Network Working Group
Request for Comments: 4758
Category: Informational

M. Nystroem
RSA Security

November 2006

Cryptographic Token Key Initialization Protocol (CT-KIP) Version 1.0 Revision 1

Status of This Memo

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

Copyright Notice

Abstract

This document constitutes Revision 1 of Cryptographic Token Key Initialization Protocol (CT-KIP) Version 1.0 from RSA Laboratories' One-Time Password Specifications (OTPS) series. The body of this document, except for the intellectual property considerations section, is taken from the CT-KIP Version 1.0 document, but comments received during the IETF review are reflected; hence, the status of a revised version. As no "bits-on-the-wire" have changed, the protocol specified herein is compatible with CT-KIP Version 1.0.

CT-KIP is a client-server protocol for initialization (and configuration) of cryptographic tokens. The protocol requires neither private-key capabilities in the cryptographic tokens, nor an established public-key infrastructure. Provisioned (or generated) secrets will only be available to the server and the cryptographic token itself.

Table of Contents

1. Introduction

Note: This document is Revision 1 of CT-KIP Version 1.0 [12] from RSA Laboratories' OTPS series.

1.1. Scope

This document describes a client-server protocol for initialization (and configuration) of cryptographic tokens. The protocol requires neither private-key capabilities in the cryptographic tokens, nor an established public-key infrastructure.

The objectives of this protocol are:

- To provide a secure method of initializing cryptographic tokens with secret keys without exposing generated, secret material to any other entities than the server and the cryptographic token itself,
- To avoid, as much as possible, any impact on existing cryptographic token manufacturing processes,
- To provide a solution that is easy to administer and scales well.

The mechanism is intended for general use within computer and communications systems employing connected cryptographic tokens (or software emulations thereof).

1.2. Background

A cryptographic token may be a handheld hardware device, a hardware device connected to a personal computer through an electronic interface such as USB, or a software module resident on a personal computer, which offers cryptographic functionality that may be used, e.g., to authenticate a user towards some service. Increasingly, these tokens work in a connected fashion, enabling their programmatic initialization as well as programmatic retrieval of their output values. This document intends to meet the need for an open and interoperable mechanism to programmatically initialize and configure connected cryptographic tokens. A companion document entitled "A PKCS #11 Mechanism for the Cryptographic Token Key Initialization Protocol" [2] describes an application-programming interface suitable for use with this mechanism.

1.3. Document Organization

The organization of this document is as follows:

- Section 1 is an introduction.
- Section 2 defines some notation used in this document.
- Section 3 defines the protocol mechanism in detail.
- Section 4 defines a binding of the protocol to transports.
- Section 5 provides security considerations.
- Appendix A defines the XML schema for the protocol mechanism, Appendix B gives example messages, and Appendix C discusses integration with PKCS #11 [3].
- Appendix D provides example realizations of an abstract pseudorandom function defined in Section 3.
- Appendix E provides general information about the One-Time Password Specifications.

2. Acronyms and Notation

2.1. Acronyms

MAC	Message Authentication Code
PDU	Protocol Data Unit
PRF	Pseudo-Random Function
CT-KIP	Cryptographic Token Key Initialization Protocol (the protocol mechanism described herein)

2.2. Notation

||	String concatenation
[x]	Optional element x
A ^ B	Exclusive-or operation on strings A and B (A and B of equal length)
K_AUTH	Secret key used for authentication purposes

K_TOKEN Secret key used for token computations, generated in CT-KIP

K_SERVER Public key of CT-KIP server

K_SHARED Secret key shared between the cryptographic token and the CT-KIP server

K Key used to encrypt R_C (either K_SERVER or K_SHARED)

R Pseudorandom value chosen by the cryptographic token and used for MAC computations

R_C Pseudorandom value chosen by the cryptographic token

R_S Pseudorandom value chosen by the CT-KIP server

The following typographical convention is used in the body of the text: <XMLElement>.

3. CT-KIP

3.1. Overview

The CT-KIP is a client-server protocol for the secure initialization of cryptographic tokens. The protocol is meant to provide high assurance for both the server and the client (cryptographic token) that generated keys have been correctly and randomly generated and not exposed to other entities. The protocol does not require the existence of a public-key infrastructure.

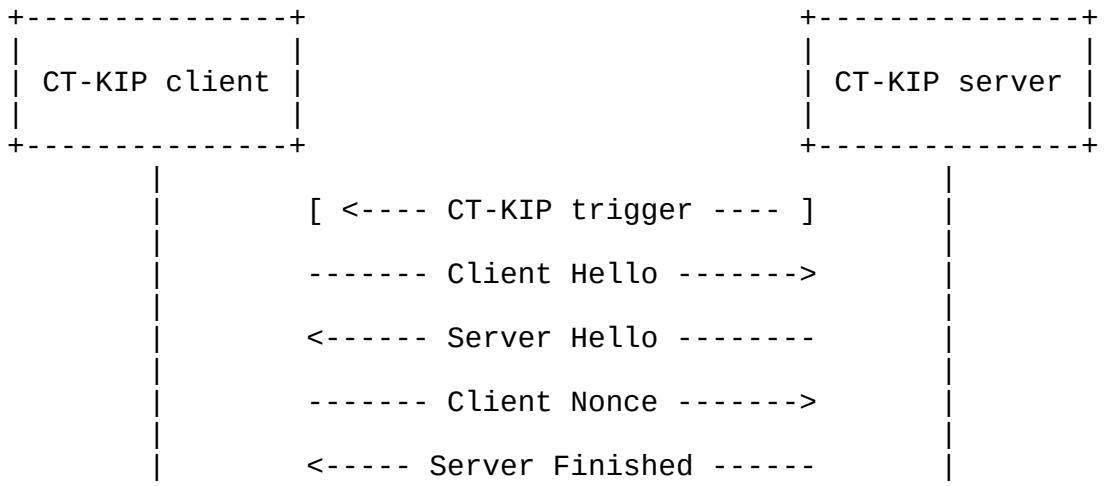

Figure 1: The 4-pass CT-KIP protocol (with optional preceding trigger)

3.2. Entities

In principle, the protocol involves a CT-KIP client and a CT-KIP server.

It is assumed that a desktop/laptop or a wireless device (e.g., a mobile phone or a PDA) will host an application communicating with the CT-KIP server as well as the cryptographic token, and collectively, the cryptographic token and the communicating application form the CT-KIP client. When there is a need to point out if an action is to be performed by the communicating application or by the token the text will make this explicit.

The manner in which the communicating application will transfer CT-KIP protocol elements to and from the cryptographic token is transparent to the CT-KIP server. One method for this transfer is described in [2].

3.3. Principles of Operation

To initiate a CT-KIP session, a user may use a browser to connect to a web server running on some host. The user may then identify (and authenticate) herself (through some means that essentially are out of scope for this document) and possibly indicate how the CT-KIP client shall contact the CT-KIP server. There are also other alternatives for CT-KIP session initiation, such as the CT-KIP client being pre-configured to contact a certain CT-KIP server, or the user being informed out-of-band about the location of the CT-KIP server. In any event, once the location of the CT-KIP server is known, the CT-KIP client and the CT-KIP server engage in a 4-pass protocol in which:

a. The CT-KIP client provides information to the CT-KIP server about the cryptographic token’s identity, supported CT-KIP versions, cryptographic algorithms supported by the token and for which keys may be generated using this protocol, and encryption and MAC algorithms supported by the cryptographic token for the purposes of this protocol.

b. Based on this information, the CT-KIP server provides a random nonce, R_S, to the CT-KIP client, along with information about the type of key to generate, the encryption algorithm chosen to protect sensitive data sent in the protocol. In addition, it provides either information about a shared secret key to use for encrypting the cryptographic token’s random nonce (see below), or its own public key. The length of the nonce R_S may depend on the selected key type.

c. The cryptographic token generates a random nonce R_C and encrypts it using the selected encryption algorithm and with a key K that is either the CT-KIP server's public key K_SERVER, or a shared secret key K_SHARED as indicated by the CT-KIP server. The length of the nonce R_C may depend on the selected key type. The CT-KIP client then sends the encrypted random nonce to the CT-KIP server. The token also calculates a cryptographic key K_TOKEN of the selected type from the combination of the two random nonces R_S and R_C, the encryption key K, and possibly some other data, using the CT-KIP-PRF function defined herein.

d. The CT-KIP server decrypts R_C, calculates K_TOKEN from the combination of the two random nonces R_S and R_C, the encryption key K, and possibly some other data, using the CT-KIP-PRF function defined herein. The server then associates K_TOKEN with the cryptographic token in a server-side data store. The intent is that the data store later on will be used by some service that needs to verify or decrypt data produced by the cryptographic token and the key.

e. Once the association has been made, the CT-KIP server sends a confirmation message to the CT-KIP client. The confirmation message includes an identifier for the generated key and may also contain additional configuration information, e.g., the identity of the CT-KIP server.

f. Upon receipt of the CT-KIP server's confirmation message, the cryptographic token associates the provided key identifier with the generated key K_TOKEN, and stores the provided configuration data, if any.

Note: Conceptually, although R_C is one pseudorandom string, it may be viewed as consisting of two components, R_C1 and R_C2, where R_C1 is generated during the protocol run, and R_C2 can be generated at the cryptographic token manufacturing time and stored in the cryptographic token. In that case, the latter string, R_C2, should be unique for each cryptographic token for a given manufacturer.

