# Part I
# System Design Considerations

## Overview

### Introduction

Part I summarizes system design considerations to be used in developing SET toolkits and applications. It provides background information and introduces the salient features and notation that will be used in subsequent parts of *SET Book 2: Programmer's Guide*.

### Organization

The following chapters are included:

| Chapter | Title | Contents | Page |
|---|---|---|---|
| 1 | Introduction | Provides background information and an overview of payment processing. | 6 |
| 2 | System Architecture | Provides an overview of the system architecture. | 37 |
| 3 | Technical Requirements | Summarizes other design considerations that affect the overall technical requirements for SET. | 52 |
| 4 | System Concepts | Summarizes other important system concepts pertinent to understanding the architecture of SET. | 84 |
| 5 | Processing | Provides a high-level overview of the step-by-step processing of common cryptographic treatments, as well as other common processing used by the payment and certificate management protocol descriptions in this Programmer's Guide. | 108 |

*Continued on next page*

## Overview, continued

**Definitive source for information**

The SET protocol is described in *SET Book 2: Programmer's Guide* and *SET Book 3: Formal Protocol Definition*. Because of the length of the documentation, it is possible that conflicts occur between the two books. In the event of conflicts, the following prioritized list should be used to determine which source is to be considered definitive (with items appearing first in the list being more definitive that items appearing later in the list):

Technical Bulletins published by SETCo
Book 3 Part II: ASN.1 Code
Book 3 Part I: Formal Protocol Definition
Book 2 Part I: System Design Considerations
Book 2 Part II: Certificate Management
Book 2 Part III: Payment System
Book 2 Appendices A, C, E, F, G, H, J, K, L, M, R
Book 2 Appendices T, U, V
Book 2 Appendices B, D, N, P, S

# Scope and audience

## Scope

This book is intended for readers who will be developing ~~software for cardholder and merchant systems. There are instances when requirements specific to the Payment Gateway and Certificate Authority systems are stated. These additional requirements, however, are informative and intended to assist the reader in understanding the processing performed by these systems.~~ applications that support electronic payment using the SET protocol as described in this specification; this includes Cardholder, Merchant, Payment Gateway, Acquirer, Issuer, and Certificate Authority software.

The processing steps in this book are requirements for these applications. Any additional processing performed by these applications, including processing related to SET messages, is outside the scope of this specification.

Specifically, the specification does not address:

- order management processing performed by merchants,
- the interface between the Payment Gateway and the existing financial system, or
- the mechanism for processing certificate requests, which depends on payment card brand and financial institution policy.

## Audience

It is assumed that the reader:

- will be developing applications that support electronic payment using the SET protocol as listed above;
- is familiar with the business requirements defined in *SET Book 1: Business Description*; and
- possesses a general understanding of cryptography and networking protocols.

# Brand and Acquirer requirements

**Brand-specific requirements**

Requirements that have been published by payment card brands can be found at *http://www.setco.org/*[brandname]. Among the requirements to be specified by the brands are the following:

- mapping of SET data elements to the brand's message formats;
- brand-specific rules for presence of optional fields;
- Certificate Authority (CA) functions, including:
  - whether to use Geopolitical CAs (GCAs), which allow brand policies to vary from one region to another as deemed necessary;
  - rules for generating Certificate Revocation Lists (CRLs), including frequency, validity periods, and the conditions under which an empty CRL may be required;
  - interval for generating Brand CRL Identifier (BCI);
  - whether a GCA or Payment Gateway CA (PCA) will handle CRL and BCI distribution on behalf of the Brand CA (BCA);
- brand policies for issuance of certificates, including:
  - whether a Cardholder certificate is required;
  - verification requirements for certificate request data;
  - for renewals, whether identification and authentication may be based on the use of the previous certificate;
- certificate contents:
  - both *brand name* and *product* in *BrandID*;
  - choices available for Descriptive Name; and
  - restrictions on validity periods;
- content and aging requirements for Cardholder, Merchant, and Payment Gateway transaction logs;
- when a party has a right to deny participation in a SET transaction;
- requirements for use of hardware tokens.

**Acquirer-specific requirements**

Acquirers have specific requirements for Merchants interfacing with their Payment Gateway. These requirements may be changed or expanded to accommodate SET. It is the responsibility of the Merchant to determine what these requirements are.

# Terminology

## Terminology

Throughout this document:

| | |
|---|---|
| SET application | Describes any software that supports electronic payment using the SET protocol. This includes Cardholder, Merchant, Payment Gateway, Acquirer, Issuer, and Certificate Authority software. |
| payment card | Refers to any of the following: credit card, debit card, charge card, and bank card. |
| shall | Indicates a requirement that is imposed by SET (see also "Processing steps" below). |
| will | Indicates ~~either a goal or an implicit requirement that is imposed~~ something you can depend on that is external to SET. |
| should | Indicates a recommended course of action. |
| must | Indicates a requirement that is imposed external to SET, such as by export requirements. |
| validate | In processing steps, means to compare the values that follow to ensure that they match. |
| end entity | Cardholder, Merchant, or Payment Gateway |
| input | The first step of most SET processing sequences described in this book lists the input to the processing sequence. With rare exceptions, the DER-encoded representation of the data is intended. That is, the processing sequences will not remind you to DER encode the data. |

**Table 1: Terminology**

## Processing steps

In processing steps, "shall" is normally implicit. That is, unless otherwise indicated, an instruction such as "verify x" is equivalent to "the application shall verify x."

The sequence of processing steps may be varied as long as the results are the same. In the event that more than one error condition applies to a given message, the **Error** message generated may report any one of the errors, at the discretion of the application.

# Chapter 1
# Introduction

## Overview

### Introduction

Chapter 1 provides background information and an overview of payment processing.

### Organization

Chapter 1 includes the following sections:

| Section | Title | Contents | Page |
|---|---|---|---|
| 1 | Background | Provides background information with emphasis on the scope of SET. | 7 |
| 2 | ~~Environment~~ Processing Overview | Describes the environment for processing payment card transactions using SET. | 12 |
| 3 | Business Flows | Provides a high-level description of typical business flows relevant to SET. | 16 |
| 4 | Capture Processing | Provides an overview of capture processing, including batch processing. | 29 |

# Section 1
# Background

## Overview

### Scope

The scope of SET ~~this document~~ is limited to the payment process and the security services necessary to support the payment aspects of the electronic shopping experience. To provide these services, SET defines not only the electronic payment protocol, but also the certificate management process.

### SET entities

The SET system is composed of a collection of entities involved in electronic commerce. These entities are listed in Table 2.

| Entity | SET definition |
|---|---|
| **Cardholder** | An authorized user of a payment card supported by an Issuer, and registered to perform electronic commerce; also, the software that processes SET transactions for a cardholder. |
| **Merchant** | A party that provides goods, services, and/or information, accepts payment for them electronically, and may provide selling services and/or electronic delivery of items such as information; also, the software that processes SET transactions for a merchant. |
| **Issuer** | A financial institution that supports issuing payment card products to individuals. |
| **Acquirer** | A financial institