Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 6, 2016

A. Biggs
S. Cooley
Cisco Systems

November 03, 2015

Key Management Service Architecture
draft-abiggs-saag-key-management-service-03

Abstract

In the interest of addressing pervasive threats to the confidentiality and integrity of online communications identified by the Internet community [I-D.barnes-pervasive-problem] this specification introduces an open architecture for the establishment, management, and secure distribution of cryptographic keys for use in the end-to-end (E2E) encryption of online communications and shared content. This architecture allows for the independent deployment of dedicated key management services in a manner that supports the adoption of third-party communications and data sharing services by individuals and organizations that require full and exclusive discretion over the confidentiality of their data.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Providers of cloud-based services commonly secure user data at the transport level using established protocols such as TLS [RFC5246] or IPSec [RFC4301]. These protocols can be effective in protecting transmitted user data from third party tampering and eavesdropping; however, by themselves these protocols do not secure user data from abuses, negligence, or coerced behavior on the part of the cloud provider. This is a concern for individuals and organizations that wish to take advantage of cloud-based communications and collaboration but cannot accept the risk of trusting cloud providers with unfettered access to the contents of their communications.

E2E encryption describes a category of solutions that can be employed to address this problem by establishing secure channels among clients. To the extent that a user can trust their collaboration client software, E2E encryption mitigates exposure of user data to untrusted parties by ensuring that intermediaries never possess unencrypted user data or have access to the keying material necessary to decrypt it.

Existing E2E strategies such as ECS [RFC5652], PGP [RFC4880], and Off-the-Record Messaging [OTR] can be effective at securing two-party communications. However, E2E encryption for the growing domain of multiparty communications and online content sharing remains a generally unsolved problem to which these existing approaches do not readily adapt. In particular, a core challenge exists in providing for the secure distribution and rotation of E2E encryption keys among an arbitrary and potentially dynamic set of communicating clients. In cases where the communications to be secured are persistent or archived, the additional challenge exists for providing trusted long-term storage and retrieval of these keys.

Equally problematic is the paucity of E2E encryption options that satisfy common organizational obligations such as regulatory compliance and legal discovery. Entities that must operate within such frameworks require mechanisms by which they (and they alone) may recover the keys used to secure their communications. Existing E2E encryption solutions are not, by themselves, well suited for this purpose.

In the interest of addressing these challenges, this document presents an architecture for the deployment of E2E encryption key management services (KMS) based on the curator role described in [I-D.abiggs-saag-primitives-for-conf-group-comms].

In the interest of addressing these challenges, this document presents an architecture for the deployment of E2E encryption key

management services (KMS). In this architecture a KMS service provides to its users a means by which their communications clients may securely create, share, rotate, and store E2E encryption keying material. It does so in a fashion that permits the decoupling of such services from the communications media, thereby permitting the former to reside under the direct control of the communicating parties or the organizations within which they do business.

1.1. Terminology

This document uses the terminology from [I-D.ietf-jose-json-web-signature], [I-D.ietf-jose-json-web-encryption], [I-D.ietf-jose-json-web-key], and [I-D.ietf-jose-json-web-algorithms] when discussing JOSE technologies.

This document uses the terminology from [I-D.abiggs-saag-primitives-for-conf-group-comms] when discussing authentication, group membership, and secure key exchange.

This document makes use of the following terminology, and additionally adopts nomenclature defined in [I-D.barnes-pervasive-problem] for the purpose of describing aspects of pervasive attacks.

communications resource

A communications resource is any uniquely identifiable continuous data channel or discrete shared content that represents an exchange of personal communications between two or more users.

communications resource client

A communications resource client consumes communications resources on behalf of a user and, when deployed in conformance with the KMS architecture, consumes the services of KMS server(s) to facilitate the E2E encryption of those communications resources.

communications resource server

A communications resource server is a provider of services through which communications resources are made available.

cloud provider

An individual or organization responsible for, and in control of, one or more communications resource servers.

E2E encryption

Shorthand for end-to-end encryption, as defined in [RFC4949], particularly as it applies to the establishment of confidentiality and integrity of communications resources.

KMS server

A key management server (KMS) is responsible for creating, storing, and providing access to E2E encryption keying material by communications resource clients.

KMS protocol

The protocol through which communications resource clients interoperate with KMS servers.

KMS provider

An individual or organization responsible for, and in control of, a KMS server deployment.

KMS transport

Any service or protocol that provides the basic transport over which KMS protocol messages are exchanged.

resource client

See communications resource client.

resource server

See communications resource server.

trusted party

A trusted party is an individual or organization that is trusted by one or more communicating users to maintain the confidentiality and integrity of their communications resources.

1.2. Security Terms

Most security-related terms in this document are to be understood in the sense defined in [RFC4949]; such terms include, but are not limited to, "attack", "authentication", "authorization", "certification authority", "certification path", "certificate",

"credential", "identity", "self-signed certificate", "trust", "trust anchor", "trust chain", "validate", "verify", and "verify".

1.3. Notational Conventions

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in BCP 14, RFC 2119 [RFC2119].

2. Architectural Overview

The architectural reference model for this specification is illustrated in Figure 1. Central to this model is the communications resource server which is presumed to be operated by a cloud provider for the purpose of offering some form of communications service. The nature of this service is not prescribed by this specification and may take the form of any of a variety of communications or collaboration services including file sharing, messaging, and VoIP. Consuming the services of the communications resource server are communications resource clients which may be supplied by the cloud provider or developed by third parties.

```
                         +-----------------+
                         | Communications  |
     +-------------------| Resource Server |-------------------+
     |                   | (Cloud Provider)|                   |
     |                   +-----------------+                   |
     |                                                         |
     |                                                         |
     |                   +-----------------+                   |
     |       +-----------|  KMS Transport  |-----------+       |
     |       |           +-----------------+           |       |
     |       |                    |                    |       |
     |       |   Untrusted        |                    |       |
- - -|- - - -|- - - - - - - - - - | - - - - - - - - - -|- - - -|- - -
     |       |   Trusted          |                    |       |
     |       |                    |                    |       |
+-----------------+      +-----------------+      +-----------------+
| Communications  |      |    KMS Server   |      | Communications  |
| Resource Client |      |  (KMS Provider) |      | Resource Client |
+-----------------+      +-----------------+      +-----------------+
         |                                                 |
+-----------------+                               +-----------------+
|      Alice      |                               |       Bob       |
+-----------------+                               +-----------------+
```

Figure 1: KMS Architecture Reference Model

In addition to the familiar elements described above, this model also includes a key management server, or KMS, operated by a KMS provider. The KMS server exposes an API through which clients may securely post and share cryptographic keying material used for the E2E encryption of content that is transited through the cloud provider's services. The API exposed by the KMS implements the curator functions described in the Centralized Groups section of [I-D.abiggs-saag-primitives-for-conf-group-comms]. It is important to note that the KMS is a dedicated curator rather than a content producing curator. This API is secured in such a way as to ensure these keys are visible to none but the KMS server itself and the clients authorized to consume the content they protect. This highlights an important distinction between the KMS provider and the cloud provider: while the KMS provider is necessarily a trusted party, the cloud provider need not be.

It is an explicit objective of this specification to promote an ecosystem of providers of KMS implementations and KMS services that are distinct and independent of the cloud providers over whose services users communicate. To that end, this specification seeks to standardize a KMS service protocol though which clients and KMS servers interoperate. This protocol provides for the establishment of a confidential and authenticated channel between each client and KMS server, and defines an API of request and response messages to be exchanged over this secure channel for the purpose of creating, retrieving, and exchanging keys.

While the KMS service protocol constitutes a central focus of this specification, the means by which this protocol is transported is expressly out of scope. This role may be readily addressed through either standards-based or proprietary protocols, and so we refer to this simply as the KMS transport for the remainder of this document. Over this transport, the communication paths between clients and KMS server are encrypted using keys established through an authenticated ephemeral key agreement. As such, the KMS transport provider need not be regarded as a trusted party, and in fact may be the cloud provider itself.

An important non-goal of this specification is the standardization of any aspect of the cloud provider's services or the means by which clients utilize shared keys for the E2E encryption of data transiting those services. By avoiding the application of constraints on the communications services and protocols we enable the use of this specification in the context of existing service deployments, both standards-based and proprietary. It is similarly a non-goal of this specification to enable federation of secure communications between vendors of different cloud services, as that is the realm of standardized application protocols. The scope of this specification

is intended to be narrowly focused on the task of separating E2E encryption key management from the communications services they secure, thereby facilitating the broadest possible adoption of secure communications though existing services.

3. Use Cases

The use cases described in this section are non-normative examples meant to illustrate how the KMS architecture may be deployed to provide E2E encryption of different types of communications resources. These use cases differ in detail, but generally follow a common logical sequence as given below.

Note that all requests to the KMS server are via the KMS transport which, for clarity, has been omitted from the sequence diagrams included in this section.

```
      Resource          Resource          Resource           KMS
      Client B          Client A            Server            Server
         |                 |                 |       (1)       |
         |                 |-----------------|---------------->|
         |                 |                 |       (2)       |
         |                 |-----------------|---------------->|
         |                 |                 |                 |
         |             (3) |                 |                 |
         |                 |       (4)       |                 |
         |                 |---------------->|                 |
         |                 |                 |       (5)       |
         |                 |-----------------|---------------->|
         |                 |                 |       (6)       |
         |                 |-----------------|---------------->|
         |                 |       (7)       |                 |
         |-----------------|---------------->|                 |
         |                 |                 |       (8)       |
         |-----------------|-----------------|---------------->|
         |                 |                 |                 |
     (9) |                 |                 |                 |
         |                 |                 |                 |
```

Figure 2: Nominal Use Case

1. Client A requests a new GMBC from the KMS server, including an initial operation to add itself as a member. The KMS creates and returns a new genesis block with the KMS as curator and client A as a member.

2. Client A requests that the KMS generate a new GK. The KMS generates the GK and returns it to the client with client A as the only recipient of the embedded JWE used to wrap the included key material.

3. Client A encrypts a resource using the key material protected by the GK.

4. Client A posts the encrypted resource to the resource server, including the URI of the GK as metadata.

5. Client A creates and signs a new GMBC block containing an operation to add client B and with the hash of the genesis block from step 1 as the "antecedent". Client A posts this block to the KMS for appending to the GMBC.

6. Client A posts a request to the KMS to update the GK and bind it to the GMBC by setting the "block" attribute of the GK to be the hash of the GMBC block posted in 5 (this has the effect of linking authorization for retrieval of the GK to the membership of the GMBC at that particular block).

7. Client B obtains the encrypted resource from the resource server, including the GK URI as metadata.

8. Client B performs a GK Get to obtain the GK from the KMS server. The KMS checks the "block" attribute on the requested GK and examines the GMBC block to which it refers. The membership of the GMBC at that block includes client B, so the server returns the GK to the client with client B as the only recipient of the JWE used to wrap the included key material.

9. Client B decrypts the resource using the key material protected by the GK.

3.1. Securing an HTTP File Sharing Service

Let A be a user that wishes to share a file with users B and C through some HTTP based file sharing service. In the context of the KMS architecture we may regard the file sharing provider's HTTP API as the resource server and the users' HTTP clients as the resource clients.

For this scenario we also assume that the file sharing service is trusted by user A with the role of providing a file sharing mechanism but is not necessarily trusted to adequately protect the confidentiality of the file contents. User A's concerns may then be addressed through the introduction of an HTTP based KMS transport

(not shown) and a KMS server deployed by an entity that A regards as a trusted party.

```
        HTTP           HTTP           HTTP       HTTP File         KMS
      Client C       Client B       Client A    Share Server      Server
         |              |              |              |             |
         |              |              |              |     (1)     |
         |              |              |--------------|------------>|
         |              |              |              |     (2)     |
         |              |              |--------------|------------>|
         |              |              |              |             |
         |              |          (3) |              |             |
         |              |              |     (4)      |             |
         |              |              |------------->|             |
         |              |              |              |     (5)     |
         |              |              |--------------|------------>|
         |              |              |              |     (6)     |
         |              |              |--------------|------------>|
         |     (7)      |     (7)      |              |             |
         |<-------------|<-------------|--------------|             |
         |              |              |     (8)      |             |
         |              |--------------|------------->|             |
         |              |              |              |     (9)     |
         |              |--------------|--------------|------------>|
         |              |              |              |             |
         |         (10) |              |              |             |
         |              |              |              |             |
    (11) |              |              |              |             |
         |              |              |              |             |
```

Figure 3: File Sharing Use Case

This sequence begins with the assumption that each client has, at some point, already established a secure channel to the KMS via authenticated key agreement.

1. Client A requests a new GMBC from the KMS server, including an initial operation to add itself as a member. The KMS creates and returns a new genesis block with the KMS as curator and client A as a member.

2. Client A requests that the KMS generate a new GK. The KMS generates the GK and returns it to the client with client A as the only recipient of the embedded JWE used to wrap the included key material.

3. Client A encrypts a file using the key material protected by the GK.

4. Client A posts the encrypted file to the file sharing service, including the URI of the GK as metadata.

5. Client A creates and signs a new GMBC block containing an operation to add clients B and C, and with the hash of the genesis block from step 1 as the "antecedent". Client A posts this block to the KMS for appending to the GMBC.

6. Client A posts a request to the KMS to bind the GK to the GMBC by setting the "block" attribute of the GK to be the hash of the GMBC block posted in 5.

7. Clients B and C learn of the newly shared file from the file sharing service (the mechanism by which this occurs is out of scope for this specification).

8. Client B retrieves the encrypted file from the file sharing service, including the GK URI as metadata.

9. Client B performs a GK Get to obtain the GK from the KMS server. The KMS checks the "block" attribute on the requested GK and examines the GMBC block to which it refers. The membership of the GMBC at that block includes client B, so the server returns the GK to the client with client B as the only recipient of the JWE used to wrap the included key material.

10. Client B decrypts the file using the key material protected by the GK.

11. Client C performs steps 8 through 10 in the same fashion as client B.

It is worth noting that a race condition does exist where step 9 could occur before steps 5 and 6 complete. This will result in a client being temporarily denied access to the GK used to encrypt the shared file.

3.2. Securing an XMPP Multi-User Chat

Let A, B and C be users that wish to engage in secure chat through an existing XMPP multi-user chat room. In the context of the KMS architecture we may regard the XMPP MUC service as the resource server, the users' XMPP clients as the resource clients, and the XMPP service itself (not shown) as the KMS transport.

```
        XMPP           XMPP           XMPP        XMPP MUC       KMS
      Client C       Client B       Client A       Server       Server
         |              |              |              |             |
         |              |              |              |     (1)     |
         |              |              |--------------|------------>|
         |              |              |              |     (2)     |
         |              |              |--------------|------------>|
         |              |              |      (3)     |             |
         |              |              |------------->|             |
         |      (4)     |      (4)     |              |             |
         |<-------------|<-------------|--------------|             |
         |              |              |              |     (5)     |
         |              |--------------|--------------|------------>|
         |              |              |              |             |
         |          (6) |              |              |             |
         |              |              |              |             |
     (7) |              |              |              |             |
         |              |              |              |             |
```

Figure 4: Multi-User Chat Use Case

This sequence begins with the assumption that a MUC room already exists on the MUC server and that each client has already established a secure channel to the KMS via authenticated key agreement. All messages are transmitted over XMPP, with the presumption that appropriate XMPP extensions are developed to provide bindings for KMS operations.

1. Client A requests a new GMBC from the KMS server, providing initial operations to add clients A, B, and C. The KMS creates and returns a new genesis block with the KMS as curator and clients A, B, and C as members.

2. Client A requests that the KMS generate a new GK, and to have it immediately bound to the genesis block created in step 1. The KMS generates the GK and returns it to the client with client A as the only recipient of the embedded JWE used to wrap the included key material.

3. Client A encrypts the content of an XMPP message using the key material from the GK created in step 2, and sends the encrypted message to the MUC room. The GK URI is included within the XMPP message as metadata.

4. The MUC service delivers client A's encrypted message to clients B and C.

5. Client B performs a GK get operation to retrieve the GK from the KMS server using the GK URI included in the encrypted message's metadata.

6. Client B decrypts the messages using the key material protected by the GK.

7. Client C performs steps 5 and 6 in the same fashion as Client B.

3.3. KMS to KMS Key Federation

This use case illustrates two KMS instances federating keys associated with a resource. As KMS servers are deployed to serve groups of users it is inevitable that users will want to share resources across groups or organizations. This cross-organization sharing of keys leads to several problems. First, each user is only known to and only knows of one logical KMS. Second, each organization might have very different archiving requirements due to differing legal compliance regulations due to jurisdiction or industry differences. Lastly, one or both of the users might be employees of enterprises that need to be able to respond to legal discovery requests. To address these issues, KMS servers may federate in such a way as to allow for limited copying of keys from one KMS to another. This permits each KMS' owning organization the ability to control the ongoing policy regarding access to keys for which their respective users are authorized.

Let Alice@DomainA and Bob@DomainB be users of a common file sharing service and who happen to use different KMS servers to secure their communications. Assume then that Alice wishes to share a file with Bob and therefore relies on KMS server federation to facilitate the key exchange.

```
   HTTP Client   HTTP Client     HTTP File      KMS Server   KMS Server
   Bob@DomainB  Alice@DomainA   Share Server     DomainA       DomainB
        |             |              |      (1)     |             |
        |             |--------------|------------->|             |
        |             |              |      (2)     |             |
        |             |--------------|------------->|             |
        |             |      (3)     |              |             |
        |             |------------->|              |             |
        |     (4)     |              |              |             |
        |<------------|--------------|              |             |
        |             |      (5)     |              |             |
        |-------------|------------->|              |             |
        |             |              |              |     (6)     |
        |-------------|--------------|--------------|------------>|
        |             |              |              |     (7)     |
        |             |              |              |<------------|
        |             |              |              |     (8)     |
        |             |              |              |------------>|
        |     (9)     |              |              |             |
        |<------------|--------------|--------------|-------------|
        |             |              |              |             |
   (10) |             |              |              |             |
        |             |              |              |             |
```

Figure 5: File Sharing with KMS Federation Use Case

This sequence begins with the assumption that each client has, at some point, already established a secure channel to their respective KMS via authenticated key agreement.

1. Alice@DomainA requests a new GMBC from the KMS server, providing initial operations to add Alice@DomainA and Bob@DomainB as members. The KMS creates and returns a new genesis block with the KMS as curator.

2. Alice@DomainA requests that the KMS generate a new GK, and to have it immediately bound to the genesis block created in step 1. The KMS generates the GK and returns it to the client with Alice@DomainA as the only recipient of the embedded JWE used to wrap the included key material.

3. Client A encrypts a file using the key material from the GK created in step 2, and sends the encrypted message to the file sharing service. The GK URI is included as metadata.

4. Bob@DomainB learns of the newly shared file from the file sharing service (the mechanism by which this occurs is out of scope for this specification).

5. Bob@DomainB retrieves the shared file from the file sharing service along with the GK URI contained in metadata.

6. Using the GK key URI obtained in step 7, Bob@DomainB requests the GK from his own KMS at DomainB.

7. The DomainB KMS recognizes the GK URI as actually hosted by the DomainA KMS. The DomainB KMS establishes a secure and mutually authenticated channel with the DomainA KMS via the KMS transport (if not previously established) and requests the GK from the DomainA KMS on behalf of Bob@DomainB.

8. The DomainA KMS first checks to see if Bob@DomainB is entitled to retrieve the GK. If so, it then consults the WebFinger resource of Bob@DomainB to determine whether DomainB is entitled to request GKs on behalf of Bob@DomainB. If so, DomainA KMS will respond by returning the GK to DomainB in such a way as the JWE used to wrap the key material is encrypted with the public key of the DomainB KMS server.