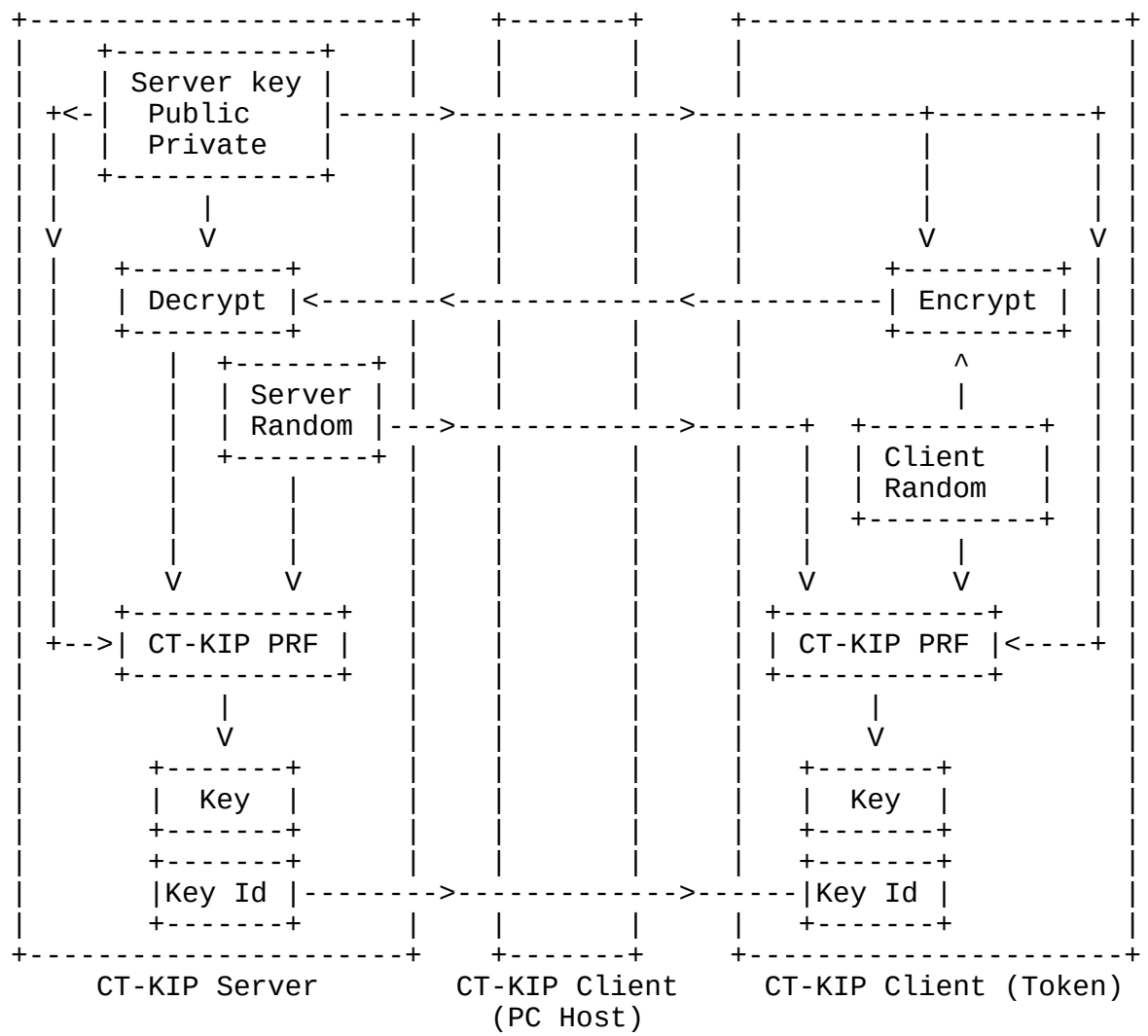

Figure 2: Principal data flow for CT-KIP key generation - using public server key

The inclusion of the two random nonces R_S and R_C in the key generation provides assurance to both sides (the token and the CT-KIP server) that they have contributed to the key's randomness and that the key is unique. The inclusion of the encryption key K ensures that no man-in-the-middle may be present, or else the cryptographic token will end up with a key different from the one stored by the legitimate CT-KIP server.

Note: A man-in-the middle (in the form of corrupt client software or a mistakenly contacted server) may present his own public key to the token. This will enable the attacker to learn the client's version

of K_TOKEN. However, the attacker is not able to persuade the legitimate server to derive the same value for K_TOKEN, since K_TOKEN is a function of the public key involved, and the attacker's public key must be different than the correct server's (or else the attacker would not be able to decrypt the information received from the client). Therefore, once the attacker is no longer "in the middle", the client and server will detect that they are "out of synch" when they try to use their keys. Therefore, in the case of encrypting R_C with K_SERVER, it is important to verify that K_SERVER really is the legitimate server's key. One way to do this is to independently validate a newly generated K_TOKEN against some validation service at the server (e.g., by using a connection independent from the one used for the key generation).

The CT-KIP server may couple an initial user authentication to the CT-KIP execution in several ways to ensure that a generated K_TOKEN ends up associated with the correct token and user. One way is to provide a one-time value to the user or CT-KIP client after successful user authentication and require this value to be used when contacting the CT-KIP service (in effect coupling the user authentication with the subsequent CT-KIP protocol run). This value could, for example, be placed in a <TriggerNonce> element of the CT-KIP initialization trigger (if triggers are used; see Section 4.2.7). Another way is for the user to provide a token identifier which will later be used in the CT-KIP protocol to the server during the authentication phase. The server may then include this token identifier in the CT-KIP initialization trigger. It is also legitimate for a CT-KIP client to initiate a CT-KIP protocol run without having received an initialization message from a server, but in this case any provided token identifier shall not be accepted by the server unless the server has access to a unique token key for the identified token and that key will be used in the protocol. Whatever the method, the CT-KIP server must ensure that a generated key is associated with the correct token and, if applicable, the correct user. For a further discussion of this and threats related to man-in-the-middle attacks in this context, see Section 5.5.

3.4. The CT-KIP One-Way Pseudorandom Function, CT-KIP-PRF

3.4.1. Introduction

The general requirements on CT-KIP-PRF are the same as on keyed hash functions: It shall take an arbitrary length input, and be one-way and collision-free (for a definition of these terms, see, e.g., [4]). Further, the CT-KIP-PRF function shall be capable of generating a variable-length output, and its output shall be unpredictable even if other outputs for the same key are known.

It is assumed that any realization of CT-KIP-PRF takes three input parameters: A secret key k, some combination of variable data, and the desired length of the output. Examples of the variable data include, but are not limited to, a current token counter value, the current token time, and a challenge. The combination of variable data can, without loss of generalization, be considered as a salt value (see PKCS #5 Version 2.0 [5], Section 4), and this characterization of CT-KIP-PRF should fit all actual PRF algorithms implemented by tokens. From the point of view of this specification, CT-KIP-PRF is a "black-box" function that, given the inputs, generates a pseudorandom value.

Separate specifications may define the implementation of CT-KIP-PRF for various types of cryptographic tokens. Appendix D contains two example realizations of CT-KIP-PRF.

3.4.2. Declaration

CT-KIP-PRF (k, s, dsLen)

Input:

k secret key in octet string format

s octet string of varying length consisting of variable data distinguishing the particular string being derived

dsLen desired length of the output

Output:

DS pseudorandom string, dsLen-octets long

For the purposes of this document, the secret key k shall be 16 octets long.

3.5. Generation of Cryptographic Keys for Tokens

In CT-KIP, keys are generated using the CT-KIP-PRF function, a secret random value R_C chosen by the CT-KIP client, a random value R_S chosen by the CT-KIP server, and the key k used to encrypt R_C. The input parameter s of CT-KIP-PRF is set to the concatenation of the (ASCII) string "Key generation", k, and R_S, and the input parameter dsLen is set to the desired length of the key, K_TOKEN (the length of K_TOKEN is given by the key’s type):

dsLen = (desired length of K_TOKEN)

K_TOKEN = CT-KIP-PRF (R_C, "Key generation" || k || R_S, dsLen)

When computing K_TOKEN above, the output of CT-KIP-PRF may be subject to an algorithm-dependent transform before being adopted as a key of the selected type. One example of this is the need for parity in DES keys.

3.6. Encryption of Pseudorandom Nonces Sent from the CT-KIP Client

CT-KIP client random nonce(s) are either encrypted with the public key provided by the CT-KIP server or by a shared secret key. For example, in the case of a public RSA key, an RSA encryption scheme from PKCS #1 [6] may be used.

In the case of a shared secret key, to avoid dependence on other algorithms, the CT-KIP client may use the CT-KIP-PRF function described herein with the shared secret key K_SHARED as input parameter k (in this case, K_SHARED should be used solely for this purpose), the concatenation of the (ASCII) string "Encryption" and the server's nonce R_S as input parameter s, and dsLen set to the length of R_C:

dsLen = len(R_C)

DS = CT-KIP-PRF(K_SHARED, "Encryption" || R_S, dsLen)

This will produce a pseudorandom string DS of length equal to R_C. Encryption of R_C may then be achieved by XOR-ing DS with R_C:

Enc-R_C = DS ^ R_C

The CT-KIP server will then perform the reverse operation to extract R_C from Enc-R_C.

Note: It may appear that an attacker, who learns a previous value of R_C, may be able to replay the corresponding R_S and, hence, learn a new R_C as well. However, this attack is mitigated by the requirement for a server to show knowledge of K_AUTH (see below) in order to successfully complete a key re-generation.

3.7. CT-KIP Schema Basics

3.7.1. Introduction

Core parts of the XML schema for CT-KIP, found in Appendix A, are explained in this section. Specific protocol message elements are defined in Section 3.8. Examples can be found in Appendix B.

The XML format for CT-KIP messages have been designed to be extensible. However, it is possible that the use of extensions will harm interoperability; therefore, any use of extensions should be carefully considered. For example, if a particular implementation relies on the presence of a proprietary extension, then it may not be able to interoperate with independent implementations that have no knowledge of this extension.

XML types defined in this sub-section are not CT-KIP messages; rather they provide building blocks that are used by CT-KIP messages.

3.7.2. General XML Schema Requirements

Some CT-KIP elements rely on the parties being able to compare received values with stored values. Unless otherwise noted, all elements in this document that have the XML Schema "xs:string" type, or a type derived from it, must be compared using an exact binary comparison. In particular, CT-KIP implementations must not depend on case-insensitive string comparisons, normalization or trimming of white space, or conversion of locale-specific formats such as numbers.

Implementations that compare values that are represented using different character encodings must use a comparison method that returns the same result as converting both values to the Unicode character encoding, Normalization Form C [1], and then performing an exact binary comparison.

No collation or sorting order for attributes or element values is defined. Therefore, CT-KIP implementations must not depend on specific sorting orders for values.

3.7.3. The AbstractRequestType Type

All CT-KIP requests are defined as extensions to the abstract AbstractRequestType type. The elements of the AbstractRequestType, therefore, apply to all CT-KIP requests. All CT-KIP requests must contain a Version attribute. For this version of this specification, Version shall be set to "1.0".

```
<xs:complexType name="AbstractRequestType" abstract="true">
  <xs:attribute name="Version" type="VersionType"
   use="required"/>
</xs:complexType>
```

3.7.4. The AbstractResponseType type

All CT-KIP responses are defined as extensions to the abstract AbstractResponseType type. The elements of the AbstractResponseType, therefore, apply to all CT-KIP responses. All CT-KIP responses contain a Version attribute indicating the version that was used. A Status attribute, which indicates whether the preceding request was successful or not must also be present. Finally, all responses may contain a SessionID attribute identifying the particular CT-KIP session. The SessionID attribute needs only be present if more than one roundtrip is required for a successful protocol run (this is the case with the protocol version described herein).

```
<xs:complexType name="AbstractResponseType" abstract="true">
  <xs:attribute name="Version" type="VersionType" use="required"/>
  <xs:attribute name="SessionID" type="IdentifierType"/>
  <xs:attribute name="Status" type="StatusCode" use="required"/>
</xs:complexType>
```

3.7.5. The StatusCode Type

The StatusCode type enumerates all possible return codes:

```
<xs:simpleType name="StatusCode">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Continue"/>
    <xs:enumeration value="Success"/>
    <xs:enumeration value="Abort"/>
    <xs:enumeration value="AccessDenied"/>
    <xs:enumeration value="MalformedRequest"/>
    <xs:enumeration value="UnknownRequest"/>
    <xs:enumeration value="UnknownCriticalExtension"/>
    <xs:enumeration value="UnsupportedVersion"/>
    <xs:enumeration value="NoSupportedKeyTypes"/>
    <xs:enumeration value="NoSupportedEncryptionAlgorithms"/>
    <xs:enumeration value="NoSupportedMACAlgorithms"/>
    <xs:enumeration value="InitializationFailed"/>
  </xs:restriction>
</xs:simpleType>
```

Upon transmission or receipt of a message for which the Status attribute's value is not "Success" or "Continue", the default behavior, unless explicitly stated otherwise below, is that both the

CT-KIP server and the CT-KIP client shall immediately terminate the CT-KIP session. CT-KIP servers and CT-KIP clients must delete any secret values generated as a result of failed runs of the CT-KIP protocol. Session identifiers may be retained from successful or failed protocol runs for replay detection purposes, but such retained identifiers shall not be reused for subsequent runs of the protocol.

When possible, the CT-KIP client should present an appropriate error message to the user.

These status codes are valid in all CT-KIP-Response messages unless explicitly stated otherwise.

- "Continue" indicates that the CT-KIP server is ready for a subsequent request from the CT-KIP client. It cannot be sent in the server's final message.

- "Success" indicates successful completion of the CT-KIP session. It can only be sent in the server's final message.

- "Abort" indicates that the CT-KIP server rejected the CT-KIP client's request for unspecified reasons.

- "AccessDenied" indicates that the CT-KIP client is not authorized to contact this CT-KIP server.

- "MalformedRequest" indicates that the CT-KIP server failed to parse the CT-KIP client's request.

- "UnknownRequest" indicates that the CT-KIP client made a request that is unknown to the CT-KIP server.

- "UnknownCriticalExtension" indicates that a critical CT-KIP extension (see below) used by the CT-KIP client was not supported or recognized by the CT-KIP server.

- "UnsupportedVersion" indicates that the CT-KIP client used a CT-KIP protocol version not supported by the CT-KIP server. This error is only valid in the CT-KIP server's first response message.

- "NoSupportedKeyTypes" indicates that the CT-KIP client only suggested key types that are not supported by the CT-KIP server. This error is only valid in the CT-KIP server's first response message. Note that the error will only occur if the CT-KIP server does not support any of the CT-KIP client's suggested key types.

- "NoSupportedEncryptionAlgorithms" indicates that the CT-KIP client only suggested encryption algorithms that are not supported by the CT-KIP server. This error is only valid in the CT-KIP server's first response message. Note that the error will only occur if the CT-KIP server does not support any of the CT-KIP client's suggested encryption algorithms.

- "NoSupportedMACAlgorithms" indicates that the CT-KIP client only suggested MAC algorithms that are not supported by the CT-KIP server. This error is only valid in the CT-KIP server's first response message. Note that the error will only occur if the CT-KIP server does not support any of the CT-KIP client's suggested MAC algorithms.

- "InitializationFailed" indicates that the CT-KIP server could not generate a valid key given the provided data. When this status code is received, the CT-KIP client should try to restart CT-KIP, as it is possible that a new run will succeed.

3.7.6. The IdentifierType Type

The IdentifierType type is used to identify various CT-KIP elements, such as sessions, users, and services. Identifiers must not be longer than 128 octets.

```
<xs:simpleType name="IdentifierType">
  <xs:restriction base="xs:string">
    <xs:maxLength value="128"/>
  </xs:restriction>
</xs:simpleType>
```

3.7.7. The NonceType Type

The NonceType type is used to carry pseudorandom values in CT-KIP messages. A nonce, as the name implies, must be used only once. For each CT-KIP message that requires a nonce element to be sent, a fresh nonce shall be generated each time. Nonce values must be at least 16 octets long.

```
<xs:simpleType name="NonceType">
  <xs:restriction base="xs:base64Binary">
    <xs:minLength value="16"/>
  </xs:restriction>
</xs:simpleType>
```

3.7.8. The ExtensionsType and the AbstractExtensionType Types

The ExtensionsType type is a list of type-value pairs that define optional CT-KIP features supported by a CT-KIP client or server. Extensions may be sent with any CT-KIP message. Please see the description of individual CT-KIP messages in Section 3.8 of this document for applicable extensions. Unless an extension is marked as Critical, a receiving party need not be able to interpret it. A receiving party is always free to disregard any (non-critical) extensions.

```
<xs:complexType name="AbstractExtensionsType">
  <xs:sequence maxOccurs="unbounded">
    <xs:element name="Extension" type="AbstractExtensionType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="AbstractExtensionType" abstract="true">
  <xs:attribute name="Critical" type="xs:boolean"/>
</xs:complexType>
```

3.8. CT-KIP Messages

3.8.1. Introduction

In this section, CT-KIP messages, including their parameters, encodings and semantics are defined.

3.8.2. CT-KIP Initialization

The CT-KIP server may initialize the CT-KIP protocol by sending a <CT-KIPTrigger> message. This message may, e.g., be sent in response to a user requesting token initialization in a browsing session.

```
<xs:complexType name="InitializationTriggerType">
  <xs:sequence>
    <xs:element name="TokenID" type="xs:base64Binary" minOccurs="0"/>
    <xs:element name="KeyID" type="xs:base64Binary" minOccurs="0"/>
    <xs:element name="TokenPlatformInfo"
      type="TokenPlatformInfoType" minOccurs="0"/>
    <xs:element name="TriggerNonce" type="NonceType"/>
    <xs:element name="CT-KIPURL" type="xs:anyURI" minOccurs="0"/>
    <xs:any namespace="##other" processContents="strict"
      minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:ID"/>
</xs:complexType>
```

```
<xs:element name="CT-KIPTrigger" type="CT-KIPTriggerType"/>

<xs:complexType name="CT-KIPTriggerType">
  <xs:annotation>
    <xs:documentation xml:lang="en">
       Message used to trigger the device to initiate a
       CT-KIP run.
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:choice>
      <xs:element name="InitializationTrigger"
        type="InitializationTriggerType"/>
      <xs:any nameSpace="##other" processContents="strict"/>
    </xs:choice>
  </xs:sequence>
  <xs:attribute name="Version" type="ct-kip:VersionType"/>
</xs:complexType>
```

The <CT-KIPTrigger> element is intended for the CT-KIP client and may inform the CT-KIP client about the identifier for the token that is to be initialized, and, optionally, of the identifier for the key on that token. The latter would apply when re-seeding. The trigger always contains a nonce to allow the server to couple the trigger with a later CT-KIP <ClientHello> request. Finally, the trigger may contain a URL to use when contacting the CT-KIP server. The <xs:any> elements are for future extensibility. Any provided <TokenID> or <KeyID> values shall be used by the CT-KIP client in the subsequent <ClientHello> request. The optional <TokenPlatformInfo> element informs the CT-KIP client about the characteristics of the intended token platform, and applies in the public-key variant of CT-KIP in situations when the client potentially needs to decide which one of several tokens to initialize.

The Version attribute shall be set to "1.0" for this version of CT-KIP.

3.8.3. The CT-KIP Client's Initial PDU

This message is the initial message sent from the CT-KIP client to the CT-KIP server.

```
<xs:element name="ClientHello" type="ClientHelloPDU"/>

<xs:complexType name="ClientHelloPDU">
  <xs:annotation>
    <xs:documentation xml:lang="en">
       Message sent from CT-KIP client to CT-KIP server to
```

```
          initiate a CT-KIP session.
        </xs:documentation>
      </xs:annotation>
      <xs:complexContent>
        <xs:extension base="AbstractRequestType">
          <xs:sequence>
            <xs:element name="TokenID"
              type="xs:base64Binary" minOccurs="0"/>
            <xs:element name="KeyID"
              type="xs:base64Binary" minOccurs="0"/>
            <xs:element name="ClientNonce"
              type="NonceType" minOccurs="0"/>
            <xs:element name= "TriggerNonce"
              type="NonceType" minOccurs="0"/>
            <xs:element name="SupportedKeyTypes"
              type="AlgorithmsType"/>
            <xs:element name="SupportedEncryptionAlgorithms"
              type="AlgorithmsType"/>
            <xs:element name="SupportedMACAlgorithms"
              type="AlgorithmsType"/>
            <xs:element name="Extensions"
              type="ExtensionsType" minOccurs="0"/>
          </xs:sequence>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
```

The components of this message have the following meaning:

- Version: (attribute inherited from the AbstractRequestType type) The highest version of this protocol the client supports. Only version one ("1.0") is currently specified.

- <TokenID>: An identifier for the cryptographic token (allows the server to find, e.g., a correct shared secret for MACing purposes). The identifier shall only be present if such shared secrets exist or if the identifier was provided by the server in a <CT-KIPTrigger> element (see Section 4.2.7 below). In the latter case, it must have the same value as the identifier provided in that element.

