**Crypto Server**

Table of Contents

[1. Introduction 2](#_Toc469326138)

[1.1 Overview 2](#_Toc469326139)

[2 Design 2](#_Toc469326140)

[2.1 Client and Server Authentications 2](#_Toc469326141)

[2.1.1 Mutual Authentication Key Generation 3](#_Toc469326142)

[3 Crypto Server API 4](#_Toc469326143)

[3.1 REST API 4](#_Toc469326144)

[3.2 Request 5](#_Toc469326145)

[3.3 Response 5](#_Toc469326146)

[4 Assumptions 5](#_Toc469326147)

# Introduction

This document explains in detail the Crypto Server and the HTTP client used for testing the Crypto Server end-points.

## Overview

Crypto Server is a web server which exposes REST APIs for crypto operations.

Entire Web Server is implemented using Node.js. Following modules have been used for the implementation.

1. Express: It is a minimal and flexible Node.js web application framework which provides a robust set of features for web.
2. Crypto: It provides all needed crypto functionalities.
3. TLS: It provides an implementation of the Transport Layer Security (TLS) and Secure Socket Layer (SSL) protocols that is built on top of OpenSSL.

OpenSSL version used is: 1.0.1e-fips 11 Feb 2013 or higher.

1. HTTPS: It is the HTTP protocol over TLS/SSL

# Design

Crypto Server uses TCP port 8443 to listen for incoming HTTPS client connections. Standard HTTPS runs on port 443, but since the HTTPS server would run as non-root user, the server does not use a port from the reserved port range and so TCP port 8443 is used.

Data sent and received between Crypto Server and the HTTP client are passed as JSON objects.

Server is based on MVC design. However, there is no implementation for model and view and the crypto processing is done in the controller. When the request reaches the server, it routes the URI to the controller. Then controller process it and response back with the prepared response JSON.

Model can be implemented in future if we want to extend this for a specific user and the respective crypto operations for an authorized user. Also if we want to persist some data, we can make use of model.

View is normally used for a website.

## Client and Server Authentications

Usually, in browser based HTTPS communications, such as e-commerce, one-way authentication is done, where the browser verifies the credentials of the HTTP server it communicates with via the certificate that the server sends to the browser. This is done during HTTPS hand-shake.

In Crypto Server, the HTTPS protocol used between server and the HTTP client is used to authenticate both communicating parties. Server will verify the certificate that the client provides against the self-signed CA certificate stored on the Server, thus authenticating the client. Likewise, the HTTP client would verify the certificate that the CM appliance provides against the self-signed CA certificate stored on the client, thus authenticating the serve it is communicating with.

Additionally, both the server and the client use private-keys generated at the time of respective certificate creation.

Default cipher suite used during negotiation is TLS\_ECDHE\_RSA\_WITH\_AES\_128\_GCM\_SHA256

### Mutual Authentication Key Generation

Below diagram shows the keys and certificates deployed in server and client.



Commands used to generate CA, Server and Client certificate, keys are:

Openssl> genrsa -des3 -out ca.key 2048

Openssl> req -new -x509 -days 365 -key ca.key -out ca.crt

Openssl> x509 -in ca.crt -text -noout

Create server certificate, request signing, sign with our CA and test:

Openssl> genrsa -out server.key 2048

Openssl> req -new -key server.key -out server.csr

Openssl> x509 -req -in server.csr -out server.crt -CA ca.crt -CAkey ca.key -CAcreateserial -days 365

Openssl> x509 -in server.crt -text -noout

Create Client certificate, request signing, sign with our CA and test:

Openssl> genrsa -out clientA.key 2048

Openssl> req -new -key clientA.key -out clientA.csr

Openssl> x509 -req -in clientA.csr -out clientA.crt -CA ca.crt -CAkey ca.key -CAcreateserial -days 365

Openssl> x509 -in clientA.crt -text –noout

# Crypto Server API

Crypto Server primarily exposes 1 API call to the HTTP client to send and receive data from it. This is “Crypto-Operation”

Most data exchanged for these API calls will use JSON that gets sent in the body of the HTTP request and/or HTTP response.

All the above functionalities exposed to the clients by the Crypto Server are mapped to a set of REST APIs that are described in the next section.

## REST API

Crypto Server exposes 1 types of resources and HTTP-POST operation is only allowed.



The URI (named: “/api/crypto”) is used as a crypto operation resource. When an HTTP ‘POST’ is sent for this URI, the Crypto Server responds with a JSON containing the desired crypto operation output.

## Request

JSON for encryption request using AES-256-CTR is:

{

type: 'encrypt',

algo: 'aes-256-ctr',

text: 'A1B2C3D4E5G6A1B2C3D4E5G6'

}

JSON for encryption request using AES-256-CTR is:

{

type: 'decrypt',

algo: 'aes-256-ctr',

text: '2ba125a5edb74c0a2c336ef54fc0e39b8e0f7bc9dd8a1cdf'

}

Other options for algo is aes-256-cbc.

## Response

JSON for any valid response is:

{

message: success message / error message,

data: cipher or decipher data

}

Here ‘data’ is optional and in case of error JSON will contain only message.

# Assumptions

Following assumptions are considered during implementation.

1. Crypto Server will generate the keys for cipher and decipher using a constant salt/seed. It is not provided by the client.
2. From the available symmetric algorithms (like DES, 3DES, AES, …) AES is chosen.
3. There are few AES modes for encryption and decryption. Implemented modes are 256 CTR. 256 CBC encryption is implemented and tested. But decryption has a bug and could not decrypt the last block.
4. No authentication is needed at application layer. E.g. user management.
5. HTTP header used Transfer-Encoding as 'chunked'.
6. Error handling is done for all flows.