA	GSM Association
HTTP(S)	Hypertext Transfer Protocol (Secure)
Identity Token	Provides a set of metadata regarding the Authentication to the SP. This includes the PCR, authenticator used, Level of Assurance etc.

Abbreviation	Description
IDS	Intrusion Detection System
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPS	Intrusion Prevention System
IV	Init Vector
JOSE	JavaScript Object Signing and Encryption
JSON	JavaScript Object Notation
JWA	JSON Web Algorithm
JWE	JSON Web Encryption
JWK	JSON Web Key
JWS	JSON Web Signing
JWT	JSON Web Token
KPI	Key Performance Indicator
MAC	Message Authentication Code
MITM	Man-in-the-middle attack - is an attack where the attacker secretly relays and possibly alters the communication between two parties who believe they are directly communicating with each other
MLS	Message-Level Security - focuses on ensuring the integrity and privacy of individual messages, without regard for the network
MMP	Mobile Money Platform
MNO	Mobile Network Operator
OIDC	OpenID Connect
OWASP	Open Web Application Security Project
PCR	Pseudonymous Customer Reference
PII	Personally Identifiable Information
PKI	Public Key Infrastructure
REST	Representational State Transfer
RFC	Request for Comments
RSA	Asymmetric Encryption algorithm named after inventors: Rivest, Shamir, Adleman
RP	Relying Party (The application/service that needs the authentication and identity services). It can either be API Client or API Gateway.
Scope	Pre-defined collection of attributes that are logical to group together either for sharing or for simplifying policy management
SHA	Secure Hash Algorithm
SIEM	Security Information and Event Management
SLA	Service Level Agreement
SP	Service Provider
SSL	Transport Layer Security is based on Secure Sockets Layer (SSL) - The SSL use to be the industry accepted standard protocol for secured encrypted communications over TCP/IP
TLS	Transport Layer Security, is the usual "first line of defense", as securing the transport mechanism itself
XML	Extensible Markup Language
UMA	User Managed Access
USSD	Unstructured Supplementary Service Data
VPN	Virtual Private Network

1.12 References

Ref.	Title	Author	Date
[1]	RFC7515 - JSON Web Signature (JWS)	IETF	05-2015
[2]	RFC7516 - JSON Web Encryption (JWE)	IETF	05-2015
[3]	RFC7517 - JSON Web Key (JWK)	IETF	05-2015
[4]	RFC7518 - JSON Web Algorithms (JWA)	IETF	05-2015
[5]	RFC7519 - JSON Web Token (JWT)	IETF	05-2015
[6]	RFC7520 - Examples of Protecting Content Using JSON Object Signing and Encryption (JOSE)	IETF	05-2015
[7]	RFC4648 - The Base16, Base32, and Base64 Data Encodings	IETF	10-2006
[8]	RFC5246 - The Transport Layer Security (TLS) Protocol Version 1.2	IETF	08-2008
[9]	REST Security Cheat Sheet https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html	OWASP	04-2015
[10]	RESTful Service Best Practices, Recommendations for Creating Web Services https://raw.githubusercontent.com/tfredrich/RestApiTutorial.com/master/media/RESTfulBesPractices-v1_1.pdf	Todd Fredrich	04-2012
[11]	NIST Special Publication 800-122: Guide to Protecting the Confidentiality of Personally Identifiable Information (PII)	NIST	04-2010
[12]	ISO/IEC 27000:2014: Information technology -- Security techniques -- Information security management systems -- Overview and vocabulary	ISO	2014
[13]	ICT guidelines for TLS: https://english.ncsc.nl/publications/publications/2019/juni/01/it-security-guidelines-for-transport-layer-security-tls	Dutch Ministry of Safety and Justice	01-2020
[14]	PayPal security guidelines and best practices https://developer.paypal.com/docs/classic/lifecycle/info-security-guidelines/	PayPal	
[15]	JSON Threat Protection Policies https://docs.mulesoft.com/api-manager/2.x/policy-mule3-json-threat	MuleSoft	
[16]	JSON Threat Protection policy http://docs.apigee.com/api-services/reference/json-threat-protection-policy	APIGEE	
[17]	JSON Threat Protection	SAP HANA	
[18]	Quota policy https://github.com/apigee-127/a127-documentation/wiki/Quota-reference	Will Witman	08-2015

Ref.	Title	Author	Date
[19]	Spike Arrest Quick Start https://github.com/apigee-127/a127-documentation/wiki/Spike-Arrest-Quick-Start	Will Witman	03-2015
[20]	SANS Institute InfoSec Reading Room - Four Attacks on OAuth - How to Secure Your OAuth Implementation https://www.sans.org/reading-room/whitepapers/application/attacks-oauth-secure-oauth-implementation-33644	SANS	
[21]	RFC7617 – Basic HTTP Authentication Scheme	IETF	09-2015
[22]	RFC6749 - The OAuth 2.0 Authorisation Framework	IETF	10-2012
[23]	RFC6750 - The OAuth 2.0 Authorisation Framework: Bearer Token Usage	IETF	10-2012
[24]	RFC2119 - Key words for use in RFCs to Indicate Requirement Levels	S. Bradner	03-1997
[25]	OWASP Top Ten Project – https://owasp.org/www-project-top-ten/	OWASP	2017
[26]	OWASP Cryptographic Storage Cheat Sheet - https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html	OWASP	08-2016
[27]	OpenID Connect “An interoperable authentication protocol based on the OAuth 2.0 family of specifications” available at http://openid.net/specs/openid-connect-core-1_0.html https://openid.net/specs/openid-connect-basic-1_0.html	IETF	11-2014
[28]	RFC8443 – The Transport Layer Security (TLS) Protocol Version 1.3	IETF	08-2018
[29]	Draft-ietf-oauth-jwt-bcp-07 – JSON Web Token Best Current Practices https://tools.ietf.org/html/draft-ietf-oauth-jwt-bcp-07	IETF	10-2019
[30]	ETSI GS NFV-SEC 022 V2.7.1 (2020-01) Network Functions Virtualisation (NFV) Release 2; Security; Access Token Specification for API Access	ETSI	01-2020
[31]	Mozilla guidelines for TLS and OIDC (Browser session oriented) https://wiki.mozilla.org/Security/Server_Side_TLS https://infosec.mozilla.org/guidelines/iam/openid_connect.html	Mozilla	
[32]	RFC6125 - Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)	IETF	03-2011

2 Security Design Options Summary

This section summarizes and ranks the security design options described in this document. They are the only ones that should be considered when implementing the Mobile Money API specification version 1.1.

The use of TLS is mandatory as it provides the primary layer of protections for both parties' authentication and data in transit.
TLS can be implemented following four options ranked by increasing strength:

1. Server authentication without PFS (i.e. RSA encryption for key exchange)
2. Server authentication with PFS (i.e. DHE like key agreement)
3. Mutual (Client / Server) authentication without PFS
4. Mutual (Client / Server) authentication with PFS

For API Client authentication by the API Gateway at HTTP level, the use of OAuth2.0 Client Credentials grant type, is strongly recommended.
OAuth2.0 Client Credentials grant type can be implemented following four options ranked by increasing strength:

1. Alone
2. By combining and binding it with authentication by API Key
3. By combining and binding it with TLS Client authentication X509 certificate
4. By combining and binding it with both authentication by API Key and TLS Client authentication X509 certificate

End-user authentication can be performed following three options ranked by increasing strength

1. By username/password sent to the Mobile Money Platform (End-user credentials known by API Client)
2. OIDC/OAuth2.0 authentication by an IDP either hosted by the API Gateway or a 3rd Party (End-user credentials never known by API Client)

In addition to data in transit message authentication offers by TLS, authenticity and integrity of sensitive data can be enhanced at the HTTP level by one of those two options ranked by increasing strength:

1. Basic data integrity and authenticity check
2. JSON Signature (JWS)

In addition to data in transit encryption offers by TLS, the confidentiality of sensitive data should be enhanced at the HTTP level by implementing JSON Encryption (JWE).

3 API Client – API Gateway Authentication – Security Design

3.1 Solution overview

The API Gateway is the end point of the TLS secure channel to be established over interface 1 (see Figure 1) by API Client instances acting as TLS clients. The API Gateway is responsible for hosting valid server certificates for TLS server authentication by clients. It is also responsible for decrypting request data (messages) and to verify data authenticity and integrity.
At the API level, the API Gateway is in charged to authenticate API Client instances and when implemented, to provide them with OAuth2.0 authorisation tokens (both Access Token and Refresh Token).
The Mobile Money Platform is responsible for end-user authentication and authorisation - whether this entity is allowed to access, create or modify the information, e.g. Transaction, Quotation.
In some cases The Mobile Money platform authenticate users by itself using own proprietary authentication means, in some cases it can delegate the authentication of users to an OIDC/OAuth2.0 Identity Provider (see Figure 5) hosted or public (e.g. GSMA’s Mobile Connect, Facebook Connect, Google IDP, etc…)

Security services	**Responsible component**
TLS server authentication of the API Gateway	API Client
Message encryption	API Client
Message integrity and authenticity tag computation	API Client
API Client authentication	API Gateway
API Client authorisation	API Gateway
Message decryption	API Gateway
Message integrity and authenticity verification	API Gateway
User authentication	MM Platform or OIDC IDP
User authorisation	MM Platform

Table 1: Responsibilities of components

It is not possible to establish an end to end secure channel from the API Client to the Mobile Money Platform as the gateway needs to be able to read the data to perform the mapping of the data elements to the format used by the Mobile Money Platform which depends on choices made by the vendor that delivered the platform. During this mapping it would be possible to change the data. This is why the API Gateway and mapping functionality should be performed in a trusted system. However, if the Mobile Money Platform allows it, it is recommended to establish a TLS link on interface 2 between the API Gateway and the Mobile Money Platform (see Figure 1). If so, the proprietarily remapped messages would be protected.

