Internet Engineering Task Force (IETF)
Request for Comments: 6063
Category: Standards Track
ISSN: 2070-1721

A. Doherty
RSA, The Security Division of EMC
M. Pei
VeriSign, Inc.
S. Machani
Diversinet Corp.
M. Nystrom
Microsoft Corp.

December 2010

Dynamic Symmetric Key Provisioning Protocol (DSKPP)

Abstract

The Dynamic Symmetric Key Provisioning Protocol (DSKPP) is a client-server protocol for initialization (and configuration) of symmetric keys to locally and remotely accessible cryptographic modules. The protocol can be run with or without private key capabilities in the cryptographic modules and with or without an established public key infrastructure.

Two variations of the protocol support multiple usage scenarios. With the four-pass variant, keys are mutually generated by the provisioning server and cryptographic module; provisioned keys are not transferred over-the-wire or over-the-air. The two-pass variant enables secure and efficient download and installation of pre-generated symmetric keys to a cryptographic module.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in 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/rfc6063.

Copyright Notice

Table of Contents

1. Introduction

Symmetric-key-based cryptographic systems (e.g., those providing authentication mechanisms such as one-time passwords and challenge-response) offer performance and operational advantages over public key schemes. Such use requires a mechanism for the provisioning of symmetric keys providing equivalent functionality to mechanisms such as the Certificate Management Protocol (CMP) [RFC4210] and Certificate Management over CMS (CMC) [RFC5272] in a Public Key Infrastructure.

Traditionally, cryptographic modules have been provisioned with keys during device manufacturing, and the keys have been imported to the cryptographic server using, e.g., a CD-ROM disc shipped with the devices. Some vendors also have proprietary provisioning protocols, which often have not been publicly documented (the Cryptographic Token Key Initialization Protocol (CT-KIP) is one exception [RFC4758]).

This document describes the Dynamic Symmetric Key Provisioning Protocol (DSKPP), a client-server protocol for provisioning symmetric keys between a cryptographic module (corresponding to DSKPP Client) and a key provisioning server (corresponding to DSKPP Server).

DSKPP provides an open and interoperable mechanism for initializing and configuring symmetric keys to cryptographic modules that are accessible over the Internet. The description is based on the information contained in [RFC4758], and contains specific enhancements, such as user authentication and support for the [RFC6030] format for transmission of keying material.

DSKPP has two principal protocol variants. The four-pass protocol variant permits a symmetric key to be established that includes randomness contributed by both the client and the server. The two-pass protocol requires only one round trip instead of two and permits a server specified key to be established.

1.1. Key Words

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

1.2. Version Support

There is a provision made in the syntax for an explicit version number. Only version "1.0" is currently specified.

The purpose for versioning the protocol is to provide a mechanism by which changes to required cryptographic algorithms (e.g., SHA-256) and attributes (e.g., key size) can be deployed without disrupting existing implementations; likewise, outdated implementations can be de-commissioned without disrupting operations involving newer protocol versions.

The numbering scheme for DSKPP versions is "<major>.<minor>". The major and minor numbers MUST be treated as separate integers and each number MAY be incremented higher than a single digit. Thus, "DSKPP 2.4" would be a lower version than "DSKPP 2.13", which in turn would be lower than "DSKPP 12.3". Leading zeros (e.g., "DSKPP 6.01") MUST be ignored by recipients and MUST NOT be sent.

The major version number should be incremented only if the data formats or security algorithms have changed so dramatically that an older version implementation would not be able to interoperate with a newer version (e.g., removing support for a previously mandatory-to-implement algorithm now found to be insecure). The minor version number indicates new capabilities (e.g., introducing a new algorithm option) and MUST be ignored by an entity with a smaller minor version number but be used for informational purposes by the entity with the larger minor version number.

1.3. Namespace Identifiers

This document uses Uniform Resource Identifiers (URIs) [RFC3986] to identify resources, algorithms, and semantics.

1.3.1. Defined Identifiers

The XML namespace [XMLNS] URI for Version 1.0 of DSKPP is:

"urn:ietf:params:xml:ns:keyprov:dskpp"

References to qualified elements in the DSKPP schema defined herein use the prefix "dskpp", but any prefix is allowed.

1.3.2. Identifiers Defined in Related Specifications

This document relies on qualified elements already defined in the Portable Symmetric Key Container [RFC6030] namespace, which is represented by the prefix "pskc" and declared as:

xmlns:pskc="urn:ietf:params:xml:ns:keyprov:pskc"

1.3.3. Referenced Identifiers

Finally, the DSKPP syntax presented in this document relies on algorithm identifiers defined in the XML Signature [XMLDSIG] namespace:

xmlns:ds="http://www.w3.org/2000/09/xmldsig#"

References to algorithm identifiers in the XML Signature namespace are represented by the prefix "ds".

2. Terminology

2.1. Definitions

Terms are defined below as they are used in this document. The same terms may be defined differently in other documents.

Authentication Code (AC): User Authentication Code comprised of a string of hexadecimal characters known to the device and the server and containing at a minimum a client identifier and a password. This ClientID/password combination is used only once and may have a time limit, and then discarded.

Authentication Data (AD): User Authentication Data that is derived from the Authentication Code (AC)

Client ID: An identifier that the DSKPP Server uses to locate the real username or account identifier on the server. It can be a short random identifier that is unrelated to any real usernames.

Cryptographic Module: A component of an application, which enables symmetric key cryptographic functionality

Device: A physical piece of hardware, or a software framework, that hosts symmetric key cryptographic modules

Device ID (DeviceID): A unique identifier for the device that houses the cryptographic module, e.g., a mobile phone

DSKPP Client: Manages communication between the symmetric key cryptographic module and the DSKPP Server

DSKPP Server: The symmetric key provisioning server that participates in the DSKPP run

DSKPP Server ID (ServerID): The unique identifier of a DSKPP Server

Key Agreement: A key establishment protocol whereby two or more parties can agree on a key in such a way that both influence the outcome

Key Confirmation: The assurance of the rightful participants in a key-establishment protocol that the intended recipient of the shared key actually possesses the shared key

Key Issuer: An organization that issues symmetric keys to end-users

Key Package (KP): An object that encapsulates a symmetric key and its configuration data

Key ID (KeyID): A unique identifier for the symmetric key

Key Protection Method (KPM): The key transport method used during two-pass DSKPP

Key Protection Method List (KPML): The list of key protection methods supported by a cryptographic module

Key Provisioning Server: A lifecycle management system that provides a key issuer with the ability to provision keys to cryptographic modules hosted on end-users' devices

Key Transport: A key establishment procedure whereby the DSKPP Server selects and encrypts the keying material and then sends the material to the DSKPP Client [NIST-SP800-57]

Key Transport Key: The private key that resides on the cryptographic module. This key is paired with the DSKPP Client's public key, which the DSKPP Server uses to encrypt keying material during key transport [NIST-SP800-57]

Key Type: The type of symmetric key cryptographic methods for which the key will be used (e.g., Open AUTHentication HMAC-Based One-Time Password (OATH HOTP) or RSA SecurID authentication, AES encryption, etc.)

Key Wrapping: A method of encrypting keys for key transport [NIST-SP800-57]

Key Wrapping Key: A symmetric key encrypting key used for key wrapping [NIST-SP800-57]

Keying Material: The data necessary (e.g., keys and key configuration data) necessary to establish and maintain cryptographic keying relationships [NIST-SP800-57]

Manufacturer's Key: A unique master key pre-issued to a hardware device, e.g., a smart card, during the manufacturing process. If present, this key may be used by a cryptographic module to derive secret keys

Protocol Run: Complete execution of the DSKPP that involves one exchange (two-pass) or two exchanges (four-pass)

Security Attribute List (SAL): A payload that contains the DSKPP version, DSKPP variant (four- or two-pass), key package formats, key types, and cryptographic algorithms that the cryptographic module is capable of supporting

2.2. Notation

||	String concatenation
[x]	Optional element x
A ^ B	Exclusive-OR operation on strings A and B (where A and B are of equal length)
<XMLElement>	A typographical convention used in the body of the text
DSKPP-PRF(k,s,dsLen)	A keyed pseudorandom function
E(k,m)	Encryption of m with the key k
K	Key used to encrypt R_C (either K_SERVER or K_SHARED), or in MAC or DSKPP_PRF computations
K_AC	Secret key that is derived from the Authentication Code and used for user authentication purposes
K_MAC	Secret key derived during a DSKPP exchange for use with key confirmation
K_MAC'	A second secret key used for server authentication
K_PROV	A provisioning master key from which two keys are derived: K_TOKEN and K_MAC
K_SERVER	Public key of the DSKPP Server; used for encrypting R_C in the four-pass protocol variant

K_SHARED	Secret key that is pre-shared between the DSKPP Client and the DSKPP Server; used for encrypting R_C in the four-pass protocol variant
K_TOKEN	Secret key that is established in a cryptographic module using DSKPP
R	Pseudorandom value chosen by the DSKPP Client and used for MAC computations
R_C	Pseudorandom value chosen by the DSKPP Client and used as input to the generation of K_TOKEN
R_S	Pseudorandom value chosen by the DSKPP Server and used as input to the generation of K_TOKEN
URL_S	DSKPP Server address, as a URL

2.3. Abbreviations

AC	Authentication Code
AD	Authentication Data
DSKPP	Dynamic Symmetric Key Provisioning Protocol
HTTP	Hypertext Transfer Protocol
KP	Key Package
KPM	Key Protection Method
KPML	Key Protection Method List
MAC	Message Authentication Code
PC	Personal Computer
PDU	Protocol Data Unit
PKCS	Public Key Cryptography Standards
PRF	Pseudorandom Function
PSKC	Portable Symmetric Key Container
SAL	Security Attribute List (see Section 2.1)
TLS	Transport Layer Security
URL	Uniform Resource Locator
USB	Universal Serial Bus
XML	eXtensible Markup Language

3. DSKPP Overview

The following sub-sections provide a high-level view of protocol internals and how they interact with external provisioning applications. Usage scenarios are provided in Appendix A.

3.1. Protocol Entities

A DSKPP provisioning transaction has three entities:

Server: The DSKPP provisioning server.

Cryptographic Module: The cryptographic module to which the symmetric keys are to be provisioned, e.g., an authentication token.

Client: The DSKPP Client that manages communication between the cryptographic module and the key provisioning server.

The principal syntax is XML [XML] and it is layered on a transport mechanism such as HTTP [RFC2616] and HTTP Over TLS [RFC2818]. While it is highly desirable for the entire communication between the DSKPP Client and server to be protected by means of a transport providing confidentiality and integrity protection such as HTTP over Transport Layer Security (TLS), such protection is not sufficient to protect the exchange of the symmetric key data between the server and the cryptographic module and DSKPP is designed to permit implementations that satisfy this requirement.

The server only communicates to the client. As far as the server is concerned, the client and cryptographic module may be considered to be a single entity.

From a client-side security perspective, however, the client and the cryptographic module are separate logical entities and may in some implementations be separate physical entities as well.

It is assumed that a device will host an application layered above the cryptographic module, and this application will manage communication between the DSKPP Client and cryptographic module. The manner in which the communicating application will transfer DSKPP elements to and from the cryptographic module is transparent to the DSKPP Server. One method for this transfer is described in [CT-KIP-P11].

3.2. Basic DSKPP Exchange

3.2.1. User Authentication

In a DSKPP message flow, the user has obtained a new hardware or software device embedded with a cryptographic module. The goal of DSKPP is to provision the same symmetric key and related information to the cryptographic module and the key management server, and

associate the key with the correct username (or other account identifier) on the server. To do this, the DSKPP Server MUST authenticate the user to be sure he is authorized for the new key.

User authentication occurs within the protocol itself *after* the DSKPP Client initiates the first message. In this case, the DSKPP Client MUST have access to the DSKPP Server URL.

Alternatively, a DSKPP web service or other form of web application can authenticate a user *before* the first message is exchanged. In this case, the DSKPP Server MUST trigger the DSKPP Client to initiate the first message in the protocol transaction.

3.2.2. Protocol Initiated by the DSKPP Client

In the following example, the DSKPP Client first initiates DSKPP, and then the user is authenticated using a Client ID and Authentication Code.

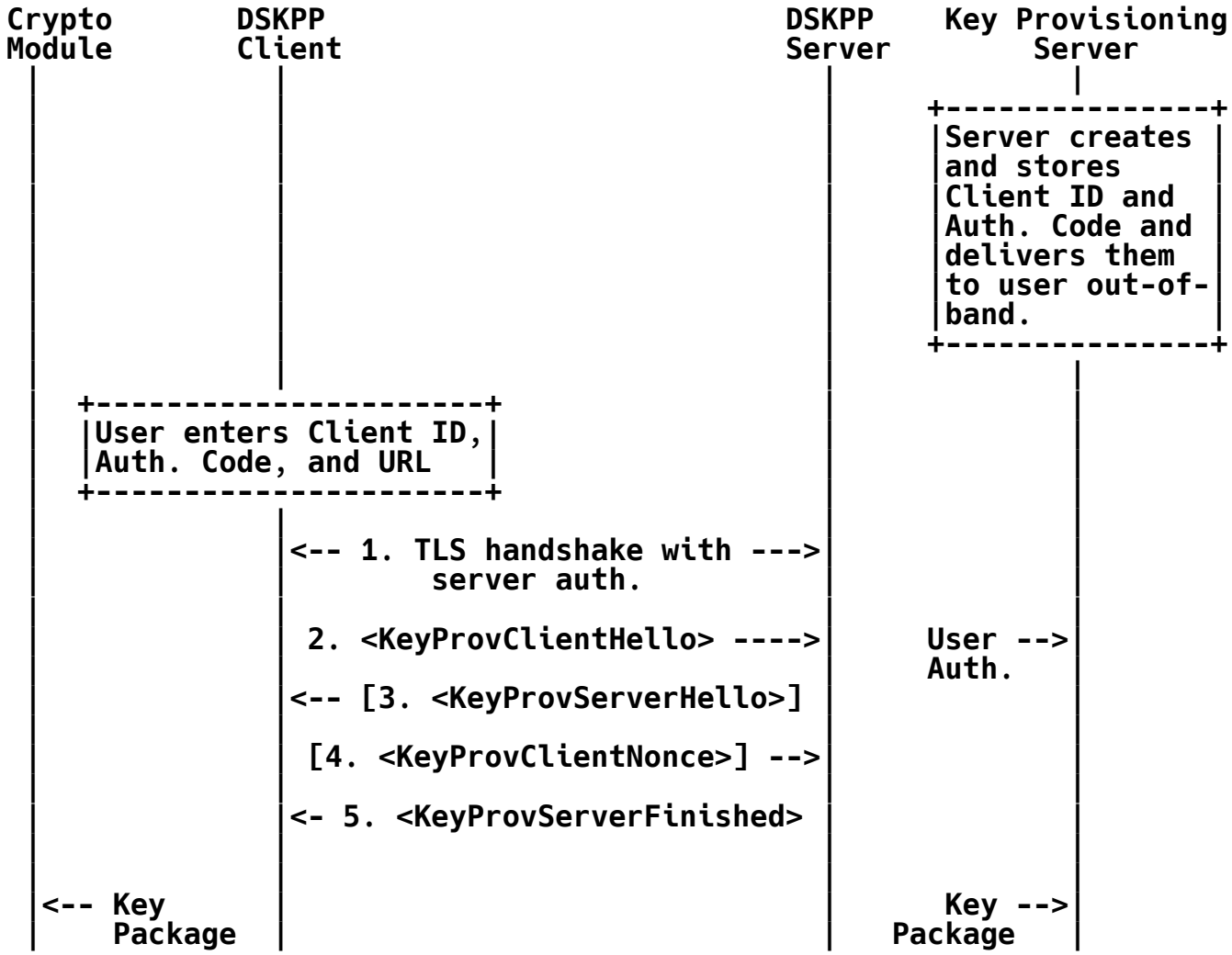


Figure 1: Basic DSKPP Exchange

Before DSKPP begins:

- The Authentication Code is generated by the DSKPP Server, and delivered to the user via an out-of-band trustworthy channel (e.g., a paper slip delivered by IT department staff).
- The user typically enters the Client ID and Authentication Code manually, possibly on a device with only a numeric keypad. Thus, they are often short numeric values (for example, 8 decimal digits). However, the DSKPP Server is free to generate them in any way it wishes.
- The DSKPP Client needs the URL [RFC3986] of the DSKPP Server (which is not user specific or secret, and may be pre-configured somehow), and a set of trust anchors for verifying the server certificate.
- There must be an account for the user that has an identifier and long-term username (or other account identifier) to which the token will be associated. The DSKPP Server will use the Client ID to find the corresponding Authentication Code for user authentication.

In Step 1, the client establishes a TLS connection, authenticates the server (that is, validates the certificate, and compares the host name in the URL with the certificate) as described in Section 3.1 of [RFC2818].

Next, the DSKPP Client and DSKPP Server exchange DSKPP messages (which are sent over HTTPS). In these messages:

- The client and server negotiate which cryptographic algorithms they want to use, which algorithms are supported for protecting DSKPP messages, and other DSKPP details.
- The client sends the Client ID to the server, and proves that it knows the corresponding Authentication Code.
- The client and server agree on a secret key (token key or K_TOKEN); depending on the negotiated protocol variant, this is either a fresh key derived during the DSKPP run (called "four-pass variant", since it involves four DSKPP messages) or is generated by (or pre-exists on) the server and transported to the client (called "two-pass variant" in the rest of this document, since it involves two DSKPP messages).
- The server sends a "key package" to the client. The package only includes the key itself in the case of the "two-pass variant"; with either variant, the key package contains attributes that influence how the provisioned key will be later used by the cryptographic module and cryptographic server. The exact contents depend on the cryptographic algorithm (e.g., for a one-time password algorithm that supports variable-length OTP values, the length of the OTP value would be one attribute in the key package).

After the protocol run has been successfully completed, the cryptographic modules stores the contents of the key package. Likewise, the DSKPP provisioning server stores the contents of the key package with the cryptographic server, and associates these with the correct username. The user can now use the their device to perform symmetric-key based operations.

The exact division of work between the cryptographic module and the DSKPP Client -- and key Provisioning server and DSKPP Server -- are not specified in this document. The figure above shows one possible case, but this is intended for illustrative purposes only.

3.2.3. Protocol Triggered by the DSKPP Server

In the first message flow (previous section), the Client ID and Authentication Code were delivered to the client by some out-of-band means (such as paper sent to the user).

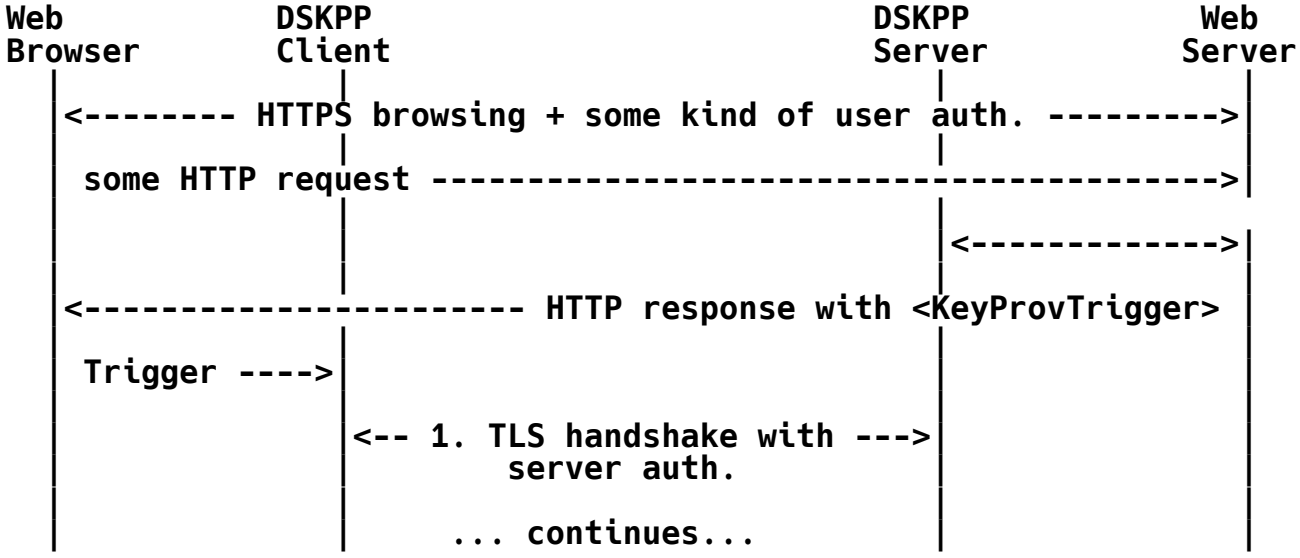


Figure 2: DSKPP Exchange with Web-Based Authentication

In the second message flow, the user first authenticates to a web server (for example, an IT department's "self-service" Intranet page), using an ordinary web browser and some existing credentials.

The user then requests (by clicking a link or submitting a form) provisioning of a new key to the cryptographic module. The web server will reply with a <KeyProvTrigger> message that contains the Client ID, Authentication Code, and URL of the DSKPP Server. This information is also needed by the DSKPP Server; how the web server and DSKPP Server interact is beyond the scope of this document.

The <KeyProvTrigger> message is sent in an HTTP response, and it is marked with MIME type "application/dskpp+xml". It is assumed the web browser has been configured to recognize this MIME type; the browser will start the DSKPP Client and provide it with the <KeyProvTrigger> message.

The DSKPP Client then contacts the DSKPP Server and uses the Client ID and Authentication Code (from the <KeyProvTrigger> message) the same way as in the first message flow.

3.2.4. Variants

As noted in the previous section, once the protocol has started, the client and server MAY engage in either a two-pass or four-pass message exchange. The four-pass and two-pass protocols are appropriate in different deployment scenarios. The biggest differentiator between the two is that the two-pass protocol supports transport of an existing key to a cryptographic module, while the four-pass involves key generation on-the-fly via key agreement. In either case, both protocol variants support algorithm agility through the negotiation of encryption mechanisms and key types at the beginning of each protocol run.

3.2.4.1. Criteria for Using the Four-Pass Variant

The four-pass protocol is needed under one or more of the following conditions:

- Policy requires that both parties engaged in the protocol jointly contribute entropy to the key. Enforcing this policy mitigates the risk of exposing a key during the provisioning process as the key is generated through mutual agreement without being transferred over-the-air or over-the-wire. It also mitigates risk of exposure after the key is provisioned, as the key will not be vulnerable to a single point of attack in the system.

- A cryptographic module does not have private key capabilities.

- The cryptographic module is hosted by a device that neither was pre-issued with a manufacturer's key or other form of pre-shared key (as might be the case with a smart card or Subscriber Identity Module (SIM) card) nor has a keypad that can be used for entering a passphrase (such as present on a mobile phone).

3.2.4.2. Criteria for Using the Two-Pass Variant

The two-pass protocol is needed under one or more of the following conditions:

- Pre-existing (i.e., legacy) keys must be provisioned via transport to the cryptographic module.
- The cryptographic module is hosted on a device that was pre-issued with a manufacturer's key (such as may exist on a smart card), or other form of pre-shared key (such as may exist on a SIM-card), and is capable of performing private key operations.
- The cryptographic module is hosted by a device that has a built-in keypad with which a user may enter a passphrase, useful for deriving a key wrapping key for distribution of keying material.

3.3. Status Codes

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 DSKPP Server and the DSKPP Client MUST immediately terminate the DSKPP run. DSKPP Servers and DSKPP Clients MUST delete any secret values generated as a result of failed runs of DSKPP. Session identifiers MAY be retained from successful or failed protocol runs for replay detection purposes, but such retained identifiers MUST NOT be reused for subsequent runs of the protocol.

When possible, the DSKPP Client SHOULD present an appropriate error message to the user.

These status codes are valid in all DSKPP Response messages unless explicitly stated otherwise:

Continue: The DSKPP Server is ready for a subsequent request from the DSKPP Client. It cannot be sent in the server's final message.

Success: Successful completion of the DSKPP session. It can only be sent in the server's final message.

Abort: The DSKPP Server rejected the DSKPP Client's request for unspecified reasons.

AccessDenied: The DSKPP Client is not authorized to contact this DSKPP Server.

MalformedRequest: The DSKPP Server failed to parse the DSKPP Client's request.

UnknownRequest: The DSKPP Client made a request that is unknown to the DSKPP Server.

UnknownCriticalExtension: A DSKPP extension marked as "Critical" could not be interpreted by the receiving party.

UnsupportedVersion: The DSKPP Client used a DSKPP version not supported by the DSKPP Server. This error is only valid in the DSKPP Server's first response message.

NoSupportedKeyTypes: "NoSupportedKeyTypes" indicates that the DSKPP Client only suggested key types that are not supported by the DSKPP Server. This error is only valid in the DSKPP Server's first response message.

NoSupportedEncryptionAlgorithms: The DSKPP Client only suggested encryption algorithms that are not supported by the DSKPP Server. This error is only valid in the DSKPP Server's first response message.

NoSupportedMacAlgorithms: The DSKPP Client only suggested MAC algorithms that are not supported by the DSKPP Server. This error is only valid in the DSKPP Server's first response message.

NoProtocolVariants: The DSKPP Client did not suggest a required protocol variant (either two-pass or four-pass). This error is only valid in the DSKPP Server's first response message.

NoSupportedKeyPackages: The DSKPP Client only suggested key package formats that are not supported by the DSKPP Server. This error is only valid in the DSKPP Server's first response message.

AuthenticationDataMissing: The DSKPP Client didn't provide Authentication Data that the DSKPP Server required.

AuthenticationDataInvalid: The DSKPP Client supplied User Authentication Data that the DSKPP Server failed to validate.

InitializationFailed: The DSKPP Server could not generate a valid key given the provided data. When this status code is received, the DSKPP Client SHOULD try to restart DSKPP, as it is possible that a new run will succeed.

ProvisioningPeriodExpired: The provisioning period set by the DSKPP Server has expired. When the status code is received, the DSKPP Client SHOULD report the reason for key initialization failure to the user and the user MUST register with the DSKPP Server to initialize a new key.

3.4. Basic Constructs

The following calculations are used in both DSKPP variants.

3.4.1. User Authentication Data (AD)

User Authentication Data (AD) is derived from a Client ID and Authentication Code that the user enters before the first DSKPP message is sent.

Note: The user will typically enter the Client ID and Authentication Code manually, possibly on a device with only numeric keypad. Thus, they are often short numeric values (for example, 8 decimal digits). However, the DSKPP Server is free to generate them in any way it wishes.

3.4.1.1. Authentication Code Format

AC is encoded in Type-Length-Value (TLV) format. The format consists of a minimum of two TLVs and a variable number of additional TLVs, depending on implementation.

The TLV fields are defined as follows:

Type (1 character)	A hexadecimal character identifying the type of information contained in the Value field.
Length (2 characters)	Two hexadecimal characters indicating the length of the Value field to follow. The field value MAY be up to 255 characters. The Length value 00 MAY be used to specify custom tags without any field values.
Value (variable length)	A variable-length string of hexadecimal characters containing the instance-specific information for this TLV.

The following table summarizes the TLVs defined in this document. Optional TLVs are allowed for vendor-specific extensions with the constraint that the high bit MUST be set to indicate a vendor-specific type. Other TLVs are left for later revisions of this protocol.

Type	TLV Name	Conformance	Example Usage
1	Client ID	Mandatory	{ "AC00000A" }
2	Password	Mandatory	{ "3582AF0C3E" }
3	Checksum	Optional	{ "4D5" }

The Client ID is a mandatory TLV that represents the requester's identifier of maximum length 255. The value is represented as a string of hexadecimal characters that identifies the key request. For example, suppose Client ID is set to "AC00000A", the Client ID TLV in the AC will be represented as "108AC00000A".

The Password is a mandatory TLV the contains a one-time use shared secret known by the user and the Provisioning Server. The Password value is unique and SHOULD be a random string to make AC more difficult to guess. The string MUST contain hexadecimal characters only. For example, suppose password is set to "3582AF0C3E", then the Password TLV would be "20A3582AF0C3E".

The Checksum is an OPTIONAL TLV, which is generated by the issuing server and sent to the user as part of the AC. If the TLV is provided, the checksum value MUST be computed using the CRC16 algorithm [ISO3309]. When the user enters the AC, the typed AC string of characters is verified with the checksum to ensure it is correctly entered by the user. For example, suppose the AC with combined Client ID tag and Password tag is set to "108AC00000A20A3582AF0C3E", then the CRC16 calculation would generate a checksum of 0x356, resulting in a Checksum TLV of "334D5". The complete AC string in this example would be "108AC00000A20A3582AF0C3E3034D5".

Although this specification recommends using hexadecimal characters only for the AC at the application's user interface layer and making the TLV triples non-transparent to the user as described in the example above; implementations MAY additionally choose to use other printable Unicode characters [UNICODE] at the application's user interface layer in order to meet specific local, context or usability requirements. When non-hexadecimal characters are desired at the

user interface layer such as when other printable US-ASCII characters or international characters are used, SASLprep [RFC4013] MUST be used to normalize user input before converting it to a string of hexadecimal characters. For example, if a given application allows the use of any printable US-ASCII characters and extended ASCII characters for Client ID and Password fields, and the Client ID is set to "myclient!D" and the associated Password is set to "mYpas&#rD", the user enters through the keyboard or other means a Client ID of "myclient!D" and a Password of "mYpas&#rD" in separate form fields or as instructed by the provider. The application's layer processing user input MUST then convert the values entered by the user to the following string for use in the protocol: "1146D79636C69656E7421442126D5970617326237244" (note that in this example the Checksum TLV is not added).

The example is explained further below in detail:

Assume that the raw Client ID value or the value entered by the use is: myclient!ID

The Client ID value as characters names is:

U+006D LATIN SMALL LETTER M character
U+0079 LATIN SMALL LETTER Y character
U+0063 LATIN SMALL LETTER C character
U+006C LATIN SMALL LETTER L character
U+0069 LATIN SMALL LETTER I character
U+0065 LATIN SMALL LETTER E character
U+006E LATIN SMALL LETTER N character
U+0074 LATIN SMALL LETTER T character
U+0021 EXCLAMATION MARK character (!)
U+0044 LATIN CAPITAL LETTER D character

The UTF-8 conversion of the Client ID value is: 6D 79 63 6C 69 65 6E 74 21 44

The length of the Client ID value in hexadecimal characters is: 14

The TLV presentation of the Client ID field is: 1146D79636C69656E742144

The raw Password value or the value entered by the user is: mYpas&#rD

The Password value as character names is:

U+006D LATIN SMALL LETTER M character
U+0059 LATIN LARGE LETTER Y character
U+0070 LATIN SMALL LETTER P character

U+0061 LATIN SMALL LETTER A character
U+0073 LATIN SMALL LETTER S character
U+0026 AMPERSAND character (&)
U+0023 POUND SIGN character (#)
U+0072 LATIN SMALL LETTER R character
U+0044 LATIN LARGE LETTER D character

The UTF-8 conversion of the password value is: 6D 59 70 61 73 26 23 72 44

The length of the password value in hexadecimal characters is: 12

The TLV presentation of the password field is: 2126D5970617326237244

The combined Client ID and password fields value or the AC value is: 1146D79636C69656E7421442126D5970617326237244

3.4.1.2. User Authentication Data Calculation

The Authentication Data consists of a Client ID (extracted from the AC) and a value, which is derived from AC as follows (refer to Section 3.4.2 for a description of DSKPP-PRF in general and Appendix D for a description of DSKPP-PRF-AES):

MAC = DSKPP-PRF(K_AC, AC->ClientID||URL_S||R_C||[R_S], 16)

In four-pass DSKPP, the cryptographic module uses R_C, R_S, and URL_S to calculate the MAC, where URL_S is the URL the DSKPP Client uses when contacting the DSKPP Server. In two-pass DSKPP, the cryptographic module does not have access to R_S, therefore only R_C is used in combination with URL_S to produce the MAC. In either case, K_AC MUST be derived from AC->password as follows [PKCS-5]:

K_AC = PBKDF2(AC->password, R_C || K, iter_count, 16)

One of the following values for K MUST be used:

a. In four-pass:
   * The public key of the DSKPP Server (K_SERVER), or (in the pre-shared key variant) the pre-shared key between the client and the server (K_SHARED).
b. In two-pass:
   * The public key of the DSKPP Client, or the public key of the device when a device certificate is available.
   * The pre-shared key between the client and the server (K_SHARED).
   * A passphrase-derived key.

The iteration count, iter_count, MUST be set to at least 100,000 except in the last two two-pass cases (where K is set to K_SHARED or a passphrase-derived key), in which case iter_count MUST be set to 1.

3.4.2. The DSKPP One-Way Pseudorandom Function, DSKPP-PRF

Regardless of the protocol variant employed, there is a requirement for a cryptographic primitive that provides a deterministic transformation of a secret key k and a varying length octet string s to a bit string of specified length dsLen.

This primitive must meet the same requirements as for a keyed hash function: it MUST take an arbitrary length input and generate an output that is one way and collision free (for a definition of these terms, see, e.g., [FAQ]). Further, its output MUST be unpredictable even if other outputs for the same key are known.

From the point of view of this specification, DSKPP-PRF is a "black-box" function that, given the inputs, generates a pseudorandom value and MAY be realized by any appropriate and competent cryptographic technique. Appendix D contains two example realizations of DSKPP-PRF.

DSKPP-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 MUST be at least 16 octets long.

3.4.3. The DSKPP Message Hash Algorithm

When sending its last message in a protocol run, the DSKPP Server generates a MAC that is used by the client for key confirmation. Computation of the MAC MUST include a hash of all DSKPP messages sent by the client and server during the transaction. To compute a message hash for the MAC given a sequence of DSKPP messages msg_1, ..., msg_n, the following operations MUST be carried out:

a. The sequence of messages contains all DSKPP Request and Response messages up to but not including this message.
b. Re-transmitted messages are removed from the sequence of messages.
   Note: The resulting sequence of messages MUST be an alternating sequence of DSKPP Request and DSKPP Response messages
c. The contents of each message is concatenated together.
d. The resultant string is hashed using SHA-256 in accordance with [FIPS180-SHA].

4. Four-Pass Protocol Usage

This section describes the methods and message flow that comprise the four-pass protocol variant. Four-pass DSKPP depends on a client-server key agreement mechanism.

4.1. The Key Agreement Mechanism

With four-pass DSKPP, the symmetric key that is the target of provisioning, is generated on-the-fly without being transferred between the DSKPP Client and DSKPP Server. The data flow and computation are described below.

4.1.1. Data Flow

A sample data flow showing key generation during the four-pass protocol is shown in Figure 3.

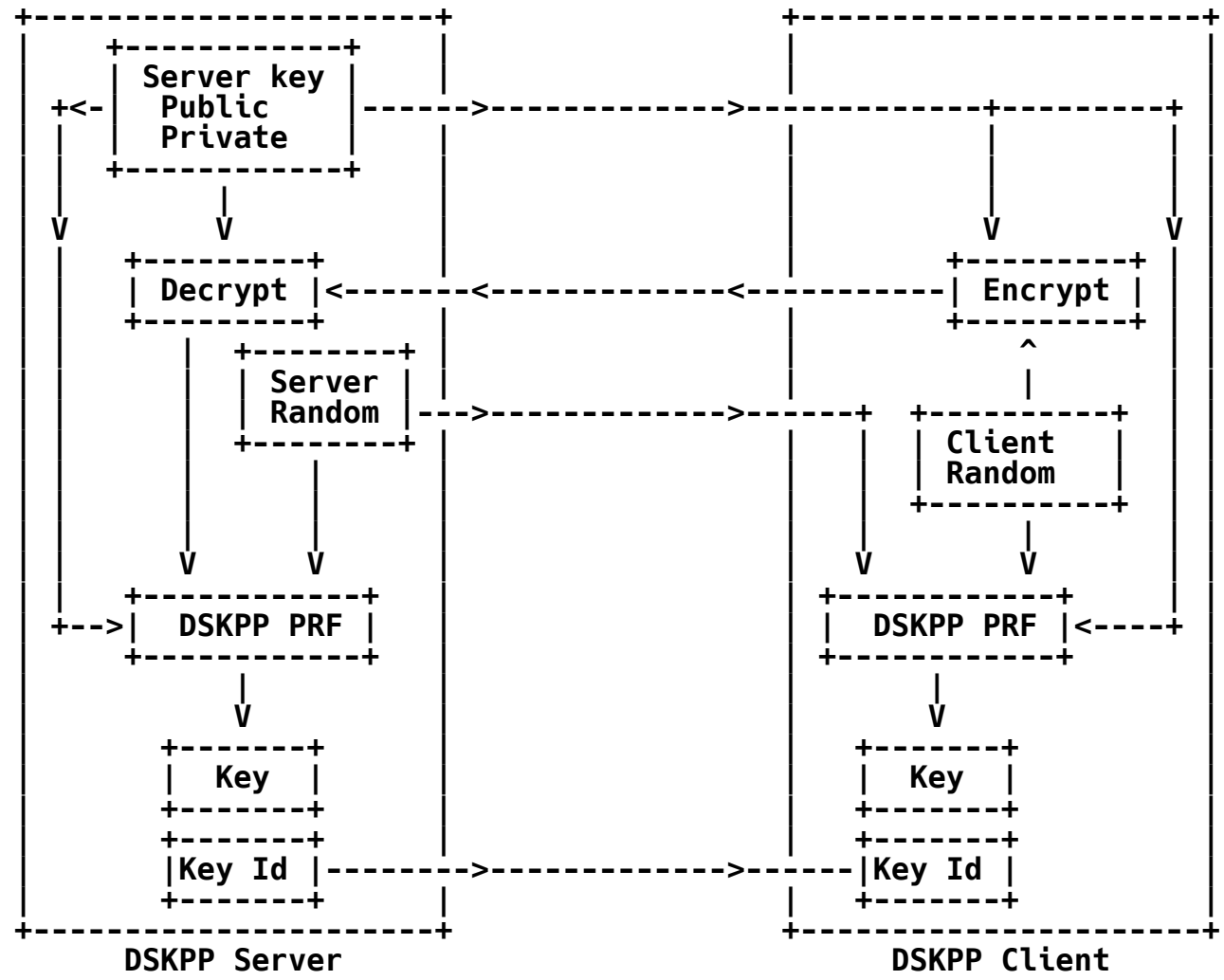


Figure 3: Principal Data Flow for DSKPP Key Generation Using Public Server Key

The inclusion of the two random nonces (R_S and R_C) in the key generation provides assurance to both sides (the cryptographic module and the DSKPP 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 module will end up with a key different from the one stored by the legitimate DSKPP Server.

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 pre-generated and

loaded on the cryptographic module before the device is issued to the user. In that case, the latter string, R_C2, SHOULD be unique for each cryptographic module.

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 cryptographic module. 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 sync" when they try to use their keys. In the case of encrypting R_C with K_SERVER, it is therefore 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., using a connection independent from the one used for the key generation).

4.1.2. Computation

In four-pass DSKPP, the client and server both generate K_TOKEN and K_MAC by deriving them from a provisioning key (K_PROV) using the DSKPP-PRF (refer to Section 3.4.2) as follows:

K_PROV = DSKPP-PRF(k,s,dsLen), where

> k = R_C (i.e., the secret random value chosen by the DSKPP Client)
> s = "Key generation" || K || R_S (where K is the key used to encrypt R_C and R_S is the random value chosen by the DSKPP Server)
> dsLen = (desired length of K_PROV whose first half constitutes K_MAC and second half constitutes K_TOKEN)

Then, K_TOKEN and K_MAC are derived from K_PROV, where

> K_PROV = K_MAC || K_TOKEN

When computing K_PROV, the derived keys, K_MAC and K_TOKEN, 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.

Note that this computation pertains to four-pass DSKPP only.

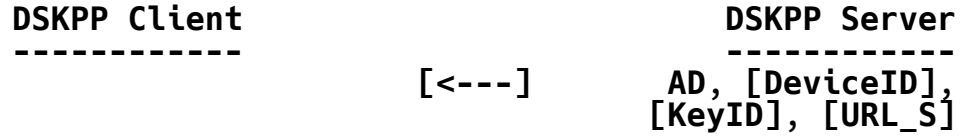
4.2. Message Flow

The four-pass protocol flow consists of two message exchanges:
1: Pass 1 = <KeyProvClientHello>, Pass 2 = <KeyProvServerHello>
2: Pass 3 = <KeyProvClientNonce>, Pass 4 = <KeyProvServerFinished>

The first pair of messages negotiate cryptographic algorithms and exchange nonces. The second pair of messages establishes a symmetric key using mutually authenticated key agreement.

The purpose and content of each message are described below. XML format and examples are in Section 8 and Appendix B.

4.2.1. KeyProvTrigger



When this message is sent:
The "trigger" message is optional. The DSKPP Server sends this message after the following out-of-band steps are performed:
1. A user directed their browser to a key provisioning web application and signs in (i.e., authenticates).
2. The user requests a key.
3. The web application processes the request and returns an Authentication Code to the user, e.g., in response to an enrollment request via a secure web session.
4. The web application retrieves the Authentication Code from the user (possibly by asking the user to enter it using a web form, or alternatively by the user selecting a URL in which the Authentication Code is embedded).
5. The web application derives Authentication Data (AD) from the Authentication Code as described in Section 3.4.1.
6. The web application passes AD, and possibly a DeviceID (identifies a particular device to which the key is to be provisioned) and/or KeyID (identifies a key that will be replaced) to the DSKPP Server.

