Independent Submission
Request for Comments: 7425
Category: Informational
ISSN: 2070-1721

M. Thornburgh
Adobe

December 2014

Adobe's RTMFP Profile for Flash Communication

Abstract

This memo describes how to use Adobe's Secure Real-Time Media Flow Protocol (RTMFP) to transport the video, audio, and data messages of Adobe Flash platform communications. Aspects of this application profile include cryptographic methods and data formats, flow metadata formats, and protocol details for client-server and peer-to-peer communication.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This is a contribution to the RFC Series, independently of any other RFC stream. The RFC Editor has chosen to publish this document at its discretion and makes no statement about its value for implementation or deployment. Documents approved for publication by the RFC Editor are not a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7425.

Table of Contents

1. Introduction

Adobe's Secure Real-Time Media Flow Protocol (RTMFP) [RFC7016] is a general-purpose transport service for real-time media and bulk data in IP networks, and it is suited to client-server and peer-to-peer (P2P) communication. RTMFP provides a generalized framework for securing its communications according to the needs of its application.

The Flash platform comprises the Flash runtime (including Flash Player) from Adobe Systems Incorporated, communication servers such as Adobe Media Server, and interoperable clients and servers provided by other parties.

Real-time streaming network communication for the Flash platform of video, audio, and data typically uses Adobe's Real-Time Messaging Protocol (RTMP) [RTMP] messages. RTMP messages were originally designed to be transported over RTMP Chunk Stream in TCP [RTMP]; however, other transports (such as the one described in this memo) are possible.

This memo specifies the syntax and semantics for transporting RTMP messages over RTMFP, and it extends Flash communication semantics to include direct P2P communication. This memo further specifies a concrete Cryptography Profile for RTMFP tailored to the application and cryptographic needs of Flash platform client-server and P2P communications.

These protocols and profiles were developed by Adobe Systems Incorporated and are not the product of an IETF activity.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

"HMAC" means the Keyed-Hash Message Authentication Code (HMAC) algorithm [RFC2104].

"HMAC-SHA256" means HMAC using the SHA-256 Secure Hash Algorithm [SHA256] [RFC6234].

"HMAC-SHA256(K, M)" means the calculation of the HMAC-SHA256 of message M using key K.

3. Common Syntax Elements

Definitions of types and structures in this specification use traditional text diagrams paired with procedural descriptions using a C-like syntax. The C-like procedural descriptions SHALL be construed as definitive.

Structures are packed to take only as many bytes as explicitly indicated. There is no 32-bit alignment constraint, and fields are not padded for alignment unless explicitly indicated or described. Text diagrams may include a bit ruler across the top; this is a convenience for counting bits in individual fields and does not necessarily imply field alignment on a multiple of the ruler width.

Unless specified otherwise, reserved fields SHOULD be set to 0 by a sender and MUST be ignored by a receiver.

The procedural syntax of this specification defines correct and error-free encoded inputs to a parser. The procedural syntax does not describe a fully featured parser, including error detection and handling. Implementations MUST include means to identify error circumstances, including truncations causing elementary or composed types not to fit inside containing structures, fields, or elements. Unless specified otherwise, an error circumstance SHALL abort the parsing and processing of an element and its enclosing elements.

This memo uses the elementary and composed types described in Section 2.1 of RFC 7016. The definitions of that section are incorporated by reference as though fully set forth here.

4. Cryptography Profile

RTMFP defines a general security framework but delegates specifics, such as packet encryption ciphers and key agreement algorithms, to an application-defined Cryptography Profile.

This section defines the RTMFP Cryptography Profile for Flash platform communication.

4.1. Default Session Key

RTMFP uses a Default Session Key and associated default cipher configuration during session startup handshaking, where session-specific keys and ciphers are negotiated.

The default cipher is the Advanced Encryption Standard [AES] with 128-bit keys operating in Cipher Block Chaining [CBC] mode, as described in Section 4.7.1. The Default Session Key is the 16 bytes of the string "Adobe Systems 02" encoded in UTF-8 [RFC3629]:

Hex: 41 64 6F 62 65 20 53 79 73 74 65 6D 73 20 30 32

The Default Session Key uses checksum mode for packet verification and does not use session sequence numbers (Section 4.7.3).

4.2. Diffie-Hellman Groups

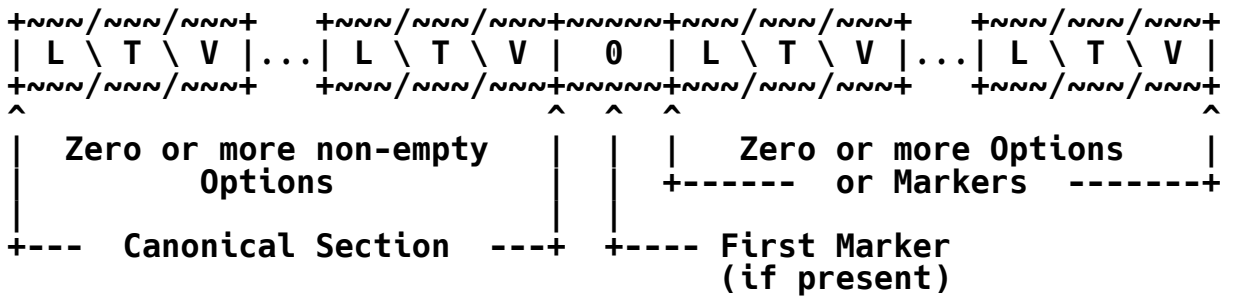
Implementations conforming to this profile MUST support Diffie-Hellman [DH] modular exponentiation (MODP) group 2 (1024 bits) as defined in [RFC7296], and SHOULD support Diffie-Hellman MODP group 5 (1536 bits) and group 14 (2048 bits) as defined in [RFC3526]. Implementations MAY support additional groups.

4.3. Certificates

This section defines the certificate format for this Cryptography Profile, and the mapping to the abstract properties and semantics for RTMFP endpoint identities.

4.3.1. Format

A certificate in this profile is encoded as a sequence of zero or more RTMFP Options and Markers (Section 2.1.3 of RFC 7016). The first marker (if any) in the certificate separates the canonical section of the certificate from the remainder. Some options are ignored if they occur outside of the canonical section (that is, after the first marker).




```
struct certificate_t
{
    canonicalStart = remainder();
    canonicalEnd = remainder();
    markerFound = false;

    while(remainder() > 0)
    {
        option_t option :variable*8;

        if(0 == option.length)
            markerFound = true;
        else if(!markerFound)
            canonicalEnd = remainder();
    };

    canonicalSectionLength = canonicalStart - canonicalEnd;
} :variable*8;
```

4.3.2. Fingerprint

A certificate's fingerprint is the SHA-256 hash [SHA256] of the canonical section of the certificate (that is, the hash of the first canonicalSectionLength bytes of the certificate).

The certificate's fingerprint is also called the "peer ID".

4.3.3. Options

This section lists options that can appear in a certificate. The following option type codes are defined:

0x00: Hostname (must be in canonical section) (Section 4.3.3.1)

0x0a: Accepts Ancillary Data (must be in canonical section) (Section 4.3.3.2)

0x0e: Extra Randomness (Section 4.3.3.3)

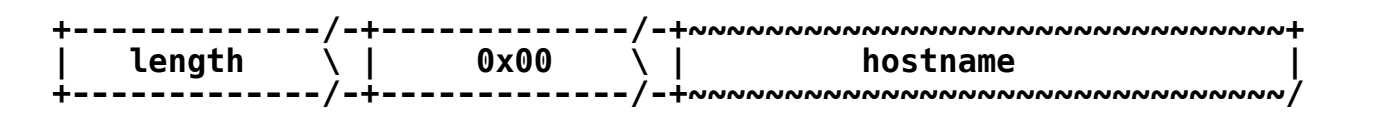
0x15: Supported Ephemeral Diffie-Hellman Group (must be in canonical section) (Section 4.3.3.4)

0x1d: Static Diffie-Hellman Public Key (must be in canonical section) (Section 4.3.3.5)

An implementation MUST ignore a certificate option type that is not understood.

4.3.3.1. Hostname

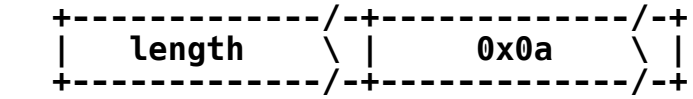
This option gives an optional hostname for the endpoint. This option MUST be ignored if is not in the canonical section. This option MUST NOT occur more than once in a certificate.




```
struct hostnameCertOptionValue_t
{
    uint8_t hostname[remainder()];
} :remainder()*8;
```

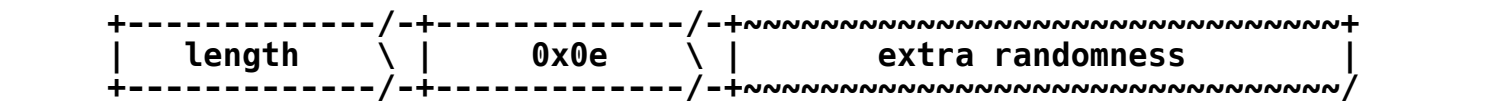
4.3.3.2. Accepts Ancillary Data

This option indicates that the endpoint will accept an Endpoint Discriminator encoding an Ancillary Data option (Section 4.4.2.2). This option MUST be ignored if it is not in the canonical section.



4.3.3.3. Extra Randomness

This option can be used to add extra entropy or randomness to a certificate that doesn't have any other cryptographic pseudorandom members (such as a public key). This option is typically used so that endpoints using ephemeral Diffie-Hellman keying can have a unique certificate fingerprint.

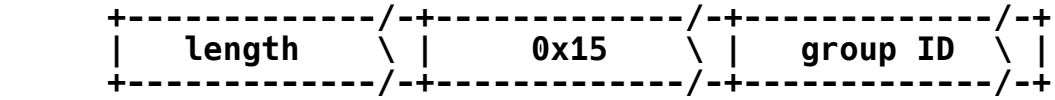


```
+-------------/-+-------------/-+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
|   length    \ |     0x0e    \ |       extra randomness        |
+-------------/-+-------------/-+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~/
```

```
struct extraRandomnessCertOptionValue_t
{
    uint_t extraRandomness[remainder()];
} :remainder()*8;
```

4.3.3.4. Supported Ephemeral Diffie-Hellman Group

This option specifies a Diffie-Hellman group ID that is supported for ephemeral keying. This option MUST be ignored if it is not in the canonical section. This option may occur more than once in the certificate; each instance indicates an additional group that is supported for key agreement.



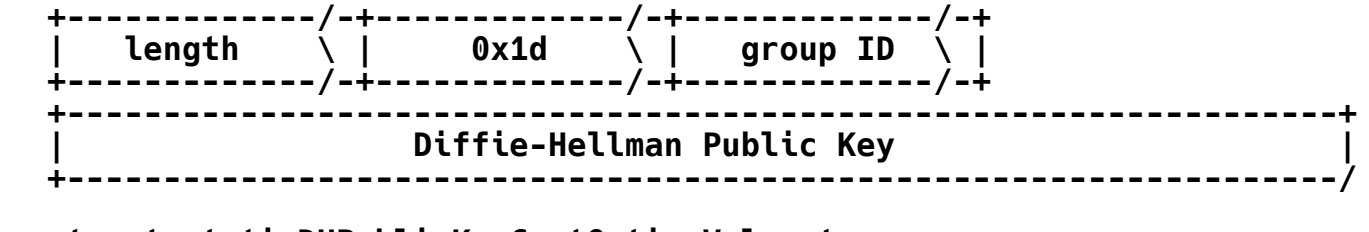
```
+-------------/-+-------------/-+-------------/-+
|   length    \ |     0x15    \ |   group ID  \ |
+-------------/-+-------------/-+-------------/-+
```

```
struct ephemeralDHGroupCertOptionValue_t
{
    vlu_t groupID :variable*8;
} :variable*8;
```

The presence of this option means that the certificate uses ephemeral Diffie-Hellman public keys only. The certificate MUST NOT contain a Static Diffie-Hellman public key (Section 4.3.3.5).

4.3.3.5. Static Diffie-Hellman Public Key

This option specifies a Diffie-Hellman group ID and static public key in that group. This option MUST be ignored if it is not in the canonical section. This option MAY occur more than once in the certificate; however, this option SHOULD NOT occur more than once for each group ID. The behavior for specifying more than one public key per group ID is not defined.




```
struct staticDHPublicKeyCertOptionValue_t
{
    vlu_t   groupID :variable*8;
    uintn_t publicKey :remainder()*8; // network byte order
} :remainder()*8;
```

The presence of this option means that the certificate uses static Diffie-Hellman public keys only. The certificate MUST NOT contain any Supported Ephemeral Diffie-Hellman Group options (Section 4.3.3.4).

4.3.4. Authenticity

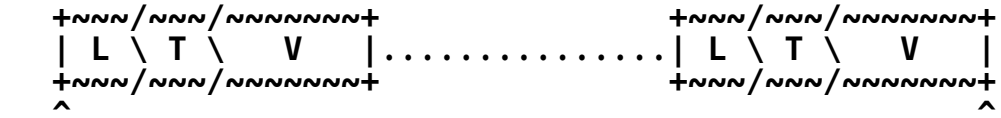
This profile does not use a public key infrastructure, nor are there signing keys present in certificates. Therefore, any properly encoded certificate is considered authentic according to Section 3.2 of RFC 7016.

A certificate containing a static public key can only be used successfully for session communication if the holder of the certificate actually holds the private key associated with the public key. Authenticity of an identity and its peer ID (Section 4.3.2) having a certificate containing a static public key is implied by successful encrypted communication with the associated endpoint (Section 4.6).

See Section 7 for further discussion of security issues related to identities.

4.3.5. Signing and Verifying Messages

RTMFP Initiator Initial Keying and Responder Initial Keying messages have a field for the sender's digital signature of the keying parameters (Sections 2.3.7 and 2.3.8 of RFC 7016). In this profile, the signature field of those messages is encoded as a sequence of zero or more RTMFP Options.




```
struct initialKeyingSignature_t
{
    while(remainder() > 0)
        option_t option :variable*8;
} :remainder()*8;
```

If a signer has no signature options to send, it MAY encode a signature as a UTF-8 capital "X" (hex 58) or as empty. A verifier MUST interpret a malformed signature field or a signature field consisting only of a UTF-8 capital "X" as though it was empty.

If a verifier does not require a signature, it SHALL consider any signature field (including an empty or malformed one) to be valid. A verifier MAY require a signature comprising one or more non-empty options that are valid according to their respective types.

This profile does not use a public key infrastructure, nor are there signing keys present in certificates. Section 4.3.5.1.1 defines a simple ID/password credential system.

4.3.5.1. Options

This section lists options that can appear in an RTMFP Initial Keying signature field. The following option type code is defined:

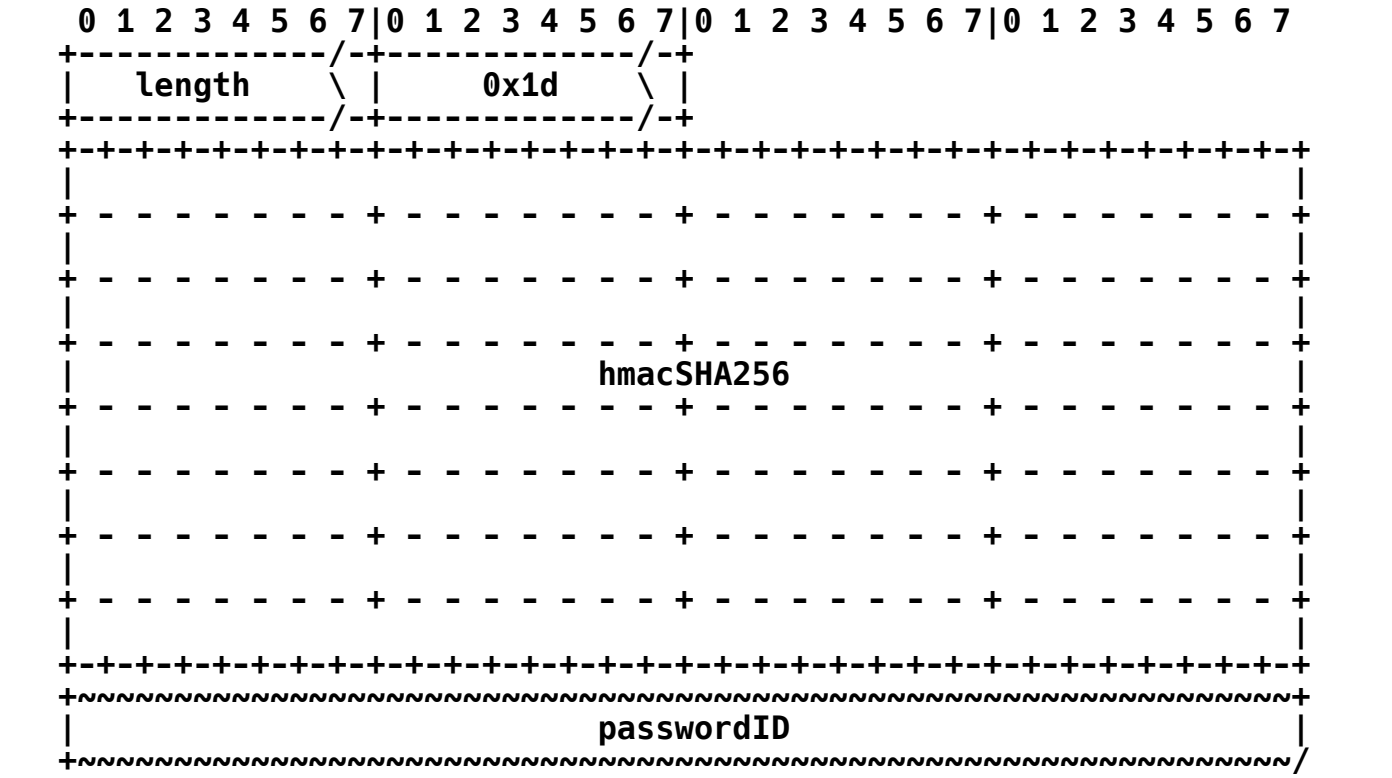
0x1d: Simple Password (Section 4.3.5.1.1)

Future or derived profiles may define additional signature field options and semantics; therefore, a verifier SHOULD ignore option types that are not understood.

4.3.5.1.1. Simple Password

This option encodes a password identifier (such as a user name, or an application-specific or implementation-specific selector) and an HMAC over the signed parameters using the identified password as the HMAC key. This option can occur more than once (for example, to allow interoperation between a current and a previous version of an implementation using implementation-specific passwords).

To support the versioning use case, a verifier SHOULD ignore a Simple Password option encoding an unrecognized password identifier. A verifier SHOULD treat the entire signature as invalid if any Simple Password option encodes a recognized password identifier with an invalid password HMAC.




```
struct simplePasswordSignatureOptionValue_t
{
    uint8_t hmacSHA256[32];
    uint8_t passwordID[remainder()];
} :remainder()*8;
```

hmacSHA256: HMAC-SHA256(K, M), where K is the password associated with passwordID, and M is the signed parameters.

passwordID: The identifier (such as a user name) for the password used as the HMAC key.

4.3.6. Glare Resolution

Glare occurs when two endpoints initiate a session each to the other concurrently.

Compare the near end's certificate to the far end's with a binary lexicographic comparison, one byte at a time, up to the length of the shorter certificate. At the first corresponding byte from each certificate that is different, the certificate having the differing byte (treated as an unsigned 8-bit integer) with the lower value is ordered before the other certificate. If the certificates are not the same length and they are identical up to the length of the shorter certificate, then the shorter certificate is ordered before the longer.

The near end prevails as the Initiator in case of glare if its certificate is ordered before, or is identical to, the certificate of the far end. Otherwise, the near end's certificate is ordered after the far end's certificate, and the near end assumes the role of Responder.

4.3.7. Session Override

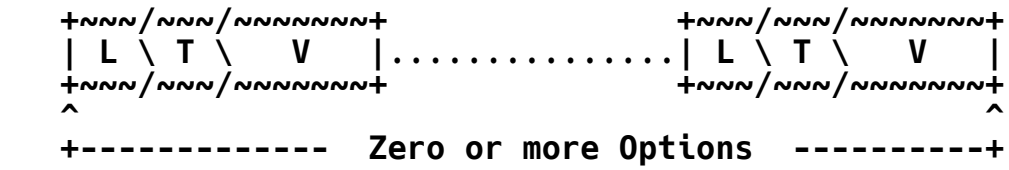
A new incoming session overrides an existing session only if the certificate for the new session is identical to the certificate for the existing session.

4.4. Endpoint Discriminators

This section describes the Endpoint Discriminator (EPD) (Section 3.2 of RFC 7016) format and semantics for this Cryptography Profile, and the mapping to RTMFP's abstract certificate and identity selection semantics.

4.4.1. Format

An EPD in this profile is encoded as a sequence of zero or more RTMFP Options.




```
struct endpointDiscriminator_t
{
    while(remainder() > 0)
        option_t option :variable*8;
} :remainder()*8;
```

4.4.2. Options

This section lists options that can appear in an EPD. The following option type codes are defined:

0x00: Required Hostname (Section 4.4.2.1)

0x0a: Ancillary Data (Section 4.4.2.2)

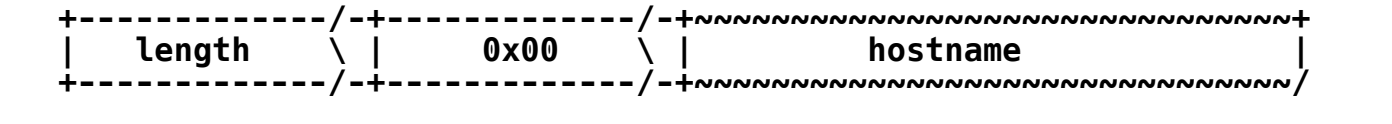
0x0f: Fingerprint (Section 4.4.2.3)

The use of these options for selecting certificates is described in Section 4.4.3.

An implementation MUST ignore EPD option types that are not understood.

4.4.2.1. Required Hostname

This option indicates the hostname to match against the certificate's Hostname option (Section 4.3.3.1).

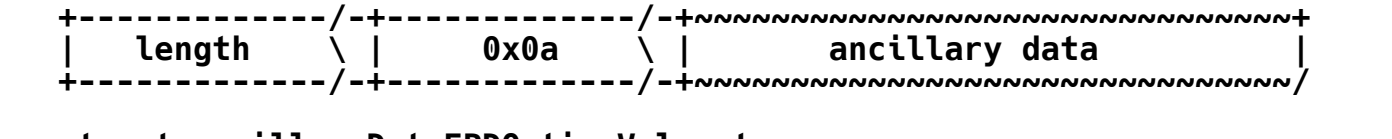



```
struct hostnameEPDOptionValue_t
{
    uint8_t hostname[remainder()];
} :remainder()*8;
```

This option MUST NOT occur more than once in an EPD.

4.4.2.2. Ancillary Data

In this profile, this option indicates the server Uniform Resource Identifier (URI) [RFC3986] encoded in UTF-8 to which a client is connecting on this session, for example, "rtmfp://server.example.com/app/instance".

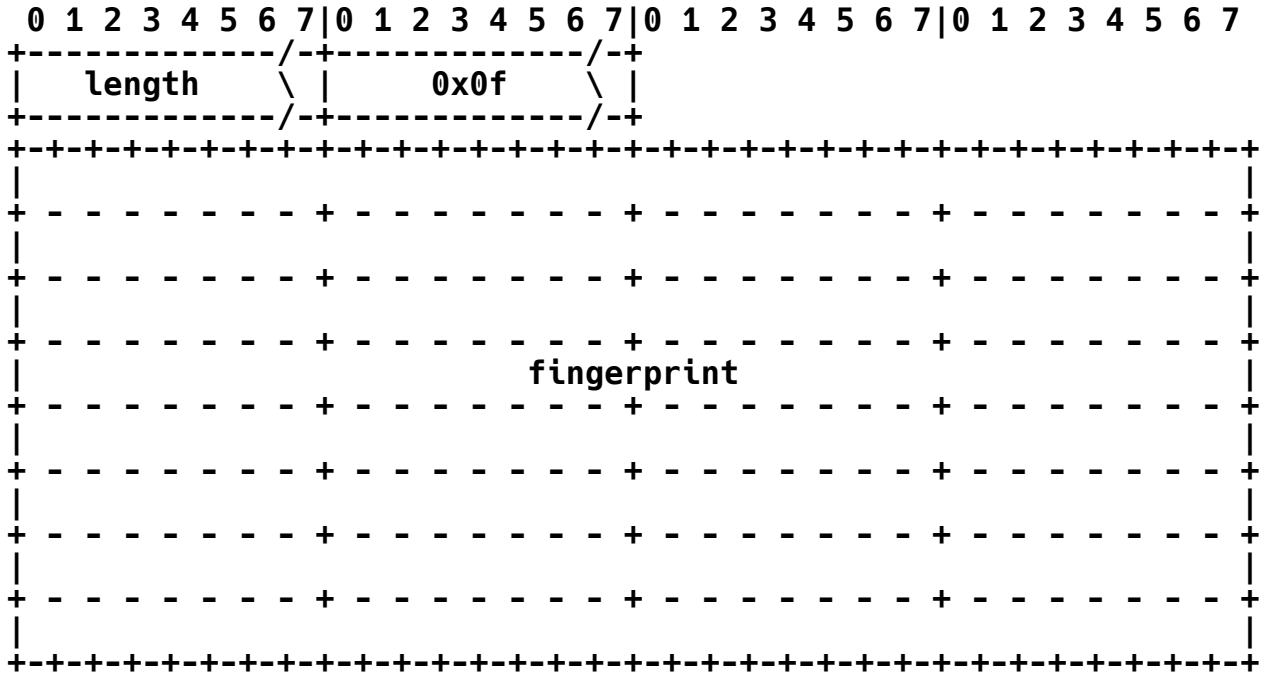



```
struct ancillaryDataEPDOptionValue_t
{
    uint8_t ancillaryData[remainder()];
} :remainder()*8;
```

This option MUST NOT occur more than once in an EPD.

4.4.2.3. Fingerprint

This option indicates the 256-bit (32-byte) fingerprint (Section 4.3.2) of a certificate.




```
struct fingerprintEPDOptionValue_t
{
    uint8_t fingerprint[32];
} :256;
```

This option MUST NOT occur more than once in an EPD.

4.4.3. Certificate Selection

This section describes the REQUIRED method of determining whether an EPD selects a certificate.

An EPD MUST contain at least one of Fingerprint, Required Hostname, or Ancillary Data options to select any certificate.

A Fingerprint EPD option selects or rejects a certificate no matter what other options are present.

Without a Fingerprint option, a Required Hostname EPD option, if present, REQUIRES an identical Hostname option in the certificate.

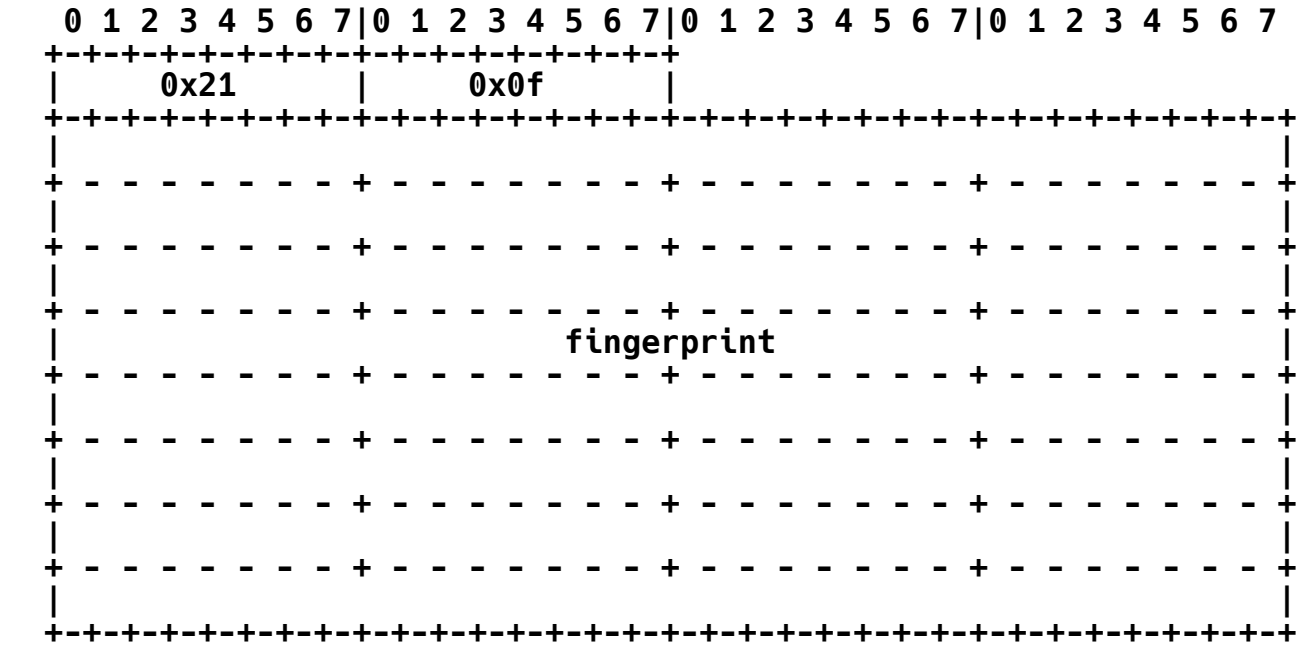
Without a Fingerprint option, an Ancillary Data EPD option, if present, REQUIRES that the certificate has an Accepts Ancillary Data option.

```
if EPD contains a Fingerprint option:
    if certificate.fingerprint == option.fingerprint:
        certificate is selected. stop.
    else:
        certificate is not selected. stop.
else:
    if EPD contains a Required Hostname option:
        if certificate contains a Hostname option:
            if certificate.hostname != option.hostname:
                certificate is not selected. stop.
        else:
            certificate is not selected. stop.
    if EPD contains an Ancillary Data option:
        if certificate doesn't have an Accepts Ancillary Data option:
            certificate is not selected. stop.
    else if EPD does not contain a Required Hostname option:
        certificate is not selected. stop.
    certificate is selected. stop.
```

Figure 1: Algorithm to Test Whether an EPD Selects a Certificate

4.4.4. Canonical Endpoint Discriminator

In this profile, a Canonical Endpoint Discriminator (Section 3.2 of RFC 7016) contains only a Fingerprint option (Section 4.4.2.3) and no other options. The option length and type code MUST be encoded as 1-byte VLUs, even though VLU encoding allows those fields to be encoded in an arbitrary number of bytes. That is, the Canonical Endpoint Discriminator MUST be exactly 34 bytes long, with a length field of 0x21 encoded as one byte, a type code of 0x0f encoded as one byte, and 32 bytes of fingerprint.



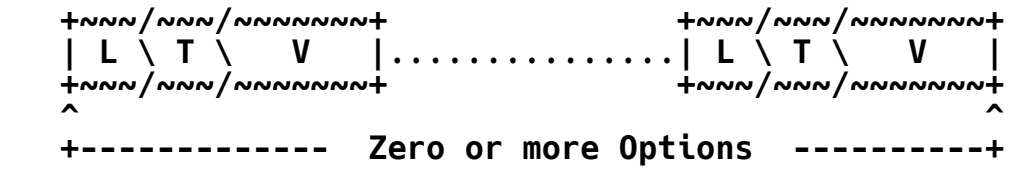

```
struct canonicalEndpointDiscriminator_t
{
    uint8_t length = 0x21;
    uint8_t type = 0x0f;
    uint8_t fingerprint[32];
} :272;
```

4.5. Session Keying Components

This section describes the format of the Session Key Initiator Component of the Initiator Initial Keying RTMFP chunk and the Session Key Responder Component of the Responder Initial Keying RTMFP chunk (Sections 2.3.7 and 2.3.8 of RFC 7016). The Initiator and Responder Session Keying Components have the same format.

4.5.1. Format

A Session Keying Component in this profile is encoded as a sequence of zero or more RTMFP Options.




```
struct sessionKeyingComponent_t
{
    while(remainder() > 0)
        option_t option :variable*8;
} :remainder()*8;
```

4.5.2. Options

This section lists options that can appear in a Session Keying Component. The following option type codes are defined:

0x0d: Ephemeral Diffie-Hellman Public Key (Section 4.5.2.1)

0x0e: Extra Randomness (Section 4.5.2.2)

0x1d: Diffie-Hellman Group Select (Section 4.5.2.3)

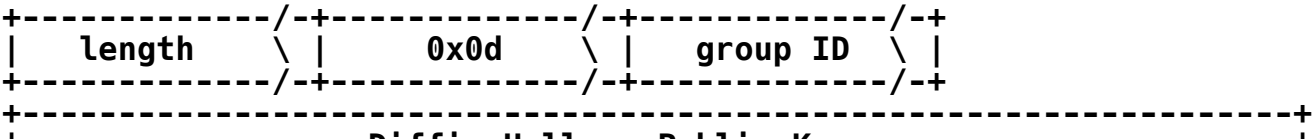
0x1a: HMAC Negotiation (Section 4.5.2.4)

0x1e: Session Sequence Number Negotiation (Section 4.5.2.5)

An implementation MUST ignore a session keying component option type that is not understood.