9. DomainB KMS returns the GK to Bob@DomainB in such a way as the JWE used to wrap the key material is encrypted with Bob@DomainB's public key, and the GK itself is signed with the DomainB KMS private key.

10. Bob@DomainB decrypts the shared file using the key obtained in step (11).

Note that in step 8 the DomainB KMS is being trusted by DomainA KMS to not share the GK key material with anyone other than those users on whose behalf it has acted and successfully retrieved the GK. This is a necessary consequence of KMS federation, where the act of authorizing access to a GK by a user residing in a federated domain engenders an implicit trust of the KMS server that controls the federated domain. For that reason, a KMS provider should restrict federation of its KMS servers to domains that the KMS provider regards as trusted.

4. KMS Protocol

The KMS protocol is composed of a message oriented request and response API and a secure channel over which those messages are exchanged. The API provides clients with the ability to post and retrieve GMBC and GK objects. The secure channel provides a mutually authenticated and E2E encrypted channel over which the clients and KMS server may exchange API messages securely. The API and secure channel are described in detail through the remainder of this section.

4.1. Secure Channel

The secure channel is an encrypted and mutually authenticated communication path between each client and the KMS server. It transits the KMS transport which, in the context of this document, represents any service or protocol that may be utilized for the relaying of KMS API request and response messages. This specification presumes the KMS transport to be untrusted.

```
                          (3) Secure Channel
        +====================================================+
        V                                                    V
+-----------------+  (2)   +----------------+   (1)  +---------------+
| Resource Client |<------>| KMS Transport  |<------>|   KMS Server  |
+-----------------+        +----------------+        +---------------+
                      |                          |
     Trusted          |        Untrusted         |        Trusted
```

Figure 6: Establishment of a Secure Channel

At a high level, the communication path between a resource client and KMS is established as follows.

1. The KMS actively connects to the KMS transport. The protocol used for this connection is out of scope for this document, however it MUST support the asynchronous flow of encrypted KMS request and response messages between the KMS and KMS transport.

2. A resource client connects to the KMS transport. The protocol used for this connection is out of scope for this document, however it MUST support the flow of encrypted KMS request and response messages between the resource client and the KMS transport.

3. Through an elliptic curve Diffie-Helman key exchange, augmented by server and client authentication, the resource client and KMS establish a shared secret and derived ephemeral key. This is discussed in greater detail in Section 4.6.1.

On successful mutual authentication and ephemeral key agreement, the resource client and KMS communicate through the exchange of sequenced request and response messages encrypted using the ephemeral key.

4.2. User Identity

Central to the KMS server's role as a key store is its ability to both restrict access to stored keying material and to rekey keying material to only authorized users. This requires robust user

authentication and a means for the unambiguous and unique identification of each user.

Conforming KMS architecture deployments MUST rely on an identity provider that supports the generation of OAuth 2.0 [RFC6749] access tokens. The KMS server MUST rely on same identity provider for the purpose of validating access tokens received from the client. Access tokens used by clients to demonstrate identity and authorization for access to KMS resources MUST NOT be used for any other service. Any exposure of a KMS recognized access token to third parties (such as providers of other services or a resource server) jeopardizes the security of all GMBC and GK objectgs for which the user whose identity is represented by the token is authorized.

The identity provider on which the KMS server relies MAY be the same identity provider as relied upon by the resource server(s) whose communications resources are encrypted with GK objects managed by the KMS server. Note, however, the reliable authentication and authorization of clients to the KMS server is critical to the security of the KMS keys it holds. The identity provider on which the KMS relies must therefore necessarily be regarded as a trusted party within the context of the KMS architecture.

Access tokens MUST be conveyed to the KMS server as part of the payload of encrypted KMS API requests as described in Section 4.5.1 and MUST NOT be conveyed in any other manner.

4.3. KMS Identity

Given the untrusted nature of the KMS transport by both the KMS and clients, it is critical for clients to be able to verify the identity of their KMS and ensure that no MITM attacks are carried out on client to KMS or KMS to client communications. Therefore, the KMS MUST make use of at least one PKIX certificate [RFC5280] and clients MUST validate the PKIX certificate presented by the KMS through the comparison of the certificate's common name (CN) or subject alternative name (SAN) [RFC6125] fields to the Internet domain portion of the user's Addr-spec [RFC2822] formatted unique identifier using the procedures defined in section 6 of [RFC6125]. An acceptable alternative to direct CN or SAN validation is defined in [I-D.ietf-xmpp-posh].

PKIX certificates presented by the KMS can be issued by either a public or private certification authority with the stipulation that clients MUST be able to validate the KMS's entire certificate path through the pre-established trust of the root certificate used to anchor the certificate path. The mechanism for establishing trust of the root certificate is out of scope for this specification, but it

is usually carried out through pre-installed trusted root certificates on various operating systems for public certification authorities or through enterprise endpoint management solutions or manual installation tasks for private certification authorities.

4.4. Object Types

The KMS protocol is based on operations on GMBC and GK objects. Specifically, these include the following JSON object types defined using using JSON content rules [I-D.newton-json-content-rules] in [I-D.abiggs-saag-primitives-for-conf-group-comms]:

gmbc-genesis-block

gmbc-appended-block

group-key

It is through the creation and retrieval of instances of these object types that clients interact with the KMS.

4.5. Message Structure

Every KMS request and response message is composed of a JSON [RFC7159] formatted payload encapsulated within either a JWE [I-D.ietf-jose-json-web-encryption] or JWS [I-D.ietf-jose-json-web-signature] object. These messages may be divided into three types.

Common Messages

Common messages include all those which do not meet the definition of either key agreement message or error message. Common messages are encrypted as JWE objects using the shared ephemeral key established during initial key agreement between the client and KMS (see Section 4.6.1). The value of the JWE header "kid" attribute of a common message MUST match that of the KMS ephemeral key object URI attribute established during initial key agreement.

Ephemeral Key Agreement Messages

Ephemeral key agreement messages are those exchanged between the client and KMS for the purpose of establishing a new shared ephemeral key (see Section 4.6.1). Key agreement request payloads are encrypted as JWE objects using the authenticated and validated static public key of the KMS. Key agreement response payloads are signed as JWS objects using the static private key of the KMS. The value of the JWE or JWS header "kid" attribute of an ephemeral

key agreement message MUST be a well known key identifier for the KMS static public key.

Error Messages

Error messages are those originated by the KMS to indicate a failed request. Error messages are composed in the same fashion as common messages; however, in the event that the KMS does not recognize the ephemeral key used in the request, or that key is determined to have expired, the KMS MUST respond with an unencrypted message composed as a JWS, with a payload as described in Section 4.5.3, and signed using the KMS server's static public key.

The basic JSON representations for the request and response payloads are defined in the following sections.

4.5.1. Basic Request Payload

The basic JSON representation for KMS request message payloads is defined as follows using JSON content rules [I-D.newton-json-content-rules] with references to rules defined in previous sections.

```
requestId (
  "requestId" : integer
)

credential {
  "userId": ?string
  "bearer": string / "jwk": jwk
}

client {
  "clientId": string,
  "credential": credential
)

method: string /create|retrieve|update|delete/

request (
  "client" : client,
  "method" : method,
  "uri" : uri,
  requestId
)
```

The attributes of a KMS request message payload are defined as follows.

requestId

A string selected by the client and provided in each request to uniquely identify the request. The string is treated opaquely by the server and returned verbatim in the associated response.

userId

The unique identifier of the user making the request. This field is optional, and MUST be disregarded if the requesting user's identity can be securely derived from either the bearer token or jwk.

bearer

An [RFC6749] access token issued by the client's identity provider and validated by the KMS in cooperation with the identity provider. See Section 4.2.

jwk

A JWK object, in JSON format as defined in [I-D.ietf-jose-json-web-key], containing the public key of the client (presumably a server). This JWK MUST contain an x5c header with a certificate chain that may be used to positively validate the public key.

clientId

An opaque unique identifier provided by the client (not used for authentication, only to assist multiple clients of a single user in differentiating between their respective unbound keys).

method

Indicates the request type: create, retrieve, update, or delete.

uri

A URI identifying a KMS object or object type (e.g. GMBC or GK) to which the request applies.

The JSON content rules above are used in conjunction with additional request type specific rules, defined later in this document, to produce the full request payload definition for each KMS operation.

4.5.2. Basic Response Payload

The basic JSON representation for KMS request message payloads is defined as follows using JSON content rules with references to rules defined in previous sections.

```
response (
  "status" : integer,
  ?"reason" : string,
  requestId
)
```

The attributes of a KMS request message payload are defined as follows.

status

Indicates the success or failure of the request. The value returned in a response status attribute SHOULD be that of an [RFC7231] defined status code with semantics that correspond to the success or failure condition of the KMS request.

reason

An optional natural language string to describe the response status in terms that are useful for tracing and troubleshooting the API.

requestId

An echo of the requestId provided in the request.

The JSON content rules above are used in conjunction with additional response type specific rules, defined later in this document, to produce the full response payload definition for each KMS operation.

4.5.3. Error Response Payload

The JSON representation for KMS error response message payloads is defined as follows using JSON content rules with references to rules defined in previous sections.

Error response payload definition:

```
root {
  response
}
```

Error response message example:

```
JWS(K_kms_priv, {
  "status": 403,
  "reason": "The ephemeral key used in the request has expired.",
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5"
})
```

4.6. Requests

The following sections provide detailed descriptions for each of the request and response operations that may occur between a resource client and the KMS.

4.6.1. Create Ephemeral Key

The first operation between a client and KMS MUST be the establishment of a shared secret and derived ephemeral key. This is necessary as all other requests and responses are encrypted with the ephemeral key.

The client request for creating an ephemeral key conforms to the basic request message payload, where the method is "create" and the uri is "/ecdhe". In addition to the basic payload, the client provides a jwk attribute for which the value is a JWK object [I-D.ietf-jose-json-web-key] containing the public part of an EC key pair generated by the client. Unlike a basic request message, however, the request payload is encrypted as the content of a JWE [I-D.ietf-jose-json-web-key] secured with the static public key of the KMS server (K_kms_pub) as obtained from the server's validated PKIX certificate [RFC5280].

Note, the client MUST generate a new EC key pair for every create ephemeral key request sent to the KMS server.

Request payload definition:

```
root {
  request,
  jwk
}
```

Request message example:

```
JWE(K_kms_pub, {
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  },
  "method": "create",
  "uri": "/ecdhe",
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
  "jwk" : {
    "kty": "EC",
    "crv": "P-256",
    "x": "VoFkf6Wk5kDQ1ob6csBmiMPHU8jALwdtaap35Fsj20M",
    "y": "XymwN6u2PmsKbIPy5iij6qZ-mIyej5dvZWB_75lnRgQ"
  }
})
```

On receiving the ephemeral key creation request, the KMS server MUST verify the credential provided in the request. If a bearer token is provided, the KMS MUST validate the token in cooperation with the identity provider. If a jwk is provided, the KMS MUST validate the included PKIX certificate chain against the KMS server's trust root. In either case, the identity of the requesting client MUST be authenticated and verified to correspond to either an authorized user of the KMS or an authorized trusted service. If verification fails, the KMS MUST NOT use the server response to continue with key agreement.

Upon successful authentication and authorization of the request, the KMS responds by generating its own EC key pair using the same curve as indicated in the "crv" attribute of the request message JWK. The KMS server returns the public part of this key pair to the resource client in the form of an EK object within the response payload. The KMS also generates and includes within the response payload a new key uri to be regarded by both client and KMS as the key identifier of the agreed upon ephemeral key. The response payload is returned to the resource client as the content of a JWS [I-D.ietf-jose-json-web-signature] signed using the static private key of the KMS server (K_kms_priv).

Response payload definition:

```
root {
  response,
  key
}
```

Response message example:

```
JWS(K_kms_priv, {
  "status": 201,
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
  "ephemeral-key": {
    "uri": "/ecdhe/ea9f3858-1240-4328-ae22-a15f6072306f",
    "jwk" : {
      "kty": "EC",
      "crv": "P-256",
      "x": "8mdasnEZac2LWxMwKExikKU5LLacLQlcOt7A6n1ZGUC",
      "y": "lxs7ln5LtZUE_GE7yzc6BZOwBxtOftdsr8HVh-14ksS"
    },
    "userId": "842e2d82-7e71-4040-8eb9-d977fe888807",
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "createDate": "2014-10-09T15:54:48Z",
    "expirationDate": "2014-10-09T16:54:48Z"
  }
})
```

If successful, the KMS response to a create ephemeral key request MUST have a status of 201. In the case of a request failure, the KMS response status SHOULD be that of an [RFC7231] defined status code with semantics that correspond to the failure condition. In addition, the ephemeral key SHOULD have the createDate assigned as the current time and an expirationDate assigned as the latest point in time before which the key may be used for encrypting messages (both in [RFC3339] date-time format).

On receiving the ephemeral key creation response, the resource client MUST verify the received JWS against the KMS server's validated static public key. If verification fails, the client MUST NOT use the server response to continue with key agreement.

To generate the shared secret, both resource client and KMS server use ECDH shared secret derivation with the private part of the local EC key pair and the public part of the remote EC key pair. The shared secret is then provided as input to HKDF (with both extract and expand, and empty salt) [RFC5869] to generate the ephemeral key (K_ephemeral).

The ephemeral key generated by this operation is used to encrypt all subsequent KMS requests and responses passed between the resource client and KMS. When encrypting such a message, the sender MUST assign a value to the kid attribute of the header of the resulting JWE object, and this value MUST match the URL of the key as provided to the client in the KMS response message described above. This

provides the recipient with a means for identifying the key necessary to decrypt the message.

The KMS SHOULD accept messages encrypted with the ephemeral key up to and until the key expiration date as provided in the response message described above. On expiration of the ephemeral key, the KMS MUST reject all further requests submitted using this key, and a client wishing to submit further requests to the KMS MUST re-establish the secure channel by requesting the creation of a new ephemeral key.

4.6.2. Delete Ephemeral Key

In the event that a resource client's ephemeral key has become compromised, a client SHOULD submit a request to the KMS to delete the ephemeral key.

The request message conforms to the basic request message structure, where the method is "delete", and the uri is that of the ephemeral key to be deleted.

Request payload definition:

```
root {
  request
}
```

Request message example:

```
JWE(K_ephemeral, {
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  }
  "method": "delete",
  "uri": "/ecdhe/ea9f3858-1240-4328-ae22-a15f6072306f",
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5"
})
```

The response message conforms to the basic response message structure, and MUST NOT include a representation of the deleted ephemeral key.

Response payload definition:

```
root {
  response
}
```

Response message example:

```
JWE(K_ephemeral, {
  "status": 204,
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5"
})
```

If successful, the KMS response to a delete ephemeral key request MUST have a status of 204. In the case of a request failure, the KMS response status SHOULD be that of an [RFC7231] defined status code with semantics that correspond to the failure condition.

On successful deletion of an ephemeral key, the KMS MUST NOT, from that time forward, accept any requests encrypted with that ephemeral key.

4.6.3. Post GMBC Block (genesis)

When a client intends to initiate E2E encryption of a communications resource, it begins by requesting the creation of a GMBC genesis block. In this request, the client provides basic GMBC block information which the KMS uses in generating the genesis block. The KMS will assign a unique GMBC URI to the genesis block and indicate itself as the GMBC curator.

The request message conforms to the basic request message structure, where the method is "post", and the path of the URI is "/blocks".

Request payload definition:

```
root {
  request,
  "blockPayload": gmbc-block
}
```

Request message example:

```
JWE(K_ephemeral, {
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  }
  "method": "post",
  "uri": "kms://kms.example.com/blocks",
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
  "blockPayload": {
    "creator": "bob@example.com",
    "created": "2015-11-02T19:02:15Z",
    "operations": [
      {
        "entity": "bob@example.com",
        "optype": "add"
      },
      {
        "entity": "alice@example.com",
        "optype": "add"
      }
    ]
  }
})
```

The response message conforms to the basic response message structure, and includes a representation of the created GMBC genesis block in the form of a compact-serialized JWS signed with the KMS server's private static key.

Response payload definition:

```
signed-gmbc-genesis-block: JWS(K_kms_priv, gmbc-genesis-block)

root {
  response,
  "block": signed-gmbc-genesis-block
}
```

Response message example:

```
JWE(K_ephemeral, {
  "status": 201,
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
  "block": "eyAiYWxnIjogIlBTMjU2IiB9.ewogICAgICAidXJpIjo..."
}
```

Deserialized payload of the block attribute:

```
JWS(K_kms_priv, {
  "uri": "kms://kms.example.com/blocks/7f35c3eb",
  "nonce": "32088b07-1a19-466b-a779-ef8dc8c61be9",
  "curator": "kms://kms.example.com",
  "creator": "kms://kms.example.com",
  "created": "2015-11-02T19:02:15Z",
  "operations": [
    {
      "entity": "bob@example.com",
      "optype": "add"
    },
    {
      "entity": "alice@example.com",
      "optype": "add"
    }
  ]
})
```

If successful, the KMS response to a this request MUST have a status of 201. In the case of a request failure, the KMS response status SHOULD be that of an [RFC7231] defined status code with semantics that correspond to the failure condition.

4.6.4. Post GMBC Block (append)

Once a GMBC genesis block has been created, any member may append new blocks in order to modify the group membership. This is done by submitting a post GMBC block request to the KMS. In this request, the client provides a signed gmbc-appended-block and the URI of the genesis block of the GMBC to which it is to be appended.

The client may submit one or more blocks to be appended, the order of which they appear in the request representing the order in which they should be appended. The KMS will validate that the antecedent hash of the first block matches the hash of the last block of the current chain, and that the antecedent of each subsequent block matches the hash of the previous block. The KMS will also validate that each block is signed by an entity that qualifies as a member of the chain. If any of these checks fails, the KMS will fail the request in its entirety.

The request message conforms to the basic request message structure, where the method is "post", and the uri is that of the genesis block of the GMBC to which the provided block should be appended.

Request payload definition:

```
signed-gmbc-appended-block: JWS(K_user_priv, gmbc-appended-block)

root {
  request,
  "blocks" [ *: signed-gmbc-appended-block ]
}
```

Request message example:

```
JWE(K_ephemeral, {
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  }
  "method": "post",
  "uri": "kms://kms.example.com/blocks/7f35c3eb",
  "requestId": "6205452b-c555-484f-8445-bb94c8044882",
  "blocks": [
    "eyAiYWxnIjogIlBTMjU2IiB9.ewogICAgICAiYW50ZWNlZGVud..."
  ]
})
```

Deserialized payload of the block attribute:

```
JWS(K_alice_priv, {
  "antecedent": "3a2371f8fb6bb0f96e65dc535010b4004afc...",
  "creator": "alice@example.com",
  "created": "2015-11-02T19:13:15Z",
  "operations": [
    {
      "entity": "charlie@example.com",
      "optype": "add"
    },
    {
       "entity": "bob@example.com",
       "optype": "remove"
    }
  ]
})
```

The response message conforms to the basic response message structure.

Response payload definition:

```
root {
  response
}
```

Response message example:

```
JWE(K_ephemeral, {
  "status": 200,
  "requestId": "6205452b-c555-484f-8445-bb94c8044882"
})
```

If successful, the KMS response to this request MUST have a status of 200. In the case of a request failure, the KMS response status SHOULD be that of an [RFC7231] defined status code with semantics that correspond to the failure condition.

4.6.5. Get GMBC

A client may retrieve GMBC blocks from the KMS using the get GMBC operation. The KMS MAY validate that the requesting client represents an entity that is a current member of the GMBC. Alternatively, a KMS MAY validate that the requesting client represents an entity that has been a member of the GMBC at some point in time.