- <KeyID>: An identifier for the key that will be overwritten if the protocol run is successful. The identifier shall only be present if the key exists or was provided by the server in a <CT-KIPTrigger> element (see Section 4.2.7 below). In the latter case, it must have the same value as the identifier provided in that element.

- <ClientNonce>: This is the nonce R, which, when present, shall be used by the server when calculating MAC values (see below). It is recommended that clients include this element whenever the <KeyID> element is present.

- <TriggerNonce>: This optional element shall be present if and only if the CT-KIP run was initialized with a <CT-KIPTrigger> message (see Section 4.2.7 below), and shall, in that case, have the same value as the <TriggerNonce> child of that message. A server using nonces in this way must verify that the nonce is valid and that any token or key identifier values provided in the <CT-KIPTrigger> message match the corresponding identifier values in the <ClientHello> message.

- <SupportedKeyTypes>: A sequence of URIs indicating the key types for which the token is willing to generate keys through CT-KIP.

- <SupportedEncryptionAlgorithms>: A sequence of URIs indicating the encryption algorithms supported by the cryptographic token for the purposes of CT-KIP. The CT-KIP client may indicate the same algorithm both as a supported key type and as an encryption algorithm.

- <SupportedMACAlgorithms>: A sequence of URIs indicating the MAC algorithms supported by the cryptographic token for the purposes of CT-KIP. The CT-KIP client may indicate the same algorithm both as an encryption algorithm and as a MAC algorithm (e.g., http://www.rsasecurity.com/rsalabs/otps/schemas/2005/12/ct-kip#ct-kip-prf-aes defined in Appendix D)

- <Extensions>: A sequence of extensions. One extension is defined for this message in this version of CT-KIP: the ClientInfoType (see Section 3.9.1).

3.8.4. The CT-KIP server's initial PDU

This message is the first message sent from the CT-KIP server to the CT-KIP client (assuming a trigger message has not been sent to initiate the protocol, in which case, this message is the second message sent from the CT-KIP server to the CT-KIP client). It is sent upon reception of a <ClientHello> message.

```
<xs:element name="ServerHello" type="ServerHelloPDU"/>

<xs:complexType name="ServerHelloPDU">
  <xs:annotation>
    <xs:documentation xml:lang="en">
      Message sent from CT-KIP server to CT-KIP
```

```
        client in response to a received ClientHello
        PDU.
      </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="AbstractResponseType">
        <xs:sequence minOccurs="0">
          <xs:element name="KeyType"
            type="AlgorithmType"/>
          <xs:element name="EncryptionAlgorithm"
            type="AlgorithmType"/>
          <xs:element name="MacAlgorithm"
            type="AlgorithmType"/>
          <xs:element name="EncryptionKey"
            type="ds:KeyInfoType"/>
          <xs:element name="Payload"
            type="PayloadType"/>
          <xs:element name="Extensions"
            type="ExtensionsType" minOccurs="0"/>
          <xs:element name="Mac" type="MacType"
            minOccurs="0"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>


  <xs:complexType name="PayloadType">
    <xs:annotation>
      <xs:documentation xml:lang="en">
        Currently, only the nonce is defined.  In future versions,
        other payloads may be defined, e.g., for one-roundtrip
        initialization protocols.
      </xs:documentation>
    </xs:annotation>
    <xs:choice>
      <xs:element name="Nonce" type="NonceType"/>
      <any namespace="##other" processContents="strict"/>
    </xs:choice>
  </xs:complexType>

  <xs:complexType name="MacType">
    <xs:simpleContent>
      <xs:extension base="xs:base64Binary">
        <xs:attribute name="MacAlgorithm" type="xs:anyURI"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
```

The components of this message have the following meaning:

- Version: (attribute inherited from the AbstractResponseType type) The version selected by the CT-KIP server. May be lower than the version indicated by the CT-KIP client, in which case, local policy at the client will determine whether or not to continue the session.

- SessionID: (attribute inherited from the AbstractResponseType type) An identifier for this session.

- Status: (attribute inherited from the abstract AbstractResponseType type) Return code for the <ClientHello>. If Status is not "Continue", only the Status and Version attributes will be present; otherwise, all the other elements must be present as well.

- <KeyType>: The type of the key to be generated.

- <EncryptionAlgorithm>: The encryption algorithm to use when protecting R_C.

- <MacAlgorithm>: The MAC algorithm to be used by the CT-KIP server.

- <EncryptionKey>: Information about the key to use when encrypting R_C. It will either be the server's public key (the <ds:KeyValue> alternative of ds:KeyInfoType) or an identifier for a shared secret key (the <ds:KeyName> alternative of ds:KeyInfoType).

- <Payload>: The actual payload. For this version of the protocol, only one payload is defined: the pseudorandom string R_S.

- <Extensions>: A list of server extensions. Two extensions are defined for this message in this version of CT-KIP: the ClientInfoType and the ServerInfoType (see Section 3.9).

- <Mac>: The MAC must be present if the CT-KIP run will result in the replacement of an existing token key with a new one (i.e., if the <KeyID> element was present in the <ClientHello> message). In this case, the CT-KIP server must prove to the cryptographic token that it is authorized to replace it. The MAC value shall be computed on the (ASCII) string "MAC 1 computation", the client's nonce R (if sent), and the server's nonce R_S using an authentication key K_AUTH that should be a special authentication key used only for this purpose but may be the current K_TOKEN.

The MAC value may be computed by using the CT-KIP-PRF function of Section 3.4, in which case the input parameter s shall be set to the concatenation of the (ASCII) string "MAC 1 computation", R (if sent by the client), and R_S, and k shall be set to K_AUTH. The input parameter dsLen shall be set to the length of R_S:

dsLen = len(R_S)

MAC = CT-KIP-PRF (K_AUTH, "MAC 1 computation" || [R ||] R_S, dsLen)

The CT-KIP client must verify the MAC if the successful execution of the protocol will result in the replacement of an existing token key with a newly generated one. The CT-KIP client must terminate the CT-KIP session if the MAC does not verify, and must delete any nonces, keys, and/or secrets associated with the failed run of the CT-KIP protocol.

The MacType's MacAlgorithm attribute shall, when present, identify the negotiated MAC algorithm.

3.8.5. The CT-KIP Client's Second PDU

This message contains the nonce chosen by the cryptographic token, R_C, encrypted by the specified encryption key and encryption algorithm.

```
<xs:element name="ClientNonce" type="ClientNoncePDU"/>

<xs:complexType name="ClientNoncePDU">
  <xs:annotation>
    <xs:documentation xml:lang="en">
      Second message sent from CT-KIP client to
      CT-KIP server in a CT-KIP session.
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="AbstractRequestType">
      <xs:sequence>
        <xs:element name="EncryptedNonce"
          type="xs:base64Binary"/>
        <xs:element name="Extensions"
          type="ExtensionsType" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="SessionID" type="IdentifierType"
        use="required"/>
    </xs:extension>
  </xs:complexContent>
```

```
</xs:complexType>
```

The components of this message have the following meaning:

- Version: (inherited from the AbstractRequestType type) Shall be the same version as in the <ServerHello> message.
- SessionID: Shall have the same value as the SessionID attribute in the received <ServerHello> message.
- <EncryptedNonce>: The nonce generated and encrypted by the token. The encryption shall be made using the selected encryption algorithm and identified key, and as specified in Section 3.4.
- <Extensions>: A list of extensions. Two extensions are defined for this message in this version of CT-KIP: the ClientInfoType and the ServerInfoType (see Section 3.9).

3.8.6. The CT-KIP Server's Final PDU

This message is the last message of a two roundtrip CT-KIP exchange. The CT-KIP server sends this message to the CT-KIP client in response to the <ClientNonce> message.

```
<xs:element name="ServerFinished" type="ServerFinishedPDU"/>

<xs:complexType name="ServerFinishedPDU">
  <xs:annotation>
    <xs:documentation xml:lang="en">
      Final message sent from CT-KIP server to
      CT-KIP client in a CT-KIP session.
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="AbstractResponseType">
      <xs:sequence minOccurs="0">
        <xs:element name="TokenID"
          type="xs:base64Binary"/>
        <xs:element name="KeyID"
          type="xs:base64Binary"/>
        <xs:element name="KeyExpiryDate"
          type="xs:dateTime" minOccurs="0"/>
        <xs:element name="ServiceID"
          type="IdentifierType" minOccurs="0"/>
        <xs:element name="ServiceLogo"
          type="LogoType" minOccurs="0"/>
        <xs:element name="UserID"
          type="IdentifierType" minOccurs="0"/>
```

```
          <xs:element name="Extensions"
            type="ExtensionsType" minOccurs="0"/>
          <xs:element name="Mac"
            type="MacType"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
```

The components of this message have the following meaning:

- Version: (inherited from the AbstractResponseType type) The CT-KIP version used in this session.

- SessionID: (inherited from the AbstractResponseType type) The previously established identifier for this session.

- Status: (inherited from the AbstractResponseType type) Return code for the <ServerFinished> message. If Status is not "Success", only the Status, SessionID, and Version attributes will be present (the presence of the SessionID attribute is dependent on the type of reported error); otherwise, all the other elements must be present as well. In this latter case, the <ServerFinished> message can be seen as a "Commit" message, instructing the cryptographic token to store the generated key and associate the given key identifier with this key.

- <TokenID>: An identifier for the token carrying the generated key. Must have the same value as the <TokenID> element of the <ClientHello> message, if one was provided. When assigned by the CT-KIP server, the <TokenID> element shall be unique within the domain of the CT-KIP server.

- <KeyID>: An identifier for the newly generated key. The identifier shall be globally unique. Must have the same value as any key identifier provided by the CT-KIP client in the <ClientHello> message.

  The reason for requiring globally unique key identifiers is that it avoids potential conflicts when associating key holders with key identifiers. One way of achieving global uniqueness with reasonable certainty is to hash the combination of the issuer's fully qualified domain name with an (issuer-specific) serial number, assuming that each issuer makes sure their serial numbers never repeat.

CT-KIP clients must support key identifiers at least 64 octets long. CT-KIP servers should not generate key identifiers longer than 64 octets.

- <KeyExpiryDate>: This optional element provides the date and time after which the generated key should be treated as expired and invalid.