4.5.2.1. Ephemeral Diffie-Hellman Public Key

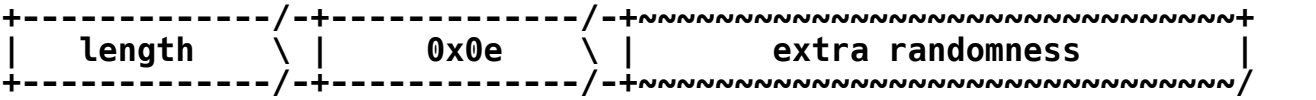
This option specifies a Diffie-Hellman group ID and public key in that group. This option MUST NOT be sent if the sender's certificate has a static Diffie-Hellman public key. This option MUST be sent if the sender's certificate does not have a static Diffie-Hellman public key. This option MUST NOT be sent more than once.




```
struct ephemeralDHPublicKeyKeyingOptionValue_t
{
    vlu_t   groupID :variable*8;
    uintn_t publicKey :remainder()*8; // network byte order
} :remainder()*8;
```

4.5.2.2. Extra Randomness

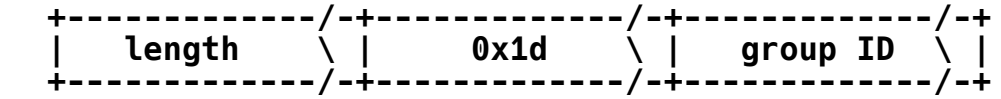
This option can be used to add extra entropy or randomness to a keying component, particularly when the sender uses a static public key. When used for that purpose, the extra randomness SHOULD be cryptographically strong pseudorandom bytes not less than 16 bytes (for cryptographically significant entropy) and not more than 64 bytes (the length of a SHA-256 input block) in length. The extra randomness serves as a salt when computing the session keys (Section 4.6).




```
struct extraRandomnessKeyingOptionValue_t
{
    uint_t extraRandomness[remainder()];
} :remainder()*8;
```

4.5.2.3. Diffie-Hellman Group Select

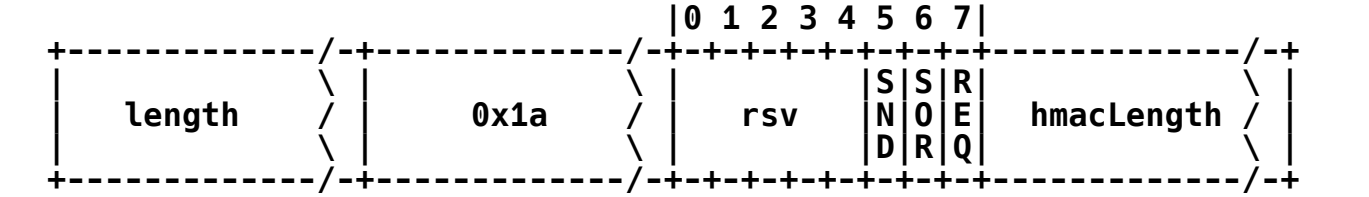
This option is sent by the Initiator to specify which Diffie-Hellman group to use for key agreement. The Initiator MUST send this option when it advertises a static Diffie-Hellman public key in its certificate and MUST NOT send this option if it sends an ephemeral Diffie-Hellman public key. This option MUST NOT be sent more than once.




```
struct staticDHGroupSelectKeyingOptionValue_t
{
    vlu_t   groupID :variable*8;
} :variable*8;
```

4.5.2.4. HMAC Negotiation

This option is used to negotiate sending and receiving of an HMAC field for packet verification.




```
struct hmacNegotiationKeyingOptionValue_t
{
    uintn_t reserved :5;          // rsv
    bool_t  willSendAlways :1;    // SND
    bool_t  willSendOnRequest :1; // SOR
    bool_t  request :1;           // REQ
    vlu_t   hmacLength :variable*8;
} :variable*8;
```

willSendAlways: If set, the sender will send an HMAC on packets in this session.

willSendOnRequest: If set, the sender will send an HMAC on packets in this session if the other end sets the request flag in its HMAC Negotiation.

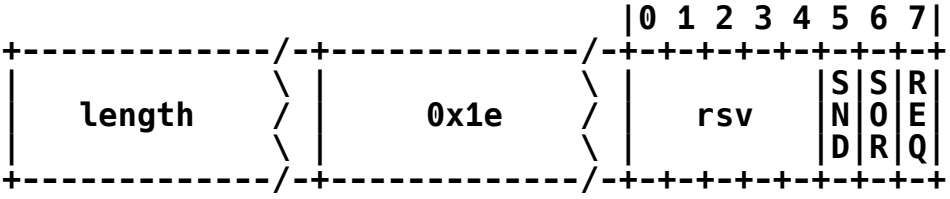
request: If set, the sender would very much like the receiver to send an HMAC on its packets. If the other end doesn't send an HMAC on its packets, the session can fail.

hmacLength: If the sender negotiates to send an HMAC on its packets, the HMAC field will be this many bytes long. This value MUST be between 4 and 32 inclusive, or 0 if and only if willSendAlways and willSendOnRequest are clear.

The handshake operational semantics for this option are described in Section 4.6.4.

4.5.2.5. Session Sequence Number Negotiation

This option is used to negotiate sending and receiving of the Session Sequence Number field for packet verification.




```
struct sseqNegotiationKeyingOptionValue_t
{
    uintn_t reserved :5;          // rsv
    bool_t  willSendAlways :1;    // SND
    bool_t  willSendOnRequest :1; // SOR
    bool_t  request :1;           // REQ
} :8;
```

willSendAlways: If set, the sender will send a session sequence number in packets in this session.

willSendOnRequest: If set, the sender will send a session sequence number in packets in this session if the other end sets the request flag in its Session Sequence Number Negotiation.

request: If set, the sender would very much like the receiver to send a session sequence number in its packets. If the other end doesn't send a session sequence number in its packets, the session can fail.

The handshake operational semantics for this option are described in Section 4.6.6.

4.6. Session Key Computation

This section describes how to compute the cryptographic keys and other settings for packet encryption and verification.

The Session Key Near Component (SKNC) means the keying component sent by the near end of the session; that is, it is the Session Key Initiator Component at the Initiator and the Session Key Responder Component at the Responder.

The Session Key Far Component (SKFC) means the keying component sent by the far end of the session; that is, it is the Session Key Responder Component at the Initiator and the Session Key Initiator Component at the Responder.

4.6.1. Public Key Selection

This section enumerates the public key selection methods for all possible combinations of static or ephemeral public key modes for each endpoint according to their certificate options (Section 4.3.3).

4.6.1.1. Initiator and Responder Ephemeral

The Initiator and Responder list one or more Supported Ephemeral Diffie-Hellman Group options (Section 4.3.3.4) in their certificates. The Initiator sends exactly one Ephemeral Diffie-Hellman Public Key option (Section 4.5.2.1) in its Session Key Initiator Component, which selects one group from among those supported by the Responder and Initiator. Responder sends exactly one Ephemeral Diffie-Hellman Public Key option in its Session Key Responder Component, in the same group as indicated by the Initiator.

4.6.1.2. Initiator Ephemeral and Responder Static

The Responder lists one or more Static Diffie-Hellman Public Key options (Section 4.3.3.5) in its certificate. The Initiator lists one or more Supported Ephemeral Diffie-Hellman Group options in its certificate. The Initiator sends exactly one Ephemeral Diffie-Hellman Public Key option in its Session Key Initiator Component, which selects one group from among those supported by the Responder and Initiator and the corresponding public key for the Responder. Responder uses its public key from the indicated group, and sends only an Extra Randomness option (Section 4.5.2.2) in its Session Key Responder Component to salt the session keys.

4.6.1.3. Initiator Static and Responder Ephemeral

The Responder lists one or more Supported Ephemeral Diffie-Hellman Group options in its certificate. The Initiator lists one or more Static Diffie-Hellman Public Key options in its certificate. The Initiator sends exactly one Diffie-Hellman Group Select option (Section 4.5.2.3) in its Session Key Initiator Component, which selects one group from among those supported by the Responder and Initiator and the corresponding public key for the Initiator, plus an Extra Randomness option to salt the session keys. The Responder sends an Ephemeral Diffie-Hellman Public Key option in its Session Key Responder Component in the same group as indicated by the Initiator.

4.6.1.4. Initiator and Responder Static

The Initiator and Responder each list one or more Static Diffie-Hellman Public Key options in their certificates. The Initiator sends exactly one Diffie-Hellman Group Select option in its Session Key Initiator Component, which selects one group and corresponding public keys from among those supported by the Responder and Initiator, and an Extra Randomness option to salt the session keys. The Responder sends an Extra Randomness option in its Session Key Responder Component to add its own salt to the session keys.

4.6.2. Diffie-Hellman Shared Secret

To be acceptable, a Diffie-Hellman public key MUST have all of the following properties:

- Be at least 16777216 (2^24);
- Be at most the group's prime modulus minus 16777216;
- Have at least 16 "1" bits;
- Have at least 16 "0" bits, not including leading zeros.

An endpoint MUST NOT complete to an S_OPEN session with a far endpoint using a public key that is not acceptable according to these criteria.

Once the group and corresponding public key of the far end is determined, the far end's public key and the near end's private key are combined according to Diffie-Hellman [DH] to compute the Diffie-Hellman Shared Secret, an integer.

In the following sections, DH_SECRET means the Diffie-Hellman Shared Secret encoded as a byte-aligned unsigned integer in network byte order with no leading zero bytes. For example, if the shared secret is 4886718345, DH_SECRET would be the five bytes:

Hex: 01 23 45 67 89

4.6.3. Packet Encrypt/Decrypt Keys

Packets are encrypted using a symmetric cipher, such as the Advanced Encryption Standard [AES]. Distinct keys are used for sending and receiving packets. Each end's sending (encrypt) key is the other end's receiving (decrypt) key.

The raw keys computed in this section for encryption and decryption are transformed in a manner specific to the cipher with which they are to be used. In this profile, AES-128 is the only currently defined cipher. For this cipher, the first 128 bits (16 bytes) of the 256-bit output of the calculation are taken to be the AES-128 key.

```
Set ENCRYPT_KEY = HMAC-SHA256(DH_SECRET, HMAC-SHA256(SKFC, SKNC));

Set DECRYPT_KEY = HMAC-SHA256(DH_SECRET, HMAC-SHA256(SKNC, SKFC));
```

The full 256 bits of ENCRYPT_KEY and DECRYPT_KEY are used in the computations in the following sections.

4.6.4. Packet HMAC Send/Receive Keys

Packets can be verified that they were not corrupted or modified by appending an HMAC to the packet. Whether to use an HMAC or a simple checksum is determined during the initial keying phase using the HMAC Negotiation option (Section 4.5.2.4). Distinct HMAC keys are used for sending and receiving packets. Each end's sending key is the other end's receiving key, and vice versa.

```
Set HMAC_SEND_KEY = HMAC_SHA256(DH_SECRET, ENCRYPT_KEY);

Set HMAC_RECV_KEY = HMAC_SHA256(DH_SECRET, DECRYPT_KEY);
```

If an endpoint sets the willSendAlways flag in its HMAC Negotiation option, then it MUST send an HMAC on packets it sends with this session key.

If an endpoint's willSendAlways flag is clear but its willSendOnRequest flag is set, then it MUST send an HMAC on packets it sends with this session key if and only if the other endpoint's request flag is set.

If a sending endpoint's willSendAlways and willSendOnRequest flags are clear, then the receiving endpoint SHOULD reject that keying component if the receiving endpoint is configured to require the sending endpoint to send HMAC.