Purpose of this message:
To start a DSKPP session: The DSKPP Server uses this message to trigger a client-side application to send the first DSKPP message.
To provide a way for the key provisioning system to get the DSKPP Server URL to the DSKPP Client.

So the key provisioning system can point the DSKPP Client to a particular cryptographic module that was pre-configured in the DSKPP provisioning server.

In the case of key renewal, to identify the key to be replaced.

What is contained in this message:
AD MUST be provided to allow the DSKPP Server to authenticate the user before completing the protocol run.

A DeviceID MAY be included to allow a key provisioning application to bind the provisioned key to a specific device.

A KeyID MAY be included to allow the key provisioning application to identify a key to be replaced, e.g., in the case of key renewal.

The Server URL MAY be included to allow the key provisioning application to inform the DSKPP Client of which server to contact.

4.2.2. KeyProvClientHello

```
DSKPP Client                          DSKPP Server
------------                          ------------
SAL, [AD],
[DeviceID], [KeyID]     --->
```

When this message is sent:
When a DSKPP Client first connects to a DSKPP Server, it is required to send the <KeyProvClientHello> as its first message. The client can also send a <KeyProvClientHello> in response to a <KeyProvTrigger>.

What is contained in this message:
The Security Attribute List (SAL) included with <KeyProvClientHello> contains the combinations of DSKPP versions, variants, key package formats, key types, and cryptographic algorithms that the DSKPP Client supports in order of the client's preference (favorite choice first).

If <KeyProvClientHello> was preceded by a <KeyProvTrigger>, then this message MUST also include the Authentication Data (AD), DeviceID, and/or KeyID that was provided with the trigger.

If <KeyProvClientHello> was not preceded by a <KeyProvTrigger>, then this message MAY contain a DeviceID that was pre-shared with the DSKPP Server, and a key ID associated with a key previously provisioned by the DSKPP provisioning server.

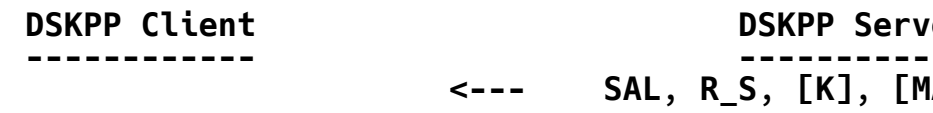
Application note:
If this message is preceded by trigger message <KeyProvTrigger>, then the application will already have AD available (see Section 4.2.1). However, if this message was not preceded by <KeyProvTrigger>, then the application MUST retrieve the User Authentication Code, possibly by prompting the user to manually enter their Authentication Code, e.g., on a device with only a numeric keypad.

The application MUST also derive Authentication Data (AD) from the Authentication Code, as described in Section 3.4.1, and save it for use in its next message, <KeyProvClientNonce>.

How the DSKPP Server uses this message:
The DSKPP Server will look for an acceptable combination of DSKPP version, variant (in this case, four-pass), key package format, key type, and cryptographic algorithms. If the DSKPP Client's SAL does not match the capabilities of the DSKPP Server, or does not comply with key provisioning policy, then the DSKPP Server will set the Status attribute to something other than "Continue". Otherwise, the Status attribute will be set to "Continue".

If included in <KeyProvClientHello>, the DSKPP Server will validate the Authentication Data (AD), DeviceID, and KeyID. The DSKPP Server MUST NOT accept the DeviceID unless the server sent the DeviceID in a preceding trigger message. Note that it is also legitimate for a DSKPP Client to initiate the DSKPP run without having received a <KeyProvTrigger> message from a server, but in this case any provided DeviceID MUST NOT be accepted by the DSKPP Server unless the server has access to a unique key for the identified device and that key will be used in the protocol.

4.2.3. KeyProvServerHello



When this message is sent:
The DSKPP Server will send this message in response to a <KeyProvClientHello> message after it looks for an acceptable combination of DSKPP version, variant (in this case, four-pass), key package format, key type, and set of cryptographic algorithms. If it could not find an acceptable combination, then it will still send the message, but with a failure status.

Purpose of this message:
With this message, the context for the protocol run is set. Furthermore, the DSKPP Server uses this message to transmit a random nonce, which is required for each side to agree upon the same symmetric key (K_TOKEN).

What is contained in this message:
A status attribute equivalent to the server's return code to <KeyProvClientHello>. If the server found an acceptable set of attributes from the client's SAL, then it sets status to Continue and returns an SAL (selected from the SAL that it received in <KeyProvClientHello>). The Server's SAL specifies the DSKPP version and variant (in this case, four-pass), key type, cryptographic algorithms, and key package format that the DSKPP Client MUST use for the remainder of the protocol run.

A random nonce (R_S) for use in generating a symmetric key through key agreement; the length of R_S may depend on the selected key type.

A key (K) for the DSKPP Client to use for encrypting the client nonce included with <KeyProvClientNonce>. K represents the server's public key (K_SERVER) or a pre-shared secret key (K_SHARED).

A MAC MUST be present if a key is being renewed so that the DSKPP Client can confirm that the replacement key came from a trusted server. This MAC MUST be computed using DSKPP-PRF (see Section 3.4.2), where the input parameter k MUST be set to the existing MAC key K_MAC' (i.e., the value of the MAC key that existed before this protocol run; the implementation MAY specify K_MAC' to be the value of the K_TOKEN that is being replaced), and input parameter dsLen MUST be set to the length of R_S.

How the DSKPP Client uses this message:
When the Status attribute is not set to "Continue", this indicates failure and the DSKPP Client MUST abort the protocol.

If successful execution of the protocol will result in the replacement of an existing key with a newly generated one, the DSKPP Client MUST verify the MAC provided in <KeyProvServerHello>. The DSKPP Client MUST terminate the DSKPP session if the MAC does not verify, and MUST delete any nonces, keys, and/or secrets associated with the failed run.

If the Status attribute is set to "Continue", the cryptographic module generates a random nonce (R_C) using the cryptographic algorithm specified in the SAL. The length of the nonce R_C will depend on the selected key type.

Encrypt R_C using K and the encryption algorithm included in the SAL.

The method the DSKPP Client MUST use to encrypt R_C:
If K is equivalent to K_SERVER (i.e., the public key of the DSKPP Server), then an RSA encryption scheme from PKCS #1 [PKCS-1] MAY be used. If K is equivalent to K_SERVER, then the cryptographic module SHOULD verify the server's certificate before using it to encrypt R_C as described in [RFC2818], Section 3.1, and [RFC5280].

If K is equivalent to K_SHARED, the DSKPP Client MAY use the DSKPP-PRF to avoid dependence on other algorithms. In this case, the client uses K_SHARED as input parameter k (K_SHARED SHOULD be used solely for this purpose) as follows:

dsLen = len(R_C), where "len" is the length of R_C
DS = DSKPP-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:

E(DS, R_C) = DS ^ R_C

The DSKPP Server will then perform the reverse operation to extract R_C from E(DS, R_C).

4.2.4. KeyProvClientNonce

When this message is sent:
The DSKPP Client will send this message immediately following a <KeyProvServerHello> message whose status was set to "Continue".

Purpose of this message:
With this message the DSKPP Client transmits User Authentication Data (AD) and a random nonce encrypted with the DSKPP Server's key (K). The client's random nonce is required for each side to agree upon the same symmetric key (K_TOKEN).

What is contained in this message:
Authentication Data (AD) that was derived from an Authentication Code entered by the user before <KeyProvClientHello> was sent (refer to Section 3.2).

The DSKPP Client's random nonce (R_C), which was encrypted as described in Section 4.2.3.

How the DSKPP Server uses this message:
The DSKPP Server MUST use AD to authenticate the user. If authentication fails, then the DSKPP Server MUST set the return code to a failure status.

If user authentication passes, the DSKPP Server decrypts R_C using its key (K). The decryption method is based on whether K that was transmitted to the client in <KeyProvServerHello> was equal to the server's public key (K_SERVER) or a pre-shared key (K_SHARED) (refer to Section 4.2.3 for a description of how the DSKPP Client encrypts R_C).

After extracting R_C, the DSKPP Server computes K_TOKEN using a combination of the two random nonces R_S and R_C and its encryption key, K, as described in Section 4.1.2. The particular realization of DSKPP-PRF (e.g., those defined in Appendix D) depends on the MAC algorithm contained in the <KeyProvServerHello> message. The DSKPP Server then generates a key package that contains key usage attributes such as expiry date and length. The key package MUST NOT include K_TOKEN since in the four-pass variant K_TOKEN is never transmitted between the DSKPP Server and Client. The server stores K_TOKEN and the key package with the user's account on the cryptographic server.

Finally, the server generates a key confirmation MAC that the client will use to avoid a false "Commit" message that would cause the cryptographic module to end up in state in which the server does not recognize the stored key.

The MAC used for key confirmation MUST be calculated as follows:

```
msg_hash = SHA-256(msg_1, ..., msg_n)

dsLen = len(msg_hash)

MAC = DSKPP-PRF (K_MAC, "MAC 1 computation" || msg_hash, dsLen)
```

where

MAC The DSKPP Pseudorandom Function defined in Section 3.4.2 is used to compute the MAC. The particular realization of DSKPP-PRF (e.g., those defined in Appendix D) depends on the MAC algorithm contained in the <KeyProvServerHello> message. The MAC MUST be computed using the existing MAC key (K_MAC), and a string that is formed by concatenating the (ASCII) string "MAC 1 computation" and a msg_hash.

K_MAC The key derived from K_PROV, as described in Section 4.1.2.

msg_hash The message hash (defined in Section 3.4.3) of messages msg_1, ..., msg_n.

4.2.5. KeyProvServerFinished

When this message is sent:
The DSKPP Server will send this message after authenticating the user and, if authentication passed, generating K_TOKEN and a key package, and associating them with the user's account on the cryptographic server.

Purpose of this message:
With this message, the DSKPP Server confirms generation of the key (K_TOKEN) and transmits the associated identifier and application-specific attributes, but not the key itself, in a key package to the client for protocol completion.

What is contained in this message:
A status attribute equivalent to the server's return code to <KeyProvClientNonce>. If user authentication passed, and the server successfully computed K_TOKEN, generated a key package, and associated them with the user's account on the cryptographic server, then it sets the Status attribute to "Success".
If the Status attribute is set to "Success", then this message acts as a "Commit" message, instructing the cryptographic module to store the generated key (K_TOKEN) and associate the given key identifier with this key. As such, a key package (KP) MUST be included in this message, which holds an identifier for the generated key (but not the key itself) and additional configuration, e.g., the identity of the DSKPP Server, key usage attributes, etc. The default symmetric key package format MUST be

based on the Portable Symmetric Key Container (PSKC) defined in [RFC6030]. Alternative formats MAY include [RFC6031], PKCS #12 [PKCS-12], or PKCS #5 XML [PKCS-5-XML] format.

With KP, the server includes a key confirmation MAC that the client uses to avoid a false "Commit" message. The MAC algorithm is the same DSKPP-PRF that was sent in the <KeyProvServerHello> message.

How the DSKPP Client uses this message:
When the Status attribute is not set to "Success", this indicates failure and the DSKPP Client MUST abort the protocol.

After receiving a <KeyProvServerFinished> message with Status = "Success", the DSKPP Client MUST verify the key confirmation MAC that was transmitted with this message. The DSKPP Client MUST terminate the DSKPP 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 protocol.

If <KeyProvServerFinished> has Status = "Success", and the MAC was verified, then the DSKPP Client MUST calculate K_TOKEN from the combination of the two random nonces R_S and R_C and the server's encryption key, K, as described in Section 4.1.2. The DSKPP-PRF is the same one used for MAC computation. The DSKPP Client associates the key package contained in <KeyProvServerFinished> with the generated key, K_TOKEN, and stores this data permanently on the cryptographic module.

After this operation, it MUST NOT be possible to overwrite the key unless knowledge of an authorizing key is proven through a MAC on a later <KeyProvServerHello> (and <KeyProvServerFinished>) message.

5. Two-Pass Protocol Usage

This section describes the methods and message flow that comprise the two-pass protocol variant. Two-pass DSKPP is essentially a transport of keying material from the DSKPP Server to the DSKPP Client. The DSKPP Server transmits keying material in a key package formatted in accordance with [RFC6030], [RFC6031], PKCS #12 [PKCS-12], or PKCS #5 XML [PKCS-5-XML].

The keying material includes a provisioning master key, K_PROV, from which the DSKPP Client derives two keys: the symmetric key to be established in the cryptographic module, K_TOKEN, and a key, K_MAC, used for key confirmation. The keying material also includes key usage attributes, such as expiry date and length.

The DSKPP Server encrypts K_PROV to ensure that it is not exposed to any other entity than the DSKPP Server and the cryptographic module itself. The DSKPP Server uses any of three key protection methods to encrypt K_PROV: Key Transport, Key Wrap, and Passphrase-Based Key Wrap Key Protection methods.

While the DSKPP Client and server may negotiate the key protection method to use, the actual key protection is carried out in the KeyPackage. The format of a KeyPackage specifies how a key should be protected using the three key protection methods. The following KeyPackage formats are defined for DSKPP:

- PSKC Key Container [RFC6030] at urn:ietf:params:xml:ns:keyprov:dskpp:pskc-key-container
- SKPC Key Container [RFC6031] at urn:ietf:params:xml:ns:keyprov:dskpp:skpc-key-container
- PKCS12 Key Container [PKCS-12] at urn:ietf:params:xml:ns:keyprov:dskpp:pkcs12-key-container
- PKCS5-XML Key Container [PKCS-5-XML] at urn:ietf:params:xml:ns:keyprov:dskpp:pkcs5-xml-key-container

Each of the key protection methods is described below.

5.1. Key Protection Methods

This section introduces three key protection methods for the two-pass variant. Additional methods MAY be defined by external entities or through the IETF process.

5.1.1. Key Transport

Purpose of this method:
This method is intended for PKI-capable devices. The DSKPP Server encrypts keying material and transports it to the DSKPP Client. The server encrypts the keying material using the public key of the DSKPP Client, whose private key part resides in the cryptographic module. The DSKPP Client decrypts the keying material and uses it to derive the symmetric key, K_TOKEN.

This method is identified with the following URN:
urn:ietf:params:xml:schema:keyprov:dskpp:transport

The DSKPP Server and Client MUST support the following mechanism:
http://www.w3.org/2001/04/xmlenc#rsa-1_5 encryption mechanism defined in [XMLENC].

5.1.2. Key Wrap

Purpose of this method:
This method is ideal for pre-keyed devices, e.g., SIM cards. The DSKPP Server encrypts keying material using a pre-shared key wrapping key and transports it to the DSKPP Client. The DSKPP Client decrypts the keying material, and uses it to derive the symmetric key, K_TOKEN.

This method is identified with the following URN:
urn:ietf:params:xml:schema:keyprov:dskpp:wrap

The DSKPP Server and Client MUST support all of the following key wrapping mechanisms:

AES128 KeyWrap
Refer to id-aes128-wrap in [RFC3394] and
http://www.w3.org/2001/04/xmlenc#kw-aes128 in [XMLENC]

AES128 KeyWrap with Padding
Refer to id-aes128-wrap-pad in [RFC5649] and
http://www.w3.org/2001/04/xmlenc#kw-aes128 in [XMLENC]

AES-CBC-128
Refer to [FIPS197-AES] and
http://www.w3.org/2001/04/xmlenc#aes128-cbc in [XMLENC]

5.1.3. Passphrase-Based Key Wrap

Purpose of this method:
This method is a variation of the Key Wrap Method that is applicable to constrained devices with keypads, e.g., mobile phones. The DSKPP Server encrypts keying material using a wrapping key derived from a user-provided passphrase, and transports the encrypted material to the DSKPP Client. The DSKPP Client decrypts the keying material, and uses it to derive the symmetric key, K_TOKEN.

To preserve the property of not exposing K_TOKEN to any other entity than the DSKPP Server and the cryptographic module itself, the method SHOULD be employed only when the device contains facilities (e.g., a keypad) for direct entry of the passphrase.

This method is identified with the following URN:
urn:ietf:params:xml:schema:keyprov:dskpp:passphrase-wrap

The DSKPP Server and Client MUST support the following:

* The PBES2 password-based encryption scheme defined in [PKCS-5] (and identified as http://www.rsasecurity.com/rsalabs/pkcs/schemas/pkcs-5#pbes2 in [PKCS-5-XML]).

* The PBKDF2 passphrase-based key derivation function also defined in [PKCS-5] (and identified as http://www.rsasecurity.com/rsalabs/pkcs/schemas/pkcs-5#pbkdf2 in [PKCS-5-XML]).

* All of the following key wrapping mechanisms:

  AES128 KeyWrap
  Refer to id-aes128-wrap in [RFC3394] and
  http://www.w3.org/2001/04/xmlenc#kw-aes128 in [XMLENC]

  AES128 KeyWrap with Padding
  Refer to id-aes128-wrap-pad in [RFC5649] and
  http://www.w3.org/2001/04/xmlenc#kw-aes128 in [XMLENC]

  AES-CBC-128
  Refer to [FIPS197-AES] and
  http://www.w3.org/2001/04/xmlenc#aes128-cbc in [XMLENC]

5.2. Message Flow

The two-pass protocol flow consists of one exchange:
1: Pass 1 = <KeyProvClientHello>, Pass 2 = <KeyProvServerFinished>

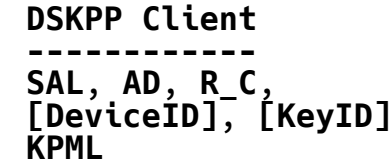
Although there is no exchange of the <ServerHello> message or the <ClientNonce> message, the DSKPP Client is still able to specify algorithm preferences and supported key types in the <KeyProvClientHello> message.

The purpose and content of each message are described below. XML format and examples are in Section 8 and Appendix B.

5.2.1. KeyProvTrigger

The trigger message is used in exactly the same way for the two-pass variant as for the four-pass variant; refer to Section 4.2.1.

5.2.2. KeyProvClientHello



When this message is sent:
When a DSKPP Client first connects to a DSKPP Server, it is required to send the <KeyProvClientHello> as its first message. The client can also send <KeyProvClientHello> in response to a <KeyProvTrigger> message.

Purpose of this message:
With this message, the DSKPP Client specifies its algorithm preferences and supported key types as well as which DSKPP versions, protocol variants (in this case "two-pass"), key package formats, and key protection methods that it supports. Furthermore, the DSKPP Client facilitates user authentication by transmitting the Authentication Data (AD) that was provided by the user before the first DSKPP message was sent.

Application note:
This message MUST send User Authentication Data (AD) to the DSKPP Server. If this message is preceded by trigger message <KeyProvTrigger>, then the application will already have AD available (see Section 4.2.1). However, if this message was not preceded by <KeyProvTrigger>, then the application MUST retrieve the User Authentication Code, possibly by prompting the user to manually enter their Authentication Code, e.g., on a device with only a numeric keypad. The application MUST also derive Authentication Data (AD) from the Authentication Code, as described in Section 3.4.1, and save it for use in its next message, <KeyProvClientNonce>.

What is contained in this message:
The Security Attribute List (SAL) included with <KeyProvClientHello> contains the combinations of DSKPP versions, variants, key package formats, key types, and cryptographic algorithms that the DSKPP Client supports in order of the client's preference (favorite choice first).

Authentication Data (AD) that was either included with <KeyProvTrigger>, or generated as described in the "Application Note" above.

The DSKPP Client's random nonce (R_C), which was used by the client when generating AD. By inserting R_C into the DSKPP session, the DSKPP Client is able to ensure the DSKPP Server is live before committing the key.

If <KeyProvClientHello> was preceded by a <KeyProvTrigger>, then this message MUST also include the DeviceID and/or KeyID that was provided with the trigger. Otherwise, if a trigger message did not precede <KeyProvClientHello>, then this message MAY include a DeviceID that was pre-shared with the DSKPP Server, and MAY contain a key ID associated with a key previously provisioned by the DSKPP provisioning server.

The list of key protection methods (KPML) that the DSKPP Client supports. Each item in the list MAY include an encryption key "payload" for the DSKPP Server to use to protect keying material that it sends back to the client. The payload MUST be of type <ds:KeyInfoType> ([XMLDSIG]). For each key protection method, the allowable choices for <ds:KeyInfoType> are:

* Key Transport
  Only those choices of <ds:KeyInfoType> that identify a public key (i.e., <ds:KeyName>, <ds:KeyValue>, <ds:X509Data>, or <ds:PGPData>). The <ds:X509Certificate> option of the <ds:X509Data> alternative is RECOMMENDED when the public key corresponding to the private key on the cryptographic module has been certified.

* Key Wrap
  Only those choices of <ds:KeyInfoType> that identify a symmetric key (i.e., <ds:KeyName> and <ds:KeyValue>). The <ds:KeyName> alternative is RECOMMENDED.