- <ServiceID>: An optional identifier for the service that has stored the generated key. The cryptographic token may store this identifier associated with the key in order to simplify later lookups. The identifier shall be a printable string.

- <ServiceLogo>: This optional element provides a graphical logo image for the service that can be displayed in user interfaces, e.g., to help a user select a certain key. The logo should contain an image within the size range of 60 pixels wide by 45 pixels high, and 200 pixels wide by 150 pixels high. The required MimeType attribute of this type provides information about the MIME type of the image. This specification supports both the JPEG and GIF image formats (with MIME types of "image/jpeg" and "image/gif").

- <UserID>: An optional identifier for the user associated with the generated key in the authentication service. The cryptographic token may store this identifier associated with the generated key in order to enhance later user experiences. The identifier shall be a printable string.

- <Extensions>: A list of extensions chosen by the CT-KIP server. For this message, this version of CT-KIP defines two extensions, the OTPKeyConfigurationDataType and the ClientInfoType (see Section 3.9).

- <Mac>: To avoid a false "Commit" message causing the token to end up in an initialized state for which the server does not know the stored key, <ServerFinished> messages must always be authenticated with a MAC. The MAC shall be made using the already established MAC algorithm. The MAC value shall be computed on the (ASCII) string "MAC 2 computation" and R_C using an authentication key K_AUTH. Again, this should be a special authentication key used only for this purpose, but may also be an existing K_TOKEN. (In this case, implementations must protect against attacks where K_TOKEN is used to pre-compute MAC values.) If no authentication key is present in the token, and no K_TOKEN existed before the CT-KIP run, K_AUTH shall be the newly generated K_TOKEN.

If CT-KIP-PRF is used as the MAC algorithm, then the input parameter s shall consist of the concatenation of the (ASCII) string "MAC 2 computation" and R_C, and the parameter dsLen shall be set to the length of R_C:

dsLen = len(R_C)

MAC = CT-KIP-PRF (K_AUTH, "MAC 2 computation" || R_C, dsLen)

When receiving a <ServerFinished> message with Status = "Success" for which the MAC verifies, the CT-KIP client shall associate the generated key K_TOKEN with the provided key identifier and store this data permanently. After this operation, it shall not be possible to overwrite the key unless knowledge of an authorizing key is proven through a MAC on a later <ServerHello> (and <ServerFinished>) message.

The CT-KIP client must verify the MAC. The CT-KIP client must terminate the CT-KIP session if the MAC does not verify, and must, in this case, also delete any nonces, keys, and/or secrets associated with the failed run of the CT-KIP protocol.

The MacType's MacAlgorithm attribute shall, when present, identify the negotiated MAC algorithm.

3.9. Protocol Extensions

3.9.1. The ClientInfoType Type

When present in a <ClientHello> or a <ClientNonce> message, the optional ClientInfoType extension contains CT-KIP client-specific information. CT-KIP servers must support this extension. CT-KIP servers must not attempt to interpret the data it carries and, if received, must include it unmodified in the current protocol run's next server response. Servers need not retain the ClientInfoType's data after that response has been generated.

```
<xs:complexType name="ClientInfoType">
  <xs:complexContent>
    <xs:extension base="AbstractExtensionType">
      <xs:sequence>
        <xs:element name="Data"
          type="xs:base64Binary"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

3.9.2. The ServerInfoType Type

When present, the optional ServerInfoType extension contains CT-KIP server-specific information. This extension is only valid in <ServerHello> messages for which Status = "Continue". CT-KIP clients must support this extension. CT-KIP clients must not attempt to interpret the data it carries and, if received, must include it unmodified in the current protocol run's next client request (i.e., the <ClientNonce> message). CT-KIP clients need not retain the ServerInfoType's data after that request has been generated. This extension may be used, e.g., for state management in the CT-KIP server.

```
<xs:complexType name="ServerInfoType">
  <xs:complexContent>
    <xs:extension base="AbstractExtensionType">
      <xs:sequence>
        <xs:element name="Data"
          type="xs:base64Binary"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

3.9.3. The OTPKeyConfigurationDataType Type

The optional OTPKeyConfigurationDataType extension contains additional key configuration data for OTP keys:

```
<xs:complexType name="OTPKeyConfigurationDataType">
  <xs:annotation>
    <xs:documentation xml:lang="en">
      This extension is only valid in ServerFinished
      PDUs.  It carries additional configuration data
      that an OTP token should use (subject to local
      policy) when generating OTP values with a newly
      generated OTP key.
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="ExtensionType">
      <xs:sequence>
        <xs:element name="OTPFormat"
          type="OTPFormatType"/>
        <xs:element name="OTPLength"
          type="xs:positiveInteger"/>
        <xs:element name="OTPMode"
          type="OTPModeType" minOccurs="0"/>
```

```
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
```

This extension is only valid in <ServerFinished> messages. It carries additional configuration data that the cryptographic token should use (subject to local policy) when generating OTP values from the newly generated OTP key. The components of this extension have the following meaning:

- OTPFormat: The default format of OTPs produced with this key.
- OTPLength: The default length of OTPs produced with this key.
- OTPMode: The default mode of operation when producing OTPs with this key.

4. Protocol Bindings

4.1. General Requirement

CT-KIP assumes a reliable transport.

4.2. HTTP/1.1 binding for CT-KIP

4.2.1. Introduction

This section presents a binding of the previous messages to HTTP/1.1 [7]. Note that the HTTP client normally will be different from the CT-KIP client, i.e., the HTTP client will only exist to "proxy" CT-KIP messages from the CT-KIP client to the CT-KIP server. Likewise, on the HTTP server side, the CT-KIP server may receive CT-KIP PDUs from a "front-end" HTTP server.

4.2.2. Identification of CT-KIP Messages

The MIME-type for all CT-KIP messages shall be

application/vnd.otps.ct-kip+xml

4.2.3. HTTP Headers

HTTP proxies must not cache responses carrying CT-KIP messages. For this reason, the following holds:

- When using HTTP/1.1, requesters should:
  - Include a Cache-Control header field set to "no-cache, no-store".
  - Include a Pragma header field set to "no-cache".
- When using HTTP/1.1, responders should:
  - Include a Cache-Control header field set to "no-cache, no-must-revalidate, private".
  - Include a Pragma header field set to "no-cache".
  - NOT include a Validator, such as a Last-Modified or ETag header.

There are no other restrictions on HTTP headers, besides the requirement to set the Content-Type header value to application/vnd.otps.ct-kip+xml.

4.2.4. HTTP Operations

Persistent connections as defined in HTTP/1.1 are assumed but not required. CT-KIP requests are mapped to HTTP POST operations. CT-KIP responses are mapped to HTTP responses.

4.2.5. HTTP Status Codes

A CT-KIP HTTP responder that refuses to perform a message exchange with a CT-KIP HTTP requester should return a 403 (Forbidden) response. In this case, the content of the HTTP body is not significant. In the case of an HTTP error while processing a CT-KIP request, the HTTP server must return a 500 (Internal Server Error) response. This type of error should be returned for HTTP-related errors detected before control is passed to the CT-KIP processor, or when the CT-KIP processor reports an internal error (for example, the CT-KIP XML namespace is incorrect, or the CT-KIP schema cannot be located). If the type of a CT-KIP request cannot be determined, the CT-KIP responder must return a 400 (Bad request) response.

In these cases (i.e., when the HTTP response code is 4xx or 5xx), the content of the HTTP body is not significant.

Redirection status codes (3xx) apply as usual.

Whenever the HTTP POST is successfully invoked, the CT-KIP HTTP responder must use the 200 status code and provide a suitable CT-KIP message (possibly with CT-KIP error information included) in the HTTP body.

4.2.6. HTTP Authentication

No support for HTTP/1.1 authentication is assumed.

4.2.7. Initialization of CT-KIP

The CT-KIP server may initialize the CT-KIP protocol by sending an HTTP response with Content-Type set to application/vnd.otps.ct-kip+xml and response code set to 200 (OK). This message may, e.g., be sent in response to a user requesting token initialization in a browsing session. The initialization message may carry data in its body. If this is the case, the data shall be a valid instance of a <CT-KIPTrigger> element.

4.2.8. Example Messages

a. Initialization from CT-KIP server:

```
HTTP/1.1 200 OK
Cache-Control: no-store
Content-Type: application/vnd.otps.ct-kip+xml
Content-Length: <some value>

CT-KIP initialization data in XML form...
```

b. Initial request from CT-KIP client:

```
POST http://example.com/cgi-bin/CT-KIP-server HTTP/1.1
Cache-Control: no-store
Pragma: no-cache
Host: example.com
Content-Type: application/vnd.otps.ct-kip+xml
Content-Length: <some value>

CT-KIP data in XML form (supported version, supported algorithms...)
```

c. Initial response from CT-KIP server:

```
HTTP/1.1 200 OK
Cache-Control: no-store
Content-Type: application/vnd.otps.ct-kip+xml
Content-Length: <some other value>

CT-KIP data in XML form (server random nonce, server public key, ...)
```

5. Security considerations

5.1. General

CT-KIP is designed to protect generated key material from exposure. No other entities than the CT-KIP server and the cryptographic token will have access to a generated K_TOKEN if the cryptographic algorithms used are of sufficient strength and, on the CT-KIP client side, generation and encryption of R_C and generation of K_TOKEN take place as specified and in the token. This applies even if malicious software is present in the CT-KIP client. However, as discussed in the following, CT-KIP does not protect against certain other threats resulting from man-in-the-middle attacks and other forms of attacks. CT-KIP should, therefore, be run over a transport providing privacy and integrity, such as HTTP over Transport Layer Security (TLS) with a suitable ciphersuite, when such threats are a concern. Note that TLS ciphersuites with anonymous key exchanges are not suitable in those situations.

5.2. Active Attacks

5.2.1. Introduction

An active attacker may attempt to modify, delete, insert, replay or reorder messages for a variety of purposes including service denial and compromise of generated key material. Sections 5.2.2 through 5.2.7 analyze these attack scenarios.

5.2.2. Message Modifications

Modifications to a <CT-KIPTrigger> message will either cause denial-of-service (modifications of any of the identifiers or the nonce) or the CT-KIP client to contact the wrong CT-KIP server. The latter is in effect a man-in-the-middle attack and is discussed further in Section 5.2.7.

