NETWORK WORKING GROUP
INTERNET-DRAFT
Intended Status: Proposed Standard
Expires: February 13, 2015

J. Schaad
Soaring Hawk Consulting
S. Turner
IECA, Inc.
P. Timmel
National Security Agency
August 12, 2014

CMC (Certificate Management over Cryptographic Message Syntax) Extensions: Server-Side Key Generation
draft-turner-cmc-serverkeygeneration-02.txt

Abstract

This document defines a set of extensions to the Certificate Management over Cryptographic Message Syntax (CMC) protocol that addresses the desire to support server-side generation of client key material for certificates. This service is provided by the definition of additional control statements within the CMC architecture. Additional CMC errors are also defined.

Table of Contents

1 Introduction

This document defines a set of extensions to and errors for Certificate Management over Cryptographic Message Syntax (CMC) [RFC5272] that allows for server-side generation of client key material for certificates [RFC5280]. The keys that are produced by this service are referred to as server-generated keys. There are strong reasons for providing this service:

- Clients may have poor, unknown, or non-existent key generation capabilities. The creation of private keys relies on the use of good key generation algorithms and a robust random number generator. Server-side key generation can use specialized hardware that may not always be available on clients.

- Central storage of keys may be desired in some environments to permit key recovery. This document only addresses a request to archive server-generated keys; archival of locally generated keys and the control to retrieve archived keys is out-of-scope.

- Server-side key generation may be useful for provisioning keys to disconnected clients (e.g., clients that receive keys from a fill device [RFC4949] because the clients are not able to connect to the server due to an air gap).

These extensions to the CMC protocol are designed to provide server-generated keys without adding any additional round trips to the enrollment process; however, additional round trips may be required based on the mechanism chosen to protect the returned key.

Section 2 describes the enrollment scenarios supported. Section 3 provides CMC requirements. Sections 4 and 5 describe the concepts and structures used in transporting private keys between the server and client applications. Section 6 describes the structure and processes for server-side key generation. Section 7 describes additional CMC error codes. Section 8 describes additional exchanges when the server requires the client provide Proof-of-Possession (POP). Appendix A provides the ASN.1 module for the CMC controls and errors. Appendix B provides example encodings.

1.1. Terminology

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

The terminology in [RFC5272] and in [RFC6402] apply to this profile. Additionally, familiarity with CMS's (Cryptographic Message Syntax) [RFC5652] SignedData, AuthenticatedData, and EnvelopedData content types is assumed.

1.2. Definitions

This section defines some of the terms that are used in this document:

- Dual-use: Applies to certificates or keys. Certificates that can be used to verify both digital signatures and to perform key management, when the KeyUsage extension [RFC5280] is set to digitalSignature and either keyAgreement or keyEncipherment, and keys whose intended use is digital signature and either key agreement or key encipherment.

- Encryption-capable: Applies to certificates or keys. Certificates that can be used for key management (i.e., the KeyUsage extension includes keyAgreement or keyEncipherment) and keys that can be used for key management. This refers to either dual-use or encryption-only certificates/keys.

- Encryption-only: Applies to certificates or keys. Certificates that can only be used for key management, when the KeyUsage extension is set to either keyAgreement or keyEncipherment, and keys whose only intended use is key agreement or key encipherment.

- Ephemeral key [SP-800-57]: A cryptographic key that is generated for each execution of a key establishment process and that meets other requirements of the key type (e.g., unique to each message or session). Often, ephemeral keys are linked to key agreement algorithms; however, this document uses the term ephemeral keys to apply to both key transport and key agreement keys. The ephemeral key has two parts: the private part and the public part. The client provides the public part to the server to allow the server to protect the server-generated keys. Note that an ephemeral key has a security advantage by being unique to the session; it SHOULD be freshly generated when possible, but MAY be pre-placed when local key generation is of poor or unknown quality (see Section 9). An ephemeral key is innately unauthenticated, and so must be carried in a suitably

authenticated protocol.

- Identification: A generic term for a process by which a name, generally assigned by a server, is used to match a request against a known entity. Identification can be either authenticated (a subject name in a certificate) or unauthenticated (a text string).

- Perfect Forward Secrecy (PFS): For this protocol, it is the property that the compromise of long-term keying material does not lead to the compromise of the new long-term keying material carried in the protocol.

- Shared Secret: A value known to two or more entities in advance of a protocol session in which it will be used, and intended to be unknown to any others. In this document, the value must be a suitable basis for derivation of a MAC (Message Authentication Code) or encryption key. Pass phrases that are used as a shared secret must be treated as confidential by the holders of the secret.

- Shrouding: A generic term to cover methods of masking the content of an object from unauthorized viewers, taken from RSA's PKCS specifications. The most common method of shrouding used is encryption of the data at the application layer. This document defines two shrouding methods that employ encryption at the application layer but other shrouding methods can be defined that do not employ encryption at the application layer.

- Signature-capable: Applies to certificates or keys. Certificates that can be used to verify signatures and keys that can be used to generate digital signatures. This refers to either dual-use or signature-only certificates/keys.

- Signature-only: Applies to certificate or keys. Certificates that can only be used to verify digital signatures, when the KeyUsage extension is set to digitalSignature, and keys whose only intended usage is digital signature.

In this document the server is the entity that generates the key. This can be either the RA (Registration Authority) or CA (Certification Authority).

2. Protocol Flows for Supported Scenarios

This section describes the supported scenarios and specifies the CMC requests and responses to support them:

1. Clients use a shared secret (e.g., password) (see Section 1.2) to provide authentication and request that the server use the same shared secret to encrypt the server-generated keys.

2. Clients use a shared secret to provide authentication and request that the server use an ephemeral key (see Section 1.2) to encrypt the server-generated keys.

3. Clients use a key pair previously certified by a CA (i.e., a private key and a certificate) to support digital signature authentication and request that the server use an ephemeral key to encrypt the server-generated keys.

4. Clients use a key pair previously certified by a CA (i.e., a private key and a certificate) to support digital signature authentication and request that the server use a certificate to encrypt the server-generated keys. Some additional details:

   * If the client's authentication certificate is signature-only, then the client also needs an encryption-capable certificate that the server will use to protect the private key.
   * If the client's certificate is dual-use, then the client only needs the one certified key pair to generate the SignedData that encapsulates the certificate request and to decrypt the EnvelopedData that encapsulates the server-generated key.

The characteristics of the four scenarios are as follows:

- Scenarios that employ shared secrets (Scenarios 1 and 2) are consider by some to be more human friendly than scenarios that employ certificates because there is less initial PKI overhead. While true, the use of shared secrets introduces its own set of key management issues.

  * When a shared secret is used for key protection (Scenario 1), secrecy of the shared secret is required for the lifetime of the key. That burden can be eased by limiting the shared secret to a single use and having all parties destroy it after the key it protects has been received by the subject. Thus, even though by definition Forward Secrecy is not possible, when the shared secret no longer exists it cannot be compromised. However the bigger problem is likely to be compromise of the shared secret before it is ever used. That cannot be mitigated except by starting over with a fresh shared secret in a way that avoids compromise.

  * When a shared secret is used for authentication only (Scenario 2), there is even greater value in limiting use to one

transaction, as that reduces the chance of identity cloning or unintended use of the shared secret as a group identity.

* When rekeying server-generated keys originally requested using a shared secret, clients that received signature-capable keys can use them to provide the POP necessary for rekey instead of using a shared secret. The same is true of clients that received encryption only keys assuming they are permitted to generate a one-time signature for rekey purposes. Clients that received encryption-only certificates that are not permitted to generate a one-time signature for rekey requests need a new shared secret for rekeys. The mechanism that distributed shared secrets is out-of-scope.

* Note distributing the new shared secret in-band is of dubious value because a) maintaining the secrecy of the new shred secret is just as hard for the user as maintaining the secrecy of the to-be-replaced shared secret b) distribution of a new shared secret protected with a to-be replaced shared secret does not provide PFS.

* Servers that place certificate request in the pending state should consider how long they are in that state as well as how long the shared secret is considered valid (i.e., do not delete the shared secret from a pending request before either rejecting the request because of timeout or fulfilling the request because the request that checks on the status will not validate).

o For scenarios that use ephemeral keys to protect the returned private key (i.e., Scenarios 2 and 3):

* PFS is provided presuming the ephemeral key is forgotten as well as any other information necessary to generate the ephemeral key after the certificate request is successfully processed by the client.

* Client use of the ephemeral key mitigates the risk of compromise of a pre-existing certificate and key while in the supply chain (see Section 9).

o For scenarios that only use certificates (i.e., Scenario 4), PFS is not provided because the client uses a long-term private key for private key protection and if compromised the next key can also be compromised.

Note that there are existing CP-based (Certificate Policy) requirements for protecting the private key(s) associated with

the certificate(s) (see Section 9).

In the scenarios above, the entity generating the key pair can be the Certification Authority (CA) or the Registration Authority (RA); in fact the RA or CA can use a separate key generator device (e.g., a hardware security module) but interactions with that device are out-of-scope. Sections 2.1-2.4 depict protocol flows for scenarios with a client and CA only where the CA generates the keys. Section 2.5 depicts protocol flows for scenarios that involve the client, RA, and CA where the RA generates the keys. Section 2.5 also depicts protocol flows for so-called "RA-involved" scenarios where the CA generates the keys but the RA performs identity checks.

All scenarios described herein require the server to have some a priori knowledge of the client. How this knowledge is obtained is out-of-scope, but the scenarios offer different opportunities/constraints for obtaining the information:

- In the shared secret cases, the method for prior knowledge has to cope with securely delivering and storing the shared secret.
- In the certificate cases, there are two alternatives:
  - The key pair was generated by the server, so the certificate is evidence of that prior knowledge.
  - The certificate was issued from another CA. Even when there are no grounds for cross-certification, the certificate can still be used as an artifact for registration/enrollment prior to the client making a certificate request, which is advantageous because the key pair bound to the identifier in the certificate enables the server to authenticate the source of the eventual certificate request and positively link it to the registration information.

Note: The initial certificate key pairs could be considered a special case of the shared secret scenario that improves on the security of the shared secret mechanism and mitigates some of the management burden and cost. For example, the certificates could be special-purpose--issued by the manufacturer and recognized by the server solely for authentication against a registration list (i.e., not usable for anything else). If a manufacturer initializes the device with a shared secret, then that shared secret has to be distributed securely to the eventual enrolling CA via the device owner, but independently of the device. If the manufacturer instead installs a key pair and generic certificate, the certificate can take the place of the shared secret that would otherwise have to be independently provided to the central key generation CA. The management process is roughly the same, but the information that has to be handled now longer has to be kept secret. That means there are many more options for how that

information can be managed and distributed.

Normally the client's identifier and the shared secret are generated by the server and then securely transported to the client; there is no practical reason why this cannot be done in the opposite direction. In this case, the client will generate an identifier, shared secret, and an ephemeral key. When the server receives the CMC message, it recognizes that it does not correspond to an existing identifier/secret pair and puts the request on hold. The client then communicates the identifier and secret to the server via an out-of-band means. The server then performs the necessary user management dealing with identity validation and certificate setup. If that passes and the ephemeral key passes applicable public key validation tests, then the certificate will be issued and the response returned protected with ephemeral key. If it does not pass checking then the certificate will fail to issue. An ephemeral key is generated by the client to ensure PFS is provided as the client may not get the same degree of confidentiality because the client is unaware how the private key has been provided.

Note that there is no scenario where the response is protected with a key/certificate that only supports digital signatures. This is because the "protection" afforded by digital signatures alone does not include confidentiality, which is required to ensure that the server-generated private key is only disclosed to the client.

Note also that there is also no scenario where the client uses an encryption-only certificate and is unable to generate a digital signature to provide authentication. This is because the "protection" afforded by encryption-only certificates does not include authentication. Technically, there are Authenticated Encryption with Associated Data (AEAD) algorithms (i.e., dual-service algorithms) that support both authentication and encryption but their use is beyond the scope of this document.

In all of the scenarios, the client can validate that the response came from the CA or RA by validating the digital signature on the SignedData to a Trust Anchor (TA). After the EnvelopedData is decrypted, the client can verify that the private key is associated with the public key in the returned certificate and that the certificate validates back to an authorized TA.