The request message conforms to the basic request message structure, where the method is "get" and the uri is that of the GMBC's genesis block. The client may also optionally request that only recently appended blocks be returned, by providing in an "antecedent" attribute the hash of a GMBC block the client already has. The KMS will return any and all blocks which were appended after the block indicated by this hash value.

Request payload definition:

```
root {
  request
  ?"antecedent": string
}
```

Request message example:

```
JWE(K_ephemeral, {
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  }
  "method": "get",
  "uri": "kms://kms.example.com/blocks/7f35c3eb",
  "requestId": "db1e4d2a-d483-4fe7-a802-ec5c0d32295f"
})
```

The response message conforms to the basic response message structure, and includes an array containing the JWS compact-serialization of GMBC blocks in chronological order.

Response payload definition:

```
signed-gmbc-block:
        signed-gmbc-genesis-block / signed-gmbc-appended-block

root {
  response,
  "blocks" [ *: signed-gmbc-block ]
}
```

Response message example:

```
JWE(K_ephemeral, {
  "status": 201,
  "requestId": "db1e4d2a-d483-4fe7-a802-ec5c0d32295f",
  "blocks": [
    "eyAiYWxnIjogIlBTMjU2IiB9.ewogICAgICAidXJpIjogImttczovL...",
    "eyAiYWxnIjogIlBTMjU2IiB9.ewogICAgICAiYW50ZWNlZGVudCI6I..."
  ]
})
```

If successful, the KMS response to this request MUST have a status of 200. In the case of a request failure, the KMS response status SHOULD be that of an [RFC7231] defined status code with semantics that correspond to the failure condition.

4.6.6. Post GK (create)

When a client intends to initiate E2E encryption of a communications resource, it obtains the necessary keying material by requesting a new GK from the KMS. The KMS generates the GK as specified in [I-D.abiggs-saag-primitives-for-conf-group-comms].

In the request, the client has the option to include a block attribute representing the hash of a GMBC block to which the GK should be immediately associated, or "bound". This is appropriate in cases where the block to which the GK is to be bound already exists and is known by the client. When this is not the case, a client may omit the block attribute from the request and receive back a GK that has its block attribute similarly omitted. Such a block-less GK is referred to as "unbound" because it is not yet associated with any GMBC block.

The request message conforms to the basic request message structure, where the method is "post", the path part of the URI is "/gks".

Request payload definition:

```
root {
  request,
  "block": string
}
```

Request message example:

```
JWE(K_ephemeral, {
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  }
  "method": "post",
  "uri": "kms://kms.example.com/gks",
  "requestId": "8c198748-36fb-4318-89c9-bfc8bb0a967c"
})
```

The response message conforms to the basic response message structure, and includes a representation of the created GK.

Response payload definition:

```
signed-group-key: JWS(K_kms_priv, group-key)

root {
  response,
  "gk": signed-group-key
}
```

Response message example:

```
JWE(K_ephemeral, {
  "status": 201,
  "requestId": "8c198748-36fb-4318-89c9-bfc8bb0a967c",
  "gk": "eyAiYWxnIjogIlBTMjU2IiB9.ewogICAgICAidXJpIjo..."
}
```

Deserialized payload of the gk attribute:

```
JWS(K_kms_priv, {
  "uri": "kms://kms.example.com/gks/8ed72cd2",
  "creator": "kms://kms.example.com",
  "created": "2015-11-02T19:19:15Z",
  "key": "eyJraWQiOiJmZjNjNWM5Ni0zOTJlLTQ2ZWYtYTg..."
})
```

If successful, the KMS response to this request MUST have a status of 201. In the case of a request failure, the KMS response status SHOULD be that of an [RFC7231] defined status code with semantics that correspond to the failure condition.

4.6.7. Post GK (update)

A GK is often generated ahead of time, before the requesting client knows which communications resource the GK will be used to secure. As such, they are created without being initially associated with any particular GMBC. These are referred to as "unbound" GKs, as discussed in the previous section. An unbound GK is not useful for E2E communications until it is bound to a GMBC block and thereby made accessible to members of that group.

A client can bind an unbound GK to a GMBC block by sending a post request to the KMS with the GKs URI and the hash of the block to which it should be bound. In response to this request the KMS will update the GK payload to include the block hash provided in the request, re-sign the GK with its private key, and return the updated GK to the client.

The request message conforms to the basic request message structure, where the method is "post", and the uri is that of the GK to be updated.

Request payload definition:

```
root {
  request,
  "block": string
}
```

Request message example:

```
JWE(K_ephemeral, {
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  }
  "method": "post",
  "uri": "kms://kms.example.com/gks/8ed72cd2",
  "requestId": "e0f9b55c-d0a5-4f70-aafd-309541fe51ab",
  "block": "14b6290c88a9b40ee519832b878ccc1896bef8900d0f9d2..."
})
```

The response message conforms to the basic response message structure, and includes a representation of the updated GK.

Response payload definition:

```
signed-group-key: JWS(K_kms_priv, group-key)

root {
  response,
  "gk": signed-group-key
}
```

Response message example:

```
JWE(K_ephemeral, {
  "status": 200,
  "requestId": "e0f9b55c-d0a5-4f70-aafd-309541fe51ab",
  "gk": "eyAiYWxnIjogIlBTMjU2IiB9.ewogICAgICAidXJpIjo..."
}
```

Deserialized payload of the gk attribute:

```
JWS(K_kms_priv, {
  "uri": "kms://kms.example.com/gks/8ed72cd2",
  "creator": "kms://kms.example.com",
  "created": "2015-11-02T19:19:15Z",
  "key": "eyJraWQiOiJmZjNjNWM5Ni0zOTJlLTQ2ZWYtYTg...",
  "block": "14b6290c88a9b40ee519832b878ccc1896bef8900d0f9d2..."
})
```

If successful, the KMS response to a create resource request MUST have a status of 200. In the case of a request failure, the KMS

response status SHOULD be that of an [RFC7231] defined status code with semantics that correspond to the failure condition.

4.6.8. Get GK

Recipients of a communications resource secured by a GK require some means by which they can retrieve the GK and subsequently decrypt the resource. Such a recipient will typically receive the URI of the GK as metadata of the encrypted resource itself and submit a get request on that URI to the KMS.

The KMS, as a curator of the GMBC to which the GK is bound, is responsible for ensuring that the keying material contained within the GK is not accessible to entities outside of the group. It does so by testing that the entity whose URI is provided in the request is a member of the GMBC and, if so, returns the GK with the keying material wrapped in a JWE encrypted with the public key of that entity.