An attacker may modify a <ClientHello> message. This means that the attacker could indicate a different key or token than the one intended by the CT-KIP client, and could also suggest other

cryptographic algorithms than the ones preferred by the CT-KIP client, e.g., cryptographically weaker ones. The attacker could also suggest earlier versions of the CT-KIP protocol, in case these versions have been shown to have vulnerabilities. These modifications could lead to an attacker succeeding in initializing or modifying another token than the one intended (i.e., the server assigning the generated key to the wrong token), or gaining access to a generated key through the use of weak cryptographic algorithms or protocol versions. CT-KIP implementations may protect against the latter by having strict policies about what versions and algorithms they support and accept. The former threat (assignment of a generated key to the wrong token) is not possible when the shared-key variant of CT-KIP is employed (assuming existing shared keys are unique per token) but is possible in the public-key variant. Therefore, CT-KIP servers must not accept unilaterally provided token identifiers in the public-key variant. This is also indicated in the protocol description. In the shared-key variant, however, an attacker may be able to provide the wrong identifier (possibly also leading to the incorrect user being associated with the generated key) if the attacker has real-time access to the token with the identified key. In other words, the generated key is associated with the correct token but the token is associated with the incorrect user. See further Section 5.5 for a discussion of this threat and possible countermeasures.

An attacker may also modify a <ServerHello> message. This means that the attacker could indicate different key types, algorithms, or protocol versions than the legitimate server would, e.g., cryptographically weaker ones. The attacker could also provide a different nonce than the one sent by the legitimate server. Clients will protect against the former through strict adherence to policies regarding permissible algorithms and protocol versions. The latter (wrong nonce) will not constitute a security problem, as a generated key will not match the key generated on the legitimate server. Also, whenever the CT-KIP run would result in the replacement of an existing key, the <Mac> element protects against modifications of R_S.

Modifications of <ClientNonce> messages are also possible. If an attacker modifies the SessionID attribute, then, in effect, a switch to another session will occur at the server, assuming the new SessionID is valid at that time on the server. It still will not allow the attacker to learn a generated K_TOKEN since R_C has been wrapped for the legitimate server. Modifications of the <EncryptedNonce> element, e.g., replacing it with a value for which the attacker knows an underlying R'C, will not result in the client changing its pre-CT-KIP state, since the server will be unable to provide a valid MAC in its final message to the client. The server

may, however, end up storing K'TOKEN rather than K_TOKEN. If the token has been associated with a particular user, then this could constitute a security problem. For a further discussion about this threat, and a possible countermeasure, see Section 5.5 below. Note that use of Secure Socket Layer (SSL) or TLS does not protect against this attack if the attacker has access to the CT-KIP client (e.g., through malicious software, "trojans").

Finally, attackers may also modify the <ServerFinished> message. Replacing the <Mac> element will only result in denial-of-service. Replacement of any other element may cause the CT-KIP client to associate, e.g., the wrong service with the generated key. CT-KIP should be run over a transport providing privacy and integrity when this is a concern.

5.2.3. Message Deletion

Message deletion will not cause any other harm than denial-of-service, since a token shall not change its state (i.e., "commit" to a generated key) until it receives the final message from the CT-KIP server and successfully has processed that message, including validation of its MAC. A deleted <ServerFinished> message will not cause the server to end up in an inconsistent state vis-a-vis the token if the server implements the suggestions in Section 5.5.

5.2.4. Message Insertion

An active attacker may initiate a CT-KIP run at any time, and suggest any token identifier. CT-KIP server implementations may receive some protection against inadvertently initializing a token or inadvertently replacing an existing key or assigning a key to a token by initializing the CT-KIP run by use of the <CT-KIPTrigger>. The <TriggerNonce> element allows the server to associate a CT-KIP protocol run with, e.g., an earlier user-authenticated session. The security of this method, therefore, depends on the ability to protect the <TriggerNonce> element in the CT-KIP initialization message. If an eavesdropper is able to capture this message, he may race the legitimate user for a key initialization. CT-KIP over a transport providing privacy and integrity, coupled with the recommendations in Section 5.5, is recommended when this is a concern.

Insertion of other messages into an existing protocol run is seen as equivalent to modification of legitimately sent messages.

5.2.5. Message Replay

Attempts to replay a previously recorded CT-KIP message will be detected, as the use of nonces ensures that both parties are live.

5.2.6. Message Reordering

An attacker may attempt to re-order messages but this will be detected, as each message is of a unique type.

5.2.7. Man in the Middle

In addition to other active attacks, an attacker posing as a man in the middle may be able to provide his own public key to the CT-KIP client. This threat and countermeasures to it are discussed in Section 3.3. An attacker posing as a man-in-the-middle may also be acting as a proxy and, hence, may not interfere with CT-KIP runs but still learn valuable information; see Section 5.3.

5.3. Passive Attacks

Passive attackers may eavesdrop on CT-KIP runs to learn information that later on may be used to impersonate users, mount active attacks, etc.

If CT-KIP is not run over a transport providing privacy, a passive attacker may learn:

- What tokens a particular user is in possession of;
- The identifiers of keys on those tokens and other attributes pertaining to those keys, e.g., the lifetime of the keys; and
- CT-KIP versions and cryptographic algorithms supported by a particular CT-KIP client or server.

Whenever the above is a concern, CT-KIP should be run over a transport providing privacy. If man-in-the-middle attacks for the purposes described above are a concern, the transport should also offer server-side authentication.

5.4. Cryptographic Attacks

An attacker with unlimited access to an initialized token may use the token as an "oracle" to pre-compute values that later on may be used to impersonate the CT-KIP server. Sections 3.6 and 3.8 contain discussions of this threat and steps recommended to protect against it.

5.5. Attacks on the Interaction between CT-KIP and User Authentication

If keys generated in CT-KIP will be associated with a particular user at the CT-KIP server (or a server trusted by, and communicating with the CT-KIP server), then in order to protect against threats where an attacker replaces a client-provided encrypted R_C with his own R'C (regardless of whether the public-key variant or the shared-secret variant of CT-KIP is employed to encrypt the client nonce), the server should not commit to associate a generated K_TOKEN with the given token (user) until the user simultaneously has proven both possession of a token containing K_TOKEN and some out-of-band provided authenticating information (e.g., a temporary password). For example, if the token is a one-time password token, the user could be required to authenticate with both a one-time password generated by the token and an out-of-band provided temporary PIN in order to have the server "commit" to the generated token value for the given user. Preferably, the user should perform this operation from another host than the one used to initialize the token, in order to minimize the risk of malicious software on the client interfering with the process.

Another threat arises when an attacker is able to trick a user to authenticate to the attacker rather than to the legitimate service before the CT-KIP protocol run. If successful, the attacker will then be able to impersonate the user towards the legitimate service, and subsequently receive a valid CT-KIP trigger. If the public-key variant of CT-KIP is used, this may result in the attacker being able to (after a successful CT-KIP protocol run) impersonate the user. Ordinary precautions must, therefore, be in place to ensure that users authenticate only to legitimate services.

6. Intellectual Property Considerations

RSA and SecurID are registered trademarks or trademarks of RSA Security Inc. in the United States and/or other countries. The names of other products and services mentioned may be the trademarks of their respective owners.

7. References

7.1. Normative References

[1] Davis, M. and M. Duerst, "Unicode Normalization Forms", March 2001, <http://www.unicode.org/unicode/reports/tr15/tr15-21.html>.

7.2. Informative References

[2] RSA Laboratories, "PKCS #11 Mechanisms for the Cryptographic Token Key Initialization Protocol", PKCS #11 Version 2.20 Amendment 2, December 2005, <ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-20/pkcs-11v2-20a2.pdf>.

[3] RSA Laboratories, "Cryptographic Token Interface Standard", PKCS #11 Version 2.20, June 2004, <ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-20/pkcs-11v2-20.pdf>.

[4] RSA Laboratories, "Frequently Asked Questions About Today's Cryptography. Version 4.1", 2000, <http://www.rsasecurity.com/rsalabs/faq/files/rsalabs_faq41.pdf>.

[5] RSA Laboratories, "Password-Based Cryptography Standard", PKCS #5 Version 2.0, March 1999, <ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf>.

[6] RSA Laboratories, "RSA Cryptography Standard", PKCS #1 Version 2.1, June 2002, <ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf>.

[7] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

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

[9] Krawzcyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[10] Iwata, T. and K. Kurosawa, "OMAC: One-Key CBC MAC. In Fast Software Encryption, FSE 2003, pages 129 - 153. Springer-Verlag", 2003, <http://crypt.cis.ibaraki.ac.jp/omac/docs/omac.pdf>.

[11] National Institute of Standards and Technology, "Secure Hash Standard", FIPS 197, February 2004, <http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf>.

[12] RSA Laboratories, "Cryptographic Token Key Initialization Protocol", OTPS Version 1.0, December 2005, <ftp://ftp.rsasecurity.com/pub/otps/ct-kip/ct-kip-v1-0.pdf>.

Appendix A. CT-KIP Schema

```
<xs:schema
  targetNamespace=
  "http://www.rsasecurity.com/rsalabs/otps/schemas/2005/12/ct-kip#"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
  xmlns=
  "http://www.rsasecurity.com/rsalabs/otps/schemas/2005/12/ct-kip#">

<xs:import namespace="http://www.w3.org/2000/09/xmldsig#"
  schemaLocation=
  "http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/
xmldsig-core-schema.xsd"/>

<!-- Basic types -->

<xs:complexType name="AbstractRequestType" abstract="true">
  <xs:attribute name="Version" type="VersionType" use="required"/>
</xs:complexType>

<xs:complexType name="AbstractResponseType" abstract="true">
  <xs:attribute name="Version" type="VersionType" use="required"/>
  <xs:attribute name="SessionID" type="IdentifierType"/>
  <xs:attribute name="Status" type="StatusCode" use="required"/>
</xs:complexType>

<xs:simpleType name="StatusCode">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Continue"/>
    <xs:enumeration value="Success"/>
    <xs:enumeration value="Abort"/>
    <xs:enumeration value="AccessDenied"/>
    <xs:enumeration value="MalformedRequest"/>
    <xs:enumeration value="UnknownRequest"/>
    <xs:enumeration value="UnknownCriticalExtension"/>
    <xs:enumeration value="UnsupportedVersion"/>
    <xs:enumeration value="NoSupportedKeyTypes"/>
    <xs:enumeration value="NoSupportedEncryptionAlgorithms"/>
    <xs:enumeration value="NoSupportedMACAlgorithms"/>
    <xs:enumeration value="InitializationFailed"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="VersionType">
  <xs:restriction base="xs:string">
    <xs:pattern value="\d{1,2}\.\d{1,3}"/>
  </xs:restriction>
```

```
   </xs:simpleType>

   <xs:simpleType name="IdentifierType">
     <xs:restriction base="xs:string">
       <xs:maxLength value="128"/>
     </xs:restriction>
   </xs:simpleType>

   <xs:simpleType name="NonceType">
     <xs:restriction base="xs:base64Binary">
       <xs:length value="16"/>
     </xs:restriction>
   </xs:simpleType>

   <xs:complexType name="LogoType">
     <xs:simpleContent>
       <xs:extension base="xs:base64Binary">
         <xs:attribute name="MimeType" type="MimeTypeType"
         use="required"/>
       </xs:extension>
     </xs:simpleContent>
   </xs:complexType>

   <xs:simpleType name="MimeTypeType">
     <xs:restriction base="xs:string">
       <xs:enumeration value="image/jpeg"/>
       <xs:enumeration value="image/gif"/>
     </xs:restriction>
   </xs:simpleType>

   <!-- Algorithms are identified through URIs -->
   <xs:complexType name="AlgorithmsType">
     <xs:sequence maxOccurs="unbounded">
       <xs:element name="Algorithm" type="AlgorithmType"/>
     </xs:sequence>
   </xs:complexType>

   <xs:simpleType name="AlgorithmType">
     <xs:restriction base="xs:anyURI"/>
   </xs:simpleType>

   <xs:complexType name="MacType">
     <xs:simpleContent>
       <xs:extension base="xs:base64Binary">
         <xs:attribute name="MacAlgorithm"
         type="xs:anyURI"/>
       </xs:extension>
     </xs:simpleContent>
```

```
   </xs:complexType>

   <!-- CT-KIP extensions (for future use) -->
   <xs:complexType name="ExtensionsType">
     <xs:sequence maxOccurs="unbounded">
       <xs:element name="Extension" type="AbstractExtensionType"/>
     </xs:sequence>
   </xs:complexType>

   <xs:complexType name="AbstractExtensionType" abstract="true">
     <xs:attribute name="Critical" type="xs:boolean"/>
   </xs:complexType>

   <xs:complexType name="ClientInfoType">
     <xs:complexContent>
       <xs:extension base="AbstractExtensionType">
         <xs:sequence>
           <xs:element name="Data" type="xs:base64Binary"/>
         </xs:sequence>
       </xs:extension>
     </xs:complexContent>
   </xs:complexType>

   <xs:complexType name="ServerInfoType">
     <xs:complexContent>
       <xs:extension base="AbstractExtensionType">
         <xs:sequence>
           <xs:element name="Data" type="xs:base64Binary"/>
         </xs:sequence>
       </xs:extension>
     </xs:complexContent>
   </xs:complexType>

   <xs:complexType name="OTPKeyConfigurationDataType">
     <xs:annotation>
       <xs:documentation xml:lang="en">
         This extension is only valid in ServerFinished PDUs.  It
         carries additional configuration data that an OTP token should
         use (subject to local policy) when generating OTP values from a
         newly generated OTP key.
       </xs:documentation>
     </xs:annotation>
     <xs:complexContent>
       <xs:extension base="AbstractExtensionType">
         <xs:sequence>
           <xs:element name="OTPFormat" type="OTPFormatType"/>
           <xs:element name="OTPLength" type="xs:positiveInteger"/>
           <xs:element name="OTPMode" type="OTPModeType" minOccurs="0"/>
```

```
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:simpleType name="OTPFormatType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Decimal"/>
      <xs:enumeration value="Hexadecimal"/>
      <xs:enumeration value="Alphanumeric"/>
      <xs:enumeration value="Binary"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="OTPModeType">
    <xs:choice maxOccurs="unbounded">
      <xs:element name="Time" type="TimeType"/>
      <xs:element name="Counter"/>
      <xs:element name="Challenge"/>
      <xs:any namespace="##other" processContents="strict"/>
    </xs:choice>
  </xs:complexType>

  <xs:complexType name="TimeType">
    <xs:complexContent>
      <xs:restriction base="xs:anyType">
        <xs:attribute name="TimeInterval" type="xs:positiveInteger"/>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="PayloadType">
    <xs:annotation>
      <xs:documentation xml:lang="en">
      </xs:documentation>
    </xs:annotation>
    <xs:choice>
      <xs:element name="Nonce" type="NonceType"/>
      <xs:any namespace="##other" processContents="strict"/>
    </xs:choice>
  </xs:complexType>

  <xs:simpleType name="PlatformType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Hardware"/>
      <xs:enumeration value="Software"/>
      <xs:enumeration value="Unspecified"/>
    </xs:restriction>
```

```
  </xs:simpleType>

  <xs:complexType name="TokenPlatformInfoType">
    <xs:annotation>
      <xs:documentation xml:lang="en">
        Carries token platform information helping the client to select
        a suitable token.
      </xs:documentation>
    </xs:annotation>
    <xs:attribute name="KeyLocation" type="PlatformType"/>
    <xs:attribute name="AlgorithmLocation" type="PlatformType"/>
  </xs:complexType>

  <xs:complexType name="InitializationTriggerType">
    <xs:sequence>
      <xs:element name="TokenID" type="xs:base64Binary" minOccurs="0"/>
      <xs:element name="KeyID" type="xs:base64Binary" minOccurs="0"/>
      <xs:element name="TokenPlatformInfo" type="TokenPlatformInfoType"
        minOccurs="0"/>
      <xs:element name="TriggerNonce" type="NonceType"/>
      <xs:element name="CT-KIPURL" type="xs:anyURI" minOccurs="0"/>
      <xs:any namespace="##other" processContents="strict"
        minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

  <!-- CT-KIP PDUs -->

  <!-- CT-KIP trigger -->
  <xs:element name="CT-KIPTrigger" type="CT-KIPTriggerType"/>

  <xs:complexType name="CT-KIPTriggerType">
    <xs:annotation>
      <xs:documentation xml:lang="en">
        Message used to trigger the device to initiate a CT-KIP run.
      </xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:choice>
        <xs:element name="InitializationTrigger"
        type="InitializationTriggerType"/>
        <xs:any namespace="##other" processContents="strict"/>
      </xs:choice>
    </xs:sequence>
    <xs:attribute name="Version" type="VersionType"/>
  </xs:complexType>

  <!-- ClientHello PDU -->
```

```
   <xs:element name="ClientHello" type="ClientHelloPDU"/>

   <xs:complexType name="ClientHelloPDU">
     <xs:annotation>
       <xs:documentation xml:lang="en">
         Message sent from CT-KIP client to CT-KIP server to initiate an
         CT-KIP session.
       </xs:documentation>
     </xs:annotation>
     <xs:complexContent>
       <xs:extension base="AbstractRequestType">
         <xs:sequence>
           <xs:element name="TokenID" type="xs:base64Binary"
             minOccurs="0"/>
           <xs:element name="KeyID" type="xs:base64Binary"
             minOccurs="0"/>
           <xs:element name="ClientNonce" type="NonceType"
             minOccurs="0"/>
           <xs:element name="TriggerNonce" type="NonceType"
             minOccurs="0"/>
           <xs:element name="SupportedKeyTypes" type="AlgorithmsType"/>
           <xs:element name="SupportedEncryptionAlgorithms"
             type="AlgorithmsType"/>
           <xs:element name="SupportedMACAlgorithms"
             type="AlgorithmsType"/>
           <xs:element name="Extensions" type="ExtensionsType"
             minOccurs="0"/>
         </xs:sequence>
       </xs:extension>
     </xs:complexContent>
   </xs:complexType>

   <!-- ServerHello PDU -->
   <xs:element name="ServerHello" type="ServerHelloPDU"/>

   <xs:complexType name="ServerHelloPDU">
     <xs:annotation>
       <xs:documentation xml:lang="en">
         Message sent from CT-KIP server to CT-KIP client in response to
         a received ClientHello PDU.
       </xs:documentation>
     </xs:annotation>
     <xs:complexContent>
       <xs:extension base="AbstractResponseType">
         <xs:sequence minOccurs="0">
           <xs:element name="KeyType" type="AlgorithmType"/>
           <xs:element name="EncryptionAlgorithm" type="AlgorithmType"/>
           <xs:element name="MacAlgorithm" type="AlgorithmType"/>
```

```
          <xs:element name="EncryptionKey" type="ds:KeyInfoType"/>
          <xs:element name="Payload" type="PayloadType"/>
          <xs:element name="Extensions" type="ExtensionsType"
            minOccurs="0"/>
          <xs:element name="Mac" type="MacType" minOccurs="0"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- ClientNonce PDU -->
  <xs:element name="ClientNonce" type="ClientNoncePDU"/>

  <xs:complexType name="ClientNoncePDU">
    <xs:annotation>
      <xs:documentation xml:lang="en">
        Second message sent from CT-KIP client to CT-KIP server to
        convey the client's chosen secret.
      </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="AbstractRequestType">
        <xs:sequence>
          <xs:element name="EncryptedNonce" type="xs:base64Binary"/>
          <xs:element name="Extensions" type="ExtensionsType"
            minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="SessionID" type="IdentifierType"
          use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- ServerFinished PDU -->
  <xs:element name="ServerFinished" type="ServerFinishedPDU"/>
  <xs:complexType name="ServerFinishedPDU">
    <xs:annotation>
      <xs:documentation xml:lang="en">
        Final message sent from CT-KIP server to CT-KIP client in an
        CT-KIP session.
      </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="AbstractResponseType">
        <xs:sequence minOccurs="0">
          <xs:element name="TokenID" type="xs:base64Binary"/>
          <xs:element name="KeyID" type="xs:base64Binary"/>
          <xs:element name="KeyExpiryDate" type="xs:dateTime"
```

```
            minOccurs="0"/>
          <xs:element name="ServiceID" type="IdentifierType"
            minOccurs="0"/>
          <xs:element name="ServiceLogo" type="LogoType"
            minOccurs="0"/>
          <xs:element name="UserID" type="IdentifierType"
            minOccurs="0"/>
          <xs:element name="Extensions" type="ExtensionsType"
            minOccurs="0"/>
          <xs:element name="Mac" type="MacType"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  </xs:schema>
```

Appendix B. Examples of CT-KIP Messages

B.1. Introduction

All examples are syntactically correct. MAC and cipher values are fictitious, however. The examples illustrate a complete CT-KIP exchange, starting with an initialization (trigger) message from the server.

B.2. Example of a CT-KIP Initialization (Trigger) Message

```
<CT-KIPTrigger
  xmlns=
  "http://www.rsasecurity.com/rsalabs/otps/schemas/2005/12/ct-kip#"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  Version="1.0">
  <InitializationTrigger>
    <TokenID>12345678</TokenID>
    <TriggerNonce>112dsdfwf312asder394jw==</TriggerNonce>
  </InitializationTrigger>
</CT-KIPTrigger>
```

B.3. Example of a <ClientHello> Message

```
<?xml version="1.0" encoding="UTF-8"?>
<ClientHello
  xmlns=
  "http://www.rsasecurity.com/rsalabs/otps/schemas/2005/12/ct-kip#"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  Version="1.0">
  <TokenID>12345678</TokenID>
```

```
    <TriggerNonce>112dsdfwf312asder394jw==</TriggerNonce>
     <SupportedKeyTypes>
       <Algorithm>http://www.rsasecurity.com/rsalabs/otps/schemas
   /2005/09/otps-wst#SecurID-AES</Algorithm>
     </SupportedKeyTypes>
     <SupportedEncryptionAlgorithms>
       <Algorithm>http://www.w3.org/2001/04/xmlenc#rsa-1_5</Algorithm>
       <Algorithm>http://www.rsasecurity.com/rsalabs/otps/schemas/
   2005/12/ct-kip#ct-kip-prf-aes</Algorithm>
     </SupportedEncryptionAlgorithms>
     <SupportedMACAlgorithms>
       <Algorithm>http://www.rsasecurity.com/rsalabs/otps/schemas/
   2005/12/ct-kip#ct-kip-prf-aes</Algorithm>
     </SupportedMACAlgorithms>
   </ClientHello>
```

B.4. Example of a <ServerHello> Message

```
   <?xml version="1.0" encoding="UTF-8"?>
   <ServerHello
     xmlns=
   "http://www.rsasecurity.com/rsalabs/otps/schemas/2005/12/ct-kip#"
     xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     Version="1.0" SessionID="4114" Status="Success">
     <KeyType>http://www.rsasecurity.com/rsalabs/otps/schemas/2005/09/
   otps-wst#SecurID-AES</KeyType>
     <EncryptionAlgorithm>http://www.rsasecurity.com/rsalabs/otps/
   schemas/2005/12/ct-kip#ct-kip-prf-aes</EncryptionAlgorithm>
     <MacAlgorithm>http://www.rsasecurity.com/rsalabs/otps/schemas/
   2005/12/ct-kip#ct-kip-prf-aes</MacAlgorithm>
     <EncryptionKey>
       <ds:KeyName>KEY-1</ds:KeyName>
     </EncryptionKey>
     <Payload>
       <Nonce>qw2ewasde312asder394jw==</Nonce>
     </Payload>
   </ServerHello>
```

B.5. Example of a <ClientNonce> Message

```
   <?xml version="1.0" encoding="UTF-8"?>
   <ClientNonce
     xmlns="http://www.rsasecurity.com/rsalabs/otps/schemas/
   2005/12/ct-kip#"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     Version="1.0" SessionID="4114">
     <EncryptedNonce>vXENc+Um/9/NvmYKiHDLaErK0gk=</EncryptedNonce>
```

```
   </ClientNonce>
```

B.6. Example of a <ServerFinished> Message

```
   <?xml version="1.0" encoding="UTF-8"?>
   <ServerFinished
     xmlns="http://www.rsasecurity.com/rsalabs/otps/schemas/
   2005/12/ct-kip#"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     Version="1.0" SessionID="4114" Status="Success">
     <TokenID>12345678</TokenID>
     <KeyExpiryDate>2009-09-16T03:02:00Z</KeyExpiryDate>
     <KeyID>43212093</KeyID>
     <ServiceID>Example Enterprise Name</ServiceID>
     <UserID>exampleLoginName</UserID>
     <Extensions>
       <Extension xsi:type="OTPKeyConfigurationDataType">
         <OTPFormat>Decimal</OTPFormat>
         <OTPLength>6</OTPLength>
         <OTPMode><Time/></OTPMode>
       </Extension>
     </Extensions>
     <Mac>miidfasde312asder394jw==</Mac>
   </ServerFinished>
```

Appendix C. Integration with PKCS #11

A CT-KIP client that needs to communicate with a connected cryptographic token to perform a CT-KIP exchange may use PKCS #11 [3] as a programming interface. When performing CT-KIP with a cryptographic token using the PKCS #11 programming interface, the procedure described in [2], Appendix B, is recommended.

Appendix D. Example CT-KIP-PRF Realizations

D.1. Introduction

This example appendix defines CT-KIP-PRF in terms of AES [8] and HMAC [9].

D.2. CT-KIP-PRF-AES

D.2.1. Identification

For tokens supporting this realization of CT-KIP-PRF, the following URI may be used to identify this algorithm in CT-KIP:

http://www.rsasecurity.com/rsalabs/otps/schemas/2005/12/

ct-kip#ct-kip-prf-aes

When this URI is used to identify the encryption algorithm to use, the method for encryption of R_C values described in Section 3.6 shall be used.

D.2.2. Definition

CT-KIP-PRF-AES (k, s, dsLen)

Input:

k encryption key to use

s octet string consisting of randomizing material. The length of the string s is sLen.

dsLen desired length of the output

Output:

DS a pseudorandom string, dsLen-octets long

Steps:

1. Let bLen be the output block size of AES in octets:

   bLen = (AES output block length in octets)

   (normally, bLen = 16)

2. If dsLen > (2**32 - 1) * bLen, output "derived data too long" and stop

3. Let n be the number of bLen-octet blocks in the output data, rounding up, and let j be the number of octets in the last block:

   n = ROUND(dsLen / bLen)

   j = dsLen - (n - 1) * bLen

4. For each block of the pseudorandom string DS, apply the function F defined below to the key k, the string s and the block index to compute the block:

   B1 = F (k, s, 1) ,

   B2 = F (k, s, 2) ,

```
    ...

    Bn = F (k, s, n)
```

The function F is defined in terms of the OMAC1 construction from [10], using AES as the block cipher:

```
F (k, s, i) = OMAC1-AES (k, INT (i) || s)
```

where INT (i) is a four-octet encoding of the integer i, most significant octet first, and the output length of OMAC1 is set to bLen.

Concatenate the blocks and extract the first dsLen octets to produce the desired data string DS:

```
DS = B1 || B2 || ... || Bn<0..j-1>
```

Output the derived data DS.

D.2.3. Example

If we assume that dsLen = 16, then:

```
n = 16 / 16 = 1

j = 16 - (1 - 1) * 16 = 16

DS = B1 = F (k, s, 1) = OMAC1-AES (k, INT (1) || S)
```

D.3. CT-KIP-PRF-SHA256

D.3.1. Identification

For tokens supporting this realization of CT-KIP-PRF, the following URI may be used to identify this algorithm in CT-KIP:

```
http://www.rsasecurity.com/rsalabs/otps/schemas/2005/12/
ct-kip#ct-kip-prf-sha256
```

When this URI is used to identify the encryption algorithm to use, the method for encryption of R_C values described in Section 3.6 shall be used.

D.3.2. Definition

CT-KIP-PRF-SHA256 (k, s, dsLen)

Input:

k encryption key to use

s octet string consisting of randomizing material. The length of the string s is sLen

dsLen desired length of the output

Output:

DS a pseudorandom string, dsLen-octets long

Steps:

1. Let bLen be the output size in octets of SHA-256 [11] (no truncation is done on the HMAC output):

   bLen = 32

2. If dsLen > (2**32 - 1) bLen, output "derived data too long" and stop

3. Let n be the number of bLen-octet blocks in the output data, rounding up, and let j be the number of octets in the last block:

   n = ROUND (dsLen / bLen)

   j = dsLen - (n - 1) * bLen

4. For each block of the pseudorandom string DS, apply the function F defined below to the key k, the string s and the block index to compute the block:

   B1 = F (k, s, 1) ,

   B2 = F (k, s, 2) ,

   ...

   Bn = F (k, s, n)

The function F is defined in terms of the HMAC construction from [9], using SHA-256 as the digest algorithm:

F (k, s, i) = HMAC-SHA256 (k, INT (i) || s)

where INT (i) is a four-octet encoding of the integer i, most significant octet first, and the output length of HMAC is set to bLen.

Concatenate the blocks and extract the first dsLen octets to produce the desired data string DS:

DS = B1 || B2 || ... || Bn<0..j-1>

Output the derived data DS.

D.3.3. Example

If we assume that sLen = 256 (two 128-octet long values) and dsLen = 16, then:

n = ROUND (16 / 32) = 1

j = 16 - (1 - 1) * 32 = 16

B1 = F (k, s, 1) = HMAC-SHA256 (k, INT (1) || s)

DS = B1<0 ... 15>

That is, the result will be the first 16 octets of the HMAC output.

Appendix E. About OTPS

The One-Time Password Specifications are documents produced by RSA Laboratories in cooperation with secure systems developers for the purpose of simplifying integration and management of strong authentication technology into secure applications, and to enhance the user experience of this technology.

Further development of the OTPS series will occur through mailing list discussions and occasional workshops, and suggestions for improvement are welcome. As for our PKCS documents, results may also be submitted to standards forums. For more information, contact:

OTPS Editor
RSA Laboratories
174 Middlesex Turnpike
Bedford, MA 01730 USA
otps-editor@rsasecurity.com
http://www.rsasecurity.com/rsalabs/

Author's Address

Magnus Nystroem
RSA Security

EMail: magnus@rsasecurity.com

Full Copyright Statement

Intellectual Property

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.