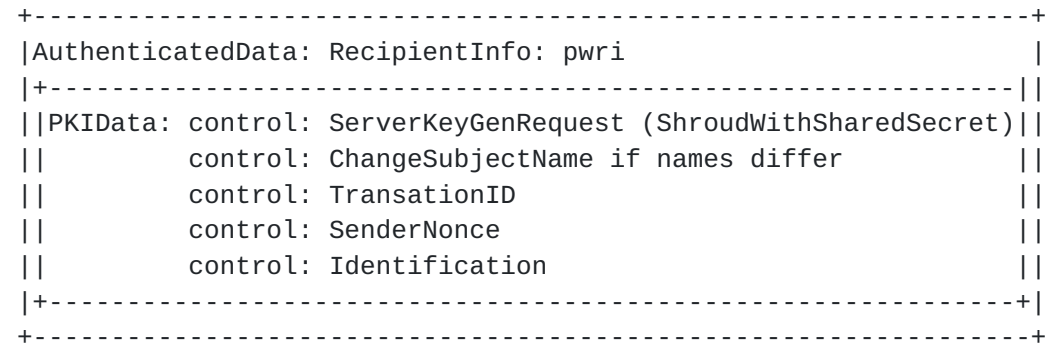
The scenarios in the subsections assume that the transaction identifier and nonce controls are used for transaction processing and replay protection, respectively, but they are optional, as specified in [RFC5272]. Also, the scenarios assume the CMC Status Information v2 control is not included when the response is a success, as allowed by [RFC5272]. See Appendix B for additional example scenarios.

A client requesting a certificate for a different name than one already issued, either based on the certificate being used in Scenarios 3 and 4 or the name associated with the shared secret in Scenarios 1 and 2, includes the Change Subject Name attribute to ensure the server will not reject the request because the name in the certificate used to sign the request does not match the name in the request. Note this is not depicted in the diagrams that follow because the attribute is included within the ServerKeyGenRequest.

2.1. Shared Secret for Authentication and Key Protection

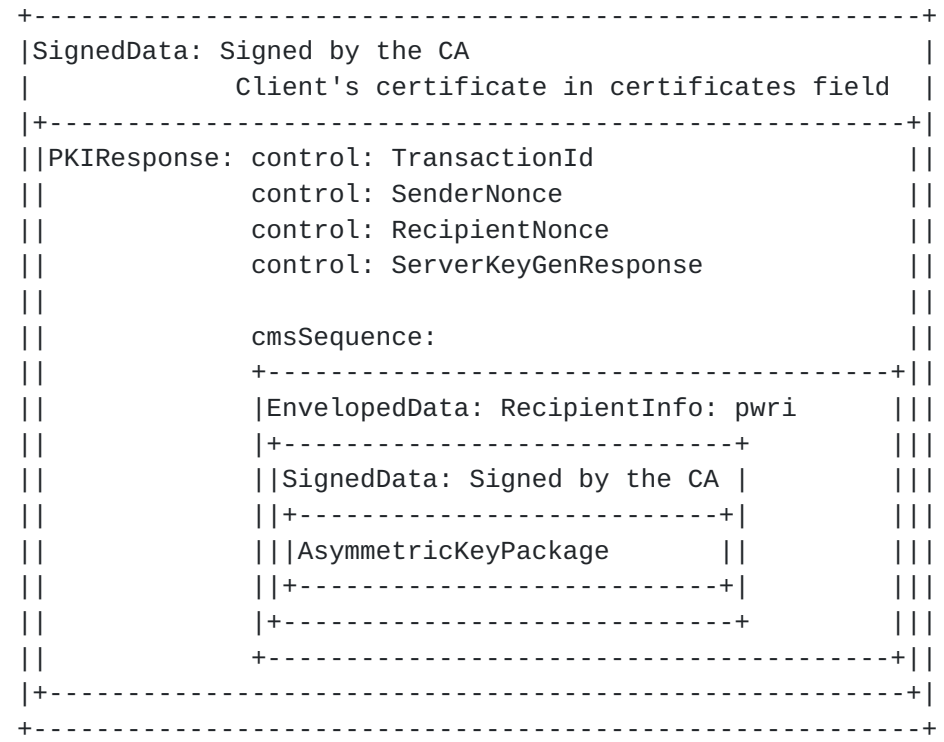
The shared secret allows the server to authenticate the client and allows the server to encrypt the server-generated key for the client. The shared secret is distributed via an out-of-band mechanism that is out-of-scope of this document. Also note that the server and client need to share a non-secret identification string that the client can assert in a request so that the server will know which shared secret is being used.

When the client generates its request, the client includes the following control attributes in a PKIData content type [RFC5272]: Server Key Generation Request (see Section 6.1), Transaction Identifier [RFC5272], Sender Nonce [RFC5272], and Identification [RFC5272]. The Server Key Generation Request control indicates that the shroudMethod is shroud with shared secret (see Section 4.2). The PKIData is encapsulated in a CMS AuthenticatedData content type and the password RecipientInfo (i.e., pwri CHOICE) is used [RFC5652]. Note that reqSequence, cmsSequence, and otherMsgSequence are not included in the PKIData for the server-side key generation request. The following depicts this:



After the server authenticates the client and verified the request, the server generates a response that includes the server-generated key and any associated parameters in an Asymmetric Key Package content type [RFC5958]. The Asymmetric Key Package is then encapsulated within a SignedData, which is signed by the server, and that is further encapsulated within an EnvelopedData using the

password RecipientInfo (i.e., pwri CHOICE). The EnvelopedData, which is encrypted for the client, is then placed in a PKIResponse cmsSequence [RFC5272] and the following controls are included in the PKIResponse: Transaction Identifier, Sender Nonce, Recipient Nonce [RFC5272], and Server Key Generation Response (see Section 4.2). The PKIResponse is then encapsulated in a SignedData, which is signed by the server, and the client's certificate associated with the server-generated key is placed in the outer-most SignedData's certificates field [RFC5652]. The following depicts this:

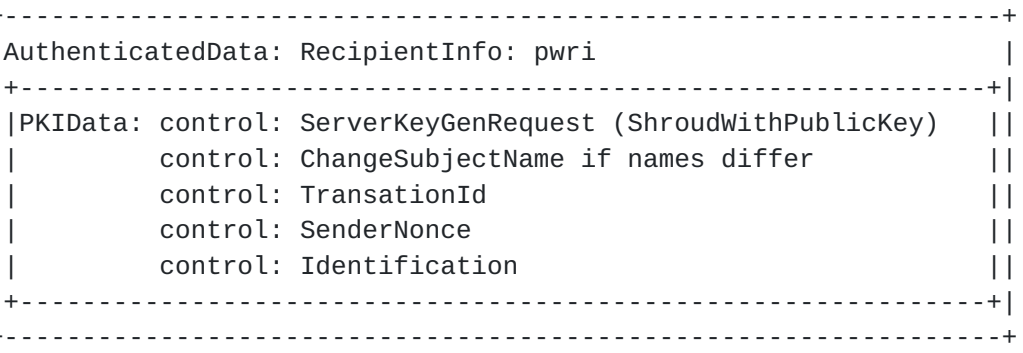


2.2 Shared Secret for Authentication and Ephemeral Key for Protection

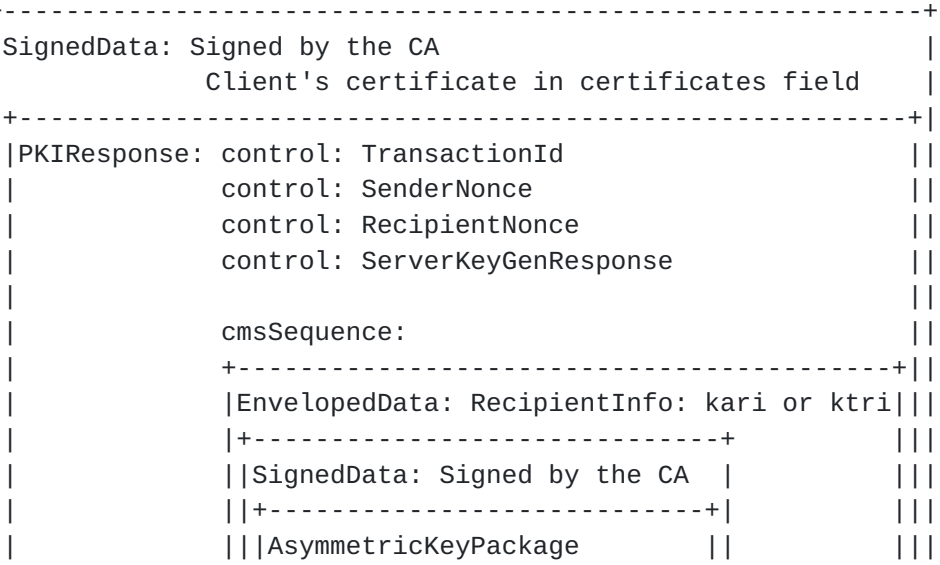
The shared secret allows the server to authenticate the client and the ephemeral key allows the server to use a different key to encrypt the server-generated key for the client. The shared secret is distributed via an out-of-band mechanism that is out-of-scope of this document. Also note that the client needs an identification string to allow the server to determine which shared secret is being used.

When the client provides an ephemeral key to protect the response, the client includes the following control attributes in a PKIData content type [RFC5272]: Server Key Generation Request control (see Section 6.1), Transaction Identifier [RFC5272], Sender Nonce [RFC5272], and Identification [RFC5272]. The Server Key Generation Request control indicates that the shroudMethod is shroud with public key and that the bareKey CHOICE is used (see Section 4.1). The

PKIData is encapsulated in an AuthenticatedData content type and the password RecipientInfo (i.e., pwri CHOICE) is used [RFC5652]. Note that reqSequence, cmsSequence, or otherMsgSequence are not included. The following depicts this:



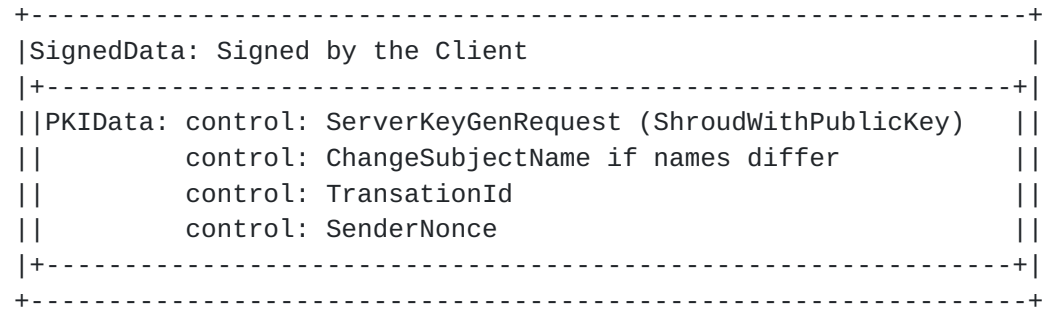
After the server has authenticated the client and verified the request, the server returns the server-generated key and any associated parameters in an Asymmetric Key Package content type [RFC5958]. The Asymmetric Key Package is then encapsulated within a SignedData, which is signed by the server, and that is further encapsulated within an EnvelopedData using the key agreement or key transport RecipientInfo (i.e., kari or ktri CHOICE). The EnvelopedData, which is encrypted for the client, is then placed in a PKIResponse cmsSequence [RFC5272] and the following controls are included: Transaction Identifier, Sender Nonce, Recipient Nonce [RFC5272], and Server Key Generation Response (see Section 6.2). The PKIResponse is then encapsulated in a SignedData, which is signed by the server, and the certificate associated with the server-generated key is placed in the outer-most SignedData's certificates field [RFC5652]. The following depicts this:



2.3. Certificate for Authentication and Ephemeral Key for Protection

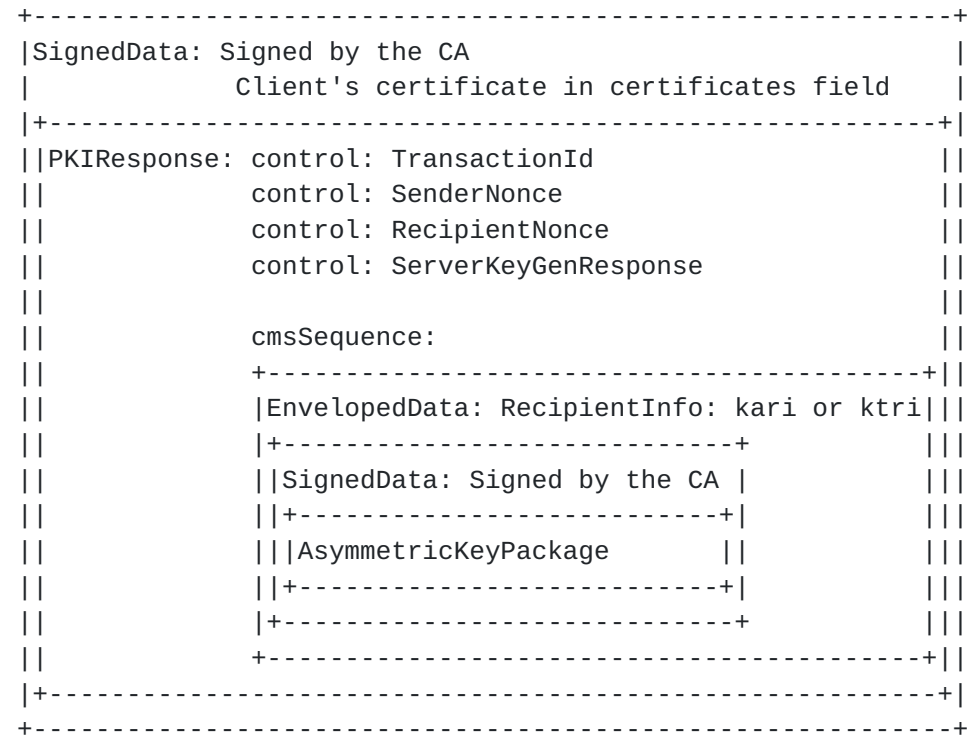
This scenario differs from the scenarios in Sections 2.1 and 2.2 in that the client encapsulates the PKIData in a SignedData instead of an AuthenticatedData (i.e., the client uses its private key associated with its signature-capable certificate to sign the PKIData) but is similar to Section 2.2 for the response. As implied in [RFC5272], clients omit the Identification and Identity Proof controls when using certificates to support digital signature authentication.

When the client generates its request, the client includes the following control attributes in a PKIData content type [RFC5272]: Server Key Generation Request (see Section 6.1), Transaction Identifier [RFC5272], and Sender Nonce [RFC5272]. The Server Key Generation Request control indicates the shroudMethod is shroud with public key and the bareKey CHOICE is used (see Section 4.1). The PKIData is encapsulated in a SignedData content type [RFC5652]. Note that reqSequence, cmsSequence, and otherMsgSequence are not included in the PKIData for the server-side key generation request. The following depicts this:



After the server has authenticated the client and verified the request, the server returns the server-generated key and any associated parameters in an AsymmetricKeyPackage content type [RFC5958]. The AsymmetricKeyPackage is then encapsulated within a SignedData, which is signed by the server, and that is encapsulated within an EnvelopedData using the key agreement or key transport RecipientInfo (i.e., kari or ktri CHOICE). The EnvelopedData, which is encrypted for the client, is then placed in a PKIResponse cmsSequence [RFC5272] and the following controls are included: Transaction Identifier, Sender Nonce, Recipient Nonce [RFC5272], and

Server Key Generation Response (see Section 6.2). The PKIResponse is then encapsulated in a SignedData, which is signed by the server and the certificate associated with the server-generated key is placed in the outer-most SignedData's certificates field [RFC5652]. The following depicts this:



2.4. Certificate for Authentication and Key Protection

If a client already has been issued a signature-capable certificate, then it can use this certificate to authenticate the requests. If the certificate also indicates support for encryption (i.e., the key usage extension is set to keyEncipherment or keyAgreement), then the client can request that the server use the same certificate to protect the server-generated key (see Section 2.4.1). If the certificate does not indicate support for encryption, then the client can provide the server with another certificate to use to encrypt the server-generated key (see Section 2.4.2). The certificate that protects the server-generated key MUST be encryption-capable.

These scenarios differ from the scenarios in Section 2.3 in that the response is protected with a previously certified key instead of an ephemeral key. As specified in [RFC5272], clients omit the Identification and Identity Proof controls when using certificates to support digital signature authentication.

2.4.1. Same Certificate for Authentication and Key Protection

This scenario is the same as in Section 2.3. except the Server Key Generation Request control includes the certIdentifier CHOICE instead of the bareKey CHOICE. The certIdentifier is sufficient information for the server since the same certificate is already provided in the SignedData.

2.4.2. Different Certificates for Authentication and Key Protection

This scenario is the same as in Section 2.3. except the Server Key Generation Request control includes the certificate CHOICE instead of the bareKey CHOICE. When using two certificates, all names in the two certificates MUST match to ensure the CA will not reject the request due to name mismatches.

2.5. RA Scenarios

In Sections 2.1-2.4, client-to-CA protocol flows were illustrated where the CA generates the client's key and no RA was involved. This section illustrates client-to-RA-to-CA protocol flows. The scenarios divide into two basic categories, according to whether the RA or the CA generates the key pair.

Regardless of whether the RA or the CA generates the key pair, the intent of the RA here is to be transparent. That is, clients initiate the same request regardless of the entity that ultimately generates the keys.

When the RA generates the key on behalf of the client, the RA augments the taggedRequest from the client with the RA-generated public key and applies POP with the corresponding private key (i.e., the RA includes the public key and signs the request). This becomes the requestSequences in a new PKIData that the RA sends to the CA, and that repeats the controls received from the client. Then the RA signs the new PKIData with its own signing key. In this way, the RA effectively becomes the client from the CA's perspective. However, the RA and CA already have an established trust relationship (i.e., the RA has been issued a certificate from the CA), which might not be true for the client. The protocol exchange between the RA and CA is identical to a client enrolling with a CMC Full PKI Request; therefore, the CA need not know about or support the Server Key Generation Request and Server Key Generation Response controls. The PKIResponse to the client now has to accommodate the fact that the asymmetric key package is generated by the RA, whereas the certificate is generated by the CA. This necessitates that the RA intercepts whatever response the CA returns to get the client's certificate and that the RA generates a signed response that includes the asymmetric key package as well as the client's certificate. See section 9 for security considerations.

When the RA participates in the process but does not generate the key, there are two possibilities. If the RA does not contribute to the protocol (its effects may be procedural or out-of-band), then it can simply pass the messages it receives to the other party when warranted. If that is not warranted, the RA would generate the usual response for the associated failure. No message flow depicting this possibility is included in this document for reasons of brevity. Alternatively, the RA may be responsible for processing certain aspects of the request and needs to vouch for that when forwarding the client request to the CA. The RA does this per [RFC5272] by embedding the client request in a Full PKI Request that it signs, containing controls for the processing that it performs. Here, as in Sections 2.1-2.4, the CA needs to fully understand and support the Server Key Generation Request and Server Key Generation Response controls, since the CA has to generate the key and construct the asymmetric key package.

In the figures that follow, the following abbreviations are used:

- o SKGReq is the Server Key Generation Request control (see Section 6.1),
- o SKGRes is the Server Key Generation Response control (see Section 6.2),
- o TransactionId is the Transaction Identifier control [RFC5272],
- o SenderNonce is the Sender Nonce control [RFC5272],
- o RecipientNonce is the Recipient Nonce control [RFC5272],
- o AKP is an Asymmetric Key Package [RFC5958] (i.e., the private key),
- o {} denotes encryption (i.e., EnvelopedData), o <> denotes signatures (i.e., SignedData), with () identifying some of the information is carried in unsignedAttrs for clarity,
- o [] denotes authentication (i.e., SignedData or AuthenticatedData). AuthenticatedData is used when clients use a shared secret for authentication.
- o control is CMC's controlSequence,
- o reqSeq is CMC's requestSequence,
- o cmsSeq is CMC's cmsSequence.

PKCS#10 [RFC2986], CRMF (Certificate Request Message Format) [RFC4211], and other request message [RFC5272] are the certification request formats supported.

2.5.1. RA-Generated Key Scenarios

There are some differences in the protocol flows when an RA generates the key:

- o The RA MUST be issued a certificate from the CA. This means all

of the RA-generated PKIData are encapsulated in a SignedData. Further, the RA's certificate can be used for identification and linking identity and POP information.

- The RA generates the certification request for the client by:

  1. Using the name from the certificate for the private key the RA will use to sign the PKIData.

  2. Using the information from the client's request.

  3. The RA includes the Change Subject Name control [RFC6402] either in the PKCS #10 or CRMF TaggedRequest because the name in the certificate used for verifying the PKIData signature must match the name in the certificate request. The Change Subject Name attribute allows the names to be different.

  4. Modifying the taggedRequest as necessary. Notably adding the public key but also adding certificate extensions, etc. Note that the Modify Certificate Template control is not needed as the RA is generating a new PKIData.

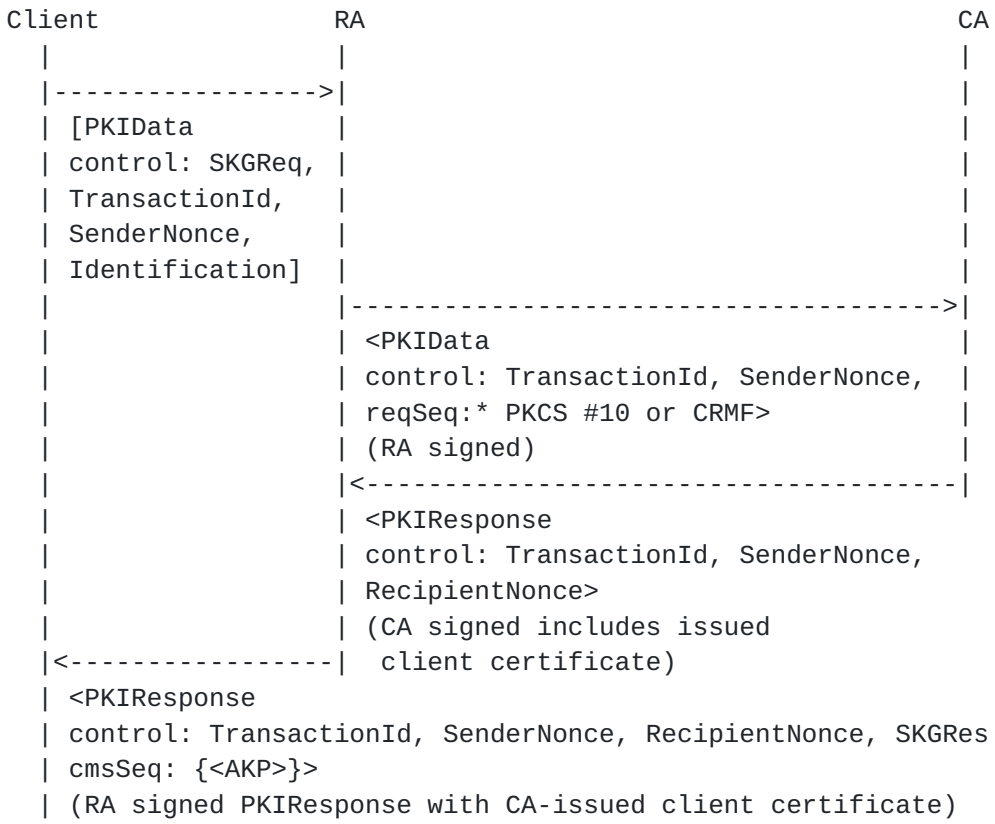
  5. Generating the POP information for the certificateRequest containing the RA-generated keys. For keys that support digital signatures, the RA includes the POPSigningKey in the CRMF or the signature in the PKCS #10. For encryption-only keys, the RA can indicate that it performed POP by including the RA POP Witness control. Note the CA could force the RA to prove it has possession of the key with the encrypted/decrypted POP mechanism for [RFC5272], but this adds additional round trips and is discussed later in this section.

  6. Signing the certification request (i.e., the PKIData) with the RA's private key.

The RA can also generate bulk requests (i.e., include more than one request in cmsSequence) with the Bulk Request control [RFC5272] but these controls are not depicted in the following sections to simplify the protocol flows. When the RA is requesting more than one key for a given client, the RA includes each request in the reqSequence.

The following message flow applies when the RA generates the key. It supports all of the previously defined choices for authentication and shrouding. The diagram below depicts use of a shared secret for authentication by including the Identification control and encapsulating the client's PKIData in an AuthenticatedData. If a digital signature is used for authentication, the Identification control is omitted and the client encapsulates its PKIData in a

SignedData. In the response, the RecipientInfo for the EnvelopedData encapsulating the <AKP> depends on whether the key was protected with a shared secret (pwri), ephemeral key (ktri or kari), or certificate (ktri or kari).