3.2 API Gateway Authentication using TLS server authentication

Server-side TLS authentication occurs during the TLS Handshake protocol. It allows a client to authenticate the server it is connecting to.
According to the result of this authentication the client refuses or accepts to go further in the establishment of a secure channel connection with the server and the setting up of a shared pre-master-secret used to generate a master-secret and secret keys for message encryption and MAC computation (2 for the client, 2 for the server).
Server-side authentication takes generally place when the server provides its public certificate for authentication to the client.
The TLS layer of the client is then in charge of authenticating the server by verifying the following points:

1- One of the certificate’s common names in the subject or in the subjectAlternativeNames match the domain name of the URL
2- The complete certificate chain is available and rely upon a trusted CA
3- The current date is in the certificate validity period
4- The certificate has not been revoked, checking a CRL (local or uploaded) or asking the OCSP responder specified in the certificate
5- The certificate signature is valid (using the public key of the issuer certificate)
6- Repeat verification from 3 to 5 for each certificate in the server certificate chain

To enforce API Gateway server certificate verification the API Client TLS and applicative layers can use a certificate pinning method to validate twice the server certificate or the trusted CA certificate on top of the server certificate chain.

3.3 API Client Authentication using TLS mutual authentication

The TLS handshake protocol always allows clients to authenticate servers, e.g. it is the usual case for websites exposed over the Internet, where clients need to trust they are connecting to genuine servers but servers are not able to know every potential client so do not care about authenticating them.
In some cases, especially when servers want to limit their exposure at HTTP level to trusted clients only, these servers can also require clients to authenticate during the TLS handshake. This is called TLS mutual authentication.
In a TLS mutual authentication, during the handshake protocol the server presents to the client not only its server certificate but also a certificate request message with a list of issuing CA it trusts for this domain. The client authenticates the server the same way as seen previously, and in addition, sends back its client certificate and a digital signature (Certificate Verify) generated with the associated private key. The server verifies the signature using the public key in the client certificate, if the verification succeed, it proves the client can use the private key.

To enrol API Clients within the API Gateway, the certificate belonging to the API Client should be enrolled within the API Gateway. This can either be performed by operating a PKI and issuing the certificates from the API Gateway or by creating a trust store of certificates that are issued by a public CA.

In both cases it is important that the certificate is linked to the API Client and known within the API Gateway.

The certificates of API clients enrolled in the API Gateway should be maintained and if the API Client is no longer trusted than the certificate should be blacklisted inside the API Gateway (in the case issued by a public CA) or placed on a Certificate Revocation List (CRL) in case the certificate was issued by a CA for the Mobile Money Platform.

When setting up a connection the certificate from the API Client should always be validated, including validation of the chain, and may not be present on the CRL (cf. previous subsection). The API Client should validate the certificate of the API Gateway, including validation of the chain, and may not be present on the CRL (cf. previous subsection).

3.4 API Client Authentication at HTTP level

At HTTP level, the GSMA strongly recommends the implementation of OAuth 2.0 Client Credentials Grant type over TLS for API Client authentication to the API Gateway.

Optionally, API Client authentication at HTTP level can be improved by adding to the OAuth 2.0 Authentication and authorisation layer an identification of the API Client by using an API KEY.

Note: This API Client authentication could be strengthened at the TLS level by implementing client authentication, described just before. In that case, a binding between both TLS and HTTP API Client identification information (e.g. X509 API Client certificate and API Client username via the provided access token).

3.4.1 API Client Authentication and authorisation using OAuth 2.0 Client Credentials grant type

The GSMA Mobile Money API utilises OAuth 2.0 authorisation framework (RFC 6749 [22]) for API Client authentication and authorisation. OAuth 2.0 is a standard way of allowing a third-party application (API Client) to obtain limited access to an HTTP service i.e., protected resources. The generic OAuth 2.0 flow is as follows:

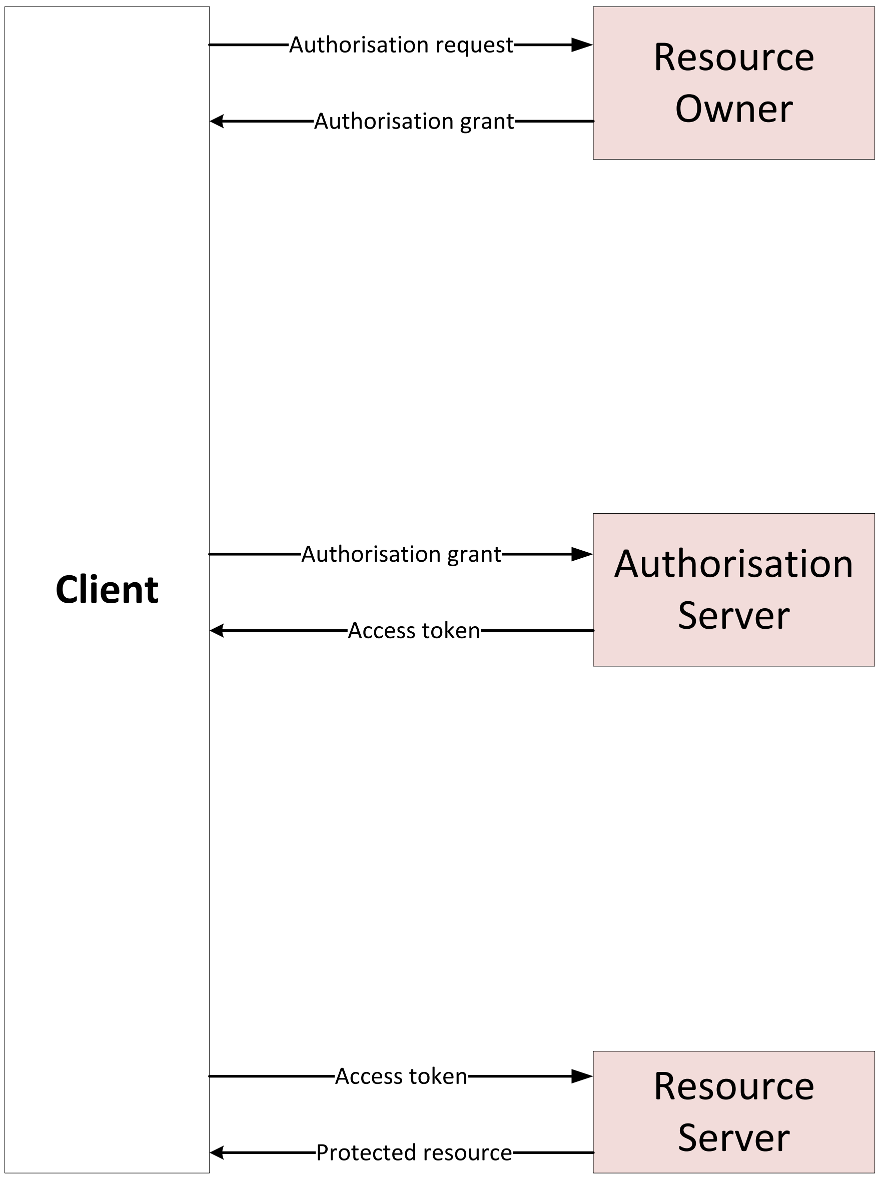

Figure 2: OAuth 2.0 standard flow

3.4.1.1 Issuance of Access Token using OAuth 2.0 Client Credentials grant type

The API Gateway is responsible for exposing an additional token endpoint over TLS connection as defined in OAuth 2.0 specifications (RFC 6749 [22]). The API Client requests an access token using only its client credentials as per client credentials grant type OAuth flow. These credentials are pre-shared as per Mobile Money platform provider's policy with API Clients. The client credentials grant type must only be used by protected and confidential clients. The client credentials flow is illustrated below:

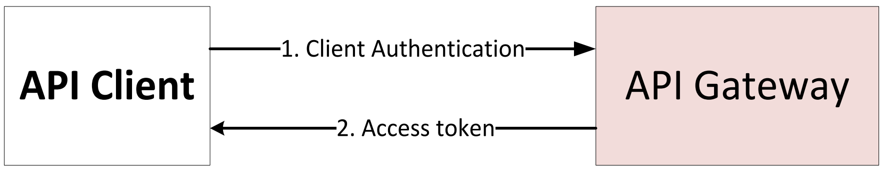

Figure 3: OAuth 2.0 client credentials flow

1. The API Client requests the access token from the token endpoint passing base64 encoded client credentials in basic authorisation header.
2. The API Gateway is responsible for performing API Client authentication. If valid, API Gateway interacts with its authorisation server to issue an access token response containing the access token, expiry time and optional scope values.

For example, the API Client makes the following HTTP request to API Gateway over TLS:

```
POST /token HTTP/1.1
Host: server.example.com
Authorisation: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded
X-API-Key: czZCaGRSa3F0MzpnWDFmQmF0M2JW88jw66

grant_type=client_credentials
```

The Authorisation header is constructed by concatenating the client's credentials with ':' and applying base64 encoding.