If HMAC is negotiated to be used, the corresponding hmacLength MUST be between 4 and 32 inclusive.

If HMAC is negotiated not to be used, a simple checksum is used for packet verification.

The Default Session Key uses the simple checksum and does not use HMAC.

4.6.5. Session Nonces

Session nonces are per-session, cryptographically strong secret values known only to the two endpoints of the session. They can be used for application-layer cryptographic challenges (such as signing or password verification). These nonces are a convenience being pre-shared and pre-agreed-upon in a secure manner during the initial keying handshake.

Each end's near nonce is the other end's far nonce, and vice versa.

```
Set NEAR_NONCE = HMAC_SHA256(DH_SECRET, SKNC);

Set FAR_NONCE = HMAC_SHA256(DH_SECRET, SKFC);
```

4.6.6. Session Sequence Number

Duplicate packets can be detected and rejected by using an optional session sequence number inside the encrypted packets. The session sequence number is a monotonically increasing unbounded integer and does not wrap. Session sequence numbers SHOULD start at zero and SHOULD increment by one for each packet sent using that session key. Implementations MUST handle session sequence numbers with no less than 64 bits of range.

If an endpoint's willSendAlways flag in its Session Sequence Number Negotiation option (Section 4.5.2.5) is set, then it MUST send a session sequence number in packets it sends with this session key.

If an endpoint's willSendAlways flag is clear but its willSendOnRequest flag is set, then it MUST send a session sequence number on packets it sends with this session key if and only if the other endpoint's request flag is set.

If a sending endpoint's willSendAlways and willSendOnRequest flags are clear, then the receiving endpoint SHOULD reject that keying component if the receiving endpoint is configured to require the sending endpoint to send session sequence numbers.

The Default Session Key does not use session sequence numbers.

4.7. Packet Encryption

This section describes the concrete syntax and operational semantics of RTMFP packet encryption for this Cryptography Profile.

4.7.1. Cipher

This profile defines AES-128 [AES] in CBC [CBC] mode as the only cipher. Extensions to this profile can specify and negotiate additional ciphers and modes by defining certificate and keying component options and associated semantics.

For AES-128-CBC, the initialization vector (IV) for each packet is 16 zero bytes. The IV is not included in the packet.

4.7.2. Format

The Encrypted Packet is the encryptedPacket field of an RTMFP Multiplex packet (Section 2.2.2 of RFC 7016); that is, the portion of the Multiplex packet following the scrambled session ID. The Encrypted Packet has the following format:

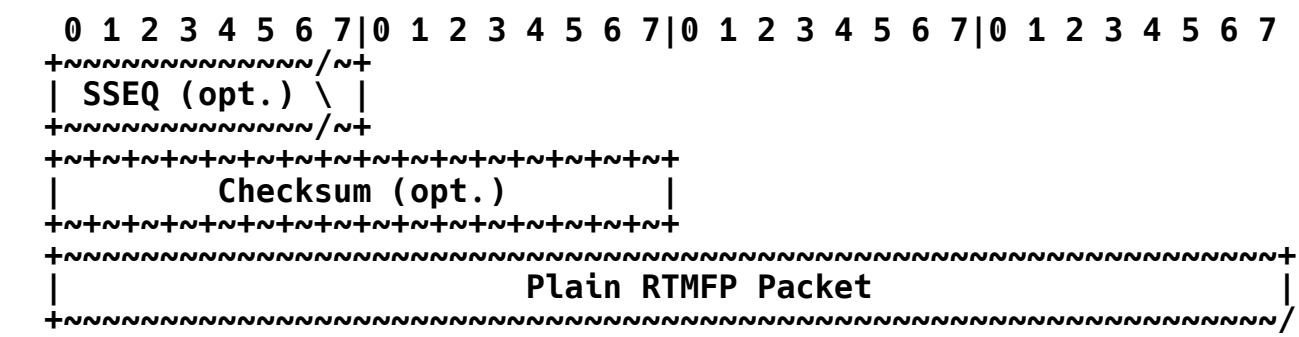

```
struct flashProfileEncryptedPacket_t
{
    if(HMAC is being used)
        hmacLength = negotiated length;
    else
        hmacLength = 0;

    struct
    {
        iv[16 bytes] = { 0 };
        blockCount = 0;
        while((remainder() > hmacLength) && (remainder() >= 16))
        {
            uint8_t cbcBlock[16];
            blockCount++;
        }
    } chainedCipherBlocks :variable*16*8;

    if(HMAC is being used)
    {
        if(remainder() == hmacLength)
            uint8_t truncatedHMAC[hmacLength];
        else
            packetVerificationFailed();
    }
    else if(remainder() > 0)
        packetVerificationFailed();
} :encryptedPacket.length*8;
```

cbcBlock: The next AES-128-CBC block.

chainedCipherBlocks: The concatenation of every cipher block in the packet (over which the HMAC is computed).

truncatedHMAC: If HMAC was negotiated to be used (Section 4.5.2.4), this field is set to the first negotiated hmacLength bytes of the HMAC of the chainedCipherBlocks.

The plaintext data before encryption or after decryption has the following format:




```
struct flashProfilePlainPacket_t
{
    if(session sequence numbers being used)
        vlu_t sessionSequenceNumber :variable*8; // SSEQ
    if(HMAC not being used)
        uint16_t checksum;
    packet_t plainRTMFPPacket :variable*8;
} :chainedCipherBlocks.blockCount*16*8;
```

sessionSequenceNumber: If session sequence numbers were negotiated to be used (Section 4.6.6), this field is present and is the VLU session sequence number of this packet.

checksum: If HMAC was not negotiated to be used, this field is present and is the simple checksum (Section 4.7.3.1) of the remaining bytes of this structure.

plainRTMFPPacket: The (plain, unencrypted) RTMFP Packet (Section 2.2.4 of RFC 7016) plus any necessary padding.

When assembling this structure and prior to calculating the checksum (if present), if the structure's total length is not an integer multiple of 16 bytes (the AES cipher block size), pad the end of plainRTMFPPacket with as many bytes having a value of 0xff as are needed to bring the structure's total length to an integer multiple of 16 bytes. The receiver's RTMFP Packet parser (Section 2.2.4 of RFC 7016) will consume this padding.

4.7.3. Verification

In RTMFP, the Cryptography Profile is responsible for packet verification. In this profile, packets are verified with an HMAC or a simple checksum, depending on the configuration of the endpoints, and optionally verified against replay or duplication using session sequence numbers. The simple checksum is inside the encrypted packet, so it becomes essentially a 16-bit cryptographic checksum.

4.7.3.1. Simple Checksum

The simple checksum is the 16-bit ones' complement of the 16-bit ones' complement sum of all 16-bit (2 bytes in network byte order) words to be checked. If there are an odd number of bytes to be checked, then for purposes of this checksum, treat the last byte as the lower 8 bits of a 16-bit word whose upper 8 bits are 0. This is also known as the "Internet Checksum" [RFC1071].

When present, the checksum is calculated over all bytes of the plaintext packet starting after the checksum field through the end of the plain packet. It cannot be calculated until the plain packet is padded, if necessary, to bring its length to an integer multiple of 16 bytes (the AES cipher block size). The session sequence number field, if present, and the checksum field itself are not included in the checksum.

On receiving a packet being verified with a checksum: calculate the checksum over all the bytes of the plaintext packet following the checksum field and compare the checksum to the value in the checksum field. If they match, the packet is verified; if they do not match, the packet is corrupt and MUST be discarded as though it was never received.

4.7.3.2. HMAC

When present, the HMAC field is the last hmacLength bytes of the packet and is calculated over all of the encrypted cipher blocks of the packet preceding the HMAC field. The value of the HMAC field is the first hmacLength bytes of the HMAC-SHA256 of the checked data, using the computed HMAC keys (Section 4.6.4) and negotiated hmacLength (Section 4.5.2.4). Note each endpoint independently specifies the length of the HMAC it will send via its hmacLength field.

When an endpoint has negotiated to send an HMAC, it encrypts the data blocks, computes the HMAC over the encrypted data blocks using its HMAC_SEND_KEY, and appends the first hmacLength bytes of that hash after the final encrypted data block.

When an endpoint has negotiated to receive an HMAC, the endpoint computes the HMAC over the encrypted data blocks using its HMAC_RECV_KEY and then compares the first receive hmacLength bytes of the computed HMAC to the HMAC field in the packet. If they are identical, the packet is verified; if they are not identical, the packet is corrupt and MUST be discarded as though it was never received.

HMAC and simple checksum verification are mutually exclusive.

4.7.3.3. Session Sequence Number

Session sequence numbers are used to detect and reject a packet that was duplicated in the network or replayed by an attacker and to ensure the first chained cipher block of every packet is unique, in lieu of a full-block initialization vector. Sequence numbers start at zero, increase by one for each packet sent in the session, do not wrap, and do not repeat.

When session sequence numbers are negotiated to be used, the receiver MUST allow for packets to be reordered in the network by up to at least 32 sequence numbers; note, however, that reordering by more than three packets can trigger loss detection and retransmission by negative acknowledgement, just as with TCP, and is therefore not likely to occur in the real Internet.

[RFC4302], [RFC4303], and [RFC6479] describe Anti-Replay Window methods that can be employed to detect duplicate sequence numbers. Other methods are possible.

Any packet received having a session sequence number that was already seen in that session, either directly or by being less than the lowest sequence number in the Anti-Replay Window, is a duplicate and MUST be discarded as though never received.

5. Flash Communication

The Flash platform uses RTMP [RTMP] messages for media streaming and communication. This section describes how to transport RTMP messages over RTMFP flows and additional messages and semantics unique to this transport.

5.1. RTMP Messages

An RTMP message comprises a virtual header and a payload. The virtual header comprises a Message Type, a Payload Length, a Timestamp, and a Stream ID. The format of the payload is dependent on the type of message.

An RTMP message is mapped onto a lower transport layer, such as RTMP Chunk Stream [RTMP] or RTMFP. RTMP messages were initially designed along with, and for transport on, RTMP Chunk Stream. This design constrains the possible values of RTMP message header fields. In particular:

Message Type is 8 bits wide, and is therefore constrained to values from 0 to 255 inclusive;

Payload Length is 24 bits wide, so messages can be at most 16777215 bytes long;

Timestamp is 32 bits wide, so timestamps range from 0 to 4294967295 and wrap around;

Stream ID is 24 bits wide, and is therefore constrained to values from 0 to 16777215 inclusive.

RTMP Chunk Stream Protocol Control messages (message types 1, 2, 3, 5, and 6) are not used when transporting RTMP messages in RTMFP flows. Messages of those types SHOULD NOT be sent and MUST be ignored.

5.1.1. Flow Metadata

All messages in RTMFP are transported in flows. In this profile, an RTMFP flow for RTMP messages carries the messages for exactly one RTMP Stream ID. Multiple flows can carry messages for the same Stream ID; for example, the video and audio messages of a stream could be sent on separate flows, allowing the audio to be given higher transmission priority.

The User Metadata for flows in this profile begins with a distinct signature to distinguish among different kinds of flows. The User Metadata for a flow used for RTMP messages begins with the two-character signature "TC".

The Stream ID is encoded in the flow's User Metadata so that it doesn't need to be sent with each message.