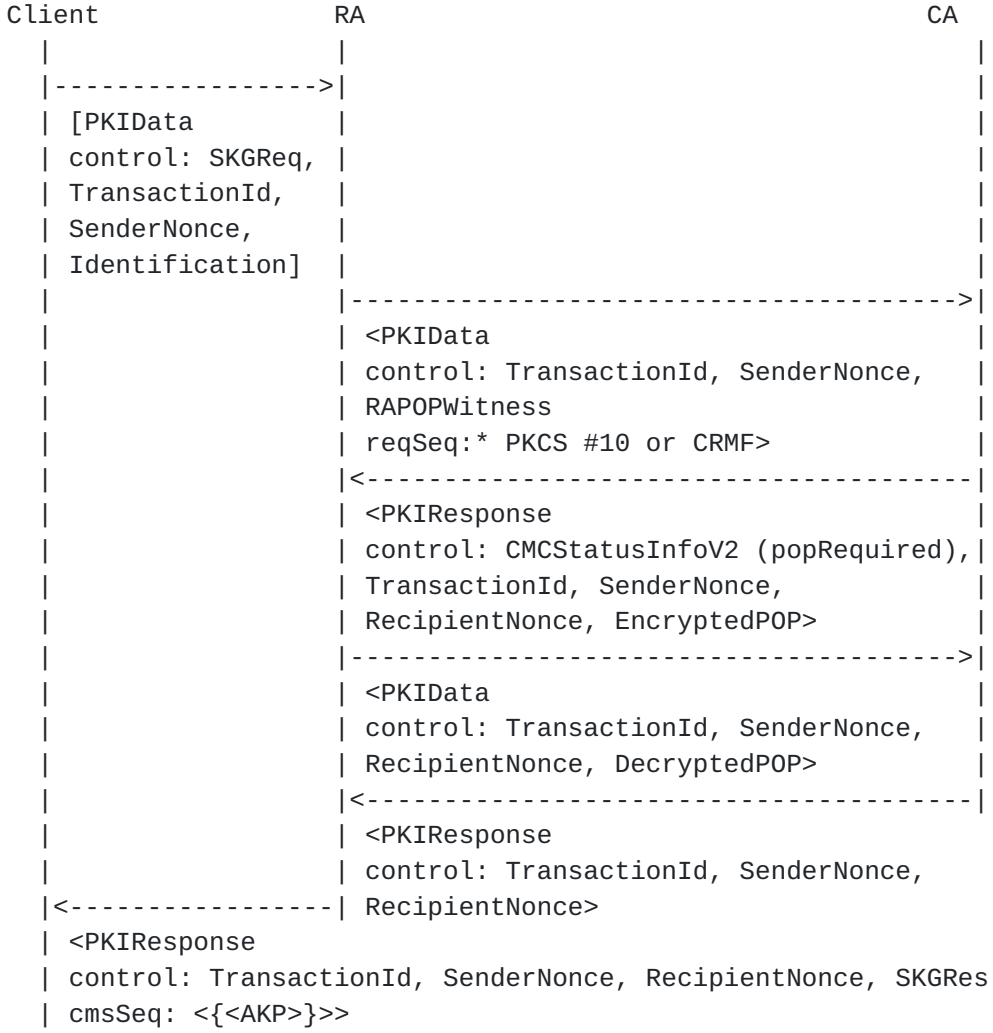
* Includes ChangeSubjectName attribute in PKCS #10 or CRMF.

NOTE: There is no need for the RA to provide the SKGReq or the {<AKP>} to the CA. The CA will not be able to access the contents of the {<AKP>} because it is encrypted for the client and the CA's response is always returned to the RA because the RA needs to provide the generated {<AKP>} back to the client.

The RA intercepts the response from the CA; it strips the CA's signature and creates a new PKIResponse for the client. The controlSequence is comprised of the Transaction Identifier and Recipient Nonce fields from the client's request, the RA's Sender Nonce, and the Server Key Generation Response (see Section 6.2); the cmsSequence includes the RA-generated {<AKP>}; and the RA signs the PKIReponse and includes the client's certificate, which was returned in the CA's SignedData.

When the RA is generating an encryption-only key pair for the client, and the CA wishes to force the RA to prove it has possession of the

private key, but the RA cannot use it to generate a one-time signature, then the flow is as follows:



* Includes ChangeSubjectName attribute in PKCS #10 or CRMF.

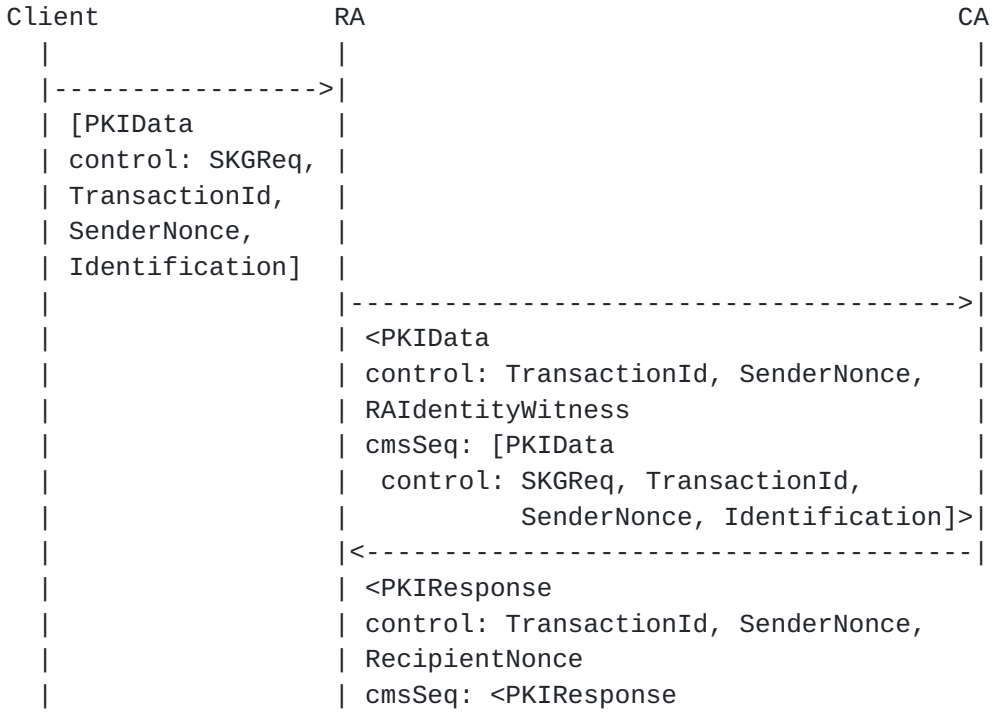
NOTE: The number of round trips between the RA and CA in the above figure is twice as many as the first figure in this Section and in Section 2.5.1.1; however, the additional round trip is specified in [RFC5272] (i.e., this document does not introduce the additional round trip). The additional round trip is necessary when the CA forces the RA to perform POP with the CA. While the additional round trip might be problematic between the client and server, the quality of communication connectivity between RA and CA should not make the additional round trips as problematic as between clients and RAs or CAs.

2.5.2. RA-Involved Scenarios

This section illustrates a message flow for when the CA generates the client's key. Here too the RA MUST be issued a certificate from the CA, which means that all of the RA-generated PKIData are encapsulated in a SignedData. Furthermore, the RA's certificate can be used for identification and linking identity and POP information. The RA can include the RA Identity Witness control to tell the CA that it performed the client identity checks; the RA will omit the control if it does not perform these checks.

The RA can include a Modify Certification Request control [RFC5272] in the PKIData that encapsulates the client's request but these controls are not shown below. The RA does this when it wishes to modify the request present in the Server Key Generation Request control. The RA MUST NOT use the RA POP Witness control if the CA is to generate the key. This control indicates that the RA performed POP, but the key for which POP is claimed has not yet been generated.

The diagram below depicts the client's use of a shared secret for authentication by including the Identification control and encapsulating the client's PKIData in an AuthenticatedData. If a digital signature is used for authentication, the Identification control is omitted and the client's PKIData is encapsulated in a SignedData. The RA encapsulates the client's request in its PKIData by placing the client request in the cmsSequence, and includes controls such as Transaction Identifier and Sender Nonce controls as well as RA Identity Witness control if the RA checks the client's identity.



```
    |                  |  control: TransactionId, SenderNonce,
    |                  |           RecipientNonce, SKGRes
    |                  |  cmsSeq: {<AKP>} >
    |                  |  (CA signed includes issued
    |                  |   client certificate)
    |<-----------------|  > (CA signed)
    | <PKIResponse
    | control: TransactionId, SenderNonce, RecipientNonce, SKGRes
    | cmsSeq: {<AKP>}>
    | (CA signed with issued client certificate)
```

When the RA receives the response from the CA, it strips the CA's response for the RA off and passes the inner response to the client unchanged. The difference between this scenario and the scenarios in Section 2.5.1 is that the signature on the PKIResponse is generated by the CA not the RA.

Note that the additional round trips to prove possession of an encryption-only key depicted in Section 2.5.1 are unnecessary here because the CA generates the asymmetric key pair and it does not need to prove to itself that it has the keys.

3. Generating PKIData and PKIResponse

[RFC5272] defines PKIData as follows:

```
  PKIData ::= SEQUENCE {
    controlSequence   SEQUENCE SIZE(0..MAX) OF TaggedAttribute,
    reqSequence       SEQUENCE SIZE(0..MAX) OF TaggedRequest,
    cmsSequence       SEQUENCE SIZE(0..MAX) OF TaggedContentInfo,
    otherMsgSequence  SEQUENCE SIZE(0..MAX) OF OtherMsg
    }
```

[RFC5272] defines PKIResponse as follows:

```
  PKIResponse ::= SEQUENCE {
    controlSequence   SEQUENCE SIZE(0..MAX) OF TaggedAttribute,
    cmsSequence       SEQUENCE SIZE(0..MAX) OF TaggedContentInfo,
    otherMsgSequence  SEQUENCE SIZE(0..MAX) OF OtherMsg
    }
```

3.1. Client Requests

When the client generates its request, the Server Key Generation Request control (see Section 6.1) is included in controlSequence; the other sequences (i.e., reqSequence, cmsSequence, and otherMsgSequence) are omitted. If a shared secret is used for

authentication, the Identification control [RFC5272] is included in the controlSequence to ensure that the server can locate the shared secret needed to authenticate the request. Additional controls can be included in controlSequence such as Sender Nonce and Transaction Identifier [RFC5272]. The reqSequence, which is included for client-generated key certification requests, is not needed as the Server Key Generation Request control includes the certification request. The client's request is either encapsulated in an AuthenticatedData or a SignedData depending on whether the client is using a shared secret or a digital signature key to authenticate the request. If the client wishes to request a certificate with a different name than the one that is present in the certificate that authenticates the request or associated with the shared secret, the client must nevertheless populate the certificateRequest with the Subject Name used in the existing certificate, and then include the Change Subject Name control to identify the Subject Name that is desired instead. Otherwise the server will reject the request as required in [RFC6402].

3.2. RA Processing of Client Requests

If an RA is involved, then it can do the following:

- Forward the request as-is to the CA. This happens when the CA authenticates the request, performs the identity checks, and generates the keys.

- Authenticate or not the request and place one or more client-authenticated PKIData in cmsSequence; reqSequence and otherMsgSequence are omitted. Here the RA does not have the shared secret necessary to authenticate the request. The RA can also include additional controls in controlSequence such as the Modify Certification Request control if the RA needs to modify the client's request and the Sender Nonce and Transaction Identifier controls for replay protection and transaction processing. If the RA performs the Identity checks it can include the RA Identity Witness control [RFC6402], otherwise it is omitted. After generation of its PKIData, the RA encapsulates it in a SignedData as part of the digital signature process.

- Authenticate the request and generate the client's keys. When the RA generates the client's key, the RA generates a new PKIData with a reqSequence; cmsSequence and otherMsgSequence are omitted. The RA must assert its own name as the Subject Name in the certificateRequest, and include the Change Subject Name attribute carrying the intended client name, as specified in [RFC6402], in the PKCS#10 or CRMF because [RFC6402] requires that the name in the request match the name in the certificate used to

authenticate the request. If the RA-generated key is signature-capable, POP is provided in the typical fashion (i.e., the embedded CRMF or PKCS#10 request includes the POP). The RA can also include additional controls in controlSequence such as Sender Nonce and Transaction Identifier. After generation of its PKIData, the RA encapsulates it in a SignedData as part of the digital signature process.

- Reject the client's request and return a PKIResponse with an appropriate reason in the CMC Status Information V2 control. Additionally, the RA includes Transaction Identifier, Sender Nonce, and Recipient Nonce if the request included Transaction Identifier and Sender Nonce controls. The PKIResponse is encapsulated in a SignedData as part of the digital signature process. This document defines three additional error conditions (see Section 7):

  * For a Server Key Generation Request control using the ShroudWithPublicKey choice of certificate or certIdentifier, the RA can check that the certificate provided to protect the returned private key validates back to an authorized TA. If the certificate does not validate back to an authorized TA, then the RA returns a PKIResponse with a CMC Status Information v2 control indicating the request failed with an extendedFailInfo indicating badCertificate (see Section 7) encapsulated in a SignedData. Note that the RA performing this check will lessen the load on the CA, but this check need only be done by the RA when the RA is generating the keys; when the CA is generating the keys, technically it is up to the CA to perform this check if it receives a server-side key generation request from a client.

  * For a Server Key Generation Request control using the ShroudWithSharedSecret choice and where the RA knows the shared secret, the RA will reject the request if the shared secret does not match the one on the RA by returning a PKIResponse with a CMC Status Information control indicating the request failed with an extendedFailInfo indicating badSharedSecret (see Section 7) encapsulated in a SignedData. This is done because client authentication failed or the HMAC output was corrupted.