On successful authentication of API Client, the API Gateway responds with an access token response. For example:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
        "access_token":"2YotnFZFEjr1zCsicMWpAA",
        "token_type":"example",
        "expires_in":3600
}
```

3.4.1.2 Usage of Access Token

The API Client is responsible for passing the received access token to every API call as a bearer token in the "Authorisation" request header field as per RFC 6750 [23]. The API Gateway is responsible for validating the token with its authorisation server, and if valid, check that the client is allowed to invoke the protected resource based on the access token scope value. It should return appropriate HTTP response codes in case of invalid access token (expired or revoked) or invalid scope.

For example:

```
GET /resource HTTP/1.1
Host: server.example.com
Authorisation: Bearer mF_9.B5f-4.1JqM
X-API-Key: czZCaGRSa3F0MzpnWDFmQmF0M2JW88jw66
```

3.4.2 API Client Authentication using HTTP Basic authentication

The HTTP Basic Authentication method defined in RFC7617 [21] is not a security method as it results in the cleartext transmission of client's username and password over the network.

For this reason, it is not recommended as a per request authentication mean by the GSMA.

3.4.3 API Client Authentication based on API key

With this option, the API Client has a pre-shared key(s) with unique identifiers. The key(s) are shared as per Mobile Money platform provider’s policy with API Clients. One of the ways of sharing the API key is through the API Gateway developer portal. Initial identity of the API Client is confirmed by providing this identifier in an “X-API-Key” request header. If the combination of API key and client credentials/OAuth access token is not correct, the request must be rejected with an error code in the response and logged in an audit log.
It should be possible to revoke this key to stop a rogue API Client from accessing the API Gateway.

Note: API Keys are assets that have to be managed (created, read, updated, stored, deleted and used) with a level of protection equivalent to the one implemented for cryptographic keys by both API Client and API Gateway.
I.e.:

- API Client should store it encrypted with an authenticity datagram at the persistent storage level. It should decrypt it in memory at runtime (non-swappable if possible). Ideally, it should verify its authenticity each time before using it and should cleanse the memory before releasing any copy of it.
- API Gateway should generate API keys randomly with enough entropy to render them not guessable. It should store them encrypted with an authenticity datagram at the persistent storage level in DB (to mitigate the risk of data breach that would disclose API Keys) and should strongly bind them to their related API Client. At runtime it should decrypt it in memory (non-swappable if possible) and verify its authenticity each time before using it. And, it should cleanse the memory before releasing any copy of it.

4 End-User Authentication – Security Design

4.1 Solution overview

This section provides implementation guidelines on end-user authentication using different security models. Some of the scenarios where the security models can be applied are:

1. Authentication and identification of end-user (debit party/credit party) to API gateway/Mobile Money platform. For example: As part of initial login process, API Gateway can authenticate the user using Authorisation Code flow[1] and API Client in turn retrieving the access token from API Gateway. The API Client can subsequently pass the access token in API calls to API Gateway for validation purpose.

2. Authorisation consent from a debit party/account holder for a financial transaction. For example: In the case of send money, cash out, buy goods; either API Gateway/API Client can authenticate the debit party using 3rd party OIDC compliant IDP and retrieve the consent proof (access token) and passing the access token to API Gateway in the API call.

3. End-user consent to share their MSISDN and in-turn identifying their wallet account. For example: In case of an ecommerce checkout, the merchant server can authenticate the user using a 3rd party IDP and acquire consent to share their MSISDN. The consent proof (access token) can then be passed to API Gateway who can validate the token and retrieve the MSISDN to identify the wallet account.

4. Third party developer has developed an app to enable customers to send money. It should not be possible for customers to enter their credentials (MSISDN + PIN) into the app and pass it in Mobile Money API. Instead, the API Gateway should prompt the user to authenticate using Mobile Money platform credential mechanism (MSISDN + PIN) and if successfully authenticated, API Gateway issues an access token to the app. The app can supply the access token in Mobile Money API to API Gateway.

The security design proposes the following security models for authenticating end-users:

1. End-user authentication by API Gateway using OAuth 2.0/OIDC Authorisation Code Flow

2. Delegated end-user (debit party) authorisation using 3rd party OIDC compliant IDP

3. End-user authentication using username and PIN

4.1.1 Overview of OpenID Connect protocol

It is recommended that any 3rd party IDP used for authorising users should be OIDC compliant. Some of the popular OIDC compliant IDPs are GSMA's Mobile Connect, Facebook Connect and Google IDP etc.
OpenID Connect (OIDC) [27] is an identity layer on top of OAuth 2.0 [22] that provides an authentication context for the end-user in the form of Who, When, How etc. in a JWT based claims set [ID Token].

[1] Also known as 3 legged OAuth flow

The key functionality provided are:

- Pseudonymous Identity (claims assertion) /Authentication of end-user [ID Token]
- JSON/REST-like API for authentication and basic profile sharing [UserInfo]

OpenID Connect provides an additional token [ID Token] along with the OAuth 2.0 access token. The ID Token is represented as a JWT and contains a claim set related to the authentication context of the subject. The JWT can be a plaintext JWT or cryptographically protected JWT – represented as signed JWT using JWS [JSON Web Signature] or as encrypted using JWE [JSON Web Encryption].
The security design recommends use of the OIDC Authorisation Code flow for the following reasons:

- Tokens not revealed to the User Agent
- RP must be authenticated
    - client_secret is used in Authorisation Code flow to retrieve access and ID tokens
- Usage of refresh token possible

4.2 End-user authentication by API Gateway

The GSMA Mobile Money API utilises industry standard OAuth 2.0/OIDC authorisation framework for end-user authentication. Please note that the authorisation server can be embedded inside API Gateway depending on the implementation of API Gateway or it can be a separate authorisation server hosted by a 3rd party. The authorisation server must be either OIDC or OAuth 2.0 compliant.
It utilises Authorisation Code flow/three legged OAuth flow for authenticating end-users. Please see section 1.10 for definition of Authorisation Code flow. This flow is considered to be highly secure as Mobile Money credentials of end-users are never requested directly by API Client.

Some of the advantages of using OAuth 2.0/OIDC authorisation framework are:

1. Use of industry standard protocols for authenticating users thereby avoiding build of bespoke solutions.
2. Use of OIDC compliant IDP providers means support for wide array of advanced authentication mechanisms including PIN and Biometrics[2].
3. Single integration model for API Client to authenticate end-users.
4. More secured as end-user credentials are never captured in API Client assets[3] directly.
5. The use of access token allows time bound/one time access, if required.

A high-level component view of various actors and flow of information is illustrated below:

[2] Finger scan or Facial recognition or Iris scan
[3] Website or apps

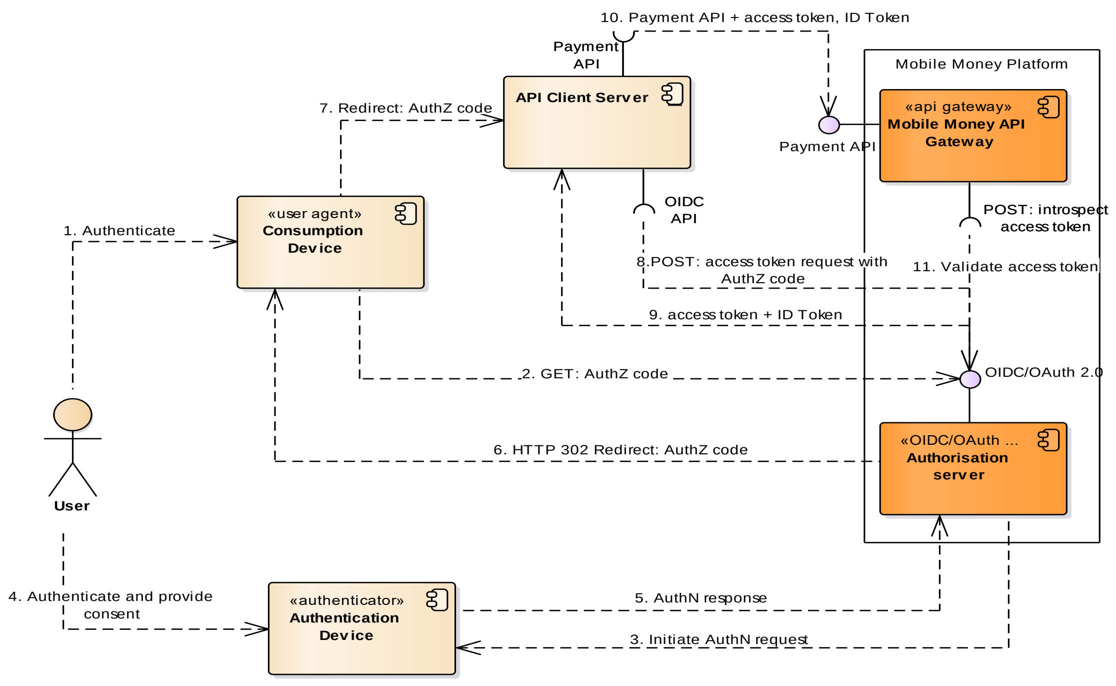
10. Payment API + access token, ID Token
Payment API
Mobile Money Platform
API Client Server
7. Redirect: AuthZ code
«api gateway» Mobile Money API Gateway
Payment API
«user agent» Consumption Device
1. Authenticate
OIDC API
POST: introspect access token
8.POST: access token request with AuthZ code
11. Validate access token
9. access token + ID Token
2. GET: AuthZ code
OIDC/OAuth 2.0
6. HTTP 302 Redirect: AuthZ code
«OIDC/OAuth ... Authorisation server
User
4. Authenticate and provide consent
«authenticator» Authentication Device
5. AuthN response
3. Initiate AuthN request

Figure 5: End-user authentication by API Gateway

The process for authenticating end-user and retrieving access token can be broken down into following steps:

1. End-user initiates authentication request on the consumption device's user agent.
2. The user agent sends an authorisation request to authorisation server for authenticating end-user passing client_id, redirect_uri, state and other parameters in the request. See section 4.2.1 for details.
3. Authorisation server initiates end-user authentication process as per Mobile Money platform's authentication mechanism. The authorisation server can authenticate the end-user by presenting an authentication page either in consumption device or separate authentication device. The actual implementation is left to the Mobile Money platform provider.
4. User is prompted to provide credentials either in consumption device or separate authentication device.
5. The authentication device generates an authentication response and returns to the authorisation server.
6. Authorisation server validates authentication response and returns a temporary authorisation code to RP server indirectly as a redirect through user agent. See section 4.2.1.1 for details.
7. RP server receives the authorisation code in the redirect URL. It extracts the authorisation code from the redirect URL's query parameter.
8. RP server exchanges the authorisation code to retrieve access token and optional ID Token with authorisation server. RP server will provide its client credentials in token API request to retrieve the tokens. See section 4.2.1.2 for details.
   Please note that if the authorisation server is not OIDC compliant, then it will only return access token. The advantage of using ID Token is to allow the RP to retrieve additional identity claims like MSISDN etc.
9. Authorisation server validates the authorisation code and client credentials, generates new access token and ID Token and returns to RP server.
10. RP server passes end-user's access token to API Gateway in API requests as a custom header value. See section 4.2.2 for details.

11. API Gateway validates the access token with authorisation server before processing the API request.

4.2.1 Authorisation request for authenticating end-user

API Client's user agent will send authorisation request to the authorisation server's 'authorisation endpoint', using HTTP GET or POST.
Communication to the authorisation server endpoint MUST use TLS. The request parameters are added using query string serialization. The prompt parameter in the request must be "login".

Sample Request:

```
POST /authorise HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

response_type=code&
client_id=s6BhdRkqt3
&redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
&scope=openid
&state=af0ifjsldkj
&nonce=n-0S6_WzA2Mj
&prompt=login
&login_hint=<MSISDN>
```

The authorisation server validates authorisation request and returns a HTML payload for authenticating the user. The actual authentication mechanism (MSISDN + PIN or Biometrics[4] or something else) is dependent on the downstream Mobile Money platform. It should also be possible for the authorisation server to perform out of band authentication using separate authenticators[5]. The authenticators can also act as consent device for displaying authentication prompt to the user. The actual implementation of authentication mechanism adopted by authorisation server is out of scope from this document.

4.2.1.1 Authorisation response

Authorisation server will generate authorisation code after authenticating the end-user.
It will return the authorisation code using redirect to the RP server[6] at the redirect_uri.

Sample Response:

```
HTTP/1.1 302 Found
Location:https://server.sp.com/authorised?Code=AsdsdsMKDsd&stat
e=af0ifjsldkj
```

4.2.1.2 Issuance of tokens using Authorisation Code flow

RP server[7] makes a token request by presenting its authorisation code to the token endpoint exposed by authorisation server. The grant type value must be "authorisation_code", as described in section 4.1.3 of OAuth 2.0 [22].

[4] Finger scan or Facial recognition or Iris scan
[5] USSD authenticator, SIM Applet authenticator, Smartphone App Authenticator
[6] API Client
[7] API Client

The RP server sends the parameters to the token endpoint using the HTTP POST method and the form serialization, as described in section 4.1.3 of OAuth 2.0 [22]. Communication to the authorisation server endpoint MUST use TLS.

The Authorisation Code flow is illustrated below:

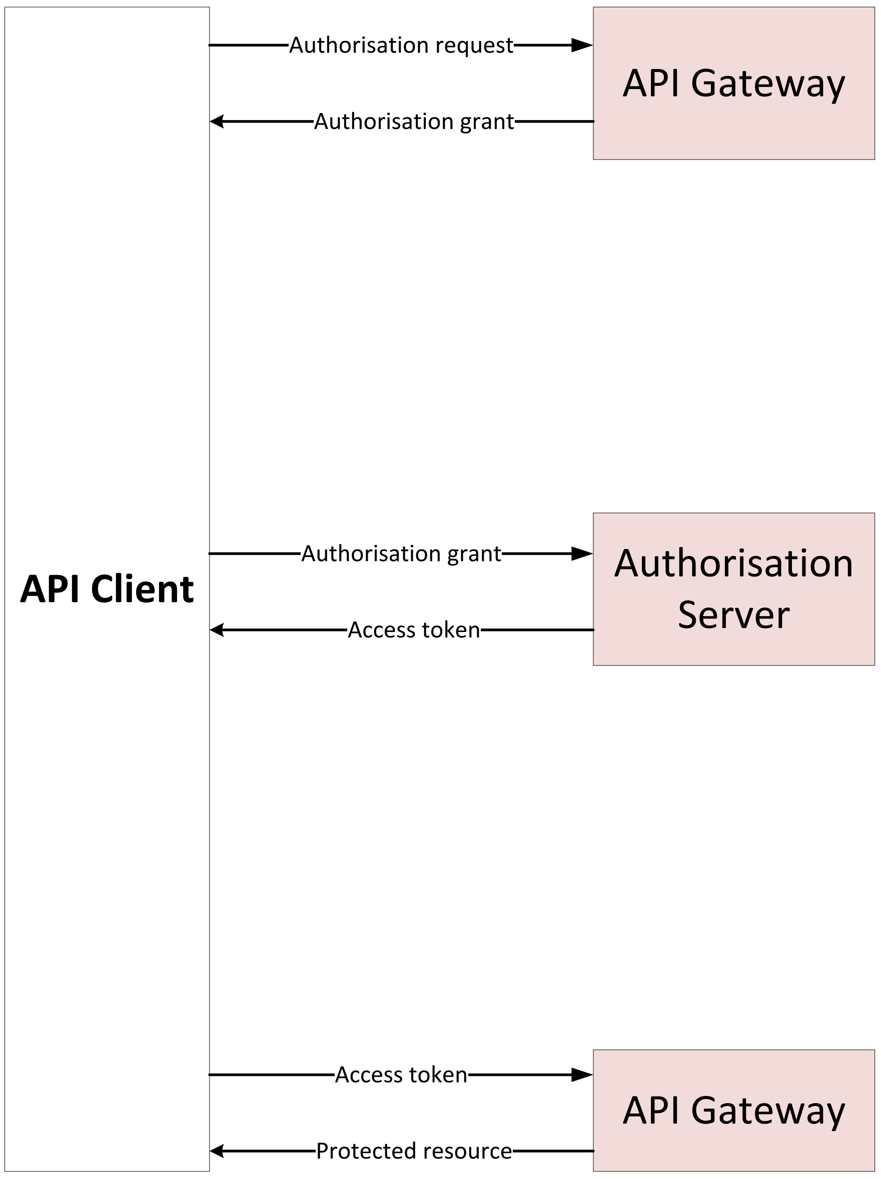

Figure 4: OAuth 2.0 Authorisation Code flow

1. The API Client requests the access token from the token endpoint of authorisation server passing base64 encoded client credentials in basic authorisation header. The request parameters includes grant type value, authorisation code received in section 4.2.1.1 and redirect URI value. These parameters are passed “x-www-form-urlencoded” values.’

2. The authorisation server validates client credentials of RP server and if valid, returns an access token response containing the access token, refresh token, expiry time and optional ID Token.

Sample token request:

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Authorisation: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

grant_type=authorisation_code&code=SplxlOBeZQQYbYS6WxSbIA
    &redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
```

Sample successful access token response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache

{
      "access_token": "SlAV32hkKG",
      "token_type": "Bearer",
      "refresh_token": "8xLOxBtZp8",
      "expires_in": 3600,
      "id_token":
"eyJhbGciOiJSUzI1NiIsImtpZCI6IjFlOWdkazcifQ.ewogImlzc

yI6ICJodHRwOi8vc2VydmVyLmV4YW1wbGUuY29tIiwKICJzdWIiOiAiMjQ4Mjg5

NzYxMDAxIiwKICJhdWQiOiAiczZCaGRSa3F0MyIsCiAibm9uY2UiOiAibi0wUzZ

fV3pBMk1qIiwKICJleHAiOiAxMzExMjgxOTcwLAogImlhdCI6IDEzMTEyODA5Nz

AKfQ.ggW8hZ1EuVLuxNuuIJKX_V8a_OMXzR0EHR9R6jgdqrOOF4daGU96Sr_P6q
       Jp6IcmD3HP99Obi1PRs-cwh3LO-
p146waJ8IhehcwL7F09JdijmBqkvPeB2T9CJ
       NqeGpe-
gccMg4vfKjkM8FcGvnzZUN4_KSP0aAp1tOJ1zZwgjxqGByKHiOtX7Tpd

QyHE5lcMiKPXfEIQILVq0pc_E2DzL7emopWoaoZTF_m0_N0YzFC6g6EJbOEoRoS

K5hoDalrcvRYLSrQAZZKflyuVCyixEoV9GfNQC3_osjzw2PAithfubEEBLuVVk4
       XUVrWOLrLl0nx7RkKU8NXNHq-rvKMzqg"
 }
```

4.2.2 Usage of Access Token

The API Client will be responsible for passing user's access token received in section 4.2.1.2 to every API call as a custom header value. The API Gateway will be responsible for validating the token with its authorisation server, and if valid, allow the processing of the API request. It should return appropriate HTTP response codes in case of invalid access token (expired or revoked).

For example:

```
        GET /resource HTTP/1.1
        Host: server.example.com
        Authorisation: Bearer mF_9.B5f-4.1JqM
        X-User-Bearer: czZCaGRSa3F0MzpnWDFmQmF0M2JW88jw66
```

4.3 Delegated end-user authorisation

There are scenarios that require debit party/account holder authentication to authorise a payment transaction. For example: In the case of Send Money, Cash Out, Buy Goods etc., it should be possible for the API Gateway to directly authenticate the debit party as described in section 4.2 or allow the API Client to use a 3rd party OIDC compliant IDP to authenticate the debit party and pass the access token as consent proof[8] to API Gateway in the API request. This allows the API Gateway to validate the access token using token introspection endpoint described in section 4.3.1. On successful validation, it should continue processing the payment request. The focus of this section is delegated authorisation of debit party/account holder using a 3rd party IDP.

Some of the advantages of delegated authorisation model are:

1. Use of industry standard protocols for authorising users thereby avoiding build of bespoke solutions.
2. Single integration model to support multiple 3rd party IDP providers.
3. User's credentials are never passed in API request thereby reducing risk and fraud.
4. The use of access token allows time bound/one time access, if required.
5. Allows API Gateway to verify the consent proof before proceeding with payment transaction. The consent proof can also provide audit trail as it contains exact timestamp of providing consent and mechanism used for authenticating the user.
6. The consent proof provides non-repudiation of payment transaction.
7. Use of OIDC compliant IDP providers means support for wide array of advanced authentication mechanisms including PIN and Biometrics[9].
8. Streamlined UX flow as the user is not required to authenticate separately with Mobile Money platform, resulting in fewer steps to complete a payment transaction.

A high-level component view of various actors and flow of information is illustrated below:

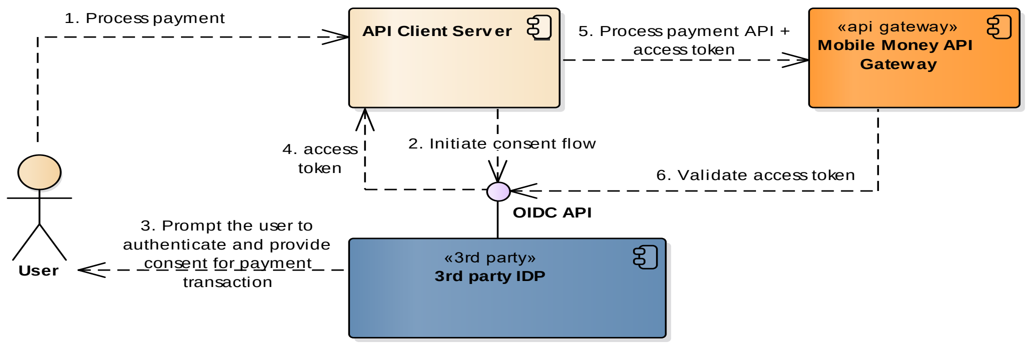
1. Process payment
API Client Server
5. Process payment API + access token
«api gateway» Mobile Money API Gateway
4. access token
2. Initiate consent flow
6. Validate access token
OIDC API
User
3. Prompt the user to authenticate and provide consent for payment transaction
«3rd party» 3rd party IDP

Figure 5: Delegated user authorisation

1. User initiates payment request (For ex: Send Money) with API Client.

[8] access token
[9] Finger scan or Facial recognition or Iris scan

2. API Client uses 3rd party IDP to authenticate the user and authorise the payment transaction.
3. 3rd party IDP prompts the user to authenticate and provide consent for the payment transaction.
4. On successful user authentication and consent, 3rd party IDP issues an access token and ID Token to API Client.
5. API Client invokes Mobile Money payment API passing the access token, ID Token and introspection endpoint URL of 3rd party IDP.
6. API Gateway validates the access token by invoking introspection endpoint URL. The introspection endpoint is a protected endpoint requiring API Gateway to pass its client credentials or bearer token when invoking this endpoint. 3rd party IDP returns meta-information of the access token if the token is still valid[10]. API Gateway can optionally introspect the ID Token to retrieve identity claims like MSISDN etc. It can also check the level of assurance[11] achieved by 3rd party IDP when authenticating the user before continuing with payment processing flow with downstream Mobile Money platform.

Please note that user must be registered with 3rd party IDP in order to authenticate and provide consent. Also, API Gateway should have client relationship with 3rd party IDP in order to invoke the introspection endpoint.

4.3.1 OAuth 2.0 Token Introspection

The token introspection endpoint allows a resource server[12] to query OAuth 2.0 authorisation server to determine the active state of an access token and to retrieve meta-information about this token. This method can be used by RP[13] to convey information about the authorisation context of the token from the authorisation server to the protected resource. In the context of Mobile Money APIs as illustrated in section 4.3, API Client can pass user's access token and introspection endpoint URL of 3rd party IDP server to API Gateway allowing the gateway to validate the access token by invoking the introspection endpoint URL and passing the access token as "application/x-www-form-urlencoded" data. The successful response contains meta-information about the token.

The endpoint also requires some form of client authorisation to access this endpoint. The calling client[14] can authenticate using the mechanisms described in section 2.3 of OAuth 2.0 [22] or by passing a separate OAuth2.0 access token as bearer token.

The following is a non-normative example request:

```
POST /introspect HTTP/1.1
Host: server.example.com
Accept: application/json
Content-Type: application/x-www-form-urlencoded
Authorisation: Bearer 23410913-abewfq.123483

token=mF_9.B5f-4.1JqM&token_type_hint=access_token
```

[10] Not expired or revoked
[11] acr_value attribute in ID Token
[12] API Gateway
[13] API Client
[14] API Gateway

4.3.2 Introspection request attributes

The protected resource[15] calls the introspection endpoint using an HTTP POST request with parameters sent as "application/x-www-form-urlencoded" data

Parameter	Required category in spec	Description
token	Mandatory	The string value of the token. • For access tokens, this is the "access_token" value returned from the token endpoint • For refresh tokens, this is the "refresh_token" value returned from the token endpoint
token_type_hint	Optional	A hint about the type of the token submitted for introspection. The possible values are: • "access_token" if the token is of type access token • "refresh_token" if the token is of type refresh token

Table 2: Introspection request attributes

4.3.2.1 Introspection response attributes

The authorisation server responds with a JSON object in "application/json" format with the following top-level members:

Attribute	Required category in spec	Description
scope	Optional	A JSON string containing a space-separated list of scopes associated with this token, in the format described in section 3.3 of 012 [22]
client_id	Optional	Client identifier for the RP that requested this token
username	Optional	User's username
token_type	Optional	Type of token as defined in section 5.1 of OAuth 2.0 [22]
exp	Optional	The expiration time after which the access token MUST NOT be accepted for processing. The format is the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the date/time specified.
iat	Optional	The time of issue of access token. The format is the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the date/time specified.
nbf	Optional	Timestamp indicating when the access token is not to be used before. The format is the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the date/time specified.
sub	Optional	Subject identifier of the user (PCR)
aud	Optional	The intended audience for the access token. It is an array of case-sensitive strings. It

[15] API Gateway

		MUST contain the client_id of the RP/Client, and MAY contains identifiers of other optional audiences. If there is one audience, the aud value MAY be a single case sensitive string OR an array of case sensitive strings with only one element. An implementation MUST support both scenarios.
iss	Optional	Issuer Identifier. It is a case-sensitive HTTPS based URL, with the host. It MAY contain the port and path element (Optional) but no query parameters.
jti	Optional	Access token string identifier

Table 3: Introspection response attributes

4.4 End-user authentication using username and PIN

There is a legacy option in some of the existing Mobile Money platforms that allows API Client to capture user's credentials[16] directly in their own assets[17] and pass it to Mobile Money platform securely in the API payload. This is not a recommended option but is still provided in this document to support existing implementation requiring minimum changes.

Some of the drawbacks of this option are:

1. User's credentials are known to API Client resulting in increased fraud due to possibility of altering the credentials.
2. API Gateway and Mobile Money platform is unable to receive explicit consent from the user thereby potentially increased customer service complaints and financial liability.
3. API Client has to support multiple authentication models for different Mobile Money platform providers.

Some of the recommendations to support this option are:

1. This option should only be used in scenarios where the user is directly controlled by Mobile Money Platform. For example: if the user is using Mobile Money platform's website or app directly and needs to authenticate.
2. User's MSISDN is passed in API and should be encrypted as defined in section 5.4.
3. The PIN is encrypted at source using pre-shared API key and symmetric encryption algorithm. The API key and algorithm details are shared during the provisioning of API Client.
4. Use of custom header in the API request to pass encrypted PIN.

[16] MSISDN and PIN
[17] Website or app

5 Data Protection – Security Design

5.1 TLS - Application Data Protocol

5.1.1 Data Encryption and Authenticity

During the handshake protocol the client and the server set up securely a shared secret called pre_master_secret, computed randomly by the client and sent to the server, or computed by both client and server using a DH like key agreement algorithm.
On both sides, client and server compute first a master_secret using a pseudo random function (PRF) taking as input a constant string and the three following parameters: pre_master_secret, the client_random (random generated by the client and sent in the Client Hello message to the server) and server_random (random generated by the server and sent in the Server Hello message to the client).
Then, using the PRF function and another constant string, from master_secret, client_random and server_random both client and server derive four session keys, and optionally if, as per below recommendation, an Authenticated Encryption with Associated Data algorithm is chosen for data encryption (e.g. AES GCM) two session init vectors (IV) :

- client_write_MAC_key
- server_write_MAC_key
- client_write_key
- server_write_key
- client_write_IV
- server_write_IV

If the encryption is not an AEAD (e.g. AES CBC), two keys are to be used by the client, the two others by the server.
The write_MAC_key keys are for data authenticity and integrity, the write_key keys are for data encryption.

If the encryption is an AEAD (e.g. AES GCM), the MAC keys are useless, and one key and one IV are to be used by the client, the other key and IV by the server.
The write_key keys are for both data authenticity and integrity and data encryption, the write_IV values are to be used as IV by the AEAD.

The encryption algorithm and the MAC algorithms are both determined by the cipher suite (see below) negotiated during the handshake protocol.

5.1.2 TLS Version and Algorithm considerations

5.1.2.1 TLS Version

The latest version of TLS is v1.3.
TLS v1.2 and v1.3 are the ones recommended.

TLS v1.1 is still acceptable when some clients do not support TLS v1.2, but must be deactivated as soon as every client supports it.
TLS v1.0 and all SSL versions 2.0 and 3.0 are banned and must be deactivated.

E.g. to configure the versions of SSL/TLS to be activated by apache, check, add or update SSLProtocol line in the ssl.conf configuration file:

```
SSLProtocol all -SSLv2 -SSLv3 -TLSv1 -TLSv1.1
```

And to test if a specific version of SSL/TLS is active, we can use the s_client tool of openssl to force the use of its specific version (-ssl2, -ssl3, -tls1, -tls1_1, -tls1_2). Here we check TLSv1.1:

```
 openssl s_client -connect server.example.com:443 -tls1_1
```

If the connection succeed then TLSv1.1 is still activated.

5.1.2.2 TLS Cipher Suites

Until version 1.2 of TLS the cipher suites are always constructed in one of two following ways:

	Construction Pattern	**Key**		MAC computation from CHF
1.	TLS_w_x_WITH_y_z	w = key agreement algorithm for the handshake protocol x = authentication algorithm for the handshake protocol y = encryption algorithm with operation mode for the application data protocol z = message digest or hash algorithm to be use with HMAC for message records	**Example** *ECDHE Elliptic-Curve Diffie-Hellman Ephemeral* *ECDSA Elliptic-Curve Digital Signature Algorithm)* *AES_128_GCM Advanced Encryption Standard 128 bits in Galois Counter Mode*	**Example** SHA256 means HMAC_SHA 256 is used
2.	TLS_v_WITH_y_z	v = key exchange and authentication algorithm for the handshake protocol when it is the same. y = encryption algorithm with operation mode for	*Usually RSA (Rivest Shamir Adleman)* *AES_128_GCM Advanced Encryption Standard 128 bits in Galois Counter Mode*	SHA256 means HMAC_SHA 256 is used

		the application data protocol z = the message digest or hash algorithm to be use with HMAC for message records		

Note:

- If RSA is used for both key exchange and server authentication, the client is in charge to compute the pre_master_secret and send it encrypted with the server public key to the server
- If a key agreement algorithm is used (e.g. DHE or ECDHE) both the client and the server participate in the computation of the pre_master_secret.

Forward Secrecy (FS), a.k.a. Perfect Forward Secrecy (PFS) is a property specific of key agreement (e.g. ECDHE) that ensures session keys will not be compromised even if the server private key is compromised.

Note: RSA for key exchange does not have this property. If server private key is compromised, anyone who recorded the traffic can decrypt the exchanged pre_master_secret and have access to both client and server random. For every recorded session he can then easily compute the session keys.

Note: recommended key size for RSA is 2048-bits or longer.

Cipher suites that implement Authenticated Encryption (e.g. AES GCM) for application data encryption and authenticity are highly recommended for at least two reasons:

1. The authenticity tag is computed from the encrypted data, if wrong, data is not decrypted (good for security and for performances)
2. CBC mode of operation was often used before but is vulnerable by design to padding oracle attacks (e.g. BEAST)

For that reason, the state-of-the-art recommended subset of the cipher suites [13] to be supported by servers is:

- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256
- TLS_RSA_WITH_AES_256_GCM_SHA384
- TLS_RSA_WITH_AES_128_GCM_SHA256

The other cipher suites defined in the TLS 1.2 specification are acceptable, but shouldn't be supported if all clients support at least one cipher suite from the recommended subset above.

A specific cipher suite shall not be hard coded in the configuration. Instead, the protocol must be allowed to negotiate the highest version automatically [14].

5.1.3 Session keys randomness

Session keys for application data authenticity and encryption are computed from multiple random generated by both sides, client and server.
If those random are of bad quality (biased or with poor entropy) the session keys could be easily guessed. To avoid that, the layer that implements TLS must be configured in such a way as to use good random source (e.g. /dev/random or /dev/urandom on linux systems).

5.2 JavaScript Object Signing and Encryption (JOSE)

JOSE is a set of IETF standards to enable cryptographic protection of JSON objects, but also others type of objects, in fact, JOSE provides a general approach to signing and encryption of any content. However, it is deliberately built on JSON and base64 encoding, RFC 4648 [7] , to be easily usable in web applications.

The standards related to JOSE are listed in the following Table.

Standard	Description	How the standard is used on interface 1	RFC Reference
JSON Web Signature (JWS)	JSON objects with digital signatures or Message Authentication Codes (MAC)	JWS is used to sign the payload of the message being transmitted	RFC7515 [1]
JSON Web Encryption (JWE)	Encrypted JSON objects	JWE is used to encrypt the payload of the message	RFC7516 [2]
JSON Web Keys (JWK)	Public and private keys (or sets of keys) represented as JSON objects	Is used to exchange the public key used to sign the message payload with JWS	RFC7517 [3]
JSON Web Algorithms (JWA)	Authorising a party to interact with a system in a prescribed manner	JWA is used to specify which algorithm is used for JWS and JWE	RFC7518 [4]
JSON Web Token (JWT)	Is a compact, URL-safe means of representing claims to be transferred between two parties Describes representation of claims encoded in JSON and protected by JWS or JWE	JWT is currently not used on interface 1 but can be used to transport OAuth tokens in future implementations. It is possible to encrypt and sign a JWT with JWE and JWS.	RFC7519 [5]

Table 4: Standards related to JOSE

JOSE has similar function to the XML Signature and XML Encryption standards, to provide message-level protection of message confidentiality, authenticity and integrity. Examples of protecting content using JSON Object Signing and Encryption can be found in RFC7520 [6].

5.2.1 JSON Payload Encryption (JWE)

Asymmetric encryption is used to exchange a symmetric CEK (Content Encryption Key). This CEK is encrypted with the public key of the receiving party to ensure that only the receiving party will be able to decrypt the CEK. From this moment on both parties are in the possession of the CEK. This Content Encryption Key can be used for the remainder of the session using a symmetric algorithm.

5.2.2 JSON Payload Signature (JWS)

The payload of the messages sent to the GSMA Mobile Money API need to be signed by the private key belonging to the certificate of the API Client which must be enrolled within the API Gateway of the Mobile Money platform. This ensures integrity of the messages exchanged from the API Client to the API Gateway.

5.2.3 Algorithm considerations

For JOSE the algorithms are defined in JWA as specified in RFC 7518 [4].

5.2.3.1 JWE Algorithms

For JWE one of the following algorithms must be applied to exchange the CEK:

- Key Agreement with Elliptic Curve Diffie-Hellman Ephemeral Static, as described in section 4.6 from [4]:
    - ECDH-ES+A256KW: ECDH-ES using Concat KDF and CEK wrapped with "A256KW" (recommended)
    - ECDH-ES+A192KW: ECDH-ES using Concat KDF and CEK wrapped with "A192KW" (recommended)
    - ECDH-ES+A128KW: ECDH-ES using Concat KDF and CEK wrapped with "A128KW" (recommended)
- Key Encryption with AES GCM, as described in section 4.7 from [4]:
    - A256GCMKW: Key wrapping with AES GCM using 256-bit key (recommended)
    - A192GCMKW: Key wrapping with AES GCM using 192-bit key (recommended)
    - A128GCMKW: Key wrapping with AES GCM using 128-bit key (recommended)
- Key Encryption with RSAES OAEP, as described in section 4.3 from [4]:
    - RSA-OAEP-256: RSAES OAEP using SHA-256 and MGF1 with SHA-256 (recommended)
    - RSA-OAEP: RSAES OAEP using default parameters (acceptable)
- Key Encryption with RSAES-PKCS1-v1_5, as described in section 4.2 from [4]:
    - RSA1_5: RSAES-PKCS1-v1_5 (banned)

Notes:

- When using ECDH-ES key agreement, the length of the output key is the one in the algorithm name (in bits). E.g. ECDH-ES+A128KW output 128-bits of agreed shared secret
- Recommended size for RSA is 2048-bits or longer keys

For the CEK one of the following algorithms must be applied:

- AES_CBC_HMAC_SHA2 Algorithms, as described in section 5.2 from [4]:
  - A256CBC-HS512: AES_256_CBC_HMAC_SHA_512 (acceptable)
  - A192CBC-HS384: AES_192_CBC_HMAC_SHA_384 (acceptable)
  - A128CBC-HS256: AES_128_CBC_HMAC_SHA_256 (acceptable)
- Content Encryption with AES GCM, as described in section 5.3 from [4]:
  - A256GCM: AES GCM using 256-bit key (recommended)
  - A192GCM: AES GCM using 192-bit key (recommended)
  - A128GCM: AES GCM using 128-bit key (recommended)

5.2.3.2 JWS Algorithms

For JWS one of the following algorithms must be applied:

- Digital Signature with ECDSA, as described in section 3.4 from [4]:
  - ES512: ECDSA using P-512 and SHA-512 (recommended)
  - ES384: ECDSA using P-384 and SHA-384 (recommended)
  - ES256: ECDSA using P-256 and SHA-256 (recommended)
- Digital Signature with RSASSA-PSS , as described in section 3.5 from [4]:
  - PS512: RSASSA-PSS using SHA-512 and MGF1 with SHA-512 (recommended)
  - PS384: RSASSA-PSS using SHA-384 and MGF1 with SHA-384 (recommended)
  - PS256: RSASSA-PSS using SHA-256 and MGF1 with SHA-256 (recommended)
- Digital Signature with RSASSA-PKCS1-v1_5, as described in section 3.3 from [4]:
  - RS512: RSASSA-PKCS1-v1_5 using SHA-512 (banned)
  - RS384: RSASSA-PKCS1-v1_5 using SHA-384 (banned)
  - RS256: RSASSA-PKCS1-v1_5 using SHA-256 (banned)

Note: recommended size for RSA is 2048-bits or longer keys.

5.2.4 CEK Key Randomness

CEK keys are assumed to be random and not predictable. If the random are of bad quality (biased or with poor entropy) the session keys could be easily guessed. To avoid that, the layer that implements random generation must be configured in such a way as to use good random source (e.g. /dev/random or /dev/urandom under linux).

5.3 Basic data integrity and authenticity check

It is not mandatory for a Mobile Money platform provider to implement JOSE technology stack for achieving data integrity and authenticity. An alternate approach to achieve basic data integrity, detection of timing issues and authenticity checks is by using the following request headers. The API Client must calculate these values and set it in the corresponding headers. These headers are optional and should only be used if JOSE is not used in a specific implementation. The different request headers are listed in Table 5.

HTTP Header Name	Description
X-Content-Hash	SHA-256 hex digest of the request content (encrypted or plain)
Content-Length	Length of request content - Requests having too long or non-matching length are rejected
Date	The date and time that the message was sent in HTTP-date format including the time zone. One policy can be to reject the requests having time deviation of more than ‘x’ minutes. It is the responsibility of API Gateway to normalize the time to server’s time zone for calculation purpose.

Table 5: Request headers for basic data integrity and authenticity check

5.4 Protection of sensitive request parameters – Query / URI Path variables

It is important to protect sensitive request parameters passed to a GET resource. These parameters can be passed either as query parameters or URI path variables.

For example:

1. MSISDN/{value}
2. /accounts/{accountIdentifier1}@{value1}${accountIdentifier2}@{value2}${accountIdentifier3}@{value3}

The following strategy can be used by API Client to protect these parameters:

1. API Client encrypts URI path variables using the pre-shared API key with pre-shared symmetric encryption algorithm. The API key and algorithm details are pre-shared during the provisioning of API Client.
2. Use a request object in a POST that can be either signed (JWS) or encrypted (JWE) using standard JOSE framework described in section 5.2.

6 API Best Practices

This chapter describes a collection of the common security practices that must be applied to RESTful API and API Gateway platform. For these common best practices, the following references have been used as a reference:

- OWASP REST Security Cheat Sheet [9].
- OWASP Top Ten [25]
- RESTful Service Best Practices [10]
- OWASP Cryptographic Storage Cheat Sheet [26]

Each section contains a table with a set of best practices encoded in the following way: {BP_Category_number]

6.1 Auditing/Monitoring

6.1.1 Logging

An important aspect of building RESTful services in a complex distributed application is to address logging functions, especially for the purpose of debugging production issues and investigating eventual points of failure. With good logging practices you can detect security issues. Keep in mind that PII (Personally Identifiable Information) [11] data should be handled with care avoiding the logging of these types of information.

Table 6 - Best Practices Logging [BP_LOG]

Code	Description
BP_LOG_1	A detailed consistent pattern should be applied to log messages across service logs. It is a good practice for a logging pattern to at least include the following: date and current time, logging level, the name of the thread, the simple logger name and the detailed message.
BP_LOG_2	It is important to anonymize sensitive data. It is important to mask or anonymized sensitive data in production logs to protect the risk of compromising confidential and critical PII information. Anonymization should rely on a cryptographic message digest mechanism.
BP_LOG_3	Identifying the caller or the initiator as part of logs.
BP_LOG_4	Do not log payloads by default.

6.1.2 Monitoring/Reporting

Monitoring activities is useful to protect your application from some misuses or external attacks, but also to keep track, with the help of a BAM (Business Activity Monitoring), of KPIs (Key Performance Indicator) to verify the adherence to the SLA agreed with the stakeholders. API Gateway can be used to monitor, throttle, and control access to the API. The following can be done by a gateway or by the RESTful service:

- Monitor usage of the API and know what activity is good and what falls out of normal usage patterns and implement appropriate reporting functionality

- Throttle API usage so that a malicious user cannot take down an API endpoint (DOS attack) and can block a malicious IP address

Table 7 - Best Practices Monitoring [BP_MON]

Code	Description
BP_MON_1	Use a monitoring system which can collect data to evaluate and to control anomalous behavior, SLA and other statistics in the background. It is a good practice to collect logs in a SIEM (Security Information and Event Management), to discover some anomalous behavior and to detect some attack patterns. Collecting information in the right manner you could do the following activities: • Identity, Audit and Authenticate Administrator and 3rd Party Access • Control and Audit of all privileged user access • Logging, monitoring user access • Track and Monitor all Access • Access Policy and reporting for Forensics and Investigations on incidents • Continuous Security Training Awareness with Recording Message • Remote Access Session Monitoring and Authentication to Servers • Logging Access, Alert on Unauthorised Access to Sensitive systems • Ports and Services Monitoring, Logging All Server and user activity • Incident Response with Session Replay on Event logs
BP_MON_2	Should implement payload protection policy. Malicious injection in the payload are mitigated using techniques described in section 5.2.2. But the attackers can, for example, use recursive techniques to consume memory resources and other techniques that can compromise the availability of the services. In the optic of a layered approach, the JSON threat protection policies help to protect applications from such intrusions and possible damages. Some example of policy to define are: • Specifies the maximum number of elements allowed in an array. • Specifies the maximum allowed containment depth, where the containers are objects or arrays. • Specifies the maximum number of entries allowed in an object • Specifies the maximum string length allowed for a property name within an object. • Specifies the maximum length allowed for a string value. Some useful reference are MuleSoft documentation [15], Apigee documentation [16] and SAP HANA documentation [17].
BP_MON_3	Should implement protection policy to monitor the traffic against malicious behavior (e.g. DOS Attack and spike arrest). • Quotas [18] and rate limits control the number of connections apps can make to the backend via the API

Code	Description
	• Spike arrest [19] capabilities protect against traffic spikes and denial-of-service attacks It is suggested to implement this feature using an API Gateway such as Apigee, SAP API Management.
BP_MON_4	It is suggested to provide real-time monitoring capability or real-time API health visibility. Define and implement a list of KPI to monitor the API health status and performance issue. Examples of KPI: 1. If you have more than 5 consecutive errors in the invocation of a service raise an alert 2. If the call for a function takes more than a fixed period ex. 4 second.

6.2 Communication

6.2.1 Transport

Switching between HTTP and HTTPS introduces security weaknesses and the best practice is to use TLS (HTTPS) by default for all the communication.

Table 8 - Best Practices Transport Communication [BP_TCOM]

Code	Description
BP_TCOM_1	Data in transit. The use of TLS should be mandated and enforced, particularly where credentials, updates, deletions, and any value transactions are performed. TLS version 1.2 [8] or newer must be utilized.

6.2.2 Data encryption

Encryption should always be used to transfer data or sensitive information between API Client and API Gateway.

Table 9 - Best Practices Transport Encryption [BP_TCRY]

Code	Description
BP_ TCRY_1	PII and sensitive information in general should be encrypted (i.e. JSON encryption [2]).
BP_ TCRY_2	Data at rest. It is necessary to prevent database bypass, which occurs when an attacker threatens to gain access to sensitive data by targeting operating system files and backup media. In this case he may avoid most database authentication and auditing mechanisms. The most common way of preventing this is encrypting the data-at-rest, i.e. whenever it is committed to memory. This has the added benefit of also protecting against improper decommission or theft of drives.

Code	Description
BP_ TCRY_3	Message Authentication Code (MAC) should be used. When using Hash-based message authentication code (HMAC) use SHA-2 and up, avoid SHA and MD5.
BP_TCRY_4	Any PII and sensitive request parameters passed as query parameters or path variables must be encrypted. Please see section 5.4 for more details.

6.2.3 Storage of cryptographic keys and credentials

Some of the best practices for cryptographic design and storage of cryptographic keys are summarized below:

Table 10 - Best Practices for storage of crypto keys [BP_CRPS]

Code	Description
BP_CPRS_1	All protocols and algorithms for authentication and secure communication should be well vetted by the cryptographic community.
BP_CPRS_2	Ensure certificates are properly validated against the hostnames/users i.e. whom they are meant for
BP_CPRS_3	Avoid using wildcard certificates unless there is a business need for it
BP_CPRS_4	Maintain a cryptographic standard to ensure that the developer community knows about the approved cipher suites for network security protocols, algorithms, permitted use, crypto periods and Key Management
BP_CPRS_5	Store a one-way and salted value of passwords - Use PBKDF2, bcrypt or scrypt for password storage
BP_CPRS_6	Ensure that the cryptographic protection remains secure even if access controls fail - This rule supports the principle of defense in depth. Access controls (usernames, passwords, privileges, etc.) are one layer of protection. Applicative encryption and MAC add an additional layer of protection that will continue protecting the data even if an attacker subverts the database access control layer
BP_CPRS_7	Ensure that any secret key is protected from unauthorised access
BP_CPRS_8	Store keys away from the encrypted data
BP_CPRS_9	Protect keys in a key vault
BP_CPRS_10	Document concrete procedures for managing keys through the lifecycle
BP_CPRS_11	Under PCI DSS requirement 3, you must protect cardholder data
BP_CPRS_12	Render PAN (Primary Account Number), at minimum, unreadable anywhere it is stored
BP_CPRS_13	Protect any keys used to secure cardholder data against disclosure and misuse. As the requirement name above indicates, we are required to securely store the encryption keys themselves. This will mean implementing strong access control, auditing and logging for your keys. The keys must be stored in a location which is both secure and "away" from the encrypted data. This means key data shouldn't be stored on web servers, database servers etc. Access to the keys must be restricted to the smallest number of users possible. This group of users will ideally be users who are highly trusted and

Code	Description
	trained to perform Key Custodian duties. There will obviously be a requirement for system/service accounts to access the key data to perform encryption/decryption of data. The keys themselves shouldn't be stored in the clear but encrypted with a KEK (Key Encrypting Key). The KEK must not be stored in the same location as the encryption keys it is encrypting.

6.3 Identity Management

The API Client and/or end-user should be authenticated and authorised prior to completing an access control decision. All access control decisions should be logged.

6.3.1 Authentication and session management

Authentication validates if you are the right person who can login to the software system.
A RESTful API should be stateless. This means that request authentication should not depend on cookies or sessions.

Table 11 - Best Practices Identity Management Authentication [BP_IMAU]

Code	Description
BP_IMAU_1	Session-based authentication must be used, either by establishing a session token via a POST or by using an API key as a POST body argument or as a cookie. Username, passwords, session tokens, and API keys must not appear in the URL.
BP_ IMAU_2	Protect session state. Most Web services and APIs are designed to be stateless, with a state blob being sent within a transaction. For a more secure design, consider using the API key to maintain client state if the API is using a server-side cache. It's a commonly used method in Web applications and provides additional security by preventing anti-replay. Replay is where attackers cut and paste a blob to become an authorised user. In order to be effective, include a time-limited encryption key that is measured against the API key, date and time, and incoming IP address
BP_ IMAU_3	All access control decisions shall be logged.
BP_ IMAU_4	Protect against replay attacks. A replay attack (also known as playback attack) is a form of network attack in which a valid data transmission is maliciously or fraudulently repeated or delayed. This is carried out either by the originator or by an adversary who intercepts the data and retransmits it, possibly as part of a masquerade attack by IP packet substitution (such as stream cipher attack). There are a lot of countermeasures that you can take in place that use a time limited encryption key, keyed against the session token or API key, date and time, and incoming IP address. For example, some mechanisms are session tokens, one-time passwords, MAC (Message Authentication Code) and time stamping.

Code	Description
	One common best practice mechanism, also used by OAuth is to have the following combination: • Nonce (number used once): It identifies each unique signed request and prevent requests from being used more than once. This nonce value is included in the signature, so it cannot be changed by an attacker • Timestamp: We can add a timestamp value to each request. When a request comes with an old timestamp, the request will be rejected. From a security standpoint, the combination of the timestamp value and nonce string provide a perpetual unique value that cannot be used by an attacker. A useful reference is a SANS whitepaper: Four Attacks on OAuth - How to Secure OAuth Implementation [20].

6.3.2 Authorisation

Authorisation validates if you are the right person to have access to the resources.

Table 12 - Best Practices Identity Management Authorisation [BP_IMAZ]

Code	Description
BP_ IMAZ_1	Protect HTTP methods. RESTful API often use GET (read), POST (create), PUT (replace/update) and DELETE (to delete a record) methods. Not all of these are valid choices for every single resource collection, user, or action. Make sure the incoming HTTP method is valid for the session token/API key and associated resource collection, action, and record.
BP_ IMAZ_2	Whitelist allowable methods. For an entity the permitted operations should be defined. For example, a GET request might read the entity while PUT would update an existing entity, POST would create a new entity, and DELETE would delete an existing entity. It is important for the service to properly restrict the allowable verbs such that only the allowed verbs would work, while all others would return a proper response code (for example, a 403 Forbidden).
BP_ IMAZ_3	Protect privileged actions and sensitive resource collections. Not every user has a right to every web service. The session token or API key should be sent along as a cookie or body parameter to ensure that privileged collections or actions are properly protected from unauthorised use.
BP_ IMAZ_4	Protect against cross-site request forgery. For resources exposed by RESTful web services, it's important to make sure any PUT, POST, and DELETE request is protected from Cross Site Request Forgery. Typically, one would use a token-based approach.

6.4 Validating RESTful services

When exposing RESTful service APIs, it is important to validate that the API behaves correctly.

6.4.1 Input validation

Table 13 - Best Practices Input Validation [BP_VALI]

Code	Description
BP_VALI_1	Use a secure parser for parsing the incoming messages.
BP_VALI_2	It is suggested the using of strongly type techniques for incoming data. Limit and define the permitted values for an input parameter.
BP_VALI_3	Validate incoming content-types. The service should never assume the *Content-Type*. When is not present in the header the server should reject the content with a generic *404 Not Found*
BP_VALI_4	Validate response types. It is common for REST services to allow multiple response type (in this case: *application/json*) and the client specifies the preferred order of response types by the Accept header in the request. Do not accept the request if the content type is not one of the allowable types. Reject the request (ideally with a generic HTTP *404 Not Found response*)
BP_VALI_5	Use some framework (e.g. Jersey) that enable validation constrains to be enforced automatically at request or response time. This kind of framework provide automatic validation after unmarshaling.
BP_VALI_6	To prevent abuse, it is standard practice to add some sort of rate limiting to an API. RFC 6585 introduced a HTTP status code *429 Too Many Requests* to accommodate this. However, it can be very useful to notify the consumer of their limits before they hit it.

6.4.2 Output encoding

Table 14 - Best Practices Output Validation [BP_VALO]

Code	Description
BP_VALO_1	Send security headers. To make sure the content of a given resources is interpreted correctly by the browser, the server should always send the *Content-Type* header. The server should also send an *X-Content-Type-Options: nosniff* to make sure the browser does not try to detect different *Content-Type* than what is actually sent (can lead to XSS).
BP_VALO_2	JSON encoding. A key concern with JSON is preventing arbitrary JavaScript remote code execution within the browser. When inserting values into the browser DOM, strongly consider using *.value/.innerText/.textContent* rather than *.innerHTML* updates, as this protects against simple DOM XSS attacks.
BP_VALO_3	XML as JSON should never be built by string concatenation. It should always be constructed using an appropriate serializer. This should be useful to be sure that the content is parsable and does not contain injected elements.

6.4.3 Error handling

You must take in consideration that unhandled exceptions could reveal, to an attacker, useful information about your API.

Table 15 - Best Practices Error Handling [BP_ERR]

Code	Description
BP_ERR_1	Utilize error codes. It is highly recommended that error codes are returned whenever an error is encountered. A cautionary note here is to not provide too much information (such that it would provide an adversary an advantage). Successful error codes/messages are a balance between enough information and security.

Annex A REST Security Standard Overview

This section provides a brief overview of open security standards defined by the Internet Engineering Task Force (IETF), the OpenID Foundation (OIDF), and other standards organizations for securing RESTful web interfaces.

Table 16 – Open Security Standards for RESTful Interfaces

Standard	Description
Transport Layer Security (TLS)	IETF standard for secure communications between a client and server, providing transport-layer encryption, integrity protection, and authentication of the server using X.509 certificates (with optional client authentication)
OAuth 2.0	IETF standard for an authorisation framework whereby resource owners can authorise delegated access by third-party clients to protected resources; OAuth enables access delegation without sharing resource owner credentials, with optional limits to the scope and duration of access
JavaScript Object Notation (JSON)	Ecma[18] standard text format for structured data interchange – not a security standard per se, but a key component of several standards listed here
JSON Web Signature (JWS)	IETF standard for attaching digital signatures or Message Authentication Codes (MAC) to JSON objects
JSON Web Encryption (JWE)	IETF standard for encrypted JSON objects
JSON Web Keys (JWK)	IETF standard for representing public and private keys (or sets of keys) as JSON objects
JSON Web Algorithms (JWA)	Specifies cryptographic algorithms to be used in the other JOSE standards
JavaScript Object Signing and Encryption (JOSE)	Collective name for the set of JSON-based cryptographic standards (JWS, JWE, JWK, and JWA)
JSON Web Token (JWT)	IETF standard for conveying a set of claims between two parties in a JSON object, with optional signature and encryption provided by the JOSE standards
OpenID Connect 1.0	OpenID Foundation standard for identity federation based on OAuth 2.0, using JWT to convey signed and optionally encrypted identity claims
User-Managed Access (UMA)	Proposed IETF standard for an OAuth 2.0-based access management protocol enabling resource owners to create access policies authorising requesting parties to access their resources through OAuth clients

Figure 6 below illustrates the dependencies among the security standards, with each standard depending on the others that lie directly beneath it.

[18] ECMA was originally an acronym standing for the European Computer Manufacturers Association, but the organization changed its name in 1994 to Ecma International to reflect its global focus

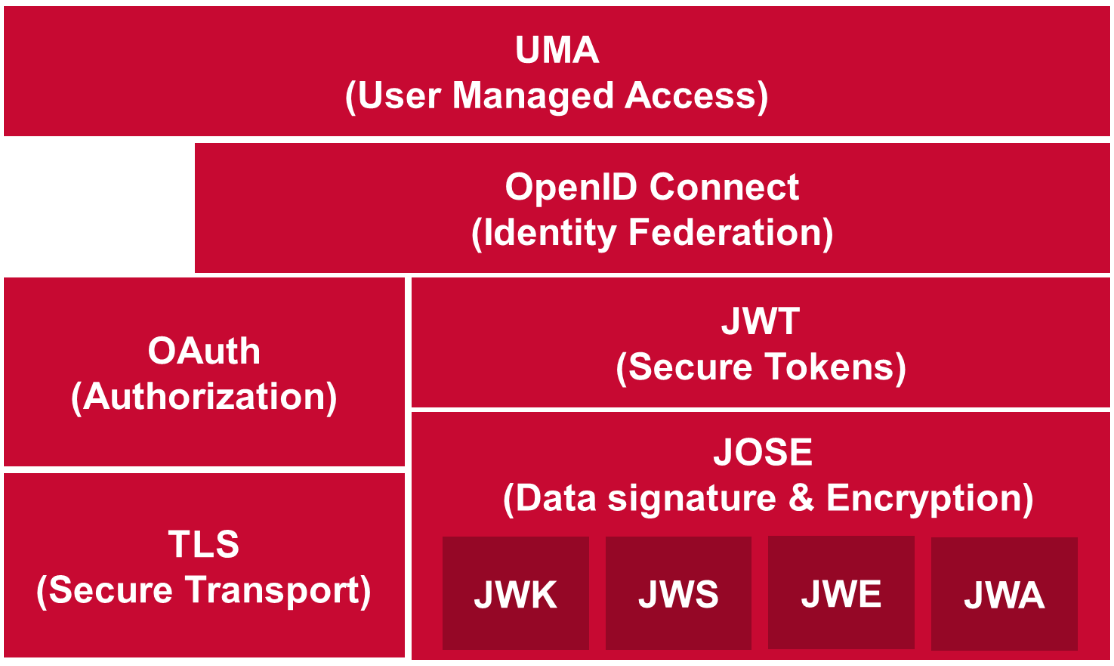

Figure 6: REST Security Standard Dependencies

Annex B Document Management

Document information

Project Owner	GSMA Inclusive Tech Lab, Mobile Money, Mobile for Development
Document Title	GSMA Mobile Money API - Security Design and Implementation Guidelines
File Name	GSMA Mobile Money API - Security Design and Implementation Guidelines1_8.docx
Key Words	GSMA, Mobile Money, Harmonised API, Security Design
Classification	Non-Confidential
Status	Release
Distribution	GSMA

Version history

Version	Date	Status	Author
1.6	06-09-2016	Published	GSMA
1.7	16-01-2020	Draft	GSMA
1.8	31-03-2020	Published	GSMA

Change history

Version	Date	Changes
1.6	06-09-2016	Final Published version of Document
1.7	16-01-2020	Internal Draft version of Document Links and Template updated
1.8	31-03-2020	Final Published version of Document 1. Reshaped introduction, regrouping considerations about main security

		concepts, adding high-level threat model considerations, updating and completed reference table Update to distinguish the concept of authentication from the one of authorization, but also remain how to protect confidentiality, integrity and authenticity of data, and availability of services, and some other concepts. Updated References subsection resolving main invalid links and adding few relevant references, TLS 1.3, documents from ETSI and IETF. 2. Reshaped document plan, isolating data encryption and authenticity / integrity in a dedicated section Restructured client and gateway authentication scenarios and outlined recommendations at different levels and using different technologies, including Layers 6 and 7. Detail the role of each parties play in different kind of authentication. Updated TLS authentication describing only the handshake protocol, not the application data one with encryption and MAC. These TLS subsections gives high level recommendations on how to verify X509 certificates. 3. Improved TLS description: handshake, authentication, key generation and cipher suites 4. Added a subsection dedicated to API Client Authentication at the HTTP level (Basic, API Key and OAuth). 5. A new section 5 has been added “Data Protection – Security Design”, it recalls the encryption and data authenticity and integrity services to be implemented at different levels (TLS, HTTP, JSON). 6. Section 6 “API Best Practices” updated in a few points (E.g. HTTP 406 error) 7. All external references updated to latest versions where applicable and updated links provided.