```
Network Working Group                                          A. Biggs
Internet-Draft                                                S. Cooley
Intended status: Informational                            Cisco Systems
Expires: February 2, 2019                                August 01, 2018
```

Key Management Service Architecture
draft-abiggs-saag-key-management-service-02

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

publication of this document. Please review these documents
carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

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). In this architecture, a KMS 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 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 service (KMS) server 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", 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        |
   +-----------------+                           +-----------------+
```
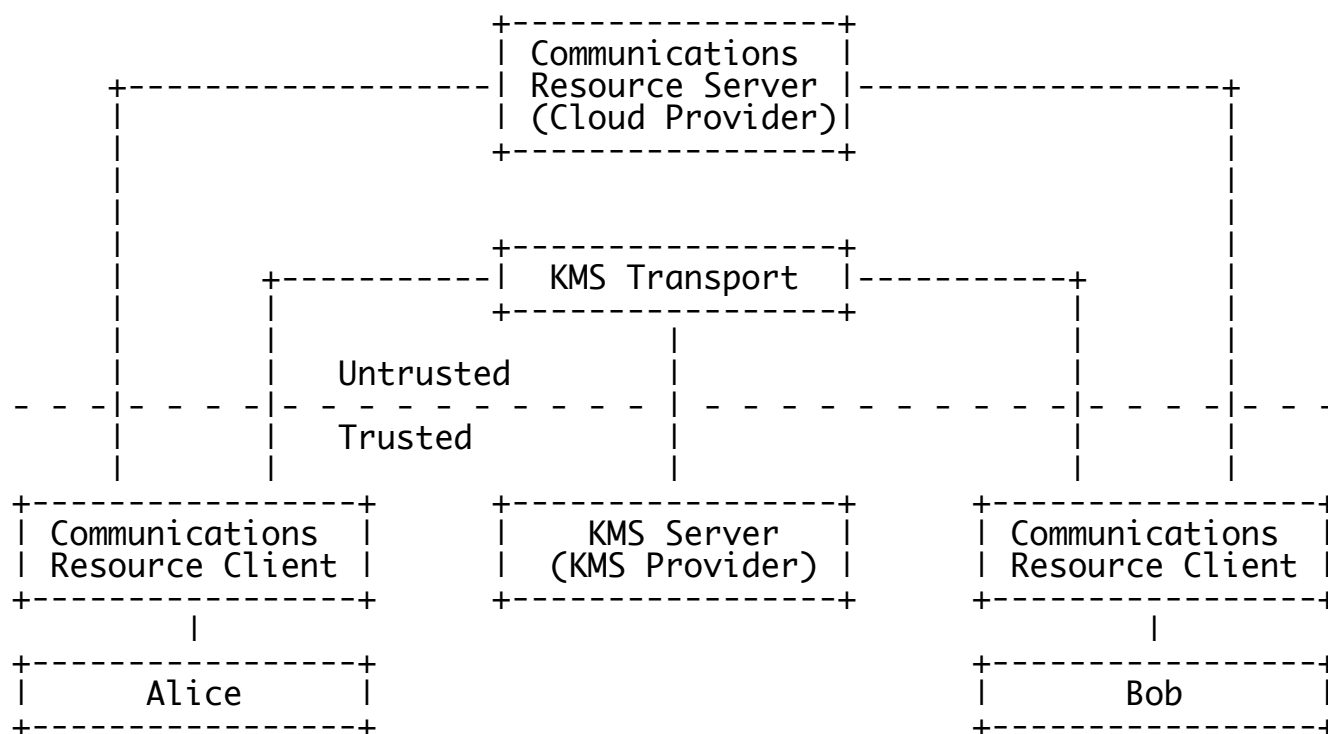
                  Figure 1: KMS Architecture Reference Model

   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 establish and share cryptographic keying material used for
the E2E encryption of content that is transited through the cloud
provider's services. 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 protocol through 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 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)  |                  |                  |                  |
        |                  |                  |                  |
```
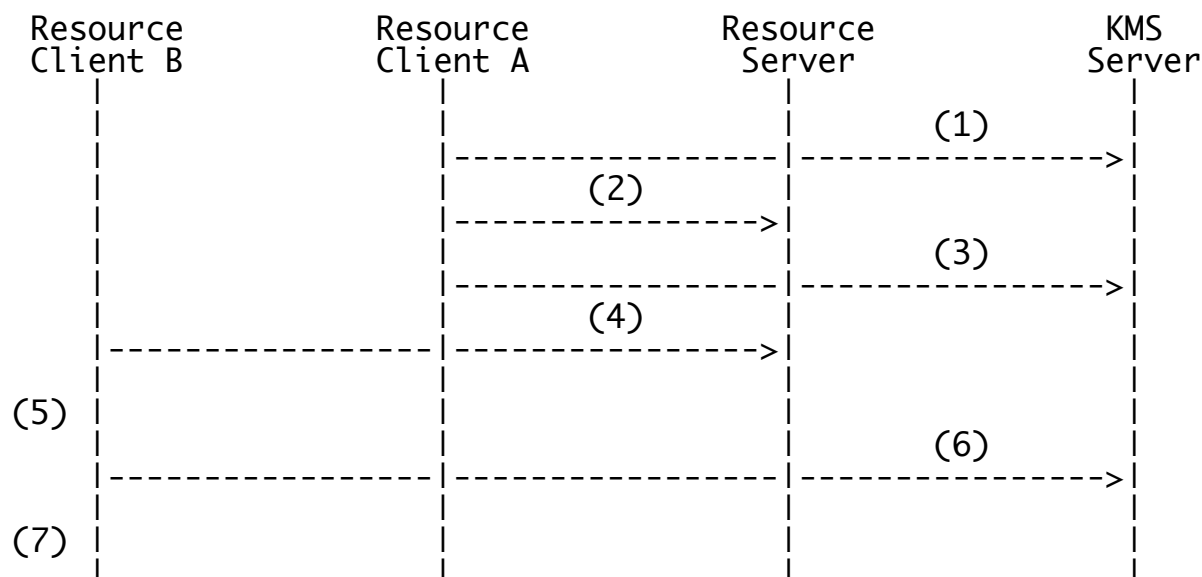
                      Figure 2: Nominal Use Case

   1. Client A requests the generation of a new unbound key from the
       KMS.

   2. Client A encrypts a communications resource using the unbound KMS
       key and shares it via a resource server.

   3. Client A requests the creation of a new KMS resource object (KRO)
       to represent the communications resource. Client A also
       instructs the KMS to bind the KMS key used in step (2) to the new
       KRO and to authorize user B to retrieve keys bound to the KRO.

   4. Client B accesses the communications resource shared by client A
       and receives the encrypted data.

   5. Client B obtains, through some means not defined by this
       specification, the URL of the KMS key used to encrypt the
       communications resource.

   6. Client B requests the KMS key from the KMS server. The KMS
       server, recognizing user B as authorized on the KRO to which the
       key is bound, returns the KMS key.