  * For a Server Key Generation Request control that has archiveKey set to TRUE, the RA is generating the client's keys, and the RA does not support archive, the RA will reject the request by returning a PKIResponse with a CMC Status Information v2 control indicating the request failed with an extendedFailInfo indicating archiveNotSupported (see Section 7) encapsulated in a SignedData. If the RA knows the CA also does not support

archival of keys, the RA, if it wishes, can reject the request in the same fashion; when the CA is generating the keys, technically it is up to the CA to perform this check if it receives a CA-generated key request from a client. Note that the RA performing this check will lessen the load on the CA, but it need only be done by the RA when the RA is generating the client's keys.

RAs can also batch more than one request together, by including each client request in a separate cmsSequence or reqSequence (for Simple PKI requests) along with a Batch Request control in the RA's PKIRequest control field. After generation of the PKIData, the RA encapsulates it in a SignedData as part of the digital signature process.

When verifying a SignedData signature, the RA verifies it back to an authorized TA.

3.3. CA Processing

CA processing of requests depends on the number of layers of encapsulation:

- Requests with a single layer of encapsulation will be validated back to an authorized TA if they are encapsulated in a SignedData or authenticated with the shared secret if they are encapsulated in an AuthenticatedData. For AuthenticatedData encapsulated requests the server locates the necessary shared secret with the information found in the Identification control. For a PKIRequest with a reqSequence, the server verifies the POP. Regardless of the encapsulation technique, the server performs the Identity checks and processes other controls such as Transaction Identifier and Sender Nonce. If any of these checks fail or processing of a control fails, the CA rejects the certification request with the appropriate error code, as specified in [RFC5272].

- Requests with multiple layers of encapsulation (i.e., those requests that are RA-involved) will first validate the signature on the outer SignedData back to an authorized TA and process any controls present such as RA Identity Witness, Modify Certificate Template, Sender Nonce, and Transaction Identifier, as per [RFC5272][RFC6402]. Inner requests are also processed as specified in the previous bullet. Failure to validate back to an authorized TA or control processing failures result in rejected requests with the appropriate error code, as specified in [RFC5272].

CAs may require that the RA prove that it has possession of encryption-only keys that do not support one-time signature use, by returning a PKIResponse indicating the request failed because POP is required and including the Encrypted POP control along with other appropriate controls. The response is signed by the CA. See Section 2.5.1.

After successfully authenticating the request and verifying the client's identity, the CA generates:

- Responses for single-layer encapsulated requests for RA-generated keys by issuing the certificate. If no controls were present in the request (see Appendix B), the PKIResponse is a Simple PKI Response [RFC5272], which includes no content and therefore no signature. With controls (e.g., Transaction Identifier, Sender Nonce, and Recipient Nonce), the PKIResponse includes the appropriate controls and is signed by the CA. The CA places the certificate in the SignedData certificates field.

- Responses for multi-layered encapsulation requests for RA-generated keys (See Appendix B) beginning as with the previous bullet to form the inner response. This is placed in cmsSequences of the outer PKIResponse, which also includes the Batch Response control as well as any other necessary controls in controlSequence. The CA generates a signature for the encapsulating SignedData.

- Responses for single layer encapsulated requests for CA-generated keys by generating the asymmetric key pair and issuing the certificate. The signed CA-generated PKIResponse includes the Server Key Generation Response control (see Section 6.2) along with other controls based on whether they were present in the controlSequence as well as the signed and then encrypted Asymmetric Key Package in cmsSequence. The CA places the certificate in the SignedData certificates field.

- Responses for multi-layered encapsulation requests for CA-generated keys beginning with the previous bullet followed by encapsulating the inner response in cmsSequence for the outer PKIResponse. The outer response also includes controls as necessary in controlSequence and the CA generates a signature for an encapsulating SignedData.

In all cases, the certificate issued is an X.509 certificate [RFC5280].

If the CA is unable to perform the request at this time or the entire request cannot be processed, it can return a signed PKIResponse with

a CMC Status Information control with a status of pending or partial along with pendInfo, which the client or RA uses to know when to ask the CA next about the request.

When the CA fails to or refuses to process the request, it returns a PKIResponse with a CMC Status Information control with the appropriate error code from [RFC5272] or from Section 7 of this document. Additionally, it includes Transaction Identifier, Sender Nonce, and Recipient Nonce in the response if the request included Transaction Identifier and Sender Nonce controls.

3.4. RA Processing of CA Responses

If the CA rejected the RA's request as indicated by a PKIResponse with CMC Status Information control that indicates "failed", then an out-of-band mechanism may be needed to determine the cause of failure in order to avoid a loop of the RA returning the same request at a later time only to have it also rejected.

If the CA returned a pending or partial response, the RA will use the information in the CMC Status Information control's pendInfo to poll the CA with a signed PKIRequest with a Query Pending control. CA processing continues as in Section 3.3.

RAs that are challenged by the CA to prove possession of an encryption-only RA-generated key validate the CA's signature back to an authorized TA, decrypt the POP, and process any other controls that are present. If any of these fail, then the RA terminates the request and informs the operator of the fault. Assuming the checks pass, the RA generates a PKIData that includes a Decrypted POP control and any other controls with no cmsSequence, reqSequence, or otherMsgSequence. The RA encapsulates the PKIData in a SignedData as part of the digital signature process and sends it to the CA. CA processing resumes as in Section 3.3.

Assuming the response is a success:

- If the CA returned a Simple PKI Response (i.e., the CA returns a certs-only message for RA-generated keys whose PKIRequest includes no additional controls), then the RA generates a PKIResponse for the client that includes the signed and then encrypted Asymmetric Key Package in cmsSequence (see Section 5), the Server Key Generation Response control (see Section 6.2), and any other controls. The RA encapsulates the PKIResponse for the client in a SignedData as part of the digital signature process and includes the client's certificate (from the returned certs-only message) in the SignedData's certificate field.

o If the CA returned a Full PKI Response, then one of three cases is possible:

* The response is for RA-generated keys. The RA generates a PKIResponse for the client that includes the signed and then encrypted Asymmetric Key Package in cmsSequence (see Section 5), the Server Key Generation Response control (see Section 6.2), and any other controls as appropriate. Finally, the RA encapsulates the PKIResponse for the client in a SignedData as part of the digital signature process and includes the client's certificate from the CA's response in the RA's SignedData certificates field.

* The response is for CA-generated keys. The RA processes any controls and assuming the processing passes, the RA strips off the outer SignedData and forwards the cmsSequence element (i.e., the inner SignedData) to the client.

Responses to batch requests (i.e., those Full PKI Requests that include the Batch Request control) are distributed by the RA to clients depending on whether the keys are RA-generated or CA-generated.

The RA, if it wishes, can also check the returned certificate to make sure it validates back to an authorized TA and that the returned certificate is consistent with the certificate request found in the Server Key Generation Request control. These checks cut down on errors at the client. If the RA detects that the certificate is not consistent, the RA SHOULD NOT return the certificate to the client and the RA SHOULD request that the certificate be revoked.

RA-generated keys for which a PKIResponse with a CMC Status Information control that is not success SHOULD NOT return the Server Key Generation Response or the encapsulated Asymmetric Key Package to the client because the CA did not certify the public key.

3.5. Client Processing of Responses

Clients validate the signature on all responses back to an authorized TA.

Responses signed by an RA with a client certificate signed by a CA whose certificate includes an id-kp-cmcCA EKU (Extended Key Usage) [RFC6402] will violate the "SHOULD" requirement found in [RFC6402] that the PKIResponse be signed by an entity with the same name as found in the certificate. Because the RA has generated the keys there are many more bad things an RA can do so this seemed like a tradeoff worth making.

4. Shrouding Algorithms

For the server-side key generation control attribute described in this document to function, clients need to tell the server in advance what encryption algorithm and what key value is to be used for encrypting the returned private key. The encrypted data returned is returned as an EnvelopedData object as defined by [RFC5652] and placed in the cmsSequence field of a PKIResponse [RFC5272]. Clients also need to tell the server what digital signature and hash algorithms they support to ensure the certification response and certificate can be verified.

Each request control for which the response includes encrypted data contains two fields to define the type of encryption used: algCapabilities and shroudMethod.

The algCapabilities field, see Section 6.1, contains the advertised capabilities of the client-side entity. This field uses the S/MIME Capabilities type defined in section 2.5.2 of [RFC5751]. The capabilities to be listed are digital signature algorithms, message digest algorithms, content encryption algorithms, key agreement algorithms, key encipherment algorithms, key-wrap algorithms, and key derivation algorithms. Encodings for SMIME Capability values for Elliptic Curve Key Agreement, Key Derivation Function, and Key Wrap algorithms can be found in [RFC5753], Message Digest and Signature algorithms can be found in [RFC5754], and AES Key Wrap with Padding can be found in [RFC5959].

The shroudMethod field (see Section 6.1) defines the method by which the server will do the key management of the content encryption key (CEK) value in EnvelopedData. The shroudMethod field uses the type ShroudMethod. This type is defined as:

```
ShroudMethod ::= AlgorithmIdentifier {
  SHROUD-ALGORITHM, { ShroudAlgorithmSet }
  }
```

When a new shroud method is defined it includes (a) the source of the key material, (b) the public or salting information, and (c) the method of protecting the Content Encryption Key (CEK) using the requested data, source key material, and salt. This document defines two shroud methods: id-cmc-shroudWithPublicKey and id-cmc-shroudWithSharedSecret. Clients and servers MUST support id-cmc-shroudWithPublicKey. Client and servers SHOULD support id-cmc-shroudWithSharedSecret.

Other shrouding methods could be defined in the future that would not involve the use of EnvelopedData. For example, one could conceive of

a shrouding method that required the use of Transport Layer Security (TLS) [RFC5246] to provide the necessary security between the server and the client. This document does not define any such mechanism.

4.1. Shroud with a Public Key

Clients can indicate that the server use a public key, either wrapped in a certificate or as a bare public key, to protect the server-generated key. For this option, the key material is either included or referenced by a key identifier. The following object identifier identifies the shroudWithPublicKey shroud method:

```
id-alg-shroudWithPublicKey OBJECT IDENTIFIER ::= { id-alg XX }
```

shroudWithPublicKey has the ASN.1 type ShroudWithPublicKey:

```
srda-shroudWithPublicKey SHROUD-ALGORITHM ::= {
   IDENTIFIED BY id-alg-shroudWithPublicKey,
   PARAMS TYPE ShroudWithPublicKey ARE required,
   SMIME-CAPS { IDENTIFIED BY id-alg-shroudWithPublicKey }
   }

ShroudWithPublicKey ::= CHOICE {
  certificate         Certificate,
  certIdentifier  [1] SignerIdentifier,
  bareKey         [2] SEQUENCE {
    publicKey  SubjectPublicKeyInfo,
    ski        SubjectKeyIdentifier
    }
  }
```

The fields of type ShroudWithPublicKey have the following meanings:

- certificate provides a public key certificate containing the public key to be used for encrypting the server-generated private key from the server to the client.

- certIdentifier provides a pointer to a public key certificate located in the SignedData that encapsulates the client's PKIData.

  For the above two fields, servers SHOULD check that the subject and, if included, subject alternative names match in some way with the entity that the private key is destined for. Servers do this to ensure the key they have made for the client is intended for the correct client. This mechanism is beyond the scope of this document.

- bareKey allows for an arbitrary public key to be used to return

the encrypted private key.

- publicKey contains the public key to be used when generating the EnvelopedData returned from the server to the client.

- ski contains the SubjectKeyIdentifier that will be used in CMS EnvelopedData to identify the public key when encrypting the private key from the server to the client.

When this method is used with the certificate option, the server validates the certificate back to a trust anchor. Further, the server checks that the client-provided certificate belongs to the same client that authenticated the certification request (e.g. the certificate subjects match or the client-provided certificate belongs to the same entity as the authentication shared secret). If either of these checks fails, then the server returns a CMCFailInfo with the value of badCertificate, which is defined in Section 7.

4.2. Shroud with a Shared Secret

Clients can indicate that servers use a shared secret value to protect the server-generated private key. For this option, the key material is identified by the identifier; the key derivation algorithms supported by the client are included in the algCapabilities field. No salting material is provided by the client. The derived key is then used as a key encryption key in the EnvelopedData recipient info structure. The following object identifier identifies the shroudWithSharedSecret shroud method:

```
id-alg-shroudWithSharedSecret OBJECT IDENTIFIER ::= {id-alg XX}
```

shroudWithSharedSecret has the ASN.1 type ShroudWithSharedSecret:

```
shrda-shroudWithSharedSecret SHROUD-ALGORITHM ::= {
  IDENTIFIED BY id-alg-shroudWithSharedSecret
  PARAMS TYPE ShroudWithSharedSecret ARE required
  SMIME-CAPS { IDENTIFIED BY id-alg-shroudWithSharedSecret }
  }

ShroudWithSharedSecret ::= UTF8String
```

The client includes an identifier in the ShroudWithSharedSecret field, which is an UTFString [RFC5280], that the server uses to locate the shared secret to be used to protect the returned server-generated private key. The secret identified by the ShroudWithSharedSecret field may be different than the secret referred to by the identification control, which is used to identify the shared secret used to authenticate the request. In addition, the

client needs to place both a key derivation function and a key wrap function in the set of capabilities advertised by the client in the algCapabilities field.