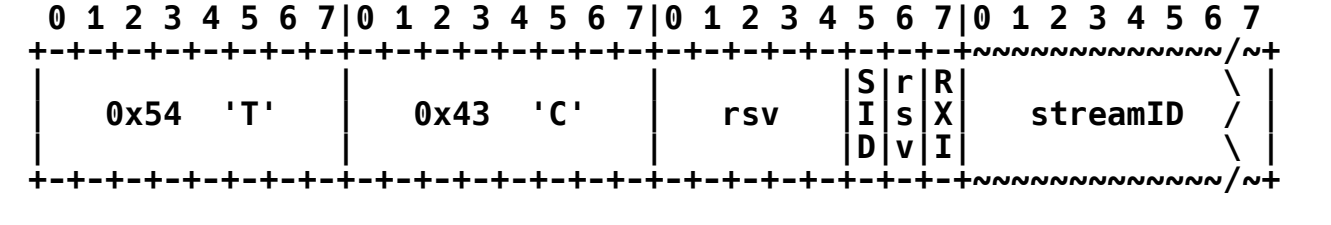
The sender can have a priori knowledge about the kind of media it intends to send on a flow and its intended use and can give the receiver a hint as to whether messages should be delivered as soon as possible or in their original queuing order. For example, the sender might be sending real-time, delay-sensitive audio messages on a flow, and hint that the receiver should take delivery of the messages on that flow as soon as they arrive in the network, to reduce the end-to-end latency of the audio.

The receiver can choose to take delivery of messages on flows as soon as they arrive in the network or in the messages' original queuing order. A receiver that chooses to take delivery of messages as soon as they arrive in the network MUST be prepared for the messages to

arrive out-of-order. For example, a receiver may choose not to render a newly received audio message having a timestamp earlier than the most recently rendered audio timestamp.

The sender can choose to abandon a message that it has queued in a flow before the message has been delivered to the receiver. For example, the sender may abandon a real-time, delay-sensitive audio message that has not been delivered within one second, to avoid spending transmission resources on stale media that is no longer relevant.

Note: A gap will cause a delay at the receiver of at least one round-trip time if the receiver is taking delivery of messages in original queuing order.




```
struct RTMPMetadata_t
{
    uint8_t signature[2] == { 'T', 'C' };
    uintn_t reserved1       :5; // rsv
    bool_t  streamIDPresent :1; // SID
    uintn_t reserved2       :1; // rsv
    uintn_t receiveIntent   :1; // RXI
        // 0: original queuing order, 1: network arrival order
    if(streamIDPresent)
        vlu_t   streamID        :variable*8;
} :variable*8;
```

signature: Metadata signature for RTMP message flows, being the two UTF-8 coded characters "TC".

streamIDPresent: A boolean flag indicating whether the streamID field is present. In this profile, this flag MUST be set.

receiveIntent: A hint by the sender as to the best order in which to take delivery of messages from the flow. A value of zero indicates a hint that the flow's messages should be received in the order they were originally queued by the sender (that is, in ascending sequence number order); a value of one indicates a hint that the flow's messages should be received in the order they arrive in the network, even if there are sequence number gaps or reordering. Network arrival order is typically hinted for live,

delay-sensitive flows, such as for audio media. To take delivery of a message as soon as it arrives in the network: receive it from the receiving flow's RECV_BUFFER as soon as it becomes complete (Section 3.6.3.3 of RFC 7016), and remove it from the RECV_BUFFER. Section 3.6.3.3 of RFC 7016 describes how to take delivery of messages in original queuing order.

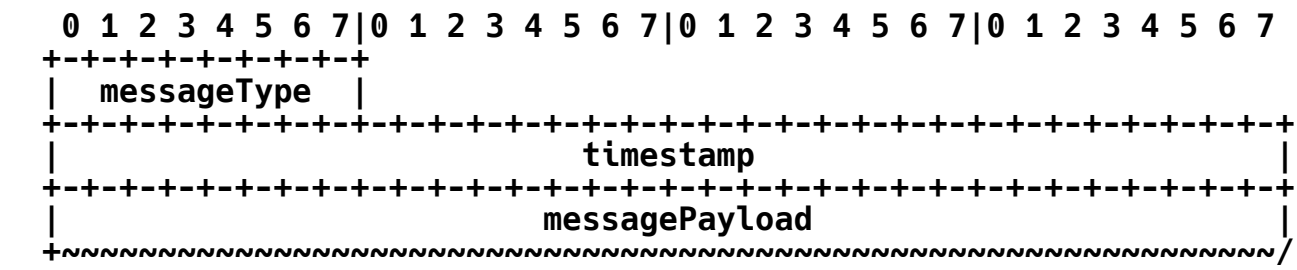
streamID: If the streamIDPresent flag is set, this field is present and is the RTMP stream ID to which the messages in this flow belong. In this profile, this field MUST be present.

A receiver SHOULD reject an RTMP message flow if its streamIDPresent flag is clear. This profile doesn't define a stream mapping for this case.

Derived or composed profiles can define additional flow types and corresponding metadata signatures. A receiver SHOULD reject a flow having an unrecognized metadata signature.

5.1.2. Message Mapping

This section describes the format of an RTMP message (Section 5.1) in an RTMFP flow.




```
struct RTMPMessage_t
{
    uint8_t  messageType;
    uint32_t timestamp;
    uint8_t  messagePayload[remainder()];
} :flowMessageLength*8;
```

messageType: The RTMP Message Type;

timestamp: The RTMP Timestamp, in network byte order;

messagePayload: The payload of the RTMP message;

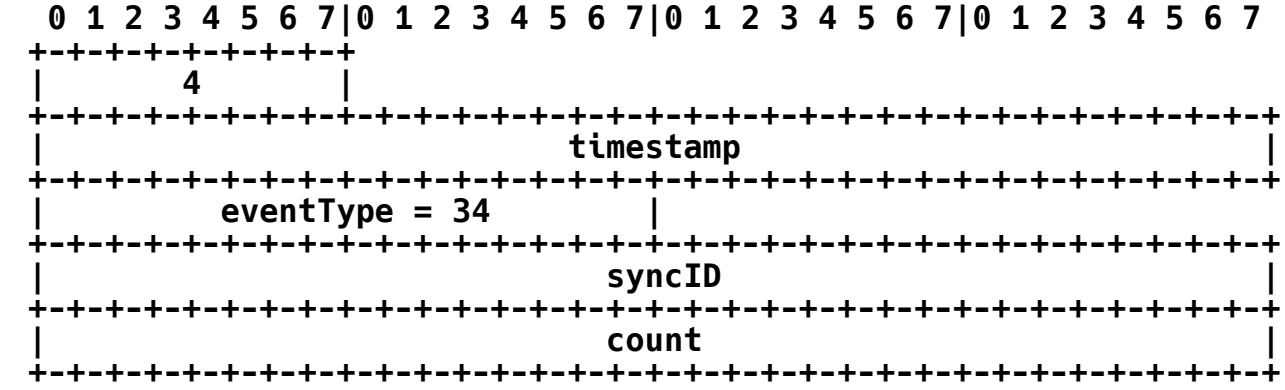
payload length: The RTMP message payload length is inferred from the length of the RTMFP message;

Stream ID: The Stream ID for this message is taken from the metadata of the flow on which this message was received.

5.2. Flow Synchronization

RTMFP flows are independent and have no inter-flow ordering guarantee. RTMP was designed for transport over a single, reliable, strictly ordered byte stream. Some RTMP message semantics take advantage of this ordering; for example, a Stream EOF User Control event must not be processed until after all media messages for the corresponding stream have been received. Flow Synchronization messages provide a barrier to align message delivery across flows when required by RTMP semantics.

A Flow Synchronization message is coded as a User Control event message (Type 4) having Event Type 34. Message timestamps are ignored and MAY be set to 0.




```
struct flowSyncUserControlMessagePayload_t
{
    uint16_t eventType = 34;
    uint32_t syncID;
    uint32_t count;
} :10*8;
```

eventType: The RTMP User Control Message Event Type. Flow Synchronization messages have type 34 (0x22);

syncID: The identifier for this barrier;

count: The number of flows being synchronized by syncID. This field MUST be at least 1 and SHOULD be at least 2.

On receipt of a Flow Synchronization message, a receiver SHOULD suspend receipt of further messages on that flow until count Flow Synchronization messages (including this one) with the same syncID have been received on flows in the same flow association tree.

Example: Consider flows F1 and F2 in the same NetConnection carrying messages M, and let Sync(syncID,count) denote a Flow Synchronization message.

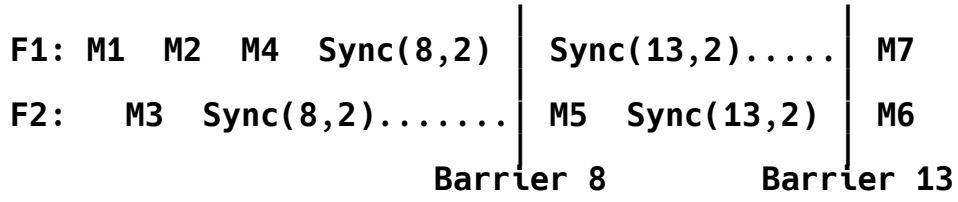


Figure 2: Example Flow Synchronization Barriers

Flow Synchronization messages form a delivery barrier to impart at least a partial message ordering across flows. In this example, message M5 comes after M1..4 and before M6..7; however, M3 could be delivered before or after any of M1, M2, or M4, and M6 could come before or after M7.

Flow Synchronization can cause a priority inversion; therefore, it SHOULD NOT be used except when necessary to preserve RTMP ordering semantics.

5.3. Client-to-Server Connection

The client connects to a server. The connection comprises one main control flow in each direction from client to server and from server to client for NetConnection messages, and zero or more flows in each direction for NetStream media messages. NetStream flows may come and go naturally over time according to media transport needs. An exception on a NetConnection control sending flow indicates the closure by the other end of the NetConnection and all associated NetStreams.

The client MUST NOT use the same client certificate for more than one server connection; that is, a client's peer ID MUST NOT be reused.

5.3.1. Connecting

The client desires a connection to a server having an RTMFP URI, for example, "rtmfp://server.example.com/app/instance". The client gathers one or more initial candidate addresses for the server named in the URI (for example, by using the Domain Name System (DNS)

[RFC1035]). The client creates an EPD having an Ancillary Data option (Section 4.4.2.2) encoding the URI. The client initiates an RTMFP session to the one or more candidate addresses using the EPD.

When the session transitions to the S_OPEN state, the client opens a new flow in that session for Stream ID 0 and Receive Intent 0 "original queuing order". This is the client's NetConnection main control flow. The client sends an RTMP "connect" command on the flow and waits for a response or exception.

5.3.2. Server-to-Client Return Control Flow

The server, on accepting the client's NetConnection control flow, and receiving and accepting the "connect" command, opens one or more return flows to the client having Stream ID 0 and associated to the control flow from the client. Flows for Stream ID 0 are the server's NetConnection control flows. The server sends a "_result" or "_error" transaction response for the client's connect command.

When the client receives the first return flow from the server for Stream ID 0 and associated to the client's NetConnection control flow, the client assumes that flow is the canonical return NetConnection control flow from the server, to which all new client-to-server flows should be associated.

On receipt of a "_result" transaction response on Stream ID 0 for the client's connect command, the connection is up.

The client MAY open additional return control flows to the server on Stream ID 0, associated to the server's canonical NetConnection control flow.