   7. Client B decrypts the communications resource using the KMS key.

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 service
   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 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)      |      (3)       |              |          |
        |<--------------|<--------------|---------------|          |
        |               |               |               |  (4)     |
        |               |               |---------------|--------------->|
        |               |               |      (5)      |          |
        |               |---------------|-------------->|          |
        |               |               |               |  (6)     |
        |               |---------------|---------------|--------------->|
        |               |               |               |          |
        |          (7)  |               |               |          |
        |               |               |               |          |
   (8)  |               |               |               |          |
        |               |               |               |          |
```
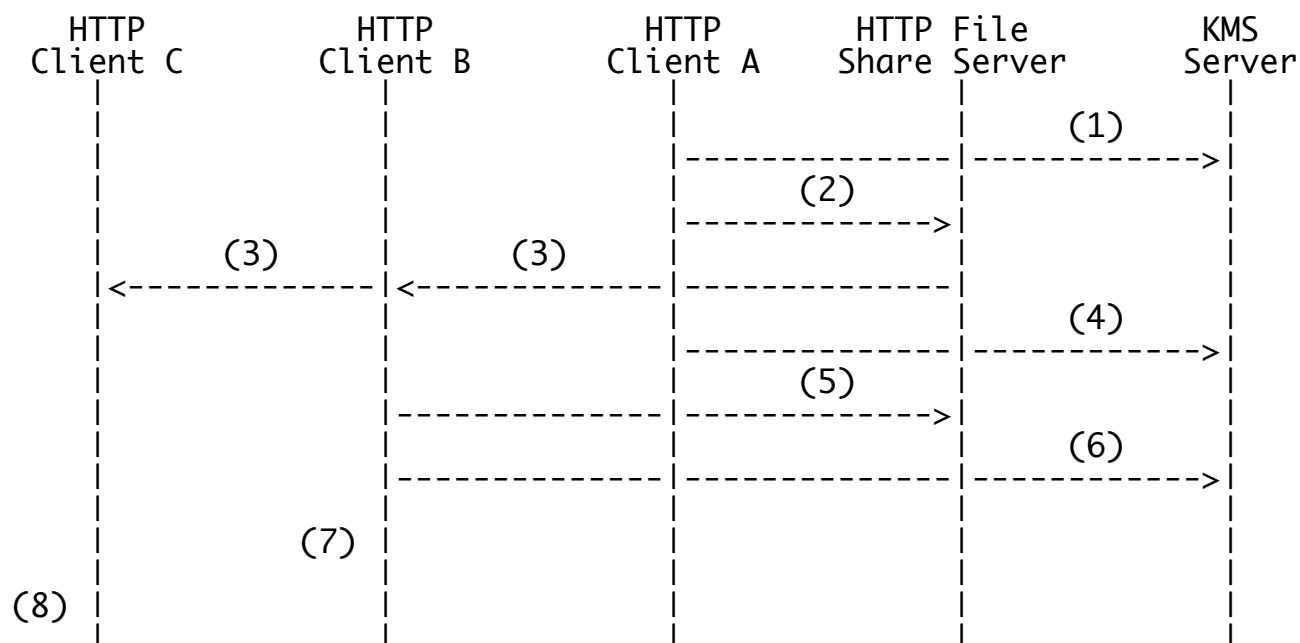
                       Figure 3: File Sharing Use Case

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 from the KMS some number of unbound KMS keys.

2. Client A selects an unbound key from the set of keys obtained
    step (1), encrypts the file to be shared, and posts the encrypted
    content to the file sharing service. The file sharing service
    responds with a URL that uniquely identifies the shared file.

3. 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).

4. Client A requests the creation of a KMS resource object (KRO) on
    the KMS to represent the shared file. In this message the client
    also requests that the key from step (2) be bound to the newly
    created KRO and that the users of clients B and C be authorized
    to retrieve keys bound to the KRO.

5. Client B retrieves the shared file from the file sharing service.

6. Client B requests from the KMS all keys bound to the KRO
    associated with the shared file's URL. Recognizing client B as
    authorized on the KRO, the KMS returns the key bound to the KRO
    by client A in step (4).

7. Client B decrypts the shared file using the key obtained in step
    (6).

8. Client C performs steps (5) through (7) in the same fashion as
    client B.

It is worth noting that a race condition does exist where step (6)
could occur before step (4) completes. This will result in a client
being temporarily denied access to the key 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 [RFC6120] multi-user chat (MUC) 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
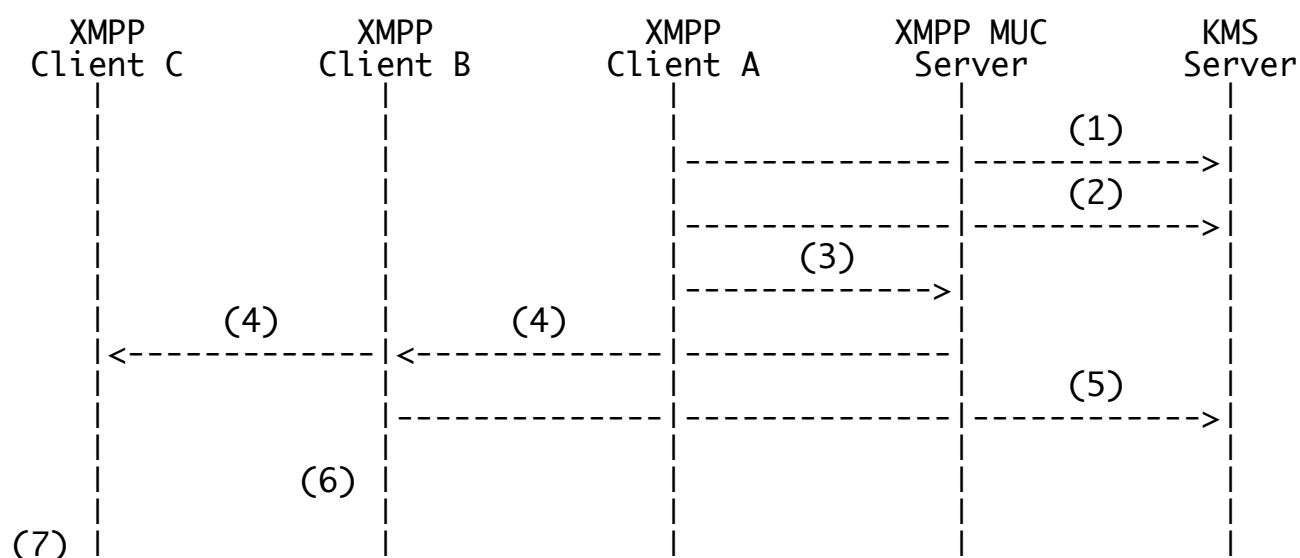the XMPP service itself (not shown) as the KMS transport.

```
        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.

1. Client A requests from the KMS some number of unbound KMS keys. Client A selects one of these keys for encrypting MUC room messages.

2. Client A requests the creation of a KMS resource object (KRO) on the KMS to represent the MUC room. In this message the client also requests that the key selected in step (1) be bound to the newly created KRO and that the users of clients B and C be authorized to retrieve keys bound to the KRO.