* Passphrase-Based Key Wrap
  The <ds:KeyName> option MUST be used and the key name MUST identify the passphrase that will be used by the server to generate the key wrapping key. The identifier and passphrase components of <ds:KeyName> MUST be set to the Client ID and Authentication Code components of AD (same AD as contained in this message).

How the DSKPP Server uses this message:
The DSKPP Server will look for an acceptable combination of DSKPP version, variant (in this case, two-pass), key package format, key type, and cryptographic algorithms. If the DSKPP Client's SAL does not match the capabilities of the DSKPP Server, or does not

comply with key provisioning policy, then the DSKPP Server will set the Status attribute to something other than "Success". Otherwise, the Status attribute will be set to "Success".

The DSKPP Server will validate the DeviceID and KeyID if included in <KeyProvClientHello>. The DSKPP Server MUST NOT accept the DeviceID unless the server sent the DeviceID in a preceding trigger message. Note that it is also legitimate for a DSKPP Client to initiate the DSKPP run without having received a <KeyProvTrigger> message from a server, but in this case any provided DeviceID MUST NOT be accepted by the DSKPP Server unless the server has access to a unique key for the identified device and that key will be used in the protocol.

The DSKPP Server MUST use AD to authenticate the user. If authentication fails, then the DSKPP Server MUST set the return code to a failure status, and MUST, in this case, also delete any nonces, keys, and/or secrets associated with the failed run of the protocol.

If user authentication passes, the DSKPP Server generates a key K_PROV. In the two-pass case, wherein the client does not have access to R_S, K_PROV is randomly generated solely by the DSKPP Server wherein K_PROV MUST consist of two parts of equal length, i.e.,

```
K_PROV = K_MAC || K_TOKEN
```

The length of K_TOKEN (and hence also the length of K_MAC) is determined by the type of K_TOKEN, which MUST be one of the key types supported by the DSKPP Client. In cases where the desired key length for K_TOKEN is different from the length of K_MAC for the underlying MAC algorithm, the greater length of the two MUST be chosen to generate K_PROV. The actual MAC key is truncated from the resulting K_MAC when it is used in the MAC algorithm when K_MAC is longer than necessary in order to match the desired K_TOKEN length. If K_TOKEN is longer than needed in order to match the K_MAC length, the provisioning server and the receiving client must determine the actual secret key length from the target key algorithm and store only the truncated portion of the K_TOKEN. The truncation MUST take the beginning bytes of the desired length from K_TOKEN or K_MAC for the actual key. For example, when a provisioning server provisions an event based HOTP secret key with length 20 and MAC algorithm DSKPP-PRF-SHA256 (Appendix D), K_PROV length will be 64. The derived K_TOKEN and K_MAC will each consist of 32 bytes. The actual HOTP key should be the first 20 bytes of the K_TOKEN.

Once K_PROV is computed, the DSKPP Server selects one of the key protection methods from the DSKPP Client's KPML, and uses that method and corresponding payload to encrypt K_PROV. The DSKPP Server generates a key package to transport the key encryption method information and the encrypted provisioning key (K_PROV). The encrypted data format is subject to the choice supported by the selected key package. The key package MUST specify and use the selected key protection method and the key information that was received in <KeyProvClientHello>. The key package also includes key usage attributes such as expiry date and length. The server stores the key package and K_TOKEN with a user account on the cryptographic server.

The server generates a MAC for key confirmation, which the client will use to avoid a false "Commit" message that would cause the cryptographic module to end up in state in which the server does not recognize the stored key.

In addition, if an existing key is being renewed, the server generates a second MAC that it will return to the client as server Authentication Data (AD) so that the DSKPP Client can confirm that the replacement key came from a trusted server.

The method the DSKPP Server MUST use to calculate the key confirmation MAC:

```
msg_hash = SHA-256(msg_1, ..., msg_n)

dsLen = len(msg_hash)

MAC = DSKPP-PRF (K_MAC, "MAC 1 computation" || msg_hash ||
ServerID, dsLen)
```

where

MAC	The MAC MUST be calculated using the already established MAC algorithm and MUST be computed on the (ASCII) string "MAC 1 computation", msg_hash, and ServerID using the existing MAC key K_MAC.
K_MAC	The key that is derived from K_PROV, which the DSKPP Server MUST provide to the cryptographic module.
msg_hash	The message hash, defined in Section 3.4.3, of messages msg_1, ..., msg_n.
ServerID	The identifier that the DSKPP Server MUST include in the <KeyPackage> element of <KeyProvServerFinished>.

If DSKPP-PRF (defined in Section 3.4.2) is used as the MAC algorithm, then the input parameter s MUST consist of the concatenation of the (ASCII) string "MAC 1 computation", msg_hash, and ServerID, and the parameter dsLen MUST be set to the length of msg_hash.

The method the DSKPP Server MUST use to calculate the server authentication MAC:

The MAC MUST be computed on the (ASCII) string "MAC 2 computation", the server identifier ServerID, and R, using a pre-existing MAC key K_MAC' (the MAC key that existed before this protocol run). Note that the implementation may specify K_MAC' to be the value of the K_TOKEN that is being replaced.

If DSKPP-PRF is used as the MAC algorithm, then the input parameter s MUST consist of the concatenation of the (ASCII) string "MAC 2 computation" ServerID, and R. The parameter dsLen MUST be set to at least 16 (i.e., the length of the MAC MUST be at least 16 octets):

dsLen >= 16

MAC = DSKPP-PRF (K_MAC', "MAC 2 computation" || ServerID || R, dsLen)

The MAC algorithm MUST be the same as the algorithm used by the DSKPP Server to calculate the key confirmation MAC.

5.2.3. KeyProvServerFinished

```
DSKPP Client                          DSKPP Server
 ------------                          ------------
                          <---          KP, MAC, AD
```

When this message is sent:
The DSKPP Server will send this message after authenticating the user and, if authentication passed, generating K_TOKEN and a key package, and associating them with the user's account on the cryptographic server.

Purpose of this message:
With this message, the DSKPP Server transports a key package containing the encrypted provisioning key (K_PROV) and key usage attributes.

What is contained in this message:

A Status attribute equivalent to the server's return code to <KeyProvClientHello>. If the server found an acceptable set of attributes from the client's SAL, then it sets Status to "Success".

The confirmation message MUST include the Key Package (KP) that holds the DSKPP Server's ID, key ID, key type, encrypted provisioning key (K_PROV), encryption method, and additional configuration information. The default symmetric key package format MUST be based on the Portable Symmetric Key Container (PSKC) defined in [RFC6030]. Alternative formats MAY include [RFC6031], PKCS #12 [PKCS-12], or PKCS #5 XML [PKCS-5-XML].

This message MUST include a MAC that the DSKPP Client will use for key confirmation. This key confirmation MAC is calculated using the "MAC 1 computation" as described in the previous section.

Finally, if an existing key is being replaced, then this message MUST also include a server authentication MAC (calculated using the "MAC 2 computation" as described in the previous section), which is passed as AD to the DSKPP Client.

How the DSKPP Client uses this message:

After receiving a <KeyProvServerFinished> message with Status = "Success", the DSKPP Client MUST verify both MACs (MAC and AD). The DSKPP Client MUST terminate the DSKPP run if either MAC does not verify, and MUST, in this case, also delete any nonces, keys, and/or secrets associated with the failed run of the protocol.

If <KeyProvServerFinished> has Status = "Success" and the MACs were verified, then the DSKPP Client MUST extract K_PROV from the provided key package, and derive K_TOKEN. Finally, the DSKPP Client initializes the cryptographic module with K_TOKEN and the corresponding key usage attributes. After this operation, it MUST NOT be possible to overwrite the key unless knowledge of an authorizing key is proven through a MAC on a later <KeyProvServerFinished> message.

6. Protocol Extensions

DSKPP has been designed to be extensible. The sub-sections below define two extensions that are included with the DSKPP schema. Since it is possible that the use of extensions will harm interoperability, protocol designers are advised to carefully consider the use of extensions. 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.

Extensions may be sent with any DSKPP message using the ExtensionsType. The ExtensionsType type is a list of Extensions containing type-value pairs that define optional features supported by a DSKPP Client or server. Each extension MAY be marked as Critical by setting the Critical attribute of the Extension to "true". 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.

6.1. The ClientInfoType Extension

The ClientInfoType extension MAY contain any client-specific data required of an application. This extension MAY be present in a <KeyProvClientHello> or <KeyProvClientNonce> message. When present, this extension MUST NOT be marked as Critical.

DSKPP Servers MUST support this extension. DSKPP 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. DSKPP Servers need not retain the ClientInfoType data.

6.2. The ServerInfoType Extension

The ServerInfoType extension MAY contain any server-specific data required of an application, e.g., state information. This extension is only valid in <KeyProvServerHello> messages for which the Status attribute is set to "Continue". When present, this extension MUST NOT be marked as Critical.

DSKPP Clients MUST support this extension. DSKPP 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 <KeyProvClientNonce> message). DSKPP Clients need not retain the ServerInfoType data.

7. Protocol Bindings

7.1. General Requirements

DSKPP assumes a reliable transport.

7.2. HTTP/1.1 Binding for DSKPP

This section presents a binding of the previous messages to HTTP/1.1 [RFC2616]. This HTTP binding is mandatory to implement, although newer versions of the specification might define additional bindings in the future. Note that the HTTP client will normally be different from the DSKPP Client (i.e., the HTTP client will "proxy" DSKPP messages from the DSKPP Client to the DSKPP Server). Likewise, on the HTTP server side, the DSKPP Server MAY receive DSKPP message from a "front-end" HTTP server. The DSKPP Server will be identified by a specific URL, which may be pre-configured, or provided to the client during initialization.

7.2.1. Identification of DSKPP Messages

The MIME type for all DSKPP messages MUST be

application/dskpp+xml

7.2.2. HTTP Headers

In order to avoid caching of responses carrying DSKPP messages by proxies, 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.

To handle content negotiation, HTTP requests MAY include an HTTP Accept header field. This header field SHOULD should be identified using the MIME type specified in Section 7.2.1. The Accept header MAY include additional content types defined by future versions of this protocol.

There are no other restrictions on HTTP headers, besides the requirement to set the Content-Type header value to the MIME type specified in Section 7.2.1.

7.2.3. HTTP Operations

Persistent connections as defined in HTTP/1.1 are OPTIONAL. DSKPP requests are mapped to HTTP requests with the POST method. DSKPP responses are mapped to HTTP responses.

For the four-pass DSKPP, messages within the protocol run are bound together. In particular, <KeyProvServerHello> is bound to the preceding <KeyProvClientHello> by being transmitted in the corresponding HTTP response. <KeyProvServerHello> MUST have a SessionID attribute, and the SessionID attribute of the subsequent <KeyProvClientNonce> message MUST be identical. <KeyProvServerFinished> is then once again bound to the rest through HTTP (and possibly through a SessionID).

7.2.4. HTTP Status Codes

A DSKPP HTTP responder that refuses to perform a message exchange with a DSKPP 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 DSKPP 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 DSKPP processor, or when the DSKPP processor reports an internal error (for example, the DSKPP XML namespace is incorrect, or the DSKPP schema cannot be located). If a request is received that is not a DSKPP Client message, the DSKPP 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 DSKPP HTTP responder MUST use the 200 status code and provide a suitable DSKPP message (possibly with DSKPP error information included) in the HTTP body.

7.2.5. HTTP Authentication

No support for HTTP/1.1 authentication is assumed.

7.2.6. Initialization of DSKPP

If a user requests key initialization in a browsing session, and if that request has an appropriate Accept header (e.g., to a specific DSKPP Server URL), the DSKPP Server MAY respond by sending a DSKPP

initialization message in an HTTP response with Content-Type set according to Section 7.2.1 and response code set to 200 (OK). The initialization message MAY carry data in its body, such as the URL for the DSKPP Client to use when contacting the DSKPP Server. If the message does carry data, the data MUST be a valid instance of a <KeyProvTrigger> element.

Note that if the user's request was directed to some other resource, the DSKPP Server MUST NOT respond by combining the DSKPP content type with response code 200. In that case, the DSKPP Server SHOULD respond by sending a DSKPP initialization message in an HTTP response with Content-Type set according to Section 7.2.1 and response code set to 406 (Not Acceptable).

7.2.7. Example Messages

a. Initialization from DSKPP Server:

```
HTTP/1.1 200 OK

Cache-Control: no-store
Content-Type: application/dskpp+xml
Content-Length: <some value>

DSKPP initialization data in XML form...
```

b. Initial request from DSKPP Client:

```
POST http://example.com/cgi-bin/DSKPP-server HTTP/1.1

Cache-Control: no-cache, no-store
Pragma: no-cache
Host: www.example.com
Content-Type: application/dskpp+xml
Content-Length: <some value>

DSKPP data in XML form (supported version, supported
algorithms...)
```

c. Initial response from DSKPP Server:

```
HTTP/1.1 200 OK

Cache-Control: no-cache, no-must-revalidate, private
Pragma: no-cache
Content-Type: application/dskpp+xml
Content-Length: <some value>

DSKPP data in XML form (server random nonce, server public key,
...)
```

8. DSKPP XML Schema

8.1. General Processing Requirements

Some DSKPP elements rely on the parties being able to compare received values with stored values. Unless otherwise noted, all elements that have the XML schema "xs:string" type, or a type derived from it, MUST be compared using an exact binary comparison. In particular, DSKPP 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 [UNICODE] and then performing an exact binary comparison.

No collation or sorting order for attributes or element values is defined. Therefore, DSKPP implementations MUST NOT depend on specific sorting orders for values.

8.2. Schema

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:dskpp="urn:ietf:params:xml:ns:keyprov:dskpp"
   xmlns:pskc="urn:ietf:params:xml:ns:keyprov:pskc"
   xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
   targetNamespace="urn:ietf:params:xml:ns:keyprov:dskpp"
   elementFormDefault="qualified" attributeFormDefault="unqualified"
      version="1.0">
   <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"/>
   <xs:import namespace="urn:ietf:params:xml:ns:keyprov:pskc"
      schemaLocation="keyprov-pskc-1.0.xsd"/>
   <xs:complexType name="AbstractRequestType" abstract="true">
      <xs:annotation>
         <xs:documentation> Basic types </xs:documentation>
      </xs:annotation>
      <xs:attribute name="Version" type="dskpp:VersionType"
         use="required"/>
   </xs:complexType>
```

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

<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="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="NoProtocolVariants" />
      <xs:enumeration value="NoSupportedKeyPackages" />
      <xs:enumeration value="AuthenticationDataMissing" />
      <xs:enumeration value="AuthenticationDataInvalid" />
      <xs:enumeration value="InitializationFailed" />
      <xs:enumeration value="ProvisioningPeriodExpired" />
   </xs:restriction>
</xs:simpleType>
```

```
<xs:complexType name="DeviceIdentifierDataType">
   <xs:choice>
      <xs:element name="DeviceId" type="pskc:DeviceInfoType" />
      <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:attribute name="KeyLocation"
      type="dskpp:PlatformType"/>
   <xs:attribute name="AlgorithmLocation"
      type="dskpp:PlatformType"/>
</xs:complexType>

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

<xs:complexType name="AlgorithmsType">
   <xs:sequence maxOccurs="unbounded">
      <xs:element name="Algorithm" type="dskpp:AlgorithmType"/>
   </xs:sequence>
</xs:complexType>

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

<xs:complexType name="ProtocolVariantsType">
   <xs:sequence>
      <xs:element name="FourPass" minOccurs="0" />
      <xs:element name="TwoPass"
         type="dskpp:KeyProtectionDataType" minOccurs="0"/>
   </xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="KeyProtectionDataType">
   <xs:annotation>
      <xs:documentation xml:lang="en">
         This element is only valid for two-pass DSKPP.
      </xs:documentation>
   </xs:annotation>
   <xs:sequence maxOccurs="unbounded">
     <xs:element name="SupportedKeyProtectionMethod"
        type="xs:anyURI"/>
     <xs:element name="Payload"
        type="dskpp:PayloadType" minOccurs="0"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="PayloadType">
   <xs:choice>
      <xs:element name="Nonce" type="dskpp:NonceType" />
      <xs:any namespace="##other" processContents="strict"/>
   </xs:choice>
</xs:complexType>

<xs:complexType name="KeyPackagesFormatType">
   <xs:sequence maxOccurs="unbounded">
      <xs:element name="KeyPackageFormat"
         type="dskpp:KeyPackageFormatType"/>
   </xs:sequence>
</xs:complexType>

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

<xs:complexType name="AuthenticationDataType">
   <xs:annotation>
      <xs:documentation xml:lang="en">
         Authentication Data contains a MAC.
      </xs:documentation>
   </xs:annotation>
   <xs:sequence>
      <xs:element name="ClientID"
         type="dskpp:IdentifierType" minOccurs="0"/>
      <xs:choice>
         <xs:element name="AuthenticationCodeMac"
            type="dskpp:AuthenticationMacType"/>
         <xs:any namespace="##other" processContents="strict" />
      </xs:choice>
   </xs:sequence>
</xs:complexType>
```

```
      <xs:complexType name="AuthenticationMacType">
         <xs:sequence>
            <xs:element minOccurs="0" name="Nonce"
               type="dskpp:NonceType"/>
            <xs:element minOccurs="0" name="IterationCount"
               type="xs:int"/>
            <xs:element name="Mac" type="dskpp:MacType" />
         </xs:sequence>
      </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>

      <xs:complexType name="KeyPackageType">
         <xs:sequence>
            <xs:element minOccurs="0" name="ServerID"
               type="xs:anyURI"/>
            <xs:element minOccurs="0" name="KeyProtectionMethod"
               type="xs:anyURI" />
            <xs:choice>
               <xs:element name="KeyContainer"
                  type="pskc:KeyContainerType"/>
               <xs:any namespace="##other" processContents="strict"/>
            </xs:choice>
         </xs:sequence>
      </xs:complexType>

      <xs:complexType name="InitializationTriggerType">
         <xs:sequence>
            <xs:element minOccurs="0" name="DeviceIdentifierData"
               type="dskpp:DeviceIdentifierDataType" />
            <xs:element minOccurs="0" name="KeyID"
               type="xs:base64Binary"/>
            <xs:element minOccurs="0" name="TokenPlatformInfo"
               type="dskpp:TokenPlatformInfoType" />
            <xs:element name="AuthenticationData"
               type="dskpp:AuthenticationDataType" />
            <xs:element minOccurs="0" name="ServerUrl"
               type="xs:anyURI"/>
            <xs:any minOccurs="0" namespace="##other"
               processContents="strict" />
         </xs:sequence>
      </xs:complexType>
```

```
<xs:complexType name="ExtensionsType">
   <xs:annotation>
      <xs:documentation> Extension types </xs:documentation>
   </xs:annotation>
   <xs:sequence maxOccurs="unbounded">
      <xs:element name="Extension"
         type="dskpp: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 mixed="false">
      <xs:extension base="dskpp:AbstractExtensionType">
         <xs:sequence>
            <xs:element name="Data" type="xs:base64Binary"/>
         </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>

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

<xs:element name="KeyProvTrigger"
   type="dskpp:KeyProvTriggerType">
   <xs:annotation>
      <xs:documentation> DSKPP PDUs </xs:documentation>
   </xs:annotation>
</xs:element>
```

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

      <xs:element name="KeyProvClientHello"
         type="dskpp:KeyProvClientHelloPDU">
         <xs:annotation>
            <xs:documentation>KeyProvClientHello PDU</xs:documentation>
         </xs:annotation>
      </xs:element>
      <xs:complexType name="KeyProvClientHelloPDU">
         <xs:annotation>
            <xs:documentation xml:lang="en">
               Message sent from DSKPP Client to DSKPP Server to
               initiate a DSKPP session.
            </xs:documentation>
         </xs:annotation>
         <xs:complexContent mixed="false">
            <xs:extension base="dskpp:AbstractRequestType">
               <xs:sequence>
                  <xs:element minOccurs="0" name="DeviceIdentifierData"
                     type="dskpp:DeviceIdentifierDataType" />
                  <xs:element minOccurs="0" name="KeyID"
                     type="xs:base64Binary" />
                  <xs:element minOccurs="0" name="ClientNonce"
                     type="dskpp:NonceType" />
                  <xs:element name="SupportedKeyTypes"
                     type="dskpp:AlgorithmsType" />
                  <xs:element name="SupportedEncryptionAlgorithms"
                     type="dskpp:AlgorithmsType" />
                  <xs:element name="SupportedMacAlgorithms"
                     type="dskpp:AlgorithmsType" />
                  <xs:element minOccurs="0"
                     name="SupportedProtocolVariants"
                     type="dskpp:ProtocolVariantsType" />
```

```
                <xs:element minOccurs="0" name="SupportedKeyPackages"
                   type="dskpp:KeyPackagesFormatType" />
                <xs:element minOccurs="0" name="AuthenticationData"
                   type="dskpp:AuthenticationDataType" />
                <xs:element minOccurs="0" name="Extensions"
                   type="dskpp:ExtensionsType" />
             </xs:sequence>
          </xs:extension>
       </xs:complexContent>
    </xs:complexType>

    <xs:element name="KeyProvServerHello"
       type="dskpp:KeyProvServerHelloPDU">
       <xs:annotation>
          <xs:documentation>KeyProvServerHello PDU</xs:documentation>
       </xs:annotation>
    </xs:element>
    <xs:complexType name="KeyProvServerHelloPDU">
       <xs:annotation>
          <xs:documentation xml:lang="en">
             Response message sent from DSKPP Server to DSKPP Client
             in four-pass DSKPP.
          </xs:documentation>
       </xs:annotation>
       <xs:complexContent mixed="false">
          <xs:extension base="dskpp:AbstractResponseType">
             <xs:sequence minOccurs="0">
                <xs:element name="KeyType"
                   type="dskpp:AlgorithmType"/>
                <xs:element name="EncryptionAlgorithm"
                   type="dskpp:AlgorithmType" />
                <xs:element name="MacAlgorithm"
                   type="dskpp:AlgorithmType"/>
                <xs:element name="EncryptionKey"
                   type="ds:KeyInfoType"/>
                <xs:element name="KeyPackageFormat"
                   type="dskpp:KeyPackageFormatType" />
                <xs:element name="Payload" type="dskpp:PayloadType"/>
                <xs:element minOccurs="0" name="Extensions"
                   type="dskpp:ExtensionsType" />
                <xs:element minOccurs="0" name="Mac"
                   type="dskpp:MacType"/>
             </xs:sequence>
          </xs:extension>
       </xs:complexContent>
    </xs:complexType>
```

```
      <xs:element name="KeyProvClientNonce"
         type="dskpp:KeyProvClientNoncePDU">
         <xs:annotation>
            <xs:documentation>KeyProvClientNonce PDU</xs:documentation>
         </xs:annotation>
      </xs:element>
      <xs:complexType name="KeyProvClientNoncePDU">
         <xs:annotation>
            <xs:documentation xml:lang="en">
               Response message sent from DSKPP Client to
               DSKPP Server in a four-pass DSKPP session.
            </xs:documentation>
         </xs:annotation>
         <xs:complexContent mixed="false">
            <xs:extension base="dskpp:AbstractRequestType">
               <xs:sequence>
                  <xs:element name="EncryptedNonce"
                     type="xs:base64Binary"/>
                  <xs:element minOccurs="0" name="AuthenticationData"
                     type="dskpp:AuthenticationDataType" />
                  <xs:element minOccurs="0" name="Extensions"
                     type="dskpp:ExtensionsType" />
               </xs:sequence>
               <xs:attribute name="SessionID"
                  type="dskpp:IdentifierType" use="required"/>
            </xs:extension>
         </xs:complexContent>
      </xs:complexType>

      <xs:element name="KeyProvServerFinished"
         type="dskpp:KeyProvServerFinishedPDU">
         <xs:annotation>
            <xs:documentation>
               KeyProvServerFinished PDU
            </xs:documentation>
         </xs:annotation>
      </xs:element>
      <xs:complexType name="KeyProvServerFinishedPDU">
         <xs:annotation>
            <xs:documentation xml:lang="en">
               Final message sent from DSKPP Server to DSKPP Client in
               a DSKPP session.  A MAC value serves for key
               confirmation, and optional AuthenticationData serves for
               server authentication.
            </xs:documentation>
         </xs:annotation>
         <xs:complexContent mixed="false">
            <xs:extension base="dskpp:AbstractResponseType">
```

```
                <xs:sequence minOccurs="0">
                   <xs:element name="KeyPackage"
                      type="dskpp:KeyPackageType" />
                   <xs:element minOccurs="0" name="Extensions"
                      type="dskpp:ExtensionsType" />
                   <xs:element name="Mac" type="dskpp:MacType" />
                   <xs:element minOccurs="0" name="AuthenticationData"
                      type="dskpp:AuthenticationMacType" />
                </xs:sequence>
             </xs:extension>
          </xs:complexContent>
       </xs:complexType>
     </xs:schema>
```

9. Conformance Requirements

In order to assure that all implementations of DSKPP can interoperate, the DSKPP Server:

a. MUST implement the four-pass variation of the protocol (Section 4)

b. MUST implement the two-pass variation of the protocol (Section 5)

c. MUST support user authentication (Section 3.2.1)

d. MUST support the following key derivation functions:
   * DSKPP-PRF-AES DSKPP-PRF realization (Appendix D)
   * DSKPP-PRF-SHA256 DSKPP-PRF realization (Appendix D)

e. MUST support the following encryption mechanisms for protection of the client nonce in the four-pass protocol:
   * Mechanism described in Section 4.2.4

f. MUST support one of the following encryption algorithms for symmetric key operations, e.g., key wrap:
   * KW-AES128 without padding; refer to http://www.w3.org/2001/04/xmlenc#kw-aes128 in [XMLENC]
   * KW-AES128 with padding; refer to http://www.w3.org/2001/04/xmlenc#kw-aes128 in [XMLENC] and [RFC5649]
   * AES-CBC-128; refer to [FIPS197-AES]

g. MUST support the following encryption algorithms for asymmetric key operations, e.g., key transport:
   * RSA Encryption Scheme [PKCS-1]

h. MUST support the following integrity/KDF MAC functions:
   * DSKPP-PRF-AES (Appendix D)
   * DSKPP-PRF-SHA256 (Appendix D)

i. MUST support the PSKC key package [RFC6030]; all three PSKC key protection methods (Key Transport, Key Wrap, and Passphrase-Based Key Wrap) MUST be implemented

j. MAY support the ASN.1 key package as defined in [RFC6031]

DSKPP Clients MUST support either the two-pass or the four-pass variant of the protocol. DSKPP Clients MUST fulfill all requirements listed in item (c) - (j).

Finally, implementations of DSKPP MUST bind DSKPP messages to HTTP/1.1 as described in Section 7.2.

Of course, DSKPP is a security protocol, and one of its major functions is to allow only authorized parties to successfully initialize a cryptographic module with a new symmetric key. Therefore, a particular implementation may be configured with any of a number of restrictions concerning algorithms and trusted authorities that will prevent universal interoperability.

10. Security Considerations

10.1. General

DSKPP is designed to protect generated keying material from exposure. No entities other than the DSKPP Server and the cryptographic module will have access to a generated K_TOKEN if the cryptographic algorithms used are of sufficient strength and, on the DSKPP Client side, generation and encryption of R_C and generation of K_TOKEN take place as specified in the cryptographic module. This applies even if malicious software is present in the DSKPP Client. However, as discussed in the following sub-sections, DSKPP does not protect against certain other threats resulting from man-in-the-middle attacks and other forms of attacks. DSKPP MUST, therefore, be run over a transport providing confidentiality and integrity, such as HTTP over Transport Layer Security (TLS) with a suitable ciphersuite [RFC2818], when such threats are a concern. Note that TLS ciphersuites with anonymous key exchanges are not suitable in those situations [RFC5246].

10.2. Active Attacks

10.2.1. 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 keying material.

10.2.2. Message Modifications

Modifications to a <KeyProvTrigger> message will either cause denial of service (modifications of any of the identifiers or the Authentication Code) or will cause the DSKPP Client to contact the wrong DSKPP Server. The latter is in effect a man-in-the-middle attack and is discussed further in Section 10.2.7.

An attacker may modify a <KeyProvClientHello> message. This means that the attacker could indicate a different key or device than the one intended by the DSKPP Client, and could also suggest other cryptographic algorithms than the ones preferred by the DSKPP Client, e.g., cryptographically weaker ones. The attacker could also suggest earlier versions of DSKPP, in case these versions have been shown to have vulnerabilities. These modifications could lead to an attacker succeeding in initializing or modifying another cryptographic module than the one intended (i.e., the server assigning the generated key to the wrong module) or gaining access to a generated key through the use of weak cryptographic algorithms or protocol versions. DSKPP 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 module) is not possible when the shared-key variant of DSKPP is employed (assuming existing shared keys are unique per cryptographic module), but is possible in the public key variation. Therefore, DSKPP Servers MUST NOT accept unilaterally provided device identifiers in the public key variation. This is also indicated in the protocol description. In the shared-key variation, 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 cryptographic module with the identified key. The result of this attack could be that the generated key is associated with the correct cryptographic module but the module is associated with the incorrect user. See Section 10.5 for a further discussion of this threat and possible countermeasures.

An attacker may also modify a <KeyProvServerHello> 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 may also provide a

different nonce than the one sent by the legitimate server. Clients MAY 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 DSKPP run would result in the replacement of an existing key, the <Mac> element protects against modifications of R_S.

Modifications of <KeyProvClientNonce> 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-DSKPP 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 cryptographic module 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 10.5 below. Note that use of TLS does not protect against this attack if the attacker has access to the DSKPP Client (e.g., through malicious software, "Trojans") [RFC5246].

Finally, attackers may also modify the <KeyProvServerFinished> message. Replacing the <Mac> element will only result in denial of service. Replacement of any other element may cause the DSKPP Client to associate, e.g., the wrong service with the generated key. DSKPP SHOULD be run over a transport providing confidentiality and integrity when this is a concern.

10.2.3. Message Deletion

Message deletion will not cause any other harm than denial of service, since a cryptographic module MUST NOT change its state (i.e., "commit" to a generated key) until it receives the final message from the DSKPP Server and successfully has processed that message, including validation of its MAC. A deleted <KeyProvServerFinished> message will not cause the server to end up in an inconsistent state vis-a-vis the cryptographic module if the server implements the suggestions in Section 10.5.

10.2.4. Message Insertion

An active attacker may initiate a DSKPP run at any time, and suggest any device identifier. DSKPP Server implementations MAY receive some protection against inadvertently initializing a key or inadvertently replacing an existing key or assigning a key to a cryptographic module by initializing the DSKPP run by use of the <KeyProvTrigger>. The <AuthenticationData> element allows the server to associate a DSKPP run e.g., with an earlier user-authenticated session. The security of this method, therefore, depends on the ability to protect the <AuthenticationData> element in the DSKPP initialization message. If an eavesdropper is able to capture this message, he may race the legitimate user for a key initialization. DSKPP over a transport providing confidentiality and integrity, coupled with the recommendations in Section 10.5, 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.

10.2.5. Message Replay

During four-pass DSKPP, attempts to replay a previously recorded DSKPP message will be detected, as the use of nonces ensures that both parties are live. For example, a DSKPP Client knows that a server it is communicating with is "live" since the server MUST create a MAC on information sent by the client.

The same is true for two-pass DSKPP thanks to the requirement that the client sends R in the <KeyProvClientHello> message and that the server includes R in the MAC computation.

10.2.6. Message Reordering

An attacker may attempt to re-order four-pass DSKPP messages but this will be detected, as each message is of a unique type. Note: Message re-ordering attacks cannot occur in two-pass DSKPP since each party sends at most one message each.

10.2.7. 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 DSKPP Client. This threat and countermeasures to it are discussed in Section 4.1.1. An attacker posing as a man in the middle may also be acting as a proxy and, hence, may not interfere with DSKPP runs but still learn valuable information; see Section 10.3.

10.3. Passive Attacks

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

If DSKPP is not run over a transport providing confidentiality, a passive attacker may learn:

- What cryptographic modules a particular user possesses
- The identifiers of keys on those cryptographic modules and other attributes pertaining to those keys, e.g., the lifetime of the keys
- DSKPP versions and cryptographic algorithms supported by a particular DSKPP Client or server
- Any value present in an <extension> that is part of <KeyProvClientHello>

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

10.4. Cryptographic Attacks

An attacker with unlimited access to an initialized cryptographic module may use the module as an "oracle" to pre-compute values that later on may be used to impersonate the DSKPP Server. Section 4.1.1 contains a discussion of this threat and steps RECOMMENDED to protect against it.

Implementers are advised that cryptographic algorithms become weaker with time. As new cryptographic techniques are developed and computing performance improves, the work factor to break a particular cryptographic algorithm will reduce. Therefore, cryptographic

algorithm implementations SHOULD be modular allowing new algorithms to be readily inserted. That is, implementers SHOULD be prepared to regularly update the algorithms in their implementations.

10.5. Attacks on the Interaction between DSKPP and User Authentication

If keys generated in DSKPP will be associated with a particular user at the DSKPP Server (or a server trusted by, and communicating with the DSKPP 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 variation or the shared-secret variation of DSKPP is employed to encrypt the client nonce), the server SHOULD NOT commit to associate a generated K_TOKEN with the given cryptographic module until the user simultaneously has proven both possession of the device that hosts the cryptographic module containing K_TOKEN and some out-of-band provided authenticating information (e.g., an Authentication Code). For example, if the cryptographic module is a one-time password token, the user could be required to authenticate with both a one-time password generated by the cryptographic module and an out-of-band provided Authentication Code in order to have the server "commit" to the generated OTP value for the given user. Preferably, the user SHOULD perform this operation from another host than the one used to initialize keys on the cryptographic module, in order to minimize the risk of malicious software on the client interfering with the process.

Note: This scenario, wherein the attacker replaces a client-provided R_C with his own R'C, does not apply to two-pass DSKPP as the client does not provide any entropy to K_TOKEN. The attack as such (and its countermeasures) still applies to two-pass DSKPP, however, as it essentially is a man-in-the-middle attack.

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

10.6. Miscellaneous Considerations

10.6.1. Client Contributions to K_TOKEN Entropy

In four-pass DSKPP, both the client and the server provide randomizing material to K_TOKEN, in a manner that allows both parties to verify that they did contribute to the resulting key. In the two-pass DSKPP version defined herein, only the server contributes to the entropy of K_TOKEN. This means that a broken or compromised (pseudo)random number generator in the server may cause more damage than it would in the four-pass variant. Server implementations SHOULD therefore take extreme care to ensure that this situation does not occur.

10.6.2. Key Confirmation

four-pass DSKPP Servers provide key confirmation through the MAC on R_C in the <KeyProvServerFinished> message. In the two-pass DSKPP variant described herein, key confirmation is provided by the MAC including R, using K_MAC.

10.6.3. Server Authentication

DSKPP Servers MUST authenticate themselves whenever a successful DSKPP two-pass protocol run would result in an existing K_TOKEN being replaced by a K_TOKEN', or else a denial-of-service attack where an unauthorized DSKPP Server replaces a K_TOKEN with another key would be possible. In two-pass DSKPP, servers authenticate by including the AuthenticationDataType extension containing a MAC as described in Section 5 for two-pass DSKPP.

Whenever a successful DSKPP two-pass protocol run would result in an existing K_TOKEN being replaced by a K_TOKEN', the DSKPP Client and Server MUST do the following to prevent a denial-of-service attack where an unauthorized DSKPP Server replaces a K_TOKEN with another key:

- The DSKPP Server MUST use the AuthenticationDataType extension to transmit a second MAC, calculated as described in Section 5.2.2.

- The DSKPP Client MUST authenticate the server using the MAC contained in the AuthenticationDataType extension received from the DSKPP Server to which it is connected.

10.6.4. User Authentication

A DSKPP Server MUST authenticate a client to ensure that K_TOKEN is delivered to the intended device. The following measures SHOULD be considered:

- When an Authentication Code is used for client authentication, a password dictionary attack on the Authentication Data is possible.
- The length of the Authentication Code when used over a non-secure channel SHOULD be longer than what is used over a secure channel. When a device, e.g., some mobile phones with small screens, cannot handle a long Authentication Code in a user-friendly manner, DSKPP SHOULD rely on a secure channel for communication.
- In the case that a non-secure channel has to be used, the Authentication Code SHOULD be sent to the server MAC'd as specified in Section 3.4.1. The Authentication Code and nonce value MUST be strong enough to prevent offline brute-force recovery of the Authentication Code from the Hashed MAC (HMAC) data. Given that the nonce value is sent in plaintext format over a non-secure transport, the cryptographic strength of the Authentication Data depends more on the quality of the Authentication Code.
- When the Authentication Code is sent from the DSKPP Server to the device in a DSKPP initialization trigger message, an eavesdropper may be able to capture this message and race the legitimate user for a key initialization. To prevent this, the transport layer used to send the DSKPP trigger MUST provide confidentiality and integrity, e.g. a secure browser session.

10.6.5. Key Protection in Two-Pass DSKPP

Three key protection methods are defined for the different usages of two-pass DSKPP, which MUST be supported by a key package format, such as [RFC6030] and [RFC6031]. Therefore, key protection in the two-pass DSKPP is dependent upon the security of the key package format selected for a protocol run. Some considerations for the Passphrase-Based Key Wrap method follow.

The Passphrase-Based Key Wrap method SHOULD depend upon the PBKDF2 function from [PKCS-5] to generate an encryption key from a passphrase and salt string. It is important to note that passphrase-based encryption is generally limited in the security that it provides despite the use of salt and iteration count in PBKDF2 to increase the complexity of attack. Implementations SHOULD therefore

take additional measures to strengthen the security of the Passphrase-Based Key Wrap method. The following measures SHOULD be considered where applicable:

- The passphrase is the same as the one-time password component of the Authentication Code (see Section 3.4.1) for a description of the AC format). The passphrase SHOULD be selected well, and usage guidelines such as the ones in [NIST-PWD] SHOULD be taken into account.

- A different passphrase SHOULD be used for every key initialization wherever possible (the use of a global passphrase for a batch of cryptographic modules SHOULD be avoided, for example). One way to achieve this is to use randomly generated passphrases.

- The passphrase SHOULD be protected well if stored on the server and/or on the cryptographic module and SHOULD be delivered to the device's user using secure methods.

- User pre-authentication SHOULD be implemented to ensure that K_TOKEN is not delivered to a rogue recipient.

- The iteration count in PBKDF2 SHOULD be high to impose more work for an attacker using brute-force methods (see [PKCS-5] for recommendations). However, it MUST be noted that the higher the count, the more work is required on the legitimate cryptographic module to decrypt the newly delivered K_TOKEN. Servers MAY use relatively low iteration counts to accommodate devices with limited processing power such as some PDA and cell phones when other security measures are implemented and the security of the Passphrase-Based Key Wrap method is not weakened.

- TLS [RFC5246] SHOULD be used where possible to protect a two-pass protocol run. Transport level security provides a second layer of protection for the newly generated K_TOKEN.

10.6.6. Algorithm Agility

Many protocols need to be algorithm agile. One reason for this is that in the past many protocols had fixed sized fields for information such as hash outputs, keys, etc. This is not the case for DSKPP, except for the key size in the computation of DSKPP-PRF. Another reason was that protocols did not support algorithm negotiation. This is also not the case for DSKPP, except for the use of SHA-256 in the MAC confirmation message. Updating the key size for DSKPP-PRF or the MAC confirmation message algorithm will require a new version of the protocol, which is supported with the Version attribute.

11. Internationalization Considerations

DSKPP is meant for machine-to-machine communications; as such, its elements are tokens not meant for direct human consumption. DSKPP exchanges information using XML. All XML processors are required to understand UTF-8 [RFC3629] encoding, and therefore all DSKPP Clients and servers MUST understand UTF-8 encoded XML. Additionally, DSKPP Servers and clients MUST NOT encode XML with encodings other than UTF-8.

12. IANA Considerations

This document requires several IANA registrations, detailed below.

12.1. URN Sub-Namespace Registration

This section registers a new XML namespace, "urn:ietf:params:xml:ns:keyprov:dskpp" per the guidelines in [RFC3688]:

URI: urn:ietf:params:xml:ns:keyprov:dskpp
Registrant Contact:
IETF, KEYPROV Working Group (keyprov@ietf.org), Andrea Doherty (andrea.doherty@rsa.com)

XML:
BEGIN