When this method is used, the server checks that the chosen shared secret belongs to the authenticated identity of the entity that generated the certification request. If this check fails, then the server returns a CMCFailInfo with the value of badSharedSecret, which is defined in Section 7. In general, while it is expected that the same identity token and shared secret used to do the identity authentication are used to derive the key encryption key this is not required.

5. Returned Key Format

Server-generated keys are returned to the client with the AsymmetricKeyPackage content type [RFC5958]. There MUST be only one OneAsymmetricKey present in the AsymmetricKeyPackage sequence and the public key SHOULD be included in the OneAsymmetricKey. The public key is provided by the server for convenience and for uniformity of message format because the client either compute the public from the private key or extract it from the certificate. If the client exchanges the public key, either in a certificate or the bare key, with another party it should check that the public key corresponds to the returned private key. If not, the client can discard the returned public key.

The AsymmetricKeyPackage is encapsulated in a CMS SignedData content type [RFC5652]; during the encapsulation process a digital signature is applied by the server. After being signed, the AsymmetricKeyPackage is cryptographically protected by encapsulating it in an EnvelopedData (i.e., the server-generated and signed private key is encrypted for the recipient). The resulting EnvelopedData is then included in a PKIResponse.cmsSequence and the entire PKIResponse is encapsulated in another SignedData. The Content Hints attribute [RFC2634] in the outer SignedData can provide a hint as to the inner most content type (i.e., the AsymmetricKeyPackage). Depending on where the key was generated the server can be either a CA or an RA.

When multiple keys are returned by the server, the server places each EnvelopedData in the cmsSequence.

6. Server-Side Key Generation

This section provides the control attributes necessary for doing server-side generation of keys for clients. The client places the request for the key generation in a request message and sends it to the server. The server will generate the key pair, create a

certificate for the public key and return the data in a response message, or the server will return a failure indication.

6.1. Server-Side Key Generation Request Attribute

The client initiates a request for server-side key generation by including the server-side key generation request attribute in the control attributes section of a PKIData object. The request attribute includes information about how to return the generated key as well as any client suggested items for the certificate. The control attribute for doing server-side key generation is identified by the following OID:

```
id-cmc-serverKeyGenRequest OBJECT IDENTIFIER ::= { id-cmc XX }
```

The Server-Side Key Generation Request control attribute has the following ASN.1 definition:

```
cmc-serverKeyGenRequest CMC-CONTROL ::= {
  ServerKeyGenRequest IDENTIFIED BY id-cmc-serverKeyGenRequest
  }

ServerKeyGenRequest ::= SEQUENCE {
  certificateRequest  TaggedRequest,
  shroudMethod        ShroudMethod,
  algCapabilities     SMimeCapabilities OPTIONAL,
  archiveKey          BOOLEAN DEFAULT TRUE
  }
```

The fields in ServerKeyGenRequest have the following meaning:

- certificateRequest contains the data fields that the client suggests for the certificate being requested for the server generated key pair. The format is TaggedRequest from [RFC5272], which supports both PKCS#10 and CRMF requests. In all instances, the bodyPartID is set to zero.

- shroudMethod contains the identifier of the type of algorithm to be used in deriving the key used to encrypt the private key.

- algCapabilities contains the set of algorithm capabilities being advertised by the client. The server uses algorithms from this set in the ServerKeyGenResponse object to encrypt the private key of the server-generated key pair. This field is optional because this information might be carried in a signed attribute, included within a certificate, or be part of the local configuration.

- archiveKey is set to TRUE if the client wishes the key to be

archived as well as generated on the server. Further processing by the server when this is set to TRUE is out-of-scope.

The client can request that the generated key be for a specific algorithm by placing data in the publicKey.attribute field of the CRMF request or in the subjectPKInfo.attribute field of the PKCS#10 request. If the publicKey or subjectPKInfo field is populated, then the subjectPublicKey is a zero-length bit string. If the client requests a specific algorithm, the server either generates a key for that algorithm (with the parameters if defined) or fails to process the request. If the request fails for this reason, the server returns a CMCFailInfo with a value of badAlg [RFC5272].

As specified in [RFC5272]:

> "A server is not required to use all of the values suggested by the client in the certificate template. Servers MUST be able to process all extensions defined in [RFC5280]. Servers are not required to be able to process other V3 X.509 extensions transmitted using this protocol, nor are they required to be able to process other, private extensions. Servers are permitted to modify client-requested extensions. Servers MUST NOT alter an extension so as to invalidate the original intent of a client-requested extension. (For example change key usage from key exchange to digital signature.) If a certification request is denied due to the inability to handle a requested extension, the server MUST respond with a CMCFailInfo with a value of unsupportedExt."

A server that does not recognize the algorithm identified in shroudMethod will reject the request. The server returns a CMCFailInfo with a value of badAlg [RFC5272].

A server that does not support at least one of the algCapabilities will reject the request. The server returns a CMCFailInfo with a value of badAlg [RFC5272].

If archiveKey is set to TRUE and the server does not support archiving of private keys, the request will be rejected by the server. The server returns a CMCFailInfo with a value of archiveNotSupported, see Section 7.

6.2. Server-side Key Generation Response

The server creates a server-side key generation response attribute for every key generation request made and successfully completed. The response message has a pointer to both the original request attribute and to the body part in the current message that holds the

encrypted private keys. The response message also can contain a pointer to the certificate issued. The key generation response control attribute is identified by the OID:

```
id-cmc-serverKeyGenResponse OBJECT IDENTIFIER ::= { id-cmc XX }
```

The Server-Side Key Generation Response control attribute has the following ASN.1 definition:

```
cmc-serverKeyGenResponse CMC-CONTROL ::= {
  ServerKeyGenResponse IDENTIFIED BY id-cmc-serverKeyGenResponse
  }

ServerKeyGenResponse ::= SEQUENCE {
  cmsBodyPartId          BodyPartID,
  requestBodyPartId      BodyPartID,
  signerIdentifier       SignerIdentifier
  }
```

The fields in ServerKeyGenResponse have the following meaning:

- cmsBodyPartId identifies a TaggedContentInfo contained within the enclosing PKIData. The ContentInfo object is of type EnvelopedData and has an encapsulated content of id-ct-KP-aKeyPackage, which is the OID for the Asymmetric Key Pacakge (see Section 5).

- requestBodyPartId contains the body part identifier of the server-side key generation request control attribute in the request message. This allows for clients to associate the resulting key and certificate with the original request.

- signerIdentifier refers the certificate issued to satisfy the request. The certificate, if present, is placed in the certificate bag of the immediately encapsulating SignedData object.

As specified in [RFC5272]:

> "Clients MUST NOT assume the certificates are in any order. Servers SHOULD include all intermediate certificates needed to form complete chains to one or more self-signed certificates, not just the newly issued certificate(s) in the certificate bag. The server MAY additionally return CRLs in the CRL bag. Servers MAY include self-signed certificates. Clients MUST NOT implicitly trust included self-signed certificate(s) merely due to its presence in the certificate bag."

7. Additional Error Codes

This section defines ExtendedFailInfo errors from this document. The ASN.1 is as follows:

```
id-cet-serverKeyGen OBJECT IDENTIFIER ::= { id-cet TBD }

cmc-err-keyGeneration EXTENDED-FAILURE-INFO ::= {
  TYPE ErrorList IDENTIFIED BY id-cet-serverKeyGen
  }

ErrorList ::= INTEGER {
  archiveNotSupported (1),
  badCertificate (2),
  badSharedSecret (3)
  }
```

The errors have the following meaning:

- archiveNotSupported indicates that the server does not support archiving of private keys.

- badCertificate indicates that the certificate to be used to encrypt the response did not validate back to an RA/CA trust anchor or the certificate does not belong to the client.

- badSharedSecret indicates that the shared secret used by the client does not match that stored by the server.

8. Proof-of-Possession

Some servers may require that the client prove that it has taken possession of the server-generated key. This proof requires an additional roundtrip beyond those previously discussed.

For certificates returned that support digital signatures the process is as described in [RFC5272]: the server indicates in CMCStatus that status is confirmRequired; the client returns the Confirm Certificate Acceptance control in PKIData signed with the server-generated private key; and the server responds with a CMCStatus of success.

For certificates returned that only support encryption, the server indicates the CMCStatus is popRequired and includes the Encrypted POP control; the client returns the Decrypted POP control. Whether the PKIRequest from the client is encapsulated in an AuthenticatedData or SignedData depends on which mechanism was used during the server key generation request.

9. Security Considerations

Central generation of digital signature keys contains risks and is not always appropriate. Organization-specific CPs (Certificate Policies) [RFC3647] define whether server-side generation of digital signature keys is permitted.

For the choice of mechanisms to protect the server-generated key, there is a balance that needs to be maintained between the use of a potentially poorly generated one-time key (i.e., the shared secret) and the use of a key externally provided. For externally provided keys, the external provider of the key will be able to decrypt the key delivery message as long as it was captured. For poorly generated one-time keys, any external party might be able to guess the key and thus decrypt the key delivery message. Different types of keys will have different requirements for what a poorly generated key means. Generators of RSA keys need to be able to do good prime checking; generators of Diffie-Hellman (DH) or Elliptic Curve Diffie-Hellman (ECDH) keys only need a moderate quality random number generator if the group parameters are externally provided and of good quality.

This specification requires implementations to generate key pairs and other random values [RFC4086]. The use of inadequate pseudo-random number generators (PRNGs) can result in little or no security. The generation of quality random numbers is difficult. NIST Special Publication 800-90 [SP-800-90] and FIPS 186 [FIPS-186] offer guidance.

Private keys, regardless of where they are generated, must be appropriately protected from disclosure or modification on the server, in transit, and on the client. Cryptographic algorithms and keys used to protect the private key should be at least as strong as the private key's intended strength.

This document describes the CA signing certificates and messages as well as encrypting messages. It should not be assumed that the CA uses the same key for all of these operations. In fact, CAs may wish to limit the exposure of their private keys by using different keys for different purposes.

Key agreement algorithms (i.e., Diffie-Hellan or Elliptic Curve Diffie-Hellman) can be used to protect the returned server-generated key. These algorithms support a number of different schemes [SP-800-56]. Normally, an Ephemeral-Static (E-S) scheme is used (more formally known as "(Cofactor) One-Pass Diffie-Hellman, C(1e,1s,ECCCDH) Scheme") see [RFC5753], but here the client provides an ephemeral key to the server so an S-E scheme is used when the key is encrypted for the client. Regardless, the client needs to generate an ephemeral key and provide it to the server and this key needs to use the same parameters (i.e., p, q, g for DH and elliptic

curve for ECDH) as the server. The client's parameters MUST be present in the publicKey or certificate field of the Server Key Generation Request control or MUST be in the certificate referred to by the ski in the same control. The client can find the server's parameters in the server's certificate; to make it easy on clients, server certificates MUST include parameters. How the client obtains the server's certificate is out-of-scope.

Servers that support the features specified herein need to document their procedures in a CPS (Certificate Practice Statement) [RFC3647]. CAs that certify server-generated private keys are certifying that they have taken due diligence to ensure that the private key is only known to and used by the subject. Depending on the Certification Policy (CP) [RFC3647], the keys have been allocated to the subject, but the keys may not be strictly owned by the subject. The CA (and the enterprise it supports) has a reason for issuing the keys (e.g., employer to employee; school to student) and because the enterprise CA generated the private keys it is accountable for the trustworthiness of the private key. But, the subject should beware of using it for other purposes.

When using an ephemeral key for protecting the server-generated key, a compromised signature key, when used by the intended party, will not automatically jeopardize the security of the server-generated keys. Procedural controls can help to ensure a one-to-one mapping between verified requests and intended parties (i.e. mitigate the risk of masquerade using a compromised authentication key and certificate), but that is outside the scope of this document.

POP is important; [RFC4211] provides some information about POP; for server-generated keys it can only be provided after the server-generated key has been returned by the client (see Section 8). Whether a server requires POP is CP dependent [RFC3647], but highly recommended.

When a shared secret is used to provide client authentication and protect the server-generated private key, the shared secret must be kept secret for the lifetime of the key or its use must be restricted to one-time use by the server. The rationale is that disclosure provides attackers access to the server-generated private key in the PKIResponse. This is different than certification requests with client-generated keys because the shared secret never protects the private key, so its loss does not comprise the private key.