3. Client A encrypts a message with the key selected in step (1) and sends it to the MUC room.

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

5. Client B requests from the KMS all keys bound to the KRO associated with the MUC room's URI. Recognizing client B as authorized on the KRO, the KMS returns the key bound to the KRO by client A in step (2).

6. Client B decrypts the shared file using the key selected in step (1).

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)  |              |               |               |           |
       |              |               |               |           |
```
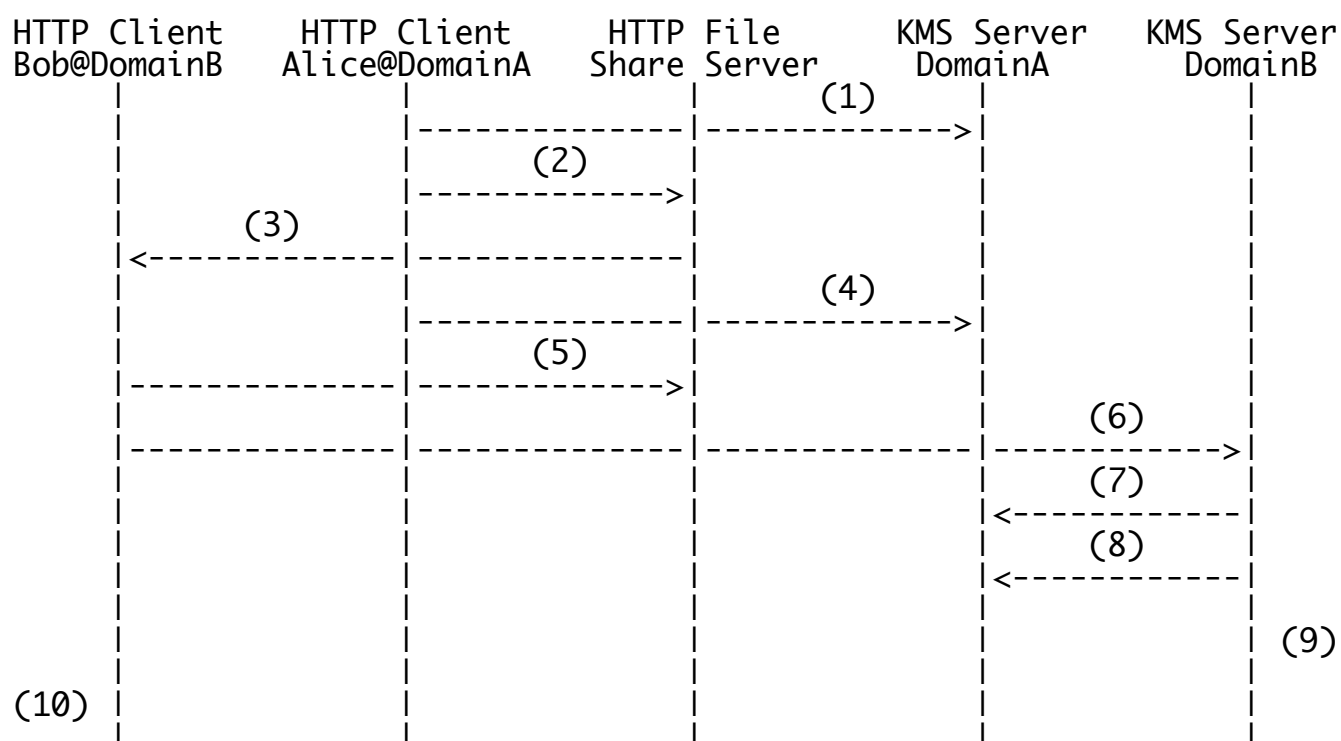
                Figure 5: File Sharing with KMS Federation Use Case

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 from the DomainA KMS some number of
      unbound KMS keys. Each KMS key is uniquely identified by a URL.

2.   Alice@DomainA selects a key from the set of KMS keys obtained in
      step (1), uses that key to encrypt the file to be shared, and
      posts the encrypted content to the file sharing service. The
      file sharing service responds with a URL that uniquely
      identifies the shared file.

3.   Bob@DomainB is notified of the newly shared file URL and
      corresponding KMS key URL through a notification from the file
      sharing service (or potentially some other means, such an an
      email from Alice).

4.   Alice@DomainA requests the creation of a KMS resource object
      (KRO) on the DomainA KMS to represent the shared file. In this
      message Alice also requests that the KMS key from step (2) be

bound to the newly created KRO and that the user Bob@DomainB be authorized to retrieve KMS keys bound to the KRO.

5.   Bob@DomainB retrieves the shared file from the file sharing service.

6.   Using the KMS key URL obtained in step (3), Bob@DomainB requests the KMS key from the DomainB KMS.

7.   The DomainB KMS recognizes the KMS key URL 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.

8.   The DomainB KMS requests from the DomainA KMS the KRO object to which the KMS key is bound, along with all DomainB user authorizations and other KMS keys that have been bound to that KRO. The DomainA KMS recognizes that the DomainB KMS is authorized to retrieve all KMS keys for which users in the DomainB domain have been authorized. It then recognizes that at least one DomainB user (Bob) has been authorized on the KRO created in step (4). The DomainA KMS therefore decides the DomainB KMS is authorized to make this request and returns the requested information.

9.   Using the information received from the DomainA KMS, the DomainB KMS verifies that Bob@DomainB is authorized on the KRO, and satisfies the request from step (6) by returning the KMS key to Bob@DomainB.

10. Client Bob@DomainB decrypts the shared file using the key obtained in step (9).

Note that in step (9) the DomainB KMS is enforcing authorization policy for the KRO hosted on the DomainA KMS as it pertains to DomainB users. This is a necessary consequence of KMS federation, where the act of authorizing access to a KRO 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 generate E2E encryption keys, associate those keys with communications resources,

and explicitly manage access authorizations on those keys. 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
```
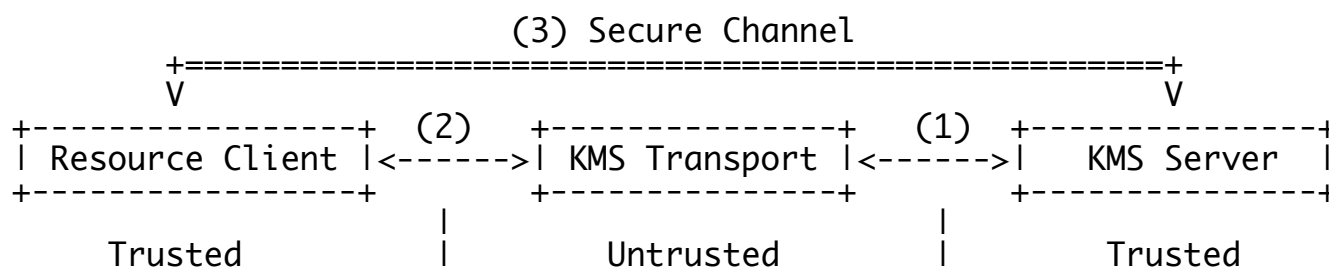
Figure 6: Establishment of a Secure Channel

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 4.7.1.