5.3.3. setPeerInfo Command

The "setPeerInfo" command is sent by the client to the server over the NetConnection control flow to inform the server of candidate socket addresses through which the client might be reachable. This list SHOULD include all directly connected interface addresses and proxy addresses except as provided below. The list MAY be empty. The list need not include the address of the server, even if the server is to act as an introducer for the client. The list SHOULD NOT include link-local or loopback addresses.

This command is sent as a regular RTMP NetConnection command; that is, as an RTMP Type 20 Command Message or an RTMP Type 17 Command Extended Message on Stream ID 0. A Type 20 Command Message SHOULD be used if the object encoding negotiated during the "connect" and

"_result" handshake is AMF0 [AMF0], and a Type 17 Command Extended Message SHOULD be used if the negotiated object encoding is AMF3 [AMF3].

Note: A Type 20 Command Message payload is a sequence of AMF objects encoded in AMF0.

Note: A Type 17 Command Extended Message payload begins with a format selector byte, followed by a sequence of objects in a format-specific encoding. At the time of writing, only format 0 is defined; therefore, the format selector byte MUST be 0. Format 0 is a sequence of AMF objects, each encoded in AMF0 by default; AMF3 encoding for an object can be selected by prefixing it with an "avmplus-object-marker" (0x11) as defined in [AMF0].

To complete the RTMFP NetConnection handshake, an RTMFP client MUST send a setPeerInfo command to the server after receiving a successful response to the "connect" command.

```
(
    "setPeerInfo", // AMF String, command name
    0.0,  // AMF Number, transaction ID
    NULL, // AMF Null, no command object
    ...   // zero or more AMF Strings, each an address
)
```

Each listed socket address includes an IPv4 or IPv6 address in presentation format and a UDP port number in decimal, separated by a colon. Since the IPv6 address presentation format uses colons, IPv6 addresses are enclosed in square brackets [RFC3986].

```
(
    "setPeerInfo",
    0.0,
    NULL,
    "192.0.2.129:50001",
    "[2001:db8:1::2]:50002"
)
```

Figure 3: Example setPeerInfo Command

A server SHOULD assume that the client is behind a Network Address Translator (NAT) if and only if the observed far endpoint address of the session for the flow on which this command was received does not appear in the setPeerInfo address list.

5.3.4. Set Keepalive Timers Command

The server can advise the client to set or change the client's session keepalive timer periods for its connection to the server and for its P2P connections. The server MAY choose keepalive periods based on static configuration, application- or deployment-specific circumstances, whether the client appears to be behind a NAT, or for any other reason.

The Set Keepalive Timers command is sent by the server to the client on Stream ID 0 as a User Control event message (Type 4) having Event Type 41. Message timestamps are ignored and MAY be set to 0.


```
struct setKeepaliveUserControlMessagePayload_t
{
    uint16_t eventType = 41;
    uint32_t serverKeepalivePeriodMsec;
    uint32_t peerKeepalivePeriodMsec;
} :10*8;
```

eventType: The RTMP User Control Message Event Type. Set Keepalive Timers messages have type 41 (0x29);

serverKeepalivePeriodMsec: The keepalive period, in milliseconds, that the client is advised to set on its RTMFP session with the server;

peerKeepalivePeriodMsec: The keepalive period, in milliseconds, that the client is advised to use on its RTMFP sessions with any peer that is not the server.

The client MUST define minimum values for these keepalive periods, below which it will not set them, regardless of the values in this message. The minimum keepalive timer periods SHOULD be at least five seconds. The client MAY define maximum values for these keepalive periods, above which it will not set them.

On receipt of this message from the server, a client SHOULD set its RTMFP server and peer keepalive timer periods to the indicated values subject to the client's minimum and maximum values. The server MAY send this message more than once, particularly if conditions that it uses to determine the timer periods change.

5.3.5. Additional Flows for Streams

The client or server opens additional flows to the other side to carry messages for any stream. Additional flows are associated to the canonical NetConnection control flow from the other side.

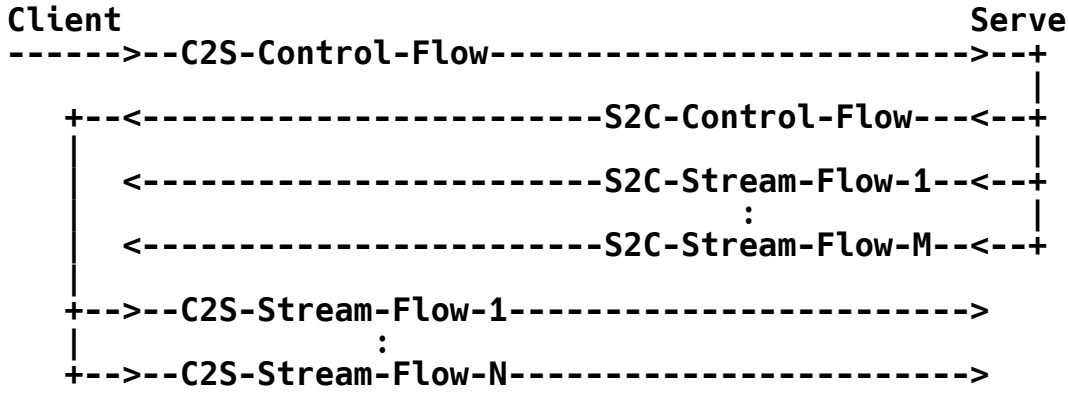


Figure 4: Schematic Flow Association Tree for a NetConnection

5.3.5.1. To Server

Additional flows from the client to the server for stream messages are opened with the Stream ID for that stream and associated in return to the server's canonical NetConnection control flow.

The client MAY create as many flows as desired for any Stream ID (including Stream ID 0) at any time.

5.3.5.2. From Server

Additional flows from the server to the client for stream messages are opened with the Stream ID for that stream, and associated in return to the client's NetConnection control flow.

The server MAY create as many flows as desired for any Stream ID (including Stream ID 0) at any time.

5.3.5.3. Closing Stream Flows

Either end MAY close a sending flow that is not for Stream ID 0 at any time with no semantic meaning for the stream.

At any time, either end MAY reject a receiving flow that is not one of the other end's NetConnection control flows. No flow exception codes are defined by this profile, so the receiving end SHOULD use exception code 0 when rejecting the flow. The sending end, on notification of any exception for a stream flow, SHOULD NOT open a new flow to take the rejected flow's place for transport of messages for that stream. If an end rejects any flow for a stream, it SHOULD reject all the flows for that stream, otherwise Flow Synchronization messages (Section 5.2) that were in flight could be discarded and some flows might become or remain stuck in a suspended state.

5.3.6. Closing the Connection

The client or server can signal an orderly close of the connection by closing its NetConnection control sending flows and all stream sending flows. The other end, on receiving a close/complete notification for the canonical NetConnection control receiving flow, closes its sending flows. When both ends observe all receiving flows have closed and completed, the connection has cleanly terminated.

Either end can abruptly terminate the connection by rejecting the NetConnection control receiving flows or by closing the underlying RTMFP session. On notification of any exception on a NetConnection control sending flow, the end seeing the exception knows the other end has terminated abruptly, and can immediately close all sending and receiving flows for that connection.

5.3.7. Example

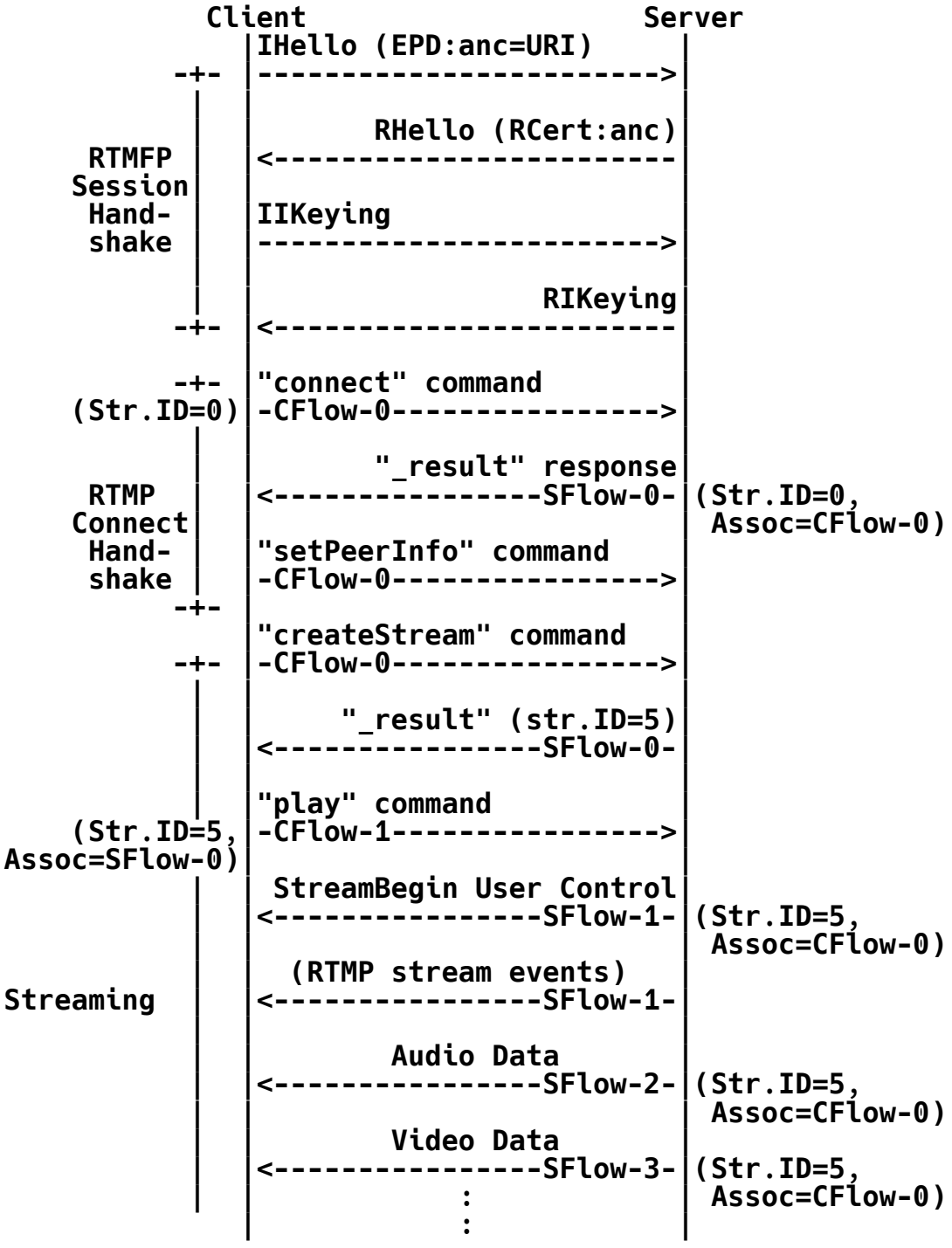


Figure 5: Example NetConnection Message Exchange

5.4. Direct Peer-to-Peer Streams

Clients can connect directly to other clients for P2P streaming and data exchange. A client MAY have multiple separate P2P NetStreams with a peer in one RTMFP session, each a separate logical connection. P2P NetStreams are unidirectional, initiated by a subscriber (the side issuing the "play" command) to a publisher. The subscribing peer has a control flow to the publisher. The publisher has zero or more return flows to the subscriber associated to the subscriber's control flow, for the stream media and data.

5.4.1. Connecting

A client desires to subscribe directly to a stream being published in P2P mode by a publishing peer. The client learns the peer ID of the publisher and the stream name through application-specific means.

If the client does not already have an RTMFP session with that peer ID, it initiates a new session, creating an EPD containing a Fingerprint option (Section 4.4.2.3) for the publisher's peer ID and using the server session's DESTADDR as the initial candidate address for the session to the peer. The server acts as an Introducer (Section 3.5.1.6 of RFC 7016), using forward and redirect messages to help the client and the peer establish a session.

When an S_OPEN session exists to the desired peer, the client creates a new independent flow to that peer. The flow MUST have a non-zero Stream ID. The client sends an RTMP "play" command over the flow, giving the name of the desired stream at the publisher. This flow is the subscriber's control flow.

5.4.2. Return Flows for Stream

The publisher, on accepting a new flow not indicating a return association with any of its sending flows and having a non-zero Stream ID, receives and processes the "play" command. If and when the request is acceptable to the publisher, it opens one or more return flows to the subscribing peer, associated to the subscriber's control flow and having the same Stream ID. The publisher sends a StreamBegin User Control message, appropriate RTMP status events, and the stream media over the one or more return flows.

The subscriber uses the return association of the media flows to the subscriber control flow to determine the stream to which the media belongs.

The publisher MAY open any number of media flows for the stream and close them at any time. The opening and closing of media flows has no semantic meaning for the stream, except that the opening of at least one flow and the reception of at least one media message or a StreamBegin User Control message indicates that the publisher is publishing the requested stream to the subscriber.

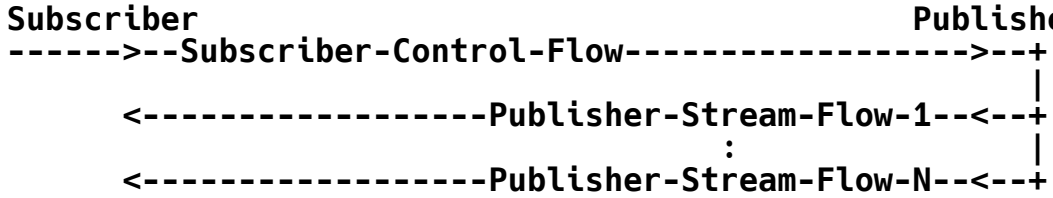


Figure 6: Schematic Flow Association Tree for a P2P Direct Connection

5.4.3. Closing the Connection

Either end can close the stream by closing or rejecting the subscriber's control flow. The publisher SHOULD close and unpublish to the subscriber on receipt of a close/complete of the control flow. The subscriber SHOULD consider the stream closed on notification of any exception on the control flow.

6. IANA Considerations

This memo specifies option type code values for Certificate fields (Section 4.3.3), Endpoint Discriminator fields (Section 4.4.2), and Session Keying Component fields (Section 4.5.2). It also specifies a flow metadata signature (Section 5.1.1). The type code values and signatures for this profile are assigned and maintained by Adobe, and therefore require no action from IANA.

6.1. RTMFP URI Scheme Registration

This memo describes use of an RTMFP URI scheme (Section 4.4.2.2, Section 5.3.1, Figure 5). Per this section, the "rtmfp" URI scheme has been registered by IANA.

The syntax and semantics of this URI scheme are described using the Augmented Backus-Naur Form (ABNF) [RFC5234] rules from RFC 3986.

URI scheme name: rtmfp

Status: provisional

URI scheme syntax:

```
rtmfp-uri-scheme = "rtmfp:"
                 / "rtmfp://" host [ ":" port ] path-abempty
```

URI scheme semantics: The first form is used in the APIs of some implementations to indicate instantiation of an RTMFP client according to this memo, but without connecting to a server. Such an instantiation might be used for pure peer-to-peer communication.

The second form provides location information for the server to which to connect and optional additional information to pass to the server. The only operation for this URI form is to connect to a server (initial candidate address(es) for which are named by host and port) according to Section 5.3. The UDP port for initial candidate addresses, if not specified, is 1935. If the host is a reg-name, the initial candidate address set SHOULD comprise all IPv4 and IPv6 addresses to which reg-name resolves. The semantics of path-abempty are specific to the server. Connections are made using RTMFP as specified by this memo.

Encoding considerations: The path-abempty component represents textual data consisting of characters from the Universal Character Set. This component SHOULD be encoded according to Section 2.5 of RFC 3986.

Applications/protocols that use this URI scheme name: The Flash runtime (including Flash Player) from Adobe Systems Incorporated, communication servers such as Adobe Media Server, and interoperable clients and servers provided by other parties, using RTMFP according to this memo.

Interoperability considerations: This scheme requires use of RTMFP as defined by RFC 7016 in the manner described by this memo.

Security considerations: See Security Considerations (Section 7) in this memo.

Contact: Michael Thornburgh, Adobe Systems Incorporated, <mthornbu@adobe.com>.

Author/Change controller: Michael Thornburgh, Adobe Systems Incorporated, <mthornbu@adobe.com>.

References:
Thornburgh, M., "Adobe's Secure Real-Time Media Flow Protocol", RFC 7016, November 2013.

This memo.

7. Security Considerations

Section 4 details the cryptographic aspects of this profile.

This profile does not define or use a Public Key Infrastructure (PKI). Clients SHOULD use static Diffie-Hellman keys in their certificates (Section 4.3.3.5). Clients MUST create a new certificate with a distinct fingerprint for each new NetConnection (Section 5.3). These constraints make client identities ephemeral but unable to be forged. A man-in-the-middle cannot successfully interpose itself in a connection to a target client addressed by its fingerprint/peer ID if the target client uses a static Diffie-Hellman public key.

Servers can have long-lived RTMFP instances, so they SHOULD use ephemeral Diffie-Hellman public keys for forward secrecy. This allows server peer IDs to be forged; however, clients do not connect to servers by peer ID, so this is irrelevant.

When a client connects to a server, the client will accept the response of any endpoint claiming to be "a server". It is assumed that an attacker that can passively observe traffic on a network segment can also inject its own packets with any source or destination and any payload. An attacker can trick a client into connecting to a rogue server or man-in-the-middle, either by observing Initiator Hello packets from the client and responding earliest with a matching Responder Hello or by using tricks such as DNS spoofing or poisoning to direct a client to connect directly to the rogue. A TCP-based transport would be vulnerable to similar attacks. Since there is no PKI, this profile gives no guarantee that the client has actually connected to the desired server, versus a rogue or man-in-the-middle. In circumstances where assurance is required that the connection is directly to the desired server, the client can use the Session Nonces (Section 4.6.5) to challenge the server, for example, over a different channel having acceptable security properties (such as an HTTPS) to transitively establish the server's identity and verify that the end-to-end communication is private and authentic.

When session sequence numbers (Section 4.7.3.3) are not used, it is possible for an attacker to use traffic analysis techniques and record encrypted packets containing the start of a new flow, and

later to replay those packets after the flow has closed, which can look to the receiver like a brand new flow. In circumstances where this can be detrimental, session sequence numbers SHOULD be used. Replay of packets for existing flows is not detrimental as the receiver detects and discards duplicate flow sequence numbers, and flow sequence numbers do not wrap or otherwise repeat.

Packet encryption uses CBC with the same (null) initialization vector for each packet. This can reveal to an observer whether two packets contain identical plaintext. However, the maximum-length RTMFP common header and User Data or Data Acknowledgement header, including flow sequence number, always fit within the first 16-byte cipher block, so each initial cipher block for most packets will already be unique even if timestamps are suppressed. Sending identical messages in a flow uses unique flow sequence numbers, so cipher blocks will be unique in this case. Keepalive pings and retransmission of lost data can result in identical cipher blocks; however, traffic analysis can also reveal likely keepalives or retransmissions, and retransmission only occurs as a result of observable network loss, so this is usually irrelevant. In circumstances where any identical cipher block is unacceptable, session sequence numbers SHOULD be used as they guarantee each initial cipher block will be unique.

Packet verification can use a 16-bit simple checksum (Section 4.7.3.1). The checksum is inside the encrypted packet, so for external packet modifications the checksum is equivalent to a 16-bit cryptographic digest. In circumstances where this is insufficient, HMAC verification (Section 4.7.3.2) SHOULD be used.

8. References

8.1. Normative References

[AES] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)", FIPS PUB 197, November 2001, <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.

[AMF0] Adobe Systems Incorporated, "Action Message Format -- AMF 0", December 2007, <http://www.adobe.com/go/spec_amf0>.

[AMF3] Adobe Systems Incorporated, "Action Message Format -- AMF 3", January 2013, <http://www.adobe.com/go/spec_amf3>.

[CBC] Dworkin, M., "Recommendation for Block Cipher Modes of Operation", NIST Special Publication 800-38A, December 2001, <http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf>.

[DH] Diffie, W. and M. Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory, V. IT-22, n. 6, June 1977.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997, <http://www.rfc-editor.org/info/rfc2104>.

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

[RFC3526] Kivinen, T. and M. Kojo, "More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)", RFC 3526, May 2003, <http://www.rfc-editor.org/info/rfc3526>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003, <http://www.rfc-editor.org/info/rfc3629>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005, <http://www.rfc-editor.org/info/rfc3986>.

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008, <http://www.rfc-editor.org/info/rfc5234>.

[RFC6234] Eastlake, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, May 2011, <http://www.rfc-editor.org/info/rfc6234>.

[RFC7016] Thornburgh, M., "Adobe's Secure Real-Time Media Flow Protocol", RFC 7016, November 2013, <http://www.rfc-editor.org/info/rfc7016>.

[RFC7296] Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., and T. Kivinen, "Internet Key Exchange Protocol Version 2 (IKEv2)", STD 79, RFC 7296, October 2014, <http://www.rfc-editor.org/info/rfc7296>.

[RTMP] Adobe Systems Incorporated, "Real-Time Messaging Protocol (RTMP) specification", December 2012, <http://www.adobe.com/go/spec_rtmp>.

[SHA256] National Institute of Standards and Technology, "Secure Hash Standard", FIPS PUB 180-4, March 2012, <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>.

8.2. Informative References

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, November 1987, <http://www.rfc-editor.org/info/rfc1035>.

[RFC1071] Braden, R., Borman, D., Partridge, C., and W. Plummer, "Computing the Internet checksum", RFC 1071, September 1988, <http://www.rfc-editor.org/info/rfc1071>.

[RFC4302] Kent, S., "IP Authentication Header", RFC 4302, December 2005, <http://www.rfc-editor.org/info/rfc4302>.

[RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, December 2005, <http://www.rfc-editor.org/info/rfc4303>.

[RFC6479] Zhang, X. and T. Tsou, "IPsec Anti-Replay Algorithm without Bit Shifting", RFC 6479, January 2012, <http://www.rfc-editor.org/info/rfc6479>.

Acknowledgements

Special thanks go to Glenn Eguchi, Matthew Kaufman, and Adam Lane for their contributions to the design of this profile.

Thanks to Philipp Hancke, Kevin Igoe, Paul Kyzivat, and Milos Trboljevac for their detailed reviews of this memo.

Author's Address

Michael C. Thornburgh
Adobe Systems Incorporated
345 Park Avenue
San Jose, CA 95110-2704
United States

Phone: +1 408 536 6000
EMail: mthornbu@adobe.com
URI: http://www.adobe.com/