The test for membership may be performed based on any one of a variety of policies, some examples of which are given below. Which policy is applied is left to the discretion of the KMS implementation.

Policy 1: An entity is considered a member for purposes of GK retrieval if and only if the entity was a member of the GMBC at the time the block to which the GK is bound was appended to the GMBC.

Policy 2: An entity is considered a member for purposes of GK retrieval if and only if the entity is a member of the GMBC as of the most recently appended block.

Policy 3: An entity is considered a member for purposes of GK retrieval if and only if the entity is a member of the GMBC as of the most recently appended block, and was a member at the time the block to which the GK is bound was appended to the GMBC.

Policy 1 represents a persistent right for current and former group members to retrieve GKs that were available to them at some point in the past. This policy does not extend the right to retrieve a GK to members added subsequently.

Policy 2 determines privilege to retrieve GKs based entirely on the current membership of the GMBC. Former members cannot retrieve and GKs, and current members can retrieve all GKs for the entire history of the GMBC.

Policy 3 allows only a current member to retrieve GKs and then only as far back as the block that introduced that member to the group.

The request message conforms to the basic request message structure, where the method is "get", and the uri is that of the GK to be retrieved.

Request payload definition:

```
root {
  request,
  entity: uri
}
```

Request message example:

```
JWE(K_ephemeral, {
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  }
  "method": "get",
  "uri": "kms://kms.example.com/gks/8ed72cd2",
  "requestId": "d83afbf1-523a-453a-8114-48c7df03ac7c",
  "entity": "bob@example.com"
})
```

The response message conforms to the basic response message structure, and includes a representation of the retrieved GK.

Response payload definition:

```
signed-group-key: JWS(K_kms_priv, group-key)

root {
  response,
  "gk": signed-group-key
}
```

Response message example:

```
JWE(K_ephemeral, {
  "status": 200,
  "requestId": "e0f9b55c-d0a5-4f70-aafd-309541fe51ab",
  "gk": "eyAiYWxnIjogIlBTMjU2IiB9.ewogICAgICAidXJpIjo..."
}
```

Deserialized payload of the gk attribute:

```
JWS(K_kms_priv, {
  "uri": "kms://kms.example.com/gks/8ed72cd2",
  "creator": "kms://kms.example.com",
  "created": "2015-11-02T19:19:15Z",
  "key": "eyJraWQiOiJmZjNjNWM5Ni0zOTJlLTQ2ZWYtYTg...",
  "block": "14b6290c88a9b40ee519832b878ccc1896bef8900d0f9d2..."
})
```

If successful, the KMS response to a retrieve resource request MUST have a status of 200. In the case of a request failure, the KMS response status SHOULD be that of an [RFC7231] defined status code with semantics that correspond to the failure condition.

4.6.9. Ping

Ping is a simple request intended to provide an efficient means for verifying the integrity of the secure channel between client and KMS. Ping MUST be implemented as a safe and idempotent operation that causes the server to do nothing more than return a basic response payload in reaction to the client request. The method of a ping request is "update" and the uri is "/ping".

Request payload definition:

```
root {
  request
}
```

Request message example:

```
JWE(K_ephemeral, {
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  }
  "method": "update",
  "uri": "/ping"
})
```

The response message conforms to the basic response message structure with no additional data.

Response payload definition:

```
root {
  response
}
```

Response message example:

```
JWE(K_ephemeral, {
{
  "status": 200,
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5"
})
```

If successful, the client may deduce that the KMS was able to successfully decrypt the received KMS request message, parse the contents, confirm the identity and authorization of the requesting client, and return a suitable response.

5. Security Considerations

Security considerations are discussed throughout this document. Additional considerations may be added here as needed.

6. Appendix A. Acknowledgments

This specification is the work of several contributors. In particular, the following individuals contributed ideas, feedback, and wording that influenced this specification:

Cullen Jennings, Matt Miller, Suhas Nandakumar, Jonathan Rosenberg

7. Appendix B. Document History

-00

- o Initial draft.

-01

- o Editorial revisions and addition of ping operation.

-02

- o Addition of new key retrieval options.

-03

- o Substantial rewrite based on [I-D.abiggs-saag-primitives-for-conf-group-comms].

8. References

8.1. Normative References

[I-D.abiggs-saag-primitives-for-conf-group-comms]
Biggs, A. and S. Cooley, "Primitives for Confidential Group Communications", draft-abiggs-saag-primitives-for-conf-group-comms-00 (work in progress), September 2015.

[I-D.ietf-jose-json-web-algorithms]
Jones, M., "JSON Web Algorithms (JWA)", draft-ietf-jose-json-web-algorithms-33 (work in progress), September 2014.

[I-D.ietf-jose-json-web-encryption]
Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", draft-ietf-jose-json-web-encryption-33 (work in progress), September 2014.

[I-D.ietf-jose-json-web-key]
Jones, M., "JSON Web Key (JWK)", draft-ietf-jose-json-web-key-33 (work in progress), September 2014.

[I-D.ietf-jose-json-web-signature]
Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", draft-ietf-jose-json-web-signature-33 (work in progress), September 2014.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2822] Resnick, P., "Internet Message Format", RFC 2822, April 2001.

[RFC3339] Klyne, G., Ed. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, July 2002.

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2", RFC 4949, August 2007.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, May 2010.

[RFC6125] Saint-Andre, P. and J. Hodges, "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)", RFC 6125, March 2011.

[RFC6749] Hardt, D., "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012.

[RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014.

[RFC7231] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, June 2014.

8.2. Informative References

[I-D.barnes-pervasive-problem] Barnes, R., Schneier, B., Jennings, C., and T. Hardie, "Pervasive Attack: A Threat Model and Problem Statement", draft-barnes-pervasive-problem-01 (work in progress), July 2014.

[I-D.ietf-xmpp-posh] Miller, M. and P. Saint-Andre, "PKIX over Secure HTTP (POSH)", draft-ietf-xmpp-posh-02 (work in progress), October 2014.

[I-D.newton-json-content-rules] Newton, A., "A Language for Rules Describing JSON Content", draft-newton-json-content-rules-02 (work in progress), August 2014.

[OTR] Borisov, N., Goldberg, I., and E. Brewer, "Off-the-Record Communication, or, Why Not To Use PGP", 2012, <https://otr.cypherpunks.ca/otr-wpes.pdf>.

[RFC4301] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, December 2005.

[RFC4880] Callas, J., Donnerhacke, L., Finney, H., Shaw, D., and R. Thayer, "OpenPGP Message Format", RFC 4880, November 2007.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, September 2009.

Authors' Addresses

Andrew Biggs
Cisco Systems

Email: adb@cisco.com

Shaun Cooley
Cisco Systems

Email: shcooley@cisco.com