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
   restrict access to stored 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) jeopardizes the security of all KMS keys
   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 keys 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 4.6.1
   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 defines three object types: resources, keys, and authorizations. It is through the creation and manipulation of instances of these object types that clients interact with the KMS.

Resource

   A resource is an object that represents, within the KMS object model, a communications resource as defined in Section 1.1. Keys and user authorizations are associated (bound) to the resource object as a means of representing their logical association with that communications resource.

Key

   A key is an object representing symmetric keying material generated and made available to authorized clients by the KMS. A key may exist in one of two states: "bound", and "unbound". An unbound key is not associated with any resource, whereas a bound key is associated with exactly one resource.

Authorization

   An authorization is the association of a user with a particular resource. When such an association exists between a user and a resource this implies that the user is entitled to retrieve any key that is bound to that resource, and to add or remove authorizations for other users on the same resource.

The KMS protocol is composed from representations of these fundamental object types. These representations are defined in the following sections.

4.4.1. KMS Key Objects

The JSON representations for KMS key objects is defined as follows using JSON content rules [I-D.newton-json-content-rules].

```
jwk : ; see [JWK]

kmsUri (
    "uri" : uri relative
)

keyRep {
    kmsUri,
    "jwk" : jwk,
    "userId" : string,
    "clientId" : string,
    "createDate" : date-time,
    "expirationDate" : date-time,
    ?"resourceUri" : kmsUri,
    ?"bindDate" : date-time
}

key (
    "key" : keyRep
)

keys (
    "keys" : [ *keyRep ]
)

keyUris (
    "keyUris" : [ *kmsUri ]
)
```

The attributes of a KMS key object are defined as follows.

uri

   A standard definition for KMS object identifiers.

jwk

   Symmetric keying material represented as a JWK object (see
   [I-D.ietf-jose-json-web-key]).

userId

   The authenticated unique identifier of the user that created the
   key.

clientId

An opaque unique identifier provided by the client that created the key.

createDate

The point in time when the key was created, in [RFC3339] date-time format.

expirationDate

The point in time after which the key may no longer be bound (if unbound) or may no longer be used for encrypting data (if bound or an ephemeral key).

resourceUri

The uri of the KMS resource object to which the key is bound.

bindDate

The point in time when the key was bound, in [RFC3339] date-time format.

4.4.2. KMS Authorization Objects

The JSON representations for KMS authorization objects is defined as follows using JSON content rules with references to rules defined in previous sections.

```
authorizationRep {
  kmsUri,
  ?"authId" : string,
  ?"bearer" : string,
  "createDate" : date-time,
  "resourceUri" : kmsUri,
  ?"roleUri" : Uri
}

authorization (
  "authorization" : authorizationRep
)

authorizations (
  "authorizations" : [ *authorizationRep ]
)

authorizationUris (
  "authorizationUris" : [ *kmsUri ]
)
```

The attributes of a KMS authorization object are defined as follows.

uri

> A standard definition for KMS object identifiers.

authId

> A unique identifier of the authorized entity. The exact semantics
> of this attribute are out of scope for this document, however it
> is RECOMMENDED that an implementation regard the value of this
> attribute as mapped to either an individual identity or a grouping
> of identities as recognized by the identity provider employed by
> the KMS. The value of this attribute may also be the URI of a
> KRO, in which case all authorizations on the indicated KRO will be
> regarded by the KMS as also applying to the KRO to which this
> authorization object belongs. Either authId or bearer needs to be
> present, but not both.

bearer

> A KMS generated JWT bearer token for a generated anonymous
> authorization that identifies the issuing KMS, the anonymous
> authorization ID, the KRO to which it applies, and a TTL that
> matches the expiration of the authorization itself. Either authId
> or bearer needs to be present, but not both.

createDate

>   The point in time when the authorization was created, in [RFC3339]
>   date-time format.

resourceUri

>   The object identifier of the resource to which the authorization
>   applies.

roleUri

>   An identifier in the form of Uri to specify the role of the
>   authorization, could be participant, cloudAgent, orgAgent, etc. by
>   default without the roleUri explicitly specified, it is
>   participant role.

Note, with respect to this specification user identifiers are opaque,
however they MUST map to unique identifiers provided as part of user
authentication.

4.4.3. KMS Resource Objects (KRO)

The JSON representation for KMS resource objects is defined as
follows using JSON content rules with references to rules defined in
previous sections.

```
resourceRep {
  kmsUri,
  keys / keyUris,
  authorizations / authorizationUris,
  ttl
}

resource (
  "resource" : resourceRep
)

resources (
  "resources" : [ *resourceRep ]
)

resourceUris (
  "resourceUris" : [ *kmsUri ]
)
```

The attributes of a KMS resource object are defined as follows.

uri

   A standard definition for KMS object identifiers.

keys

   An array of key object representations, one for each key bound to
   the resource.

keyUris

   An array of key object identifiers, one for each key bound to the
   resource. Only one of either keys or keyUris may be present in a
   resource object representation.

authorizations

   An array of authorization object representations, one for each
   authorization on the resource.

authorizationUris

   An array of authorization object identifiers, one for each
   authorization on the resource. Only one of either authorizations
   or authorizationUris may be present in a resource object
   representation.

4.5. Request Types

   The KMS protocol defines four types of requests: create, retrieve,
   update, delete, each of which may be applied to one of the three KMS
   object types. Note that not all object types support all requests
   types. A KMS need only support those combinations of request type
   and object type explicitly defined in this document.

Create

   A create operation acts upon an object type, creating one or more
   new instances of that object type.

Retrieve

   A retrieve operation acts upon an object or object type, returning
   in the response a representation of one or more object instances.

Update

An update operation acts upon an object, altering mutable
properties of that object.

Delete

A delete operation acts upon an object, removing that object from
the KMS.

4.6. 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 4.7.1). 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 4.7.1). 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 4.6.3, 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.

4.6.1. Basic Request 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.

```
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,
  kmsUri,
  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

> The KMS object or object type 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.

4.6.2. 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.

4.6.3. 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"
})
```

4.7. 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.

4.7.1. 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] object 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 a KMS key 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",
  "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.

4.7.2. 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.

4.7.3. Create Resource

When a client intends to initiate E2E encryption of a communications resource, it begins by requesting the creation of a KMS resource object. This resource object logically represents the communications resource within the KMS data model.

As part of a create resource request, a KMS server MUST create at least one authorization object on the newly created resource object to explicitly authorize the user making the request. A client MAY request the immediate creation of one or more additional authorizations such that corresponding users may be immediately authorized to access and operate on the new resource object. Also, a client MAY request number of anonymous authorizations to be created on the new resource object, specified by anonymous attribute in the request. If for any reason one or more requested authorizations cannot be applied to the new resource object, the entire create resource request MUST be failed by the KMS.

As part of a create resource request, a client MAY request the immediate binding of one or more unbound KMS keys to the new resource object. If any key indicated in the request is already bound, or is otherwise invalid (e.g. expired), the entire create resource request MUST be failed by the KMS.