```
<?xml version="1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
   <title>DSKPP Messages</title>
</head>
<body>
   <h1>Namespace for DSKPP Messages</h1>
   <h2>urn:ietf:params:xml:ns:keyprov:dskpp</h2>
   <p>See RFC 6063</p>
</body>
</html>
```

END

12.2. XML Schema Registration

This section registers an XML schema as per the guidelines in [RFC3688].

URI: urn:ietf:params:xml:ns:keyprov:dskpp
Registrant Contact:
IETF, KEYPROV Working Group (keyprov@ietf.org), Andrea Doherty (andrea.doherty@rsa.com)
Schema:
The XML for this schema can be found as the entirety of Section 8 of this document.

12.3. MIME Media Type Registration

This section registers the "application/dskpp+xml" MIME type:

To: ietf-types@iana.org
Subject: Registration of MIME media type application/dskpp+xml
MIME media type name: application
MIME subtype name: dskpp+xml
Required parameters: (none)
Optional parameters: charset
Indicates the character encoding of enclosed XML.
Encoding considerations: Uses XML, which can employ 8-bit characters, depending on the character encoding used. See [RFC3023], Section 3.2. Implementations need to support UTF-8 [RFC3629].
Security considerations: This content type is designed to carry protocol data related to key management. Security mechanisms are built into the protocol to ensure that various threats are dealt with. Refer to Section 10 of RFC 6063 for more details
Interoperability considerations: None
Published specification: RFC 6063.
Applications that use this media type: Protocol for key exchange.
Additional information:
Magic Number(s): (none)
File extension(s): .xmls
Macintosh File Type Code(s): (none)
Person & email address to contact for further information:
Andrea Doherty (andrea.doherty@rsa.com)
Intended usage: LIMITED USE
Author/Change controller: The IETF
Other information: This media type is a specialization of application/xml [RFC3023], and many of the considerations described there also apply to application/dskpp+xml.

12.4. Status Code Registration

This section registers status codes included in each DSKPP response message. The status codes are defined in the schema in the <StatusCode> type definition contained in the XML schema in Section 8. The following summarizes the registry:

Related Registry:
KEYPROV DSKPP Registries, Status codes for DSKPP

Defining RFC:
RFC 6063.
Registration/Assignment Procedures:
Following the policies outlined in [RFC3575], the IANA policy for assigning new values for the status codes for DSKPP MUST be "Specification Required" and their meanings MUST be documented in an RFC or in some other permanent and readily available reference, in sufficient detail that interoperability between independent implementations is possible. No mechanism to mark entries as "deprecated" is envisioned. It is possible to update entries from the registry.

Registrant Contact:
IETF, KEYPROV working group (keyprov@ietf.org),
Andrea Doherty (andrea.doherty@rsa.com)

12.5. DSKPP Version Registration

This section registers DSKPP version numbers. The registry has the following structure:

DSKPP Version	Specification
1.0	This document

Standards action is required to define new versions of DSKPP. It is not envisioned to deprecate, delete, or modify existing DSKPP versions.

12.6. PRF Algorithm ID Sub-Registry

This specification relies on a cryptographic primitive, called "DSKPP-PRF" that provides a deterministic transformation of a secret key k and a varying length octet string s to a bit string of specified length dsLen. From the point of view of this specification, DSKPP-PRF is a "black-box" function that, given the inputs, generates a pseudorandom value that can be realized by any

appropriate and competent cryptographic technique. Section 3.4.2 provides two realizations of DSKPP-PRF, DSKPP-PRF-AES, and DSKPP-PRF-SHA256.

This section registers the identifiers associated with these realizations. PRF Algorithm ID Sub-registries are to be subject to "Specification Required" as per RFC 5226 [RFC5226]. Updates MUST be documented in an RFC or in some other permanent and readily available reference, in sufficient detail that interoperability between independent implementations is possible.

Expert approval is required to deprecate a sub-registry. Once deprecated, the PRF Algorithm ID SHOULD NOT be used in any new implementations.

12.6.1. DSKPP-PRF-AES

This section registers the following in the IETF XML namespace registry.

Common Name:
DSKPP-PRF-AES

URI:
urn:ietf:params:xml:ns:keyprov:dskpp:prf-aes-128

Identifier Definition:
The DSKPP-PRF-AES algorithm realization is defined in Appendix D.2.2 of this document.

Registrant Contact:
IETF, KEYPROV working group (keyprov@ietf.org),
Andrea Doherty (andrea.doherty@rsa.com)

12.6.2. DSKPP-PRF-SHA256

This section registers the following in the IETF XML namespace registry.

Common Name:
DSKPP-PRF-SHA256

URI:
urn:ietf:params:xml:ns:keyprov:dskpp:prf-sha256

Identifier Definition:
The DSKPP-PRF-SHA256 algorithm realization is defined in Appendix D.3.2 of this document.

Registrant Contact:
IETF, KEYPROV working group (keyprov@ietf.org),
Andrea Doherty (andrea.doherty@rsa.com)

12.7. Key Container Registration

This section registers the Key Container type.

Key Container:
The registration name for the Key Container.

Specification:
Key Container defines a key package format that specifies how a key should be protected using the three key protection methods provided in Section 5.1.

Registration Procedure:
Following the policies outlined in [RFC3575], the IANA policy for assigning new values for the status codes for DSKPP MUST be "Specification Required" and their meanings MUST be documented in an RFC or in some other permanent and readily available reference, in sufficient detail that interoperability between independent implementations is possible.

Deprecated:
TRUE if based on expert approval this entry has been deprecated and SHOULD NOT be used in any new implementations. Otherwise, FALSE.

Identifiers:
The initial URIs for the Key Container defined for this version of the document are listed here:

Name: PSKC Key Container
URI: urn:ietf:params:xml:ns:keyprov:dskpp:pskc-key-container
Specification: [RFC6030]
Deprecated: FALSE

Name: SKPC Key Container
URI: urn:ietf:params:xml:ns:keyprov:dskpp:skpc-key-container
Specification: [RFC6031]
Deprecated: FALSE

Name: PKCS12 Key Container
URI: urn:ietf:params:xml:ns:keyprov:dskpp:pkcs12-key-container
Specification: [PKCS-12]
Deprecated: FALSE

```
      Name:  PKCS5-XML Key Container
      URI:  urn:ietf:params:xml:ns:keyprov:dskpp:pkcs5-xml-key-container
      Specification:  [PKCS-5-XML]
      Deprecated:  FALSE

   Registrant Contact:
      IETF, KEYPROV working group (keyprov@ietf.org),
      Andrea Doherty (andrea.doherty@rsa.com)
```

13. Intellectual Property Considerations

RSA and RSA Security 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.

14. Contributors

This work is based on information contained in [RFC4758], authored by Magnus Nystrom, with enhancements borrowed from an individual document coauthored by Mingliang Pei and Salah Machani (e.g., user authentication, and support for multiple key package formats).

We would like to thank Philip Hoyer for his work in aligning DSKPP and PSKC schemas.

We would also like to thank Hannes Tschofenig and Phillip Hallam-Baker for their reviews, feedback, and text contributions.

15. Acknowledgements

We would like to thank the following for review of previous DSKPP document versions:

- Dr. Ulrike Meyer (Review June 2007)
- Niklas Neumann (Review June 2007)
- Shuh Chang (Review June 2007)
- Hannes Tschofenig (Review June 2007 and again in August 2007)
- Sean Turner (Reviews August 2007 and again in July 2008)
- John Linn (Review August 2007)
- Philip Hoyer (Review September 2007)
- Thomas Roessler (Review November 2007)
- Lakshminath Dondeti (Comments December 2007)
- Pasi Eronen (Comments December 2007)
- Phillip Hallam-Baker (Review and Edits November 2008 and again in January 2009)
- Alexey Melnikov (Review May 2010)
- Peter Saint-Andre (Review May 2010)

We would also like to thank the following for their input to selected design aspects of DSKPP:

- Anders Rundgren (Key Package Format and Client Authentication Data)
- Thomas Roessler (HTTP Binding)
- Hannes Tschofenig (HTTP Binding)
- Phillip Hallam-Baker (Registry for Algorithms)
- N. Asokan (original observation of weakness in Authentication Data)

Finally, we would like to thank Robert Griffin for opening communication channels for us with the IEEE P1619.3 Key Management Group, and facilitating our groups in staying informed of potential areas (especially key provisioning and global key identifiers of collaboration) of collaboration.

16. References

16.1. Normative References

[FIPS180-SHA] National Institute of Standards and Technology, "Secure Hash Standard", FIPS 180-2, February 2004, <http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf>.

[FIPS197-AES] 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>.

[ISO3309] International Organization for Standardization, "ISO Information Processing Systems - Data Communication - High-Level Data Link Control Procedure - Frame Structure", ISO 3309, 3rd Edition, October 1984.

[PKCS-1] RSA Laboratories, "RSA Cryptography Standard", PKCS #1 Version 2.1, June 2002, <http://www.rsasecurity.com/rsalabs/pkcs/>.

[PKCS-5] RSA Laboratories, "Password-Based Cryptography Standard", PKCS #5 Version 2.0, March 1999, <http://www.rsasecurity.com/rsalabs/pkcs/>.

[PKCS-5-XML] RSA Laboratories, "XML Schema for PKCS #5 Version 2.0", PKCS #5 Version 2.0 Amd.1 (FINAL DRAFT), October 2006, <http://www.rsasecurity.com/rsalabs/pkcs/>.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

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

[RFC2616] 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.

[RFC3394] Schaad, J. and R. Housley, "Advanced Encryption Standard (AES) Key Wrap Algorithm", RFC 3394, September 2002.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC4013] Zeilenga, K., "SASLprep: Stringprep Profile for User Names and Passwords", RFC 4013, February 2005.

[RFC4210] Adams, C., Farrell, S., Kause, T., and T. Mononen, "Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP)", RFC 4210, September 2005.

[RFC5272] Schaad, J. and M. Myers, "Certificate Management over CMS (CMC)", RFC 5272, June 2008.

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

[RFC5649] Housley, R. and M. Dworkin, "Advanced Encryption Standard (AES) Key Wrap with Padding Algorithm", RFC 5649, September 2009.

[RFC6030] Hoyer, P., Pei, M., and S. Machani, "Portable Symmetric Key Container (PSKC)", RFC 6030, October 2010.

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

[XML] W3C, "Extensible Markup Language (XML) 1.0 (Fifth Edition)", W3C Recommendation, November 2008, <http://www.w3.org/TR/2006/REC-xml-20060816/>.

[XMLDSIG] W3C, "XML Signature Syntax and Processing", W3C Recommendation, February 2002, <http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/>.

[XMLENC] W3C, "XML Encryption Syntax and Processing", W3C Recommendation, December 2002, <http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/>.

16.2. Informative References

[CT-KIP-P11] RSA Laboratories, "PKCS #11 Mechanisms for the Cryptographic Token Key Initialization Protocol", PKCS #11 Version 2.20 Amd.2, December 2005, <http://www.rsasecurity.com/rsalabs/pkcs/>.

[FAQ] RSA Laboratories, "Frequently Asked Questions About Today's Cryptography", Version 4.1, 2000.

[NIST-PWD] National Institute of Standards and Technology, "Password Usage", FIPS 112, May 1985, <http://www.itl.nist.gov/fipspubs/fip112.htm>.

[NIST-SP800-38B] International Organization for Standardization, "Recommendations for Block Cipher Modes of Operation: The CMAC Mode for Authentication", NIST SP800-38B, May 2005, <http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf>.

[NIST-SP800-57] National Institute of Standards and Technology, "Recommendation for Key Management - Part I: General (Revised)", NIST 800-57, March 2007, <http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf>.

[PKCS-11] RSA Laboratories, "Cryptographic Token Interface Standard", PKCS #11 Version 2.20, June 2004, <http://www.rsasecurity.com/rsalabs/pkcs/>.

[PKCS-12] "Personal Information Exchange Syntax Standard", PKCS #12 Version 1.0, 2005, <ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-12/pkcs-12v1.pdf>.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

[RFC3023] Murata, M., St. Laurent, S., and D. Kohn, "XML Media Types", RFC 3023, January 2001.

[RFC3575] Aboba, B., "IANA Considerations for RADIUS (Remote Authentication Dial In User Service)", RFC 3575, July 2003.

[RFC3688] Mealling, M., "The IETF XML Registry", BCP 81, RFC 3688, January 2004.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC4758] Nystroem, M., "Cryptographic Token Key Initialization Protocol (CT-KIP) Version 1.0 Revision 1", RFC 4758, November 2006.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

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

[RFC6031] Turner, S. and R. , "Cryptographic Message Syntax (CMS) Symmetric Key Package Content Type", RFC 6031, December 2010.

[XMLNS] W3C, "Namespaces in XML", W3C Recommendation, January 1999, <http://www.w3.org/TR/2009/REC-xml-names-20091208>.

Appendix A. Usage Scenarios

DSKPP is expected to be used to provision symmetric keys to cryptographic modules in a number of different scenarios, each with its own special requirements, as described below. This appendix forms an informative part of the document.

A.1. Single Key Request

The usual scenario is that a cryptographic module makes a request for a symmetric key from a provisioning server that is located on the local network or somewhere on the Internet. Depending upon the deployment scenario, the provisioning server may generate a new key on-the-fly or use a pre-generated key, e.g., one provided by a legacy back-end issuance server. The provisioning server assigns a unique key ID to the symmetric key and provisions it to the cryptographic module.

A.2. Multiple Key Requests

A cryptographic module makes multiple requests for symmetric keys from the same provisioning server. The symmetric keys need not be of the same type, i.e., the keys may be used with different symmetric key cryptographic algorithms, including one-time password authentication algorithms, and the AES encryption algorithm.

A.3. User Authentication

In some deployment scenarios, a key issuer may rely on a third-party provisioning service. In this case, the issuer directs provisioning requests from the cryptographic module to the provisioning service. As such, it is the responsibility of the issuer to authenticate the user through some out-of-band means before granting him rights to acquire keys. Once the issuer has granted those rights, the issuer provides an Authentication Code to the user and makes it available to the provisioning service, so that the user can prove that he is authorized to acquire keys.

A.4. Provisioning Time-Out Policy

An issuer may provide a time-limited Authentication Code to a user during registration, which the user will input into the cryptographic module to authenticate themselves with the provisioning server. The server will allow a key to be provisioned to the cryptographic module hosted by the user's device when user authentication is required only if the user inputs a valid Authentication Code within the fixed time period established by the issuer.

A.5. Key Renewal

A cryptographic module requests renewal of the symmetric key material attached to a key ID, as opposed to keeping the key value constant and refreshing the metadata. Such a need may occur in the case when a user wants to upgrade her device that houses the cryptographic module or when a key has expired. When a user uses the same cryptographic module for example, to perform strong authentication at multiple Web login sites, keeping the same key ID removes the need for the user to register a new key ID at each site.

A.6. Pre-Loaded Key Replacement

This scenario represents a special case of symmetric key renewal in which a local administrator can authenticate the user procedurally before initiating the provisioning process. It also allows for a device issuer to pre-load a key onto a cryptographic module with a restriction that the key is replaced with a new key prior to use of the cryptographic module. Another variation of this scenario is the organization who recycles devices. In this case, a key issuer would provision a new symmetric key to a cryptographic module hosted on a device that was previously owned by another user.

Note that this usage scenario is essentially the same as the previous scenario wherein the same key ID is used for renewal.

A.7. Pre-Shared Manufacturing Key

A cryptographic module is loaded onto a smart card after the card is issued to a user. The symmetric key for the cryptographic module will then be provisioned using a secure channel mechanism present in many smart card platforms. This allows a direct secure channel to be established between the smart card chip and the provisioning server. For example, the card commands (i.e., Application Protocol Data Units, or APDUs) are encrypted with a pre-issued card manufacturer's key and sent directly to the smart card chip, allowing secure post-issuance in-the-field provisioning. This secure flow can pass Transport Layer Security (TLS) [RFC5246] and other transport security boundaries.

Note that two pre-conditions for this usage scenario are for the protocol to be tunneled and the provisioning server to know the correct pre-established manufacturer's key.

A.8. End-to-End Protection of Key Material

In this scenario, Transport Layer Security does not provide end-to-end protection of keying material transported from the provisioning server to the cryptographic module. For example, TLS may terminate at an application hosted on a PC rather than at the cryptographic module (i.e., the endpoint) located on a data storage device [RFC5246]. Mutually authenticated key agreement provides end-to-end protection, which TLS cannot provide.

Appendix B. Examples

This appendix contains example messages that illustrate parameters, encoding, and semantics in four- and two-pass DSKPP exchanges. The examples are written using XML, and are syntactically correct. MAC and cipher values are fictitious, however. This appendix forms an informative part of the document.

B.1. Trigger Message

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<dskpp:KeyProvTrigger Version="1.0"
  xmlns:dskpp="urn:ietf:params:xml:ns:keyprov:dskpp"
  xmlns:pskc="urn:ietf:params:xml:ns:keyprov:pskc">
  <dskpp:InitializationTrigger>
    <dskpp:DeviceIdentifierData>
        <dskpp:DeviceId>
            <pskc:Manufacturer>TokenVendorAcme</pskc:Manufacturer>
            <pskc:SerialNo>987654321</pskc:SerialNo>
            <pskc:StartDate>2009-09-01T00:00:00Z</pskc:StartDate>
            <pskc:ExpiryDate>2014-09-01T00:00:00Z</pskc:ExpiryDate>
        </dskpp:DeviceId>
    </dskpp:DeviceIdentifierData>
    <dskpp:KeyID>SE9UUDAwMDAwMDAx</dskpp:KeyID>
    <dskpp:TokenPlatformInfo KeyLocation="Hardware"
      AlgorithmLocation="Software"/>
    <dskpp:AuthenticationData>
      <dskpp:ClientID>31300257</dskpp:ClientID>
      <dskpp:AuthenticationCodeMac>
        <dskpp:IterationCount>512</dskpp:IterationCount>
        <dskpp:Mac>4bRJf9xXd3KchKoTenHJiw==</dskpp:Mac>
      </dskpp:AuthenticationCodeMac>
    </dskpp:AuthenticationData>
    <dskpp:ServerUrl>keyprovservice.example.com
      </dskpp:ServerUrl>
  </dskpp:InitializationTrigger>
</dskpp:KeyProvTrigger>
```

B.2. Four-Pass Protocol

B.2.1. <KeyProvClientHello> without a Preceding Trigger

```
    <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
    <dskpp:KeyProvClientHello
        xmlns:pskc="urn:ietf:params:xml:ns:keyprov:pskc"
        xmlns:dskpp="urn:ietf:params:xml:ns:keyprov:dskpp"
        xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
        xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
        Version="1.0">
        <dskpp:DeviceIdentifierData>
            <dskpp:DeviceId>
                <pskc:Manufacturer>TokenVendorAcme</pskc:Manufacturer>
                <pskc:SerialNo>987654321</pskc:SerialNo>
                <pskc:StartDate>2009-09-01T00:00:00Z</pskc:StartDate>
                <pskc:ExpiryDate>2014-09-01T00:00:00Z</pskc:ExpiryDate>
            </dskpp:DeviceId>
        </dskpp:DeviceIdentifierData>
        <dskpp:SupportedKeyTypes>
            <dskpp:Algorithm>
                urn:ietf:params:xml:ns:keyprov:pskc:hotp
            </dskpp:Algorithm>
            <dskpp:Algorithm>
    http://www.rsa.com/rsalabs/otps/schemas/2005/09/otps-wst#SecurID-AES
            </dskpp:Algorithm>
        </dskpp:SupportedKeyTypes>
        <dskpp:SupportedEncryptionAlgorithms>
            <dskpp:Algorithm>
                http://www.w3.org/2001/04/xmlenc#aes128-cbc
            </dskpp:Algorithm>
        </dskpp:SupportedEncryptionAlgorithms>
        <dskpp:SupportedMacAlgorithms>
            <dskpp:Algorithm>
                urn:ietf:params:xml:ns:keyprov:dskpp:prf-sha256
            </dskpp:Algorithm>
        </dskpp:SupportedMacAlgorithms>
        <dskpp:SupportedProtocolVariants>
            <dskpp:FourPass/>
        </dskpp:SupportedProtocolVariants>
        <dskpp:SupportedKeyPackages>
            <dskpp:KeyPackageFormat>
                urn:ietf:params:xml:ns:keyprov:dskpp:pskc-key-container
            </dskpp:KeyPackageFormat>
        </dskpp:SupportedKeyPackages>
    </dskpp:KeyProvClientHello>
```