If the key generator and the server are not collocated, then the exchange between these two entities must be protected from unauthorized disclosure and modification and both entities must have a trust relationship. However, these exchanges are beyond the scope

of this document. Note that the CA needs access to the public key to generate the certificate. If the key generator encrypts the generated key for the client, then the key generator needs to provide the public key to the CA (possibly through the RA).

Returning the key to the wrong client can be bad. If an encrypted key is returned to the wrong client, then it is only bad if the key was encrypted for the wrong client and then something much worse is afoot. If the encrypted key is returned to the wrong client and it is encrypted for the right client (i.e., it was misdirected), then it is bad but the unencrypted key has not been disclosed to an unauthorized client. The protection afforded by the confidentiality algorithm is what protects the misdirected key from unauthorized disclosure.

10. IANA Considerations

This document makes use of object identifiers; all object identifiers are defined in the PKIX arc described in [RFC7299]. IANA is requested to register the following OIDs:

1) In the SMI Security for PKIX Module Identifier arc:

Decimal	Description	References
TBD	id-mod-cmc-serverkeygen-2014-02	[This Document]

2) In the SMI Security for PKIX CMC Controls arc:

Decimal	Description	References
TBD	id-cmc-serverKeyGenRequest	[This Document]
TBD	id-cmc-serverKeyGenResponse	[This Document]

3) In the SMI Security for PKIX Algorithms arc:

Decimal	Description	References
TBD	id-alg-shroudWithPublicKey	[This Document]
TBD	id-alg-shroudWithSharedSecret	[This Document]

4) In the SMI Security for PKIX CMC Error Types arc:

Decimal	Description	References
TBD	id-cet-serverKeyGen	[This Document]

11. References

11.1 Normative References

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

[RFC2986] Nystrom, M. and B. Kaliski, "PKCS #10: Certification Request Syntax Specification Version 1.7", RFC 2986, November 2000.

[RFC4211] Schaad, J., "Internet X.509 Public Key Infrastructure Certificate Request Message Format (CRMF)", RFC 4211, 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.

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

[RFC5751] Ramsdell, B. and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification", RFC 5751, January 2010.

[RFC5958] Turner, S., "Asymmetric Key Packages", RFC 5958, August 2010.

[RFC6402] Schaad, J., "Certificate Management over CMS (CMC) Updates", RFC 6402, November 2011.

12.2 Informative References

[RFC2634] Hoffman, P., Ed., "Enhanced Security Services for S/MIME", RFC 2634, June 1999.

[RFC3647] Chokhani, S., Ford, W., Sabett, R., Merrill, C., and S. Wu, "Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practices Framework", RFC 3647, November 2003.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2", FYI 36, RFC 4949, August 2007.

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

[RFC5753] Turner, S. and D. Brown, "Use of Elliptic Curve Cryptography (ECC) Algorithms in Cryptographic Message Syntax (CMS)", RFC 5753, January 2010.

[RFC5754] Turner, S., "Using SHA2 Algorithms with Cryptographic Message Syntax", RFC 5754, January 2010.

[RFC5959] Turner, S., "Algorithms for Asymmetric Key Package Content Type", RFC 5959, August 2010.

[RFC7299] Housley, R., "Object Identifier Registry for the PKIX Working Group", RFC 7299, July 2014.

[FIPS-186] National Institute of Standards and Technology (NIST), FIPS 186-3 DRAFT: Digital Signature Standard (DSS), November 2008.

[SP-800-56] Barker, E., Johnson, D., and M. Smid, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", NIST Special Publication 800-56A Revision 1, March 2007.

[SP-800-57] National Institute of Standards and Technology (NIST), Special Publication 800-57: Recommendation for Key Management - Part 1 (Revised), March 2007.

[SP-800-90] National Institute of Standards and Technology (NIST), Special Publication 800-90: Recommendation for Random Number Generation Using Deterministic Random Number Bit Generators (Revised), March 2007.

Appendix A. ASN.1 Module

```
CMC-KeyGen
  { iso(1) identified-organization(3) dod(6) internet(1) security(5)
    mechanisms(5) pkix(7) id-mod(0)
    id-mod-cmc-serverkeygen-2014-02(TBD) }
DEFINITIONS ::=
BEGIN
    IMPORTS

AlgorithmIdentifier{}, SMIMECapabilities{}, ParamOptions, SMIME-CAPS
  FROM AlgorithmInformation-2009
    { iso(1) identified-organization(3) dod(6) internet(1)
      security(5) mechanisms(5) pkix(7) id-mod(0)
      id-mod-algorithmInformation-02(58) }

Certificate, SubjectPublicKeyInfo
  FROM PKIX1Explicit-2009
    { iso(1) identified-organization(3) dod(6) internet(1)
      security(5) mechanisms(5) pkix(7) id-mod(0)
      id-mod-pkix1-explicit-02(51) }

IssuerAndSerialNumber, SubjectKeyIdentifier, SignerIdentifier
  FROM CryptographicMessageSyntax-2010
    { iso(1) member-body(2) us(840) rsadsi(113549)
       pkcs(1) pkcs-9(9) smime(16) modules(0) id-mod-cms-2009(58) }

BodyPartID, EXTENDED-FAILURE-INFO, TaggedRequest, CMC-CONTROL, id-cmc
  FROM EnrollmentMessageSyntax-2011-v08
    { iso(1) identified-organization(3) dod(6) internet(1)
     security(5) mechanisms(5) pkix(7) id-mod(0)
     id-mod-enrollMsgSyntax-2011-08(76) }

SMimeCapsSet
  FROM SecureMimeMessageV3dot1-2009
    { iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-9(9)
      smime(16) modules(0) id-mod-msg-v3dot1-02(39) }
;

Keygen-controls CMC-CONTROL ::= {
  cmc-serverKeyGenRequest | cmc-serverKeyGenResponse, ...
}
```

```
SHROUD-ALGORITHM ::= CLASS {
  &id             OBJECT IDENTIFIER UNIQUE,
  &Params         OPTIONAL,
  &paramPresence      ParamOptions DEFAULT absent,
  &smimeCaps          SMIME-CAPS OPTIONAL
} WITH SYNTAX {
     IDENTIFIED BY &id
     [PARAMS [TYPE &Params] ARE &paramPresence]
     [SMIME-CAPS &smimeCaps]
}

ShroudAlgorithmSet SHROUD-ALGORITHM ::= {
  srda-shroudWithPublicKey | shrda-shroudWithSharedSecret, ... }

ShroudMethod ::= AlgorithmIdentifier {
  SHROUD-ALGORITHM, { ShroudAlgorithmSet }
}

id-alg OBJECT IDENTIFIER ::= { iso(1) identified-organization(3)
  dod(6) internet(1) security(5) mechanisms(5) pkix(7) alg(6)
}

id-alg-shroudWithPublicKey OBJECT IDENTIFIER ::= { id-alg TBD }

srda-shroudWithPublicKey SHROUD-ALGORITHM ::= {
  IDENTIFIED BY id-alg-shroudWithPublicKey
  PARAMS TYPE ShroudWithPublicKey ARE required
  SMIME-CAPS { IDENTIFIED BY id-alg-shroudWithPublicKey }
}

ShroudWithPublicKey ::= CHOICE {
  certificate         Certificate,
  certIdentifier  [1] SignerIdentifier,
  bareKey         [2] SEQUENCE {
    publicKey  SubjectPublicKeyInfo,
    ski        SubjectKeyIdentifier
    }
}

id-alg-shroudWithSharedSecret OBJECT IDENTIFIER ::= {id-alg TBD }

shrda-shroudWithSharedSecret SHROUD-ALGORITHM ::= {
  IDENTIFIED BY id-alg-shroudWithSharedSecret
  PARAMS TYPE ShroudWithSharedSecret ARE required
  SMIME-CAPS { IDENTIFIED BY id-alg-shroudWithSharedSecret }
}

ShroudWithSharedSecret ::= UTF8String
```

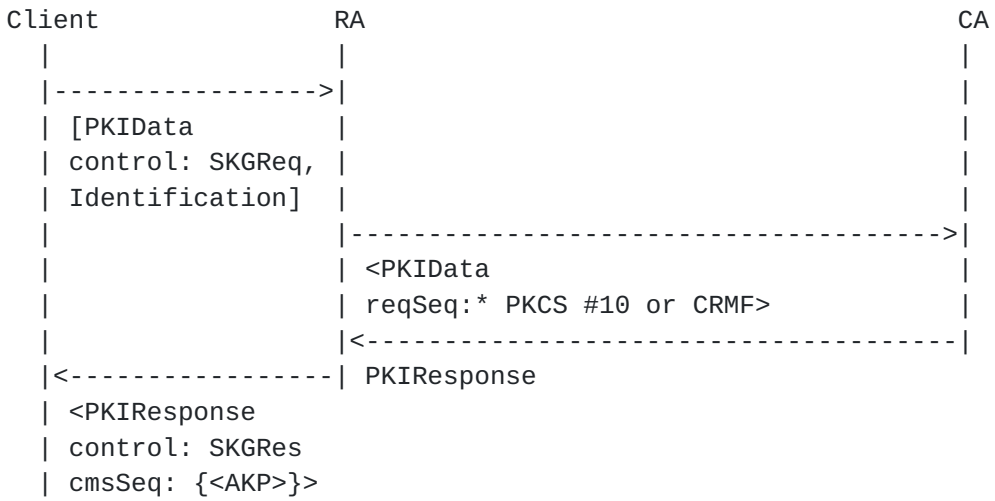
```
   id-cmc-serverKeyGenRequest OBJECT IDENTIFIER ::= { id-cmc TBD }

   cmc-serverKeyGenRequest CMC-CONTROL ::= {
     ServerKeyGenRequest IDENTIFIED BY id-cmc-serverKeyGenRequest
   }

   ServerKeyGenRequest ::= SEQUENCE {
     certificateRequest  TaggedRequest,
     shroudMethod        ShroudMethod,
     algCapabilities     SMimeCapabilities,
     archiveKey          BOOLEAN DEFAULT TRUE
   }

   SMimeCapabilities ::= SMIMECapabilities{{SMimeCapsSet}}

   id-cmc-serverKeyGenResponse OBJECT IDENTIFIER ::= { id-cmc TBD }

   cmc-serverKeyGenResponse CMC-CONTROL ::= {
     ServerKeyGenResponse IDENTIFIED BY id-cmc-serverKeyGenResponse
   }

   ServerKeyGenResponse ::= SEQUENCE {
     cmsBodyPartId          BodyPartID,
     requestBodyPartId      BodyPartID,
     signerIdentifier       SignerIdentifier
   }

   id-cet OBJECT IDENTIFIER ::= { iso(1) identified-organization(3)
     dod(6) internet(1) security(5) mechanisms(5) pkix(7) cet(15)
   }

   id-cet-serverKeyGen OBJECT IDENTIFIER ::= { id-cet TBD }

   cmc-err-keyGeneration EXTENDED-FAILURE-INFO ::= {
     TYPE ErrorList IDENTIFIED BY id-cet-serverKeyGen
   }

   ErrorList ::= INTEGER {
     archiveNotSupported (1),
     badCertificate (2),
     badSharedSecret (3)
   }

   END
```

Appendix B. Additional Message Flows

This appendix forms a non-normative part of this specification.

The main body of this document has portrayed protocol flows with optional controls. This was done to explain the more complicated scenarios. This appendix depicts the flows without those optional controls.