Also, the client MAY request a time to live for the new resource object. The TTL is in seconds with a default of 0 (never expires). Implementations MAY enforce minimum and/or maximum values for TTL. If a requested TTL value is not within the server's acceptable range, it SHOULD respond with a 400 and response message indicating the server's acceptable range.

The request message conforms to the basic request message structure, where the method is "create", the uri is "/resources", and additional

user identifiers and/or key URIs are provided in a manner consistent
with the following.

Request payload definition:

```
authIds (
  "authIds" : [ *string ]
)

root {
  request,
  ?authIds,
  ?anonymous,
  ?keyUris,
  ?ttl
}
```

Request message example:

```
JWE(K_ephemeral, {
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  }
  "method": "create",
  "uri": "/resources",
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
  "authIds": [
    "b46e8124-b6e8-47e0-af0d-e7f1a2072dac",
    "39d56a84-c6f9-459e-9fd1-40ab4ad3e89a"
  ],
  "keyUris": [
    "/keys/b4cba4da-a984-4af2-b54f-3ca04acfe461",
    "/keys/2671413c-ab80-4f19-a0a4-ae07e1a94e90"
  ],
  "ttl": 604800
})
```

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

Response payload definition:

```
root {
  response,
  resource
}
```

Response message example:

```
JWE(K_ephemeral, {
  "status": 201,
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
  "resource": {
      "uri": "/resources/7f35c3eb-95d6-4558-a7fc-1942e5f03094",
      "authorizationUris": [
        "/authorizations/50e9056d-0700-4919-b55f-84cd78a2a65e",
        "/authorizations/db4c95ab-3fbf-42a8-989f-f53c1f13cc9a"
      ],
      "keyUris": [
        "/keys/b4cba4da-a984-4af2-b54f-3ca04acfe461",
        "/keys/2671413c-ab80-4f19-a0a4-ae07e1a94e90"
      ],
      "expirationDate": "2014-10-09T16:54:48Z"
  }
})
```

If successful, the KMS response to a create resource 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.

4.7.4. Retrieve Resource

A client that is authorized on a given KMS resource object may retrieve the current state of that object as well as that of current set of KMS authorization objects and bound KMS keys.

The request message conforms to the basic request message structure, where the method is "retrieve", and the uri is that of the KMS resource object as returned by the create operation from which it originated.

Request payload definition:

```
root {
  request
}
```

Request message example:

```
JWE(K_ephemeral, {
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  }
  "method": "retrieve",
  "uri": "/resources/7f35c3eb-95d6-4558-a7fc-1942e5f03094",
  "requestId": "db1e4d2a-d483-4fe7-a802-ec5c0d32295f",
})
```

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

Response payload definition:

```
root {
  response,
  resource
}
```

Response message example:

```
JWE(K_ephemeral, {
  "status": 200,
  "requestId": "db1e4d2a-d483-4fe7-a802-ec5c0d32295f",
  "resource": {
    "uri": "/resources/7f35c3eb-95d6-4558-a7fc-1942e5f03094",
    "authorizationUris": [
      "/authorizations/50e9056d-0700-4919-b55f-84cd78a2a65e",
      "/authorizations/db4c95ab-3fbf-42a8-989f-f53c1f13cc9a"
    ],
    "keyUris": [
      "/keys/b4cba4da-a984-4af2-b54f-3ca04acfe461",
      "/keys/2671413c-ab80-4f19-a0a4-ae07e1a94e90"
    ]
  }
})
```

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.

4.7.5. Create Keys

   When a client requires a symmetric key for use in the E2E encryption
   of a communications resource, it begins by requesting the creation of
   one or more keys from the KMS. The initial state of a newly created
   key is "unbound" in the sense that it does not yet belong to a
   particular resource. A client may submit this request at any time,
   even before the communications resource exists. The keys returned by
   this request are unbound, which is to say not yet associated with any
   KMS resource object.

   The request message conforms to the basic request message structure,
   where the method is "create", the uri is "/keys", and an additional
   count attribute is introduced to indicate the number of keys to be
   created.

   Request payload definition:

   root {
     request,
     "count": integer
   }

   Request message example:

   JWE(K_ephemeral, {
     "client": {
       "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
       "credential": {
         "bearer": "ZWU5NGE2YWYtMGE2NC0..."
       }
     }
     "method": "create",
     "uri": "/keys",
     "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
     "count": 2
   })

   The response message conforms to the basic response message structure
   with the addition of an array of key object representations, one for
   each unbound key created.

   Response payload definition:

   root {
     response,
     keys / keyUris
   }

Response message example:

```
JWE(K_ephemeral, {
  "status": 201,
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
  "keys": [
    {
      "uri": "/keys/52100fa4-c222-46d0-994d-1ca885e4a3a2",
      "jwk": {
        "kid": "52100fa4-c222-46d0-994d-1ca885e4a3a2",
        "kty": "oct",
        "k": "ZMpktzGq1g6_r4fKVdnx9OaYr4HjxPjIs7l7SwAsgsg"
      }
      "userId": "842e2d82-7e71-4040-8eb9-d977fe888807",
      "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
      "createDate": "2014-10-09T15:54:48Z",
      "expirationDate": "2014-10-09T16:04:48Z"
    },
    {
      "uri": "/keys/fed33890-f9fa-43ad-a9f8-ab55a983a543",
      "jwk": {
        "kid": "fed33890-f9fa-43ad-a9f8-ab55a983a543",
        "kty": "oct",
        "k": "q2znCXQpbBPSZBUddZvchRSH5pSSKPEHlgb3CSGIdpL"
      }
      "userId": "842e2d82-7e71-4040-8eb9-d977fe888807",
      "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
      "createDate": "2014-10-09T15:54:48Z",
      "expirationDate": "2014-10-09T16:04:48Z"
    }
  ]
})
```

Each key object in the response to a create unbound keys request
includes a single JWK [I-D.ietf-jose-json-web-key] representing a new
symmetric key of 256 bits generated by a cryptographically secure
PRNG. Note that, as unbound keys, the resourceUri attribute of each
key is either undefined or null. All keys 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 bound to a resource
(both in [RFC3339] date-time format).

The clientId attribute of each created key MUST be the clientId
provided by the client in the client.clientId attribute of the
request.

As shown in the response payload definition, the KMS MUST return
either an array of key object representations or an array of key

object uris. It is at the KMS server's discretion which of these is
returned.

If successful, the KMS response to a create unbound keys 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.

4.7.6. Update Key (Bind)

To initiate the use of an unbound KMS key in securing a
communications resource, a client will create a corresponding KMS
resource object and subsequently bind the unbound key to that
resource. A client MAY begin using an unbound KMS key to encrypt a
communications resource prior to the binding of that key.

The request message conforms to the basic request message structure,
where the method is "update", the uri is that of the key to be bound,
and an additional resourceUri attribute is introduced to indicate the
KMS resource object to which the key is to be bound. If the user
making a bind unbound key request does not have an authorization on
the resource indicated by the resourceUri, or is not the user for
whom the unbound key was originally created, the KMS MUST fail the
request. The KMS SHOULD fail the request if the clientId of the
request does not match that of the unbound key.