B.2.2. <KeyProvClientHello> Assuming a Preceding Trigger

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<dskpp:KeyProvClientHello
    xmlns:pskc="urn:ietf:params:xml:ns:keyprov:pskc"
    xmlns:dskpp="urn:ietf:params:xml:ns:keyprov:dskpp"
    xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
    Version="1.0">
    <dskpp:DeviceIdentifierData>
        <dskpp:DeviceId>
            <pskc:Manufacturer>TokenVendorAcme</pskc:Manufacturer>
            <pskc:SerialNo>987654321</pskc:SerialNo>
            <pskc:StartDate>2009-09-01T00:00:00Z</pskc:StartDate>
            <pskc:ExpiryDate>2014-09-01T00:00:00Z</pskc:ExpiryDate>
        </dskpp:DeviceId>
    </dskpp:DeviceIdentifierData>
    <dskpp:KeyID>SE9UUDAwMDAwMDAx</dskpp:KeyID>
    <dskpp:SupportedKeyTypes>
        <dskpp:Algorithm>
            urn:ietf:params:xml:ns:keyprov:pskc:hotp
        </dskpp:Algorithm>
        <dskpp:Algorithm>
http://www.rsa.com/rsalabs/otps/schemas/2005/09/otps-wst#SecurID-AES
        </dskpp:Algorithm>
    </dskpp:SupportedKeyTypes>
    <dskpp:SupportedEncryptionAlgorithms>
        <dskpp:Algorithm>
            http://www.w3.org/2001/04/xmlenc#aes128-cbc
        </dskpp:Algorithm>
    </dskpp:SupportedEncryptionAlgorithms>
    <dskpp:SupportedMacAlgorithms>
        <dskpp:Algorithm>
            urn:ietf:params:xml:ns:keyprov:dskpp:prf-sha256
        </dskpp:Algorithm>
    </dskpp:SupportedMacAlgorithms>
    <dskpp:SupportedProtocolVariants>
      <dskpp:FourPass/>
    </dskpp:SupportedProtocolVariants>
    <dskpp:SupportedKeyPackages>
        <dskpp:KeyPackageFormat>
            urn:ietf:params:xml:ns:keyprov:dskpp:pskc-key-container
        </dskpp:KeyPackageFormat>
    </dskpp:SupportedKeyPackages>
</dskpp:KeyProvClientHello>
```

B.2.3. <KeyProvServerHello> Without a Preceding Trigger

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<dskpp:KeyProvServerHello
    xmlns:pskc="urn:ietf:params:xml:ns:keyprov:pskc"
    xmlns:dskpp="urn:ietf:params:xml:ns:keyprov:dskpp"
    xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
    Version="1.0"
    Status="Continue"
    SessionID="4114">
    <dskpp:KeyType>
        urn:ietf:params:xml:ns:keyprov:pskc:hotp
    </dskpp:KeyType>
    <dskpp:EncryptionAlgorithm>
        http://www.w3.org/2001/04/xmlenc#aes128-cbc
    </dskpp:EncryptionAlgorithm>
    <dskpp:MacAlgorithm>
        urn:ietf:params:xml:ns:keyprov:dskpp:prf-sha256
    </dskpp:MacAlgorithm>
    <dskpp:EncryptionKey>
      <ds:KeyName>Example-Key1</ds:KeyName>
    </dskpp:EncryptionKey>
    <dskpp:KeyPackageFormat>
        urn:ietf:params:xml:ns:keyprov:dskpp:pskc-key-container
    </dskpp:KeyPackageFormat>
    <dskpp:Payload>
        <dskpp:Nonce>EjRWeJASNFZ4kBI0VniQEg==</dskpp:Nonce>
    </dskpp:Payload>
</dskpp:KeyProvServerHello>
```

B.2.4. <KeyProvServerHello> Assuming Key Renewal

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<dskpp:KeyProvServerHello
  xmlns:dskpp="urn:ietf:params:xml:ns:keyprov:dskpp"
  xmlns:pskc="urn:ietf:params:xml:ns:keyprov:pskc"
  xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
  xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
  Version="1.0"
  SessionID="4114"
  Status="Continue">
  <dskpp:KeyType>
    urn:ietf:params:xml:schema:keyprov:otpalg#SecurID-AES
  </dskpp:KeyType>
  <dskpp:EncryptionAlgorithm>
     http://www.w3.org/2001/04/xmlenc#aes128-cbc
  </dskpp:EncryptionAlgorithm>
  <dskpp:MacAlgorithm>
     urn:ietf:params:xml:ns:keyprov:dskpp:prf-sha256
  </dskpp:MacAlgorithm>
  <dskpp:EncryptionKey>
    <ds:KeyName>Example-Key1</ds:KeyName>
  </dskpp:EncryptionKey>
  <dskpp:KeyPackageFormat>
    urn:ietf:params:xml:ns:keyprov:dskpp:pskc-key-container
  </dskpp:KeyPackageFormat>
  <dskpp:Payload>
    <dskpp:Nonce>qw2ewasde312asder394jw==</dskpp:Nonce>
  </dskpp:Payload>
  <dskpp:Mac
    MacAlgorithm="urn:ietf:params:xml:ns:keyprov:dskpp:prf-aes-128">
    cXcycmFuZG9tMzEyYXNkZXIzOTRqdw==
  </dskpp:Mac>
</dskpp:KeyProvServerHello>
```

B.2.5. <KeyProvClientNonce> Using Default Encryption

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

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<dskpp:KeyProvClientNonce
    xmlns:pskc="urn:ietf:params:xml:ns:keyprov:pskc"
    xmlns:dskpp="urn:ietf:params:xml:ns:keyprov:dskpp"
    xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
    SessionID="4114"
    Version="1.0">
    <dskpp:EncryptedNonce>
        oTvo+S22nsmS2Z/RtcoF8CTwadRa1PVsRXkZnCihHkU1rPueggrd0NpEWVZR
        16Rg16+FHuTg33GK1wH3wffDZQ==
    </dskpp:EncryptedNonce>
</dskpp:KeyProvClientNonce>
```

B.2.6. <KeyProvServerFinished> Using Default Encryption

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<dskpp:KeyProvServerFinished
    xmlns:pskc="urn:ietf:params:xml:ns:keyprov:pskc"
    xmlns:dskpp="urn:ietf:params:xml:ns:keyprov:dskpp"
    xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
    Version="1.0"
    Status="Success"
    SessionID="4114">
    <dskpp:KeyPackage>
        <dskpp:KeyContainer Version="1.0" Id="KC0001">
            <pskc:KeyPackage>
                <pskc:DeviceInfo>
                    <pskc:Manufacturer>
                       TokenVendorAcme
                    </pskc:Manufacturer>
                    <pskc:SerialNo>
                       987654321
                    </pskc:SerialNo>
                    <pskc:StartDate>
                       2009-09-01T00:00:00Z
                    </pskc:StartDate>
                    <pskc:ExpiryDate>
                       2014-09-01T00:00:00Z
                    </pskc:ExpiryDate>
                </pskc:DeviceInfo>
```

```
                    <pskc:CryptoModuleInfo>
                        <pskc:Id>CM_ID_001</pskc:Id>
                    </pskc:CryptoModuleInfo>
                    <pskc:Key
                       Id="MBK000000001"
                       Algorithm=
                          "urn:ietf:params:xml:ns:keyprov:pskc:hotp">
                       <pskc:Issuer>Example-Issuer</pskc:Issuer>
                       <pskc:AlgorithmParameters>
                           <pskc:ResponseFormat Length="6"
                              Encoding="DECIMAL"/>
                        </pskc:AlgorithmParameters>
                        <pskc:Data>
                            <pskc:Counter>
                                <pskc:PlainValue>0</pskc:PlainValue>
                            </pskc:Counter>
                        </pskc:Data>
                        <pskc:Policy>
                            <pskc:KeyUsage>OTP</pskc:KeyUsage>
                        </pskc:Policy>
                    </pskc:Key>
                </pskc:KeyPackage>
            </dskpp:KeyContainer>
        </dskpp:KeyPackage>
        <dskpp:Mac
            MacAlgorithm=
               "urn:ietf:params:xml:ns:keyprov:dskpp:prf-sha256">
            151yAR2NqU5dJzETK+SGYqN6sq6DEH5AgHohra3Jpp4=
        </dskpp:Mac>
    </dskpp:KeyProvServerFinished>
```

B.3. Two-Pass Protocol

B.3.1. Example Using the Key Transport Method

The client indicates support for all the Key Transport, Key Wrap, and Passphrase-Based Key Wrap key protection methods:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<dskpp:KeyProvClientHello
    xmlns:pskc="urn:ietf:params:xml:ns:keyprov:pskc"
    xmlns:dskpp="urn:ietf:params:xml:ns:keyprov:dskpp"
    xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
    Version="1.0">
    <dskpp:DeviceIdentifierData>
        <dskpp:DeviceId>
            <pskc:Manufacturer>TokenVendorAcme</pskc:Manufacturer>
```

```
                <pskc:SerialNo>987654321</pskc:SerialNo>
                <pskc:StartDate>2009-09-01T00:00:00Z</pskc:StartDate>
                <pskc:ExpiryDate>2014-09-01T00:00:00Z</pskc:ExpiryDate>
            </dskpp:DeviceId>
        </dskpp:DeviceIdentifierData>
        <dskpp:SupportedKeyTypes>
            <dskpp:Algorithm>
                urn:ietf:params:xml:ns:keyprov:pskc:hotp
            </dskpp:Algorithm>
            <dskpp:Algorithm>
    http://www.rsa.com/rsalabs/otps/schemas/2005/09/otps-wst#SecurID-AES
            </dskpp:Algorithm>
        </dskpp:SupportedKeyTypes>
        <dskpp:SupportedEncryptionAlgorithms>
            <dskpp:Algorithm>
                http://www.w3.org/2001/04/xmlenc#rsa_1_5
            </dskpp:Algorithm>
        </dskpp:SupportedEncryptionAlgorithms>
        <dskpp:SupportedMacAlgorithms>
            <dskpp:Algorithm>
                urn:ietf:params:xml:ns:keyprov:dskpp:prf-sha256
            </dskpp:Algorithm>
        </dskpp:SupportedMacAlgorithms>
        <dskpp:SupportedProtocolVariants>
            <dskpp:TwoPass>
                <dskpp:SupportedKeyProtectionMethod>
                    urn:ietf:params:xml:schema:keyprov:dskpp:transport
                </dskpp:SupportedKeyProtectionMethod>
                <dskpp:Payload>
                    <ds:KeyInfo>
                        <ds:X509Data>
                            <ds:X509Certificate>
    MIIB5zCCAVCgAwIBAgIESZp/vDANBgkqhkiG9w0BAQUFADA4MQ0wCwYDVQQKEwRJRVRGM
    RMwEQYDVQQLEwpLZXlQcm92IFdHMRIwEAYDVQQDEwlQU0tDIFRlc3QwHhcNMDkwMjE3MD
    kxMzMyWhcNMTEwMjE3MDkxMzMyWjA4MQ0wCwYDVQQKEwRJRVRGMRMwEQYDVQQLEwpLZXl
    Qcm92IFdHMRIwEAYDVQQDEwlQU0tDIFRlc3QwgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJ
    AoGBALCWLDa2ItYJ6su80hd1gL4cggQYdyyKK17btt/aS6Q/eDsKjsPyFIODsxeKVV/uA
    3wLT4jQJM5euKJXkDajzGGOy92+ypfzTX4zDJMkh61SZwlHNJxBKilAM5aW7C+BQ0RvCx
    vdYtzx2LTdB+X/KMEBA7uIYxLfXH2Mnub3WIh1AgMBAAEwDQYJKoZIhvcNAQEFBQADgYE
    Ae875m84sYUJ8qPeZ+NG7REgTvlHTmoCdoByU0LBBLotUKuqfrnRuXJRMeZXaaEGmzY1k
    LonVjQGzjAkU4dJ+RPmiDlYuHLZS41Pg6VMwY+03lhk6I5A/w4rnqdkmwZX/NgXg06aln
    c2pBsXWhL4O7nk0S2ZrLMsQZ6HcsXgdmHo=
                            </ds:X509Certificate>
                        </ds:X509Data>
                    </ds:KeyInfo>
                </dskpp:Payload>
            </dskpp:TwoPass>
        </dskpp:SupportedProtocolVariants>
```

```
        <dskpp:SupportedKeyPackages>
            <dskpp:KeyPackageFormat>
                urn:ietf:params:xml:ns:keyprov:dskpp:pskc-key-container
            </dskpp:KeyPackageFormat>
        </dskpp:SupportedKeyPackages>
        <dskpp:AuthenticationData>
            <dskpp:ClientID>AC00000A</dskpp:ClientID>
            <dskpp:AuthenticationCodeMac>
                <dskpp:Nonce>
                    ESIzRFVmd4iZqrvM3e7/ESIzRFVmd4iZqrvM3e7/ESI=
                </dskpp:Nonce>
                <dskpp:IterationCount>100000</dskpp:IterationCount>
                <dskpp:Mac
                    MacAlgorithm=
                    "urn:ietf:params:xml:ns:keyprov:dskpp:prf-sha256">
                    3eRz51ILqiG+dJW2iLcjuA==
                </dskpp:Mac>
            </dskpp:AuthenticationCodeMac>
        </dskpp:AuthenticationData>
    </dskpp:KeyProvClientHello>
```

In this example, the server responds to the previous request by returning a key package in which the provisioning key was encrypted using the Key Transport key protection method.

```
    <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
    <dskpp:KeyProvServerFinished
        xmlns:pskc="urn:ietf:params:xml:ns:keyprov:pskc"
        xmlns:dskpp="urn:ietf:params:xml:ns:keyprov:dskpp"
        xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
        xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
        xmlns:dkey="http://www.w3.org/2009/xmlsec-derivedkey#"
        xmlns:pkcs5=
           "http://www.rsasecurity.com/rsalabs/pkcs/schemas/pkcs-5v2-0#"
        Version="1.0"
        Status="Success"
        SessionID="4114">
        <dskpp:KeyPackage>
            <dskpp:KeyContainer Version="1.0" Id="KC0001">
                <pskc:EncryptionKey>
                    <ds:X509Data>
                        <ds:X509Certificate>
    MIIB5zCCAVCgAwIBAgIESZp/vDANBgkqhkiG9w0BAQUFADA4MQ0wCwYDVQQKEwRJRVRGM
    RMwEQYDVQQLEwpLZXlQcm92IFdHMRIwEAYDVQQDEwlQU0tDIFRlc3QwHhcNMDkwMjE3MD
    kxMzMyWhcNMTEwMjE3MDkxMzMyWjA4MQ0wCwYDVQQKEwRJRVRGMRMwEQYDVQQLEwpLZXl
    Qcm92IFdHMRIwEAYDVQQDEwlQU0tDIFRlc3QwgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJ
    AoGBALCWLDa2ItYJ6su80hd1gL4cggQYdyyKK17btt/aS6Q/eDsKjsPyFIODsxeKVV/uA
    3wLT4jQJM5euKJXkDajzGGOy92+ypfzTX4zDJMkh61SZwlHNJxBKilAM5aW7C+BQ0RvCx
```

```
   vdYtzx2LTdB+X/KMEBA7uIYxLfXH2Mnub3WIh1AgMBAAEwDQYJKoZIhvcNAQEFBQADgYE
   Ae875m84sYUJ8qPeZ+NG7REgTvlHTmoCdoByU0LBBLotUKuqfrnRuXJRMeZXaaEGmzY1k
   LonVjQGzjAkU4dJ+RPmiDlYuHLZS41Pg6VMwY+03lhk6I5A/w4rnqdkmwZX/NgXg06aln
   c2pBsXWhL4O7nk0S2ZrLMsQZ6HcsXgdmHo=
                        </ds:X509Certificate>
                    </ds:X509Data>
                </pskc:EncryptionKey>
                <pskc:KeyPackage>
                    <pskc:DeviceInfo>
                        <pskc:Manufacturer>
                           TokenVendorAcme
                        </pskc:Manufacturer>
                        <pskc:SerialNo>
                           987654321
                        </pskc:SerialNo>
                        <pskc:StartDate>
                           2009-09-01T00:00:00Z
                        </pskc:StartDate>
                        <pskc:ExpiryDate>
                           2014-09-01T00:00:00Z
                        </pskc:ExpiryDate>
                    </pskc:DeviceInfo>
                    <pskc:Key
                        Id="MBK000000001"
                        Algorithm=
                           "urn:ietf:params:xml:ns:keyprov:pskc:hotp">
                        <pskc:Issuer>Example-Issuer</pskc:Issuer>
                        <pskc:AlgorithmParameters>
                            <pskc:ResponseFormat Length="6"
                               Encoding="DECIMAL"/>
                        </pskc:AlgorithmParameters>
                        <pskc:Data>
                            <pskc:Secret>
                                <pskc:EncryptedValue>
                                    <xenc:EncryptionMethod
                                     Algorithm=
                                 "http://www.w3.org/2001/04/xmlenc#rsa_1_5"/>
                                    <xenc:CipherData>
                                        <xenc:CipherValue>
   eyjr23WMy9S2UdKgGnQEbs44T1jmX1TNWEBq48xfS20PK2VWF4ZK1iSctHj/u3uk+7+y8
   uKrAzHEm5mujKPAU4DCbb5mSibXMnAbbIoAi2cJW60/l8FlzwaU4EZsZ1LyQ1GcBQKACE
   eylG5vK8NTo47vZTatL5UxmbmOX2HvaVQ=
                                        </xenc:CipherValue>
                                    </xenc:CipherData>
                                </pskc:EncryptedValue>
                            </pskc:Secret>
                            <pskc:Counter>
                                <pskc:PlainValue>0</pskc:PlainValue>
```

```
                        </pskc:Counter>
                    </pskc:Data>
                    <pskc:Policy>
                        <pskc:KeyUsage>OTP</pskc:KeyUsage>
                    </pskc:Policy>
                </pskc:Key>
            </pskc:KeyPackage>
        </dskpp:KeyContainer>
    </dskpp:KeyPackage>
    <dskpp:Mac
        MacAlgorithm=
           "urn:ietf:params:xml:ns:keyprov:dskpp:prf-sha256">
        GHZ0H6Y+KpxdlVZ7zgcJDiDdqc8Gcmlcf+HQi4EUxYU=
    </dskpp:Mac>
</dskpp:KeyProvServerFinished>
```

B.3.2. Example Using the Key Wrap Method

The client sends a request that specifies a shared key to protect the K_TOKEN, and the server responds using the Key Wrap key protection method. Authentication Data in this example is based on an Authentication Code rather than a device certificate.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<dskpp:KeyProvClientHello
    xmlns:pskc="urn:ietf:params:xml:ns:keyprov:pskc"
    xmlns:dskpp="urn:ietf:params:xml:ns:keyprov:dskpp"
    xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
    Version="1.0">
    <dskpp:DeviceIdentifierData>
        <dskpp:DeviceId>
            <pskc:Manufacturer>TokenVendorAcme</pskc:Manufacturer>
            <pskc:SerialNo>987654321</pskc:SerialNo>
            <pskc:StartDate>2009-09-01T00:00:00Z</pskc:StartDate>
            <pskc:ExpiryDate>2014-09-01T00:00:00Z</pskc:ExpiryDate>
        </dskpp:DeviceId>
    </dskpp:DeviceIdentifierData>
    <dskpp:SupportedKeyTypes>
        <dskpp:Algorithm>
            urn:ietf:params:xml:ns:keyprov:pskc:hotp
        </dskpp:Algorithm>
        <dskpp:Algorithm>
 http://www.rsa.com/rsalabs/otps/schemas/2005/09/otps-wst#SecurID-AES
        </dskpp:Algorithm>
    </dskpp:SupportedKeyTypes>
    <dskpp:SupportedEncryptionAlgorithms>
        <dskpp:Algorithm>
```