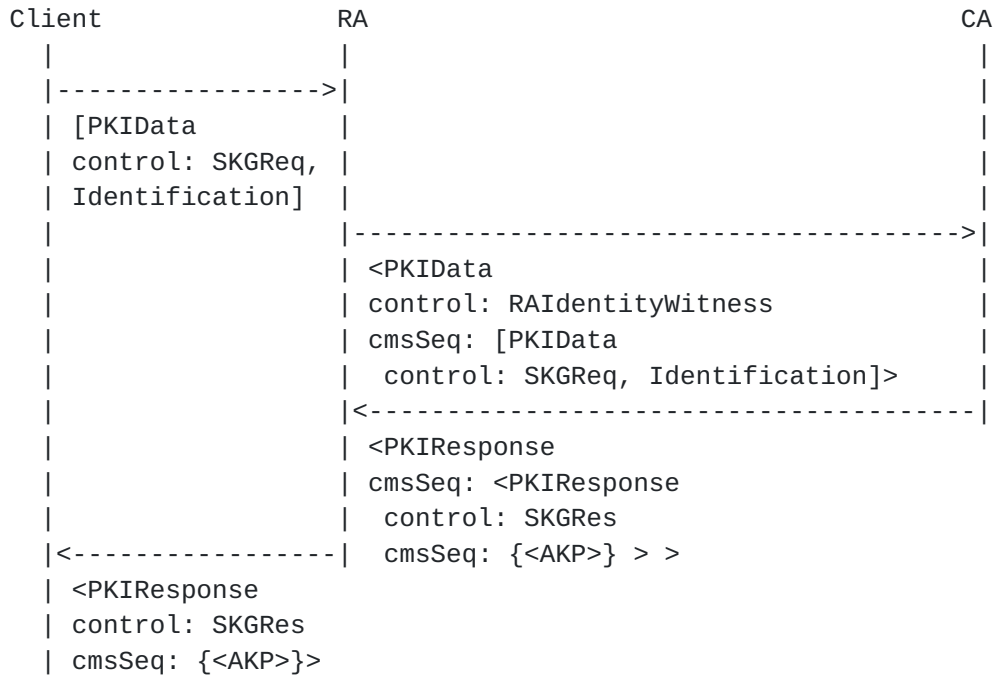
For example the figure in Section 2.5.1 without the TransactionId, SenderNonce, and RecipientNonce, appears as follows:



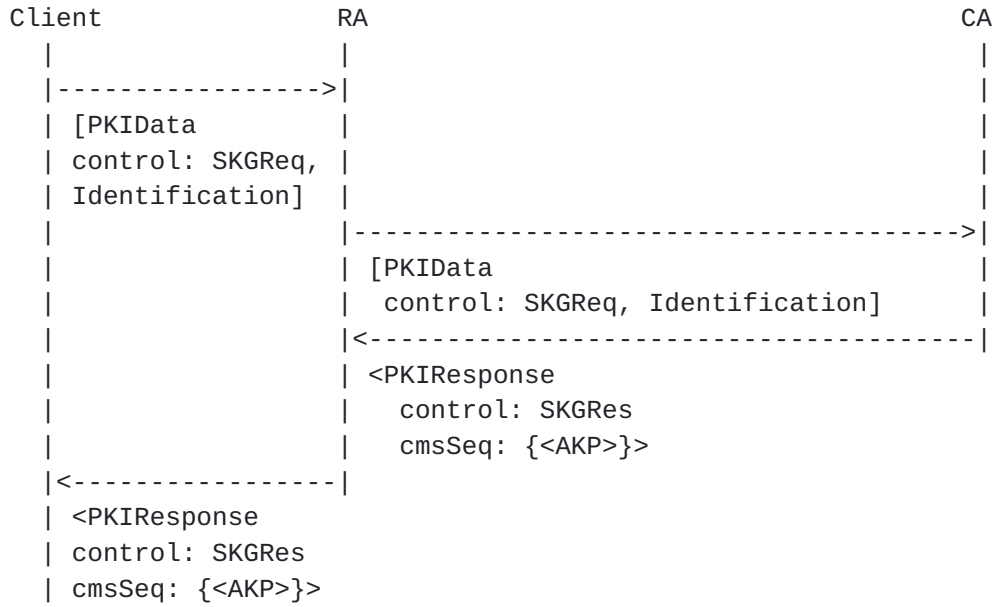
* Includes ChangeSubjectName attribute in PKCS #10 or CRMF.

The PKIResponse from the CA is a certs-only message, which does not include a signature.

Likewise for the figure in Section 2.5.2:

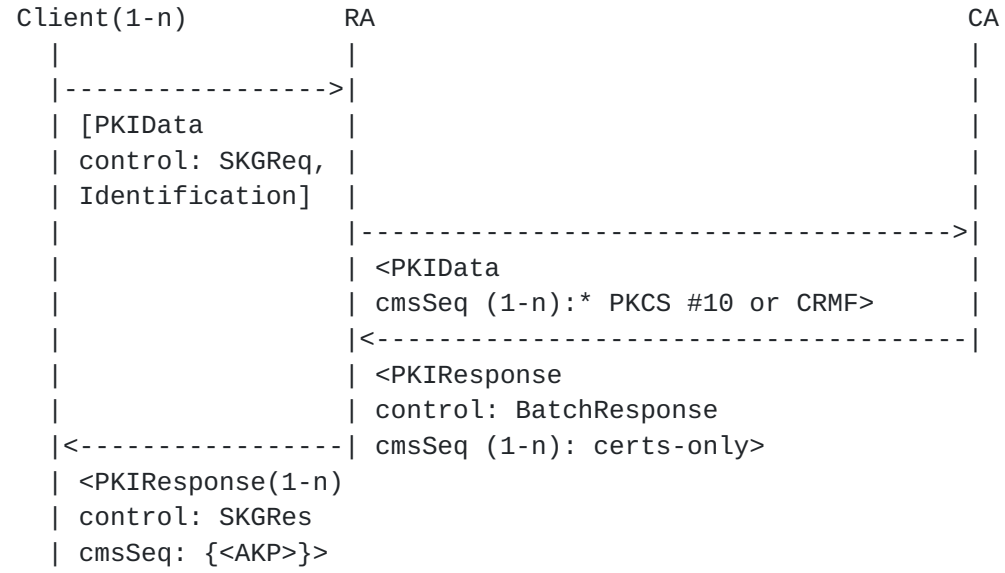


If the RA does not perform the Identity checks, then it can forward the client's request without the additional layers of encapsulation.



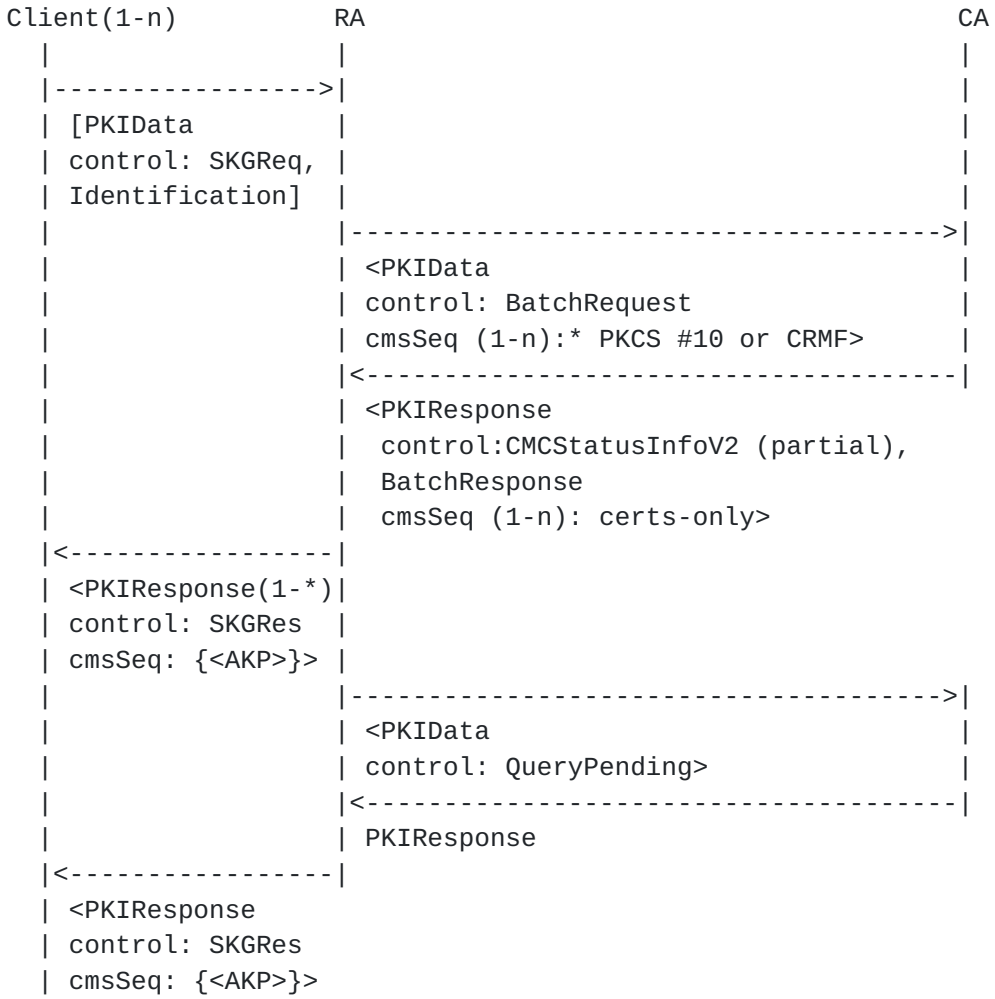
Not to be outdone, the scenarios in Section 2.5.1 can be more complicated by an RA that batches requests together. The following

depicts a number of clients sending requests together that an RA then batches together. The unsigned PKIResponse (a certs-only message) includes all of the certificates issued. The CA can also return individual responses as opposed to batching them all together or it can batch them together in some other combination.



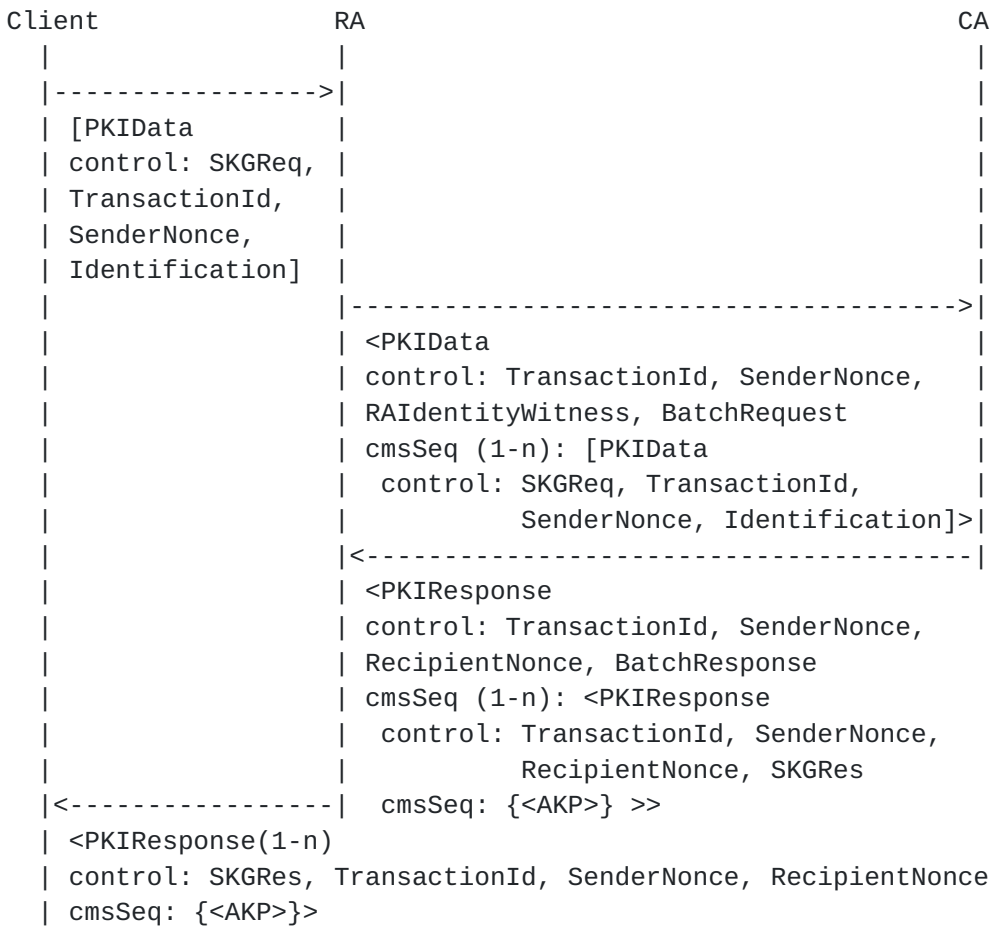
* Includes ChangeSubjectName attribute in PKCS #10 or CRMF.

In another scenario, all but one of the requests were successfully processed. The RA returns those that were successful back to the clients but later polls the CA, based on the value CMCStatusInfoV2 pendInfo, for the one that was not successful. The CA returns the one successful request.



* Includes ChangeSubjectName attribute in PKCS #10 or CRMF.

Batching the requests can also be performed for CA-generated keys as shown below. The RA Identity Witness controls indicates all those client requests that it performed Identity checks on.



Appendix B. Examples

To be supplied later.

B.1. Client Requests

B.1.1. Shroud with Certificate

B.1.2. Shroud with Public Key

B.1.3. Shroud with Shared Secret

B.2. CA-Generate Key Response

B.3. RA-Generate Key Response

Authors' Addresses

Jim Schaad
Soaring Hawk Consulting

Email: jimsch@exmsft.com

Sean Turner
IECA, Inc.
3057 Nutley Street, Suite 106
Fairfax, VA 22031
USA

Email: turners@ieca.com

Paul Timmel
National Information Assurance Research Laboratory
National Security Agency

Email: pstimme@tycho.ncsc.mil