Request payload definition:

```
root {
  request,
  "resourceUri" : kmsUri
}
```

Request message example:

```
JWE(K_ephemeral, {
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  }
  "method": "update",
  "uri": "/keys/52100fa4-c222-46d0-994d-1ca885e4a3a2",
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
  "resourceUri": "/resources/7f35c3eb-95d6-4558-a7fc-1942e5f03094"
})
```

The response message conforms to the basic response message
structure, and includes a representation of the full state of the
newly bound key.

Response payload definition:

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

Response message example:

```
JWE(K_ephemeral, {
{
   "status": 200,
   "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
   "key": {
     "uri": "/keys/52100fa4-c222-46d0-994d-1ca885e4a3a2",
     "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
     "jwk": {
       "kid": "52100fa4-c222-46d0-994d-1ca885e4a3a2",
       "kty": "oct",
       "k": "ZMpktzGq1g6_r4fKVdnx90aYr4HjxPjIs7l7SwAsgsg"
     }
     "userId": "842e2d82-7e71-4040-8eb9-d977fe888807",
     "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
     "createDate": "2014-10-09T15:54:48Z",
     "bindDate": "2014-10-09T15:55:34Z",
     "expirationDate": "2014-10-10T15:55:34Z",
     "resourceUri": "/resources/7f35c3eb-95d6-4558-a7fc-1942e5f03094"
   }
})
```

On successfully binding a formerly unbound KMS key to a resource
object, the state of the KMS key object MUST reflect the updated
resourceUri attribute, MUST reflect a bindDate as the current time,
and MUST reflect an expirationDate as the time after which clients
MUST NOT use this key for encryption as provided by KMS policy.
Subsequently, the KMS MUST regard the key as bound to the KMS
resource object identified by the resourceUri and MUST reject
subsequent requests to bind the same key to any other resource
object.

If successful, the KMS response to a bind unbound key 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.

4.7.7. Retrieve Keys

   Clients engaging in E2E encryption require a means for retrieving
   keys from the KMS. A key request may take one of three forms, it may
   be a request for

   o a specific key,

   o all keys bound to a particular resource,

   o a subset of keys bound to a particular resource.

   In all cases, the request message conforms to the basic request
   message structure with "retrieve" as the value for the method
   attribute.

   To retrieve an individual key, the uri of the request is that of the
   key object to be retrieved. If the key is unbound, the KMS MUST
   reject the request unless it originates from the user that requested
   the key's creation, and SHOULD reject the request unless it
   originates from the same client that requested the key's creation (as
   determined by clientId). If the key is bound, the KMS MUST reject
   the request if the request originates from a user for which there
   does not exist a corresponding authorization on the resource to which
   the requested key is bound.

   To retrieve all keys bound to a resource, the uri of the request is
   that of the resource concatenated with "/keys". The KMS MUST reject
   the request if the request originates from a user for which there
   does not exist a corresponding authorization on the resource.

   To retrieve a subset of keys bound to a resource, the client submits
   a request in the same fashion as for requesting all keys but also
   includes one or more additional attributes indicating selection
   criteria. These additional attributes include the following:

   o boundAfter

   o boundBefore

   Each of these parameters is optional and clients MAY employ them in
   any combination.

   If the request includes a "boundAfter" attribute, the value MUST
   conform to the [RFC3339] date-time format and the KMS response MUST
   NOT include any key with a "bindDate" that chronologically precedes
   it. Similarly, if the request includes a "boundBefore" attribute,
   the value MUST conform to the [RFC3339] date-time format and the KMS

response MUST NOT include any key with a "bindDate" that is either
equal to or chronologically subsequent to it.

To limit the number of keys returned in a KMS response, a client MAY
include a "count" attribute. If the request includes a "count"
attribute, it must be of JSON type integer and the cardinality of the
set of keys returned in the KMS response MUST NOT exceed its value.
In the event it becomes necessary for the KMS to truncate the set of
keys included in the KMS response, due to the limitations imposed by
the "count" attribute, the truncated subset MUST be composed of those
keys which satisfy the request's other selection criteria (if any)
and also constitute a subset of that selection for which the elements
possess the chronologically largest "bindDate" values.

Request payload definition:

```
root {
  request,
  ?"boundAfter" : date-time,
  ?"boundBefore" : date-time,
  ?"count" : integer
}
```

Request message example (individual key):

```
JWE(K_ephemeral, {
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  }
  "method": "retrieve",
  "uri": "/keys/52100fa4-c222-46d0-994d-1ca885e4a3a2",
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5"
})
```

Request message example (10 most recently bound keys):

```
JWE(K_ephemeral, {
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  }
  "method": "retrieve",
  "uri": "/resources/7f35c3eb-95d6-4558-a7fc-1942e5f03094/keys",
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
  "count": 10
})
```

Request message example (keys bound after a given time, 25 max):

```
JWE(K_ephemeral, {
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  }
  "method": "retrieve",
  "uri": "/resources/7f35c3eb-95d6-4558-a7fc-1942e5f03094/keys",
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
  "boundAfter": "2015-01-11T18:23:21Z",
  "count": 25
})
```

The response message conforms to the basic response message structure
and includes a representation of the key or keys selected by the
request. When responding to a request for a specific key, the KMS
will return a response that includes a KMS key object representation
as described in Section 4.4.1. When responding to a request for
multiple keys, the KMS will return a response that includes an array
of KMS key object representations.

Response payload definition:

```
root {
  response,
  key / keys
}
```

Response message example (for specific key):

```
   JWE(K_ephemeral, {
   {
     "status": 200,
     "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
     "key": {
       "uri": "/keys/52100fa4-c222-46d0-994d-1ca885e4a3a2",
       "jwk": {
         "kid": "52100fa4-c222-46d0-994d-1ca885e4a3a2",
         "kty": "oct",
         "k": "ZMpktzGq1g6_r4fKVdnx9OaYr4HjxPjIs7l7SwAsgsg"
       }
       "userId": "842e2d82-7e71-4040-8eb9-d977fe888807",
       "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
       "createDate": "2014-10-09T15:54:48Z",
       "bindDate": "2014-10-09T15:55:34Z",
       "expirationDate": "2014-10-10T15:55:34Z",
       "resourceUri": "/resources/7f35c3eb-95d6-4558-a7fc-1942e5f03094"
     }
   })
```

   Response message example (for keys bound to a specific resource):

```
    JWE(K_ephemeral, {
    {
      "status": 200,
      "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
      "keys": [
      {
        "uri": "/keys/52100fa4-c222-46d0-994d-1ca885e4a3a2",
        "jwk": {
          "kid": "52100fa4-c222-46d0-994d-1ca885e4a3a2",
          "kty": "oct",
          "k": "ZMpktzGq1g6_r4fKVdnx9OaYr4HjxPjIs7l7SwAsgsg"
        }
        "userId": "842e2d82-7e71-4040-8eb9-d977fe888807",
        "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
        "createDate": "2014-10-09T15:54:48Z",
        "bindDate": "2014-10-09T15:55:34Z",
        "expirationDate": "2014-10-10T15:55:34Z",
        "resourceUri": "/resources/7f35c3eb-95d6-4558-a7fc-1942e5f03094"
      },
      {
        "uri": "/keys/fed33890-f9fa-43ad-a9f8-ab55a983a543",
        "jwk": {
          "kid": "fed33890-f9fa-43ad-a9f8-ab55a983a543",
          "kty": "oct",
          "k": "q2znCXQpbBPSZBUddZvchRSH5pSSKPEHlgb3CSGIdpL"
        }
        "userId": "842e2d82-7e71-4040-8eb9-d977fe888807",
        "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
        "createDate": "2014-10-09T15:54:48Z",
        "bindDate": "2014-10-09T15:56:43Z",
        "expirationDate": "2014-10-10T15:56:43Z",
        "resourceUri": "/resources/7f35c3eb-95d6-4558-a7fc-1942e5f03094"
      }]
    })
```

   If successful, the KMS response to a retrieve bound keys 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.