```
            http://www.w3.org/2001/04/xmlenc#aes128-cbc
        </dskpp:Algorithm>
    </dskpp:SupportedEncryptionAlgorithms>
    <dskpp:SupportedMacAlgorithms>
        <dskpp:Algorithm>
            urn:ietf:params:xml:ns:keyprov:dskpp:prf-sha256
        </dskpp:Algorithm>
    </dskpp:SupportedMacAlgorithms>
    <dskpp:SupportedProtocolVariants>
        <dskpp:TwoPass>
            <dskpp:SupportedKeyProtectionMethod>
                urn:ietf:params:xml:schema:keyprov:dskpp:wrap
            </dskpp:SupportedKeyProtectionMethod>
            <dskpp:Payload>
                <ds:KeyInfo>
                    <ds:KeyName>Pre-shared-key-1</ds:KeyName>
                </ds:KeyInfo>
            </dskpp:Payload>
        </dskpp:TwoPass>
    </dskpp:SupportedProtocolVariants>
    <dskpp:SupportedKeyPackages>
        <dskpp:KeyPackageFormat>
            urn:ietf:params:xml:ns:keyprov:dskpp:pskc-key-container
        </dskpp:KeyPackageFormat>
    </dskpp:SupportedKeyPackages>
    <dskpp:AuthenticationData>
        <dskpp:ClientID>AC00000A</dskpp:ClientID>
        <dskpp:AuthenticationCodeMac>
            <dskpp:Nonce>
                ESIzRFVmd4iZqrvM3e7/ESIzRFVmd4iZqrvM3e7/ESI=
            </dskpp:Nonce>
            <dskpp:IterationCount>1</dskpp:IterationCount>
            <dskpp:Mac
                MacAlgorithm=
                "urn:ietf:params:xml:ns:keyprov:dskpp:prf-sha256">
                3eRz51ILqiG+dJW2iLcjuA==
            </dskpp:Mac>
        </dskpp:AuthenticationCodeMac>
    </dskpp:AuthenticationData>
</dskpp:KeyProvClientHello>
```

In this example, the server responds to the previous request by returning a key package in which the provisioning key was encrypted using the Key Wrap key protection method.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<dskpp:KeyProvServerFinished
    xmlns:pskc="urn:ietf:params:xml:ns:keyprov:pskc"
```

```
      xmlns:dskpp="urn:ietf:params:xml:ns:keyprov:dskpp"
      xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
      xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
      xmlns:dkey="http://www.w3.org/2009/xmlsec-derivedkey#"
      xmlns:pkcs5=
          "http://www.rsasecurity.com/rsalabs/pkcs/schemas/pkcs-5v2-0#"
      Version="1.0"
      Status="Success"
      SessionID="4114">
      <dskpp:KeyPackage>
           <dskpp:KeyContainer Version="1.0" Id="KC0001">
               <pskc:EncryptionKey>
                  <ds:KeyName>Pre-shared-key-1</ds:KeyName>
               </pskc:EncryptionKey>
               <pskc:MACMethod
                   Algorithm=
                      "http://www.w3.org/2000/09/xmldsig#hmac-sha1">
                   <pskc:MACKey>
                       <xenc:EncryptionMethod
                           Algorithm=
                        "http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
                       <xenc:CipherData>
                           <xenc:CipherValue>
       2GTTnLwM3I4e5IO5FkufoMUBJBuAf25hARFv0Z7MFk9Ecdb04PWY/qaeCbrgz7Es
                           </xenc:CipherValue>
                       </xenc:CipherData>
                   </pskc:MACKey>
               </pskc:MACMethod>
               <pskc:KeyPackage>
                   <pskc:DeviceInfo>
                       <pskc:Manufacturer>
                          TokenVendorAcme
                       </pskc:Manufacturer>
                       <pskc:SerialNo>
                          987654321
                       </pskc:SerialNo>
                       <pskc:StartDate>
                          2009-09-01T00:00:00Z
                       </pskc:StartDate>
                       <pskc:ExpiryDate>
                          2014-09-01T00:00:00Z
                       </pskc:ExpiryDate>
                   </pskc:DeviceInfo>
                   <pskc:CryptoModuleInfo>
                       <pskc:Id>CM_ID_001</pskc:Id>
                   </pskc:CryptoModuleInfo>
                   <pskc:Key
                       Id="MBK000000001"
```

```
                        Algorithm=
                           "urn:ietf:params:xml:ns:keyprov:pskc:hotp">
                        <pskc:Issuer>Example-Issuer</pskc:Issuer>
                        <pskc:AlgorithmParameters>
                          <pskc:ResponseFormat Length="6"
                             Encoding="DECIMAL"/>
                        </pskc:AlgorithmParameters>
                        <pskc:Data>
                            <pskc:Secret>
                                <pskc:EncryptedValue>
                                  <xenc:EncryptionMethod
                                  Algorithm=
                         "http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
                                    <xenc:CipherData>
                                        <xenc:CipherValue>
                                            oTvo+S22nsmS2Z/RtcoF8AabC6vr
                                            09sh0QIU+E224S96sZjpV+6nFYgn
                                            6525OoepbPnL/fGuuey64WCYXoqh
                                            Tg==
                                        </xenc:CipherValue>
                                    </xenc:CipherData>
                                </pskc:EncryptedValue>
                                <pskc:ValueMAC>
                                    o+e9xgMVUbYuZH9UHe0W9dIo88A=
                                </pskc:ValueMAC>
                            </pskc:Secret>
                            <pskc:Counter>
                                <pskc:PlainValue>0</pskc:PlainValue>
                            </pskc:Counter>
                        </pskc:Data>
                        <pskc:Policy>
                            <pskc:KeyUsage>OTP</pskc:KeyUsage>
                        </pskc:Policy>
                    </pskc:Key>
                </pskc:KeyPackage>
            </dskpp:KeyContainer>
        </dskpp:KeyPackage>
        <dskpp:Mac
            MacAlgorithm=
               "urn:ietf:params:xml:ns:keyprov:dskpp:prf-sha256">
            l53BmSO6qUzoIgbQegimsKk2es+WRpEl0YFqaOp5PGE=
        </dskpp:Mac>
    </dskpp:KeyProvServerFinished>
```

B.3.3. Example Using the Passphrase-Based Key Wrap Method

The client sends a request similar to that in Appendix B.3.1 with Authentication Data based on an Authentication Code, and the server responds using the Passphrase-Based Key Wrap method to encrypt the provisioning key (note that the encryption is derived from the password component of the Authentication Code). The Authentication Data is set in clear text when it is sent over a secure transport channel such as TLS [RFC5246].

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<dskpp:KeyProvClientHello
    xmlns:pskc="urn:ietf:params:xml:ns:keyprov:pskc"
    xmlns:dskpp="urn:ietf:params:xml:ns:keyprov:dskpp"
    xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
    Version="1.0">
    <dskpp:DeviceIdentifierData>
        <dskpp:DeviceId>
            <pskc:Manufacturer>TokenVendorAcme</pskc:Manufacturer>
            <pskc:SerialNo>987654321</pskc:SerialNo>
            <pskc:StartDate>2009-09-01T00:00:00Z</pskc:StartDate>
            <pskc:ExpiryDate>2014-09-01T00:00:00Z</pskc:ExpiryDate>
        </dskpp:DeviceId>
    </dskpp:DeviceIdentifierData>
    <dskpp:SupportedKeyTypes>
        <dskpp:Algorithm>
            urn:ietf:params:xml:ns:keyprov:pskc:hotp
        </dskpp:Algorithm>
        <dskpp:Algorithm>
 http://www.rsa.com/rsalabs/otps/schemas/2005/09/otps-wst#SecurID-AES
        </dskpp:Algorithm>
    </dskpp:SupportedKeyTypes>
    <dskpp:SupportedEncryptionAlgorithms>
        <dskpp:Algorithm>
            http://www.w3.org/2001/04/xmlenc#rsa_1_5
        </dskpp:Algorithm>
    </dskpp:SupportedEncryptionAlgorithms>
    <dskpp:SupportedMacAlgorithms>
        <dskpp:Algorithm>
            urn:ietf:params:xml:ns:keyprov:dskpp:prf-sha256
        </dskpp:Algorithm>
    </dskpp:SupportedMacAlgorithms>
    <dskpp:SupportedProtocolVariants>
        <dskpp:TwoPass>
            <dskpp:SupportedKeyProtectionMethod>
             urn:ietf:params:xml:schema:keyprov:dskpp:passphrase-wrap
            </dskpp:SupportedKeyProtectionMethod>
```

```
            <dskpp:Payload>
                <ds:KeyInfo>
                    <ds:KeyName>Passphrase-1</ds:KeyName>
                </ds:KeyInfo>
            </dskpp:Payload>
        </dskpp:TwoPass>
    </dskpp:SupportedProtocolVariants>
    <dskpp:SupportedKeyPackages>
        <dskpp:KeyPackageFormat>
            urn:ietf:params:xml:ns:keyprov:dskpp:pskc-key-container
        </dskpp:KeyPackageFormat>
    </dskpp:SupportedKeyPackages>
    <dskpp:AuthenticationData>
        <dskpp:ClientID>AC00000A</dskpp:ClientID>
        <dskpp:AuthenticationCodeMac>
            <dskpp:Nonce>
                ESIzRFVmd4iZqrvM3e7/ESIzRFVmd4iZqrvM3e7/ESI=
            </dskpp:Nonce>
            <dskpp:IterationCount>1</dskpp:IterationCount>
            <dskpp:Mac
                MacAlgorithm=
               "urn:ietf:params:xml:ns:keyprov:dskpp:prf-sha256">
               K4YvLMN6Q1DZvtShoCxQag==
            </dskpp:Mac>
        </dskpp:AuthenticationCodeMac>
    </dskpp:AuthenticationData>
</dskpp:KeyProvClientHello>
```

In this example, the server responds to the previous request by returning a key package in which the provisioning key was encrypted using the Passphrase-Based Key Wrap key protection method.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<dskpp:KeyProvServerFinished
    xmlns:pskc="urn:ietf:params:xml:ns:keyprov:pskc"
    xmlns:dskpp="urn:ietf:params:xml:ns:keyprov:dskpp"
    xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
    xmlns:dkey="http://www.w3.org/2009/xmlsec-derivedkey#"
    xmlns:pkcs5=
       "http://www.rsasecurity.com/rsalabs/pkcs/schemas/pkcs-5v2-0#"
    Version="1.0"
    Status="Success"
    SessionID="4114">
    <dskpp:KeyPackage>
        <dskpp:KeyContainer Version="1.0" Id="KC0002">
            <pskc:EncryptionKey>
                <dkey:DerivedKey>
```

```
                    <dkey:KeyDerivationMethod
                    Algorithm=
                    "http://www.rsasecurity.com/rsalabs/pkcs/schemas/
                    pkcs-5v2-0#pbkdf2">
                        <pkcs5:PBKDF2-params>
                            <Salt>
                                <Specified>Ej7/PEpyEpw=</Specified>
                            </Salt>
                            <IterationCount>1000</IterationCount>
                            <KeyLength>16</KeyLength>
                        </pkcs5:PBKDF2-params>
                    </dkey:KeyDerivationMethod>
                    <xenc:ReferenceList>
                        <xenc:DataReference URI="#ED"/>
                    </xenc:ReferenceList>
                    <dkey:MasterKeyName>
                       Passphrase1
                    </dkey:MasterKeyName>
                </dkey:DerivedKey>
            </pskc:EncryptionKey>
            <pskc:MACMethod
                Algorithm=
                   "http://www.w3.org/2000/09/xmldsig#hmac-sha1">
                <pskc:MACKey>
                    <xenc:EncryptionMethod
                        Algorithm=
                      "http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
                    <xenc:CipherData>
                        <xenc:CipherValue>
      2GTTnLwM3I4e5IO5FkufoOEiOhNj91fhKRQBtBJYluUDsPOLTfUvoU2dStyOwYZx
                        </xenc:CipherValue>
                    </xenc:CipherData>
                </pskc:MACKey>
            </pskc:MACMethod>
            <pskc:KeyPackage>
                <pskc:DeviceInfo>
                    <pskc:Manufacturer>
                       TokenVendorAcme
                    </pskc:Manufacturer>
                    <pskc:SerialNo>
                       987654321
                    </pskc:SerialNo>
                    <pskc:StartDate>
                       2009-09-01T00:00:00Z
                    </pskc:StartDate>
                    <pskc:ExpiryDate>
                       2014-09-01T00:00:00Z
                    </pskc:ExpiryDate>
```

```
                    </pskc:DeviceInfo>
                    <pskc:CryptoModuleInfo>
                        <pskc:Id>CM_ID_001</pskc:Id>
                    </pskc:CryptoModuleInfo>
                    <pskc:Key
                        Id="MBK000000001"
                        Algorithm=
                           "urn:ietf:params:xml:ns:keyprov:pskc:hotp">
                        <pskc:Issuer>Example-Issuer</pskc:Issuer>
                        <pskc:AlgorithmParameters>
                           <pskc:ResponseFormat Length="6"
                              Encoding="DECIMAL"/>
                        </pskc:AlgorithmParameters>
                        <pskc:Data>
                            <pskc:Secret>
                                <pskc:EncryptedValue>
                                    <xenc:EncryptionMethod
                                        Algorithm=
                                        "http://www.w3.org/2001/04/
                                        xmlenc#aes128-cbc"/>
                                    <xenc:CipherData>
                                        <xenc:CipherValue>
                                          oTvo+S22nsmS2Z/RtcoF8HX385uMWgJ
                                          myIFMESBmcvtHQXp/6T1TgCS9CsgKtm
                                          cOrF8VoK254tZKnrAjiD5cdw==
                                        </xenc:CipherValue>
                                    </xenc:CipherData>
                                </pskc:EncryptedValue>
                                <pskc:ValueMAC>
                                    pbgEbVYxoYs0x41wdeC7eDRbUEk=
                                </pskc:ValueMAC>
                            </pskc:Secret>
                            <pskc:Counter>
                                <pskc:PlainValue>0</pskc:PlainValue>
                            </pskc:Counter>
                        </pskc:Data>
                        <pskc:Policy>
                            <pskc:KeyUsage>OTP</pskc:KeyUsage>
                        </pskc:Policy>
                    </pskc:Key>
                </pskc:KeyPackage>
            </dskpp:KeyContainer>
        </dskpp:KeyPackage>
        <dskpp:Mac MacAlgorithm=
            "urn:ietf:params:xml:ns:keyprov:dskpp:prf-sha256">
            Jc4VsNODYXgfbDmTn9qQZgcL3cKoa//j/NRT7sTpKOM=
        </dskpp:Mac>
    </dskpp:KeyProvServerFinished>
```

Appendix C. Integration with PKCS #11

A DSKPP Client that needs to communicate with a connected cryptographic module to perform a DSKPP exchange MAY use PKCS #11 [PKCS-11] as a programming interface as described herein. This appendix forms an informative part of the document.

C.1. The Four-Pass Variant

When performing four-pass DSKPP with a cryptographic module using the PKCS #11 programming interface, the procedure described in [CT-KIP-P11], Appendix B, is RECOMMENDED.

C.2. The Two-Pass Variant

A suggested procedure to perform two-pass DSKPP with a cryptographic module through the PKCS #11 interface using the mechanisms defined in [CT-KIP-P11] is as follows:

a. On the client side,

    1. The client selects a suitable slot and token (e.g., through use of the <DeviceIdentifier> or the <PlatformInfo> element of the DSKPP trigger message).

    2. A nonce R is generated, e.g., by calling C_SeedRandom and C_GenerateRandom.

    3. The client sends its first message to the server, including the nonce R.

b. On the server side,

    1. A generic key K_PROV = K_TOKEN | K_MAC (where '|' denotes concatenation) is generated, e.g., by calling C_GenerateKey (using key type CKK_GENERIC_SECRET). The template for K_PROV MUST allow it to be exported (but only in wrapped form, i.e., CKA_SENSITIVE MUST be set to CK_TRUE and CKA_EXTRACTABLE MUST also be set to CK_TRUE), and also to be used for further key derivation. From K, a token key K_TOKEN of suitable type is derived by calling C_DeriveKey using the PKCS #11 mechanism CKM_EXTRACT_KEY_FROM_KEY and setting the CK_EXTRACT_PARAMS to the first bit of the generic secret key (i.e., set to 0). Likewise, a MAC key K_MAC is derived from K_PROV by calling C_DeriveKey using the CKM_EXTRACT_KEY_FROM_KEY mechanism, this time setting CK_EXTRACT_PARAMS to the length of K_PROV (in bits) divided by two.

2. The server wraps K_PROV with either the public key of the DSKPP Client or device, the pre-shared secret key, or the derived shared secret key by using C_WrapKey. If use of the DSKPP key wrap algorithm has been negotiated, then the CKM_KIP_WRAP mechanism MUST be used to wrap K. When calling C_WrapKey, the hKey handle in the CK_KIP_PARAMS structure MUST be set to NULL_PTR. The pSeed parameter in the CK_KIP_PARAMS structure MUST point to the nonce R provided by the DSKPP Client, and the ulSeedLen parameter MUST indicate the length of R. The hWrappingKey parameter in the call to C_WrapKey MUST be set to refer to the key wrapping key.

3. Next, the server needs to calculate a MAC using K_MAC. If use of the DSKPP MAC algorithm has been negotiated, then the MAC is calculated by calling C_SignInit with the CKM_KIP_MAC mechanism followed by a call to C_Sign. In the call to C_SignInit, K_MAC MUST be the signature key, the hKey parameter in the CK_KIP_PARAMS structure MUST be set to NULL_PTR, the pSeed parameter of the CT_KIP_PARAMS structure MUST be set to NULL_PTR, and the ulSeedLen parameter MUST be set to zero. In the call to C_Sign, the pData parameter MUST be set to the concatenation of the string ServerID and the nonce R, and the ulDataLen parameter MUST be set to the length of the concatenated string. The desired length of the MAC MUST be specified through the pulSignatureLen parameter and MUST be set to the length of R.

4. If the server also needs to authenticate its message (due to an existing K_TOKEN being replaced), the server MUST calculate a second MAC. Again, if use of the DSKPP MAC algorithm has been negotiated, then the MAC is calculated by calling C_SignInit with the CKM_KIP_MAC mechanism followed by a call to C_Sign. In this call to C_SignInit, the K_MAC' existing before this DSKPP run MUST be the signature key (the implementation may specify K_MAC' to be the value of the K_TOKEN that is being replaced, or a version of K_MAC from the previous protocol run), the hKey parameter in the CK_KIP_PARAMS structure MUST be set to NULL, the pSeed parameter of the CT_KIP_PARAMS structure MUST be set to NULL_PTR, and the ulSeedLen parameter MUST be set to zero. In the call to C_Sign, the pData parameter MUST be set to the concatenation of the string ServerID and the nonce R, and the ulDataLen parameter MUST be set to the length of concatenated string. The desired length of the MAC MUST be specified through the pulSignatureLen parameter and MUST be set to the length of R.

5. The server sends its message to the client, including the wrapped key K_TOKEN, the MAC and possibly also the authenticating MAC.

c. On the client side,

1. The client calls C_UnwrapKey to receive a handle to K. After this, the client calls C_DeriveKey twice: once to derive K_TOKEN and once to derive K_MAC. The client MUST use the same mechanism (CKM_EXTRACT_KEY_FROM_KEY) and the same mechanism parameters as used by the server above. When calling C_UnwrapKey and C_DeriveKey, the pTemplate parameter MUST be used to set additional key attributes in accordance with local policy and as negotiated and expressed in the protocol. In particular, the value of the <KeyID> element in the server's response message MAY be used as CKA_ID for K_TOKEN. The key K_PROV MUST be destroyed after deriving K_TOKEN and K_MAC.

2. The MAC is verified in a reciprocal fashion as it was generated by the server. If use of the CKM_KIP_MAC mechanism has been negotiated, then in the call to C_VerifyInit, the hKey parameter in the CK_KIP_PARAMS structure MUST be set to NULL_PTR, the pSeed parameter MUST be set to NULL_PTR, and ulSeedLen MUST be set to 0. The hKey parameter of C_VerifyInit MUST refer to K_MAC. In the call to C_Verify, pData MUST be set to the concatenation of the string ServerID and the nonce R, and the ulDataLen parameter MUST be set to the length of the concatenated string, pSignature to the MAC value received from the server, and ulSignatureLen to the length of the MAC. If the MAC does not verify the protocol session ends with a failure. The token MUST be constructed to not "commit" to the new K_TOKEN or the new K_MAC unless the MAC verifies.

3. If an authenticating MAC was received (REQUIRED if the new K_TOKEN will replace an existing key on the token), then it is verified in a similar vein but using the K_MAC' associated with this server and existing before the protocol run (the implementation may specify K_MAC' to be the value of the K_TOKEN that is being replaced, or a version of K_MAC from the previous protocol run). Again, if the MAC does not verify the protocol session ends with a failure, and the token MUST be constructed not to "commit" to the new K_TOKEN or the new K_MAC unless the MAC verifies.

Appendix D. Example of DSKPP-PRF Realizations

D.1. Introduction

This example appendix defines DSKPP-PRF in terms of AES [FIPS197-AES] and HMAC [RFC2104]. This appendix forms a normative part of the document.

D.2. DSKPP-PRF-AES

D.2.1. Identification

For cryptographic modules supporting this realization of DSKPP-PRF, the following URN MUST be used to identify this algorithm in DSKPP:

urn:ietf:params:xml:ns:keyprov:dskpp:prf-aes-128

When this URN is used to identify the encryption algorithm, the method for encryption of R_C values described in Section 4.2.4 MUST be used.

D.2.2. Definition

DSKPP-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 = CEILING( 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 CMAC construction from [NIST-SP800-38B], using AES as the block cipher:

```
F (k, s, i) = CMAC-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 CMAC 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) = CMAC-AES (k, INT (1) || s)
```

D.3. DSKPP-PRF-SHA256

D.3.1. Identification

For cryptographic modules supporting this realization of DSKPP-PRF, the following URN MUST be used to identify this algorithm in DSKPP:

urn:ietf:params:xml:ns:keyprov:dskpp:prf-sha256

When this URN is used to identify the encryption algorithm to use, the method for encryption of R_C values described in Section 4.2.4 MUST be used.

D.3.2. Definition

DSKPP-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 of SHA-256 in octets of [FIPS180-SHA] (no truncation is done on the HMAC output):

   bLen = 32
   (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 = CEILING(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 [RFC2104], 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 = CEILING(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.

Authors' Addresses

Andrea Doherty
RSA, The Security Division of EMC
174 Middlesex Turnpike
Bedford, MA 01730
USA

EMail: andrea.doherty@rsa.com

Mingliang Pei
VeriSign, Inc.
487 E. Middlefield Road
Mountain View, CA 94043
USA

EMail: mpei@verisign.com

Salah Machani
Diversinet Corp.
2225 Sheppard Avenue East, Suite 1801
Toronto, Ontario M2J 5C2
Canada

EMail: smachani@diversinet.com

Magnus Nystrom
Microsoft Corp.
One Microsoft Way
Redmond, WA 98052
USA

EMail: mnystrom@microsoft.com