4.7.8. Create Authorizations

   An authorization establishes a relationship between a resource and a
   user that entitles the user to retrieve bound keys from, and bind new
   keys to, that resource. The KMS resource authorization model is
   viral in the sense that, once a user has been authorized on a
   resource, that user is also entitled to authorize other users on that

resource. These authorizations are created through create
authorization requests.

The request message conforms to the basic request message structure,
where the method is "create", and the uri is "/authorizations".
Additional attributes are required to indicate the resource on which
authorizations are to be added, as well as the set of users for whom
these new authorizations are to be created, also the number of
anonymous authorization to be created.

```
root {
  request,
  "resourceUri" : kmsUri,
  "authIds" : [ *string ],
  "anonymous" : int
}
```

Request message example:

```
JWE(K_ephemeral, {
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  }
  "method": "create",
  "uri": "/authorizations",
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
  "resourceUri": "/resources/7f35c3eb-95d6-4558-a7fc-1942e5f03094",
  "authIds": [
    "119a0582-2e2b-4c0c-ba6a-753d05171803",
    "557ac05d-5751-43b4-a04b-e7eb1499ee0a"
  ],
  "anonymous": 1
})
```

The response message conforms to the basic response message
structure, and includes a representation of the set of KMS
authorization objects created by the request.

Response payload definition:

```
root {
  response,
  authorizations
}
```

Response message example:

```
JWE(K_ephemeral, {
{
  "status": 201,
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
  "authorizations": [
  {
    "uri": "/authorizations/79a39ed9-a8e5-4d1f-9ae2-e27857fc5901",
    "authId": "119a0582-2e2b-4c0c-ba6a-753d05171803",
    "resourceUri": "/resources/7f35c3eb-95d6-4558-a7fc-1942e5f03094"
  },
  {
    "uri": "/authorizations/5aaca3eb-ca4c-47c9-b8e2-b20f47568b7b",
    "authId": "557ac05d-5751-43b4-a04b-e7eb1499ee0a",
    "resourceUri": "/resources/7f35c3eb-95d6-4558-a7fc-1942e5f03094"
  },
  {
    "uri": "/authorizations/5aaca3eb-ca4c-47c9-b8e2-b20f47568b7b",
    "bearer": "EYHHBGGE2YWYtMGE2NC0...",
    "resourceUri": "/resources/7f35c3eb-95d6-4558-a7fc-1942e5f03094"
  }]
})
```

If successful, the KMS response to a create authorizations 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. If for any
reason one or more requested authorizations cannot be created or
applied to the resource object, the entire create authorizations
request MUST be failed by the KMS.

4.7.9. Retrieve Authorizations

A client may explicitly request the set of all authorizations on a
given KMS resource object. The uri of the request is that of the
resource concatenated with "/authorizations". The KMS MUST reject
the request if the request originates from a user for which there
does not exist a corresponding authorization on the resource.

Request payload definition:

```
root {
  request
}
```

Request message example:

```
JWE(K_ephemeral, {
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  }
  "method": "retrieve",
  "uri": "/resources/7f35c3eb-95d6-4558-1942e5f03094/authorizations",
  "requestId": "913d7ae3-8945-46ca-8ed1-2b287c1370ce"
})
```

The response message conforms to the basic response message structure and includes an array of KMS authorization object representations.

Response payload definition:

```
root {
  response,
  authorizations
}
```

Response message example:

```
JWE(K_ephemeral, {
{
  "status": 200,
  "requestId": "913d7ae3-8945-46ca-8ed1-2b287c1370ce",
  "authorizations": [
  {
    "uri": "/authorizations/79a39ed9-a8e5-4d1f-9ae2-e27857fc5901",
    "authId": "119a0582-2e2b-4c0c-ba6a-753d05171803",
    "resourceUri": "/resources/7f35c3eb-95d6-4558-1942e5f03094"
  },
  {
    "uri": "/authorizations/5aaca3eb-ca4c-47c9-b8e2-b20f47568b7b",
    "authId": "557ac05d-5751-43b4-a04b-e7eb1499ee0a",
    "resourceUri": "/resources/7f35c3eb-95d6-4558-1942e5f03094"
  }]
})
```

If successful, the KMS response to a retrieve bound keys 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.

4.7.10. Retrieve Authorizations By User

   To query an authorization for the specific user from a KMS resource
   object, any user currently authorized on the same resource object may
   issue retrieve authorization request for the specific user. The
   request message conforms to the basic request message structure,
   where the method is "retrieve", and the URI is "/authorizations"
   followed by authorization id.

   Request payload definition:

   root {
     request
   }

   Request message example:

```
JWE(K_ephemeral, {
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  }
  "method": "retrieve",
  "uri": "/resources/a8c3c42f-5348-4289-83f0-d833d31ba93c/authorizations?authId=
119a0582-2e2b-4c0c-ba6a-753d05171803",
  "requestId": "d4a741cd-0e16-4ff8-9a61-4a4b841d9e93"
})
```

   If successful, the KMS response to a retrieve authorization for the
   specific user request MUST have a status of 200. The response
   message conforms to the basic response message structure and includes
   an array of KMS authorization object representations. The array will
   be empty if the specified user is not authorized for the KMS resource
   object. The array will contain one single authorization entry for
   the specified user if that user is authorized for the KMS resource
   object. 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.

   Response payload definition:

   root {
     response,
     authorizations
   }

   Response message example:

```
JWE(K_ephemeral, {
{
  "status": 200,
  "requestId": "d4a741cd-0e16-4ff8-9a61-4a4b841d9e93",
  "authorizations": [
  {
    "uri": "/authorizations/ef0a5291-49cb-413f-832b-b437bc542825",
    "authId": "119a0582-2e2b-4c0c-ba6a-753d05171803",
    "resourceUri": "/resources/a8c3c42f-5348-4289-83f0-d833d31ba93c"
  }]
})
```

4.7.11. Delete Authorization

To remove an authorization from a KMS resource object, any user
currently authorized on the same resource object may issue a delete
authorization request. The request message conforms to the basic
request message structure, where the method is "delete", and the URI
is either that of the authorization object to be deleted, or the URI
of the collection of authorizations within a particular KMS resource
object appended with an authId query parameter whose value matches
that of the authorization object to be deleted.

Request payload definition:

```
root {
  request
}
```

Request message example, explicitly identifying the URI of the
authorization object to be deleted:

```
JWE(K_ephemeral, {
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  }
  "method": "delete",
  "uri": "/authorizations/5aaca3eb-ca4c-47c9-b8e2-b20f47568b7b"
})
```

Request message example, implicitly identifying the authorization
object to be deleted by the KMS resource object to which it belongs
plus the value of its authId attribute:

```
JWE(K_ephemeral, {
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
  "client": {
    "clientId": "android_a6aa012a-0795-4fb4-bddb-f04abda9e34f",
    "credential": {
      "bearer": "ZWU5NGE2YWYtMGE2NC0..."
    }
  }
  "method": "delete",
  "uri": "/resources/7f35c3eb-95d6-4558-1942e5f03094
          /authorizations?authId=557ac05d-5751-43b4-a04b-e7eb1499ee0a",
})
```

Note, in the example above, the URI attribute value is a continuous
string of non-whitespace characters. Whitespace has been added here
for readability.

The response message conforms to the basic response message
structure, and includes a representation of the authorization object
that was deleted.

Response payload definition:

```
root {
  response,
  ?authorization
}
```

Response message example:

```
JWE(K_ephemeral, {
{
  "status": 200,
  "requestId": "10992782-e096-4fd3-9458-24dca7a92fa5",
  "authorization": {
    "uri": "/authorizations/5aaca3eb-ca4c-47c9-b8e2-b20f47568b7b",
    "authId": "557ac05d-5751-43b4-a04b-e7eb1499ee0a",
    "resourceUri": "/resources/7f35c3eb-95d6-4558-a7fc-1942e5f03094"
  }
})
```

If successful, the KMS response to a delete authorization request
MUST have a status of 200 if the authorization object representation
is included, and 204 if not. 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.

4.7.12. 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. Mandatory-to-Implement

   Implementations MUST support the following JWK key types from
   [I-D.ietf-jose-json-web-algorithms]:

   o "RSA" for the KMS static public/private key

   o "EC" for the Ephemeral Diffie Hellman exchange

   o "oct" for all symmetric keys

   Implementations MUST support "PS256" (RSASSA-PSS using SHA-256 and
   MGF1 with SHA-256) from [I-D.ietf-jose-json-web-algorithms] for
   signatures using the KMS static public/private key for Section 4.7.1.

   Implementations MUST support JWK Elliptic Curve type "P-256" (NIST
   P-256 curve) from [I-D.ietf-jose-json-web-algorithms] for
   Section 4.7.1.

   Implementations MUST support "RSA-OAEP" (RSAES OAEP using default
   parameters) from [I-D.ietf-jose-json-web-algorithms] for key
   encryption using the KMS static public/private key for Section 4.7.1.

   Implementations MUST support "dir" (Direct Key Agreement Key
   Management Mode) from [I-D.ietf-jose-json-web-algorithms] for all
   operations other than Section 4.7.1.

   Implementations MUST support "A256GCM" (AES GCM using 256 bit key)
   from [I-D.ietf-jose-json-web-algorithms] for content encryption for
   all operations other than Section 4.7.1.

6. Security Considerations

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

7. 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,
   Bo Zou

8. Appendix B. Document History

    -00

    o Initial draft.

9. References

9.1. Normative References

    [I-D.ietf-jose-json-web-algorithms]
              Jones, M., "JSON Web Algorithms (JWA)", draft-ietf-jose-
              json-web-algorithms-40 (work in progress), January 2015.

    [I-D.ietf-jose-json-web-encryption]
              Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)",
              draft-ietf-jose-json-web-encryption-40 (work in progress),
              January 2015.

    [I-D.ietf-jose-json-web-key]
              Jones, M., "JSON Web Key (JWK)", draft-ietf-jose-json-web-
              key-41 (work in progress), January 2015.

    [I-D.ietf-jose-json-web-signature]
              Jones, M., Bradley, J., and N. Sakimura, "JSON Web
              Signature (JWS)", draft-ietf-jose-json-web-signature-41
              (work in progress), January 2015.

    [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119,
              DOI 10.17487/RFC2119, March 1997,
              <https://www.rfc-editor.org/info/rfc2119>.

    [RFC2822] Resnick, P., Ed., "Internet Message Format", RFC 2822,
              DOI 10.17487/RFC2822, April 2001,
              <https://www.rfc-editor.org/info/rfc2822>.

    [RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet:
              Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002,
              <https://www.rfc-editor.org/info/rfc3339>.

    [RFC4949] Shirey, R., "Internet Security Glossary, Version 2",
              FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007,
              <https://www.rfc-editor.org/info/rfc4949>.

   [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, DOI 10.17487/RFC5280, May 2008,
              <https://www.rfc-editor.org/info/rfc5280>.

   [RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand
              Key Derivation Function (HKDF)", RFC 5869,
              DOI 10.17487/RFC5869, May 2010,
              <https://www.rfc-editor.org/info/rfc5869>.

   [RFC6120] Saint-Andre, P., "Extensible Messaging and Presence
              Protocol (XMPP): Core", RFC 6120, DOI 10.17487/RFC6120,
              March 2011, <https://www.rfc-editor.org/info/rfc6120>.

   [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, DOI 10.17487/RFC6125, March
              2011, <https://www.rfc-editor.org/info/rfc6125>.

   [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework",
              RFC 6749, DOI 10.17487/RFC6749, October 2012,
              <https://www.rfc-editor.org/info/rfc6749>.

   [RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data
              Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March
              2014, <https://www.rfc-editor.org/info/rfc7159>.

   [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer
              Protocol (HTTP/1.1): Semantics and Content", RFC 7231,
              DOI 10.17487/RFC7231, June 2014,
              <https://www.rfc-editor.org/info/rfc7231>.

9.2. 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-06 (work in progress),
              September 2015.

[I-D.newton-json-content-rules]
          Newton, A. and P. Cordell, "A Language for Rules
          Describing JSON Content", draft-newton-json-content-
          rules-09 (work in progress), September 2017.

[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, DOI 10.17487/RFC4301,
          December 2005, <https://www.rfc-editor.org/info/rfc4301>.

[RFC4880] Callas, J., Donnerhacke, L., Finney, H., Shaw, D., and R.
          Thayer, "OpenPGP Message Format", RFC 4880,
          DOI 10.17487/RFC4880, November 2007,
          <https://www.rfc-editor.org/info/rfc4880>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security
          (TLS) Protocol Version 1.2", RFC 5246,
          DOI 10.17487/RFC5246, August 2008,
          <https://www.rfc-editor.org/info/rfc5246>.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70,
          RFC 5652, DOI 10.17487/RFC5652, September 2009,
          <https://www.rfc-editor.org/info/rfc5652>.

Authors' Addresses

   Andrew Biggs
   Cisco Systems

   Email: adb@cisco.com

   Shaun Cooley
   Cisco Systems

   Email: shcooley@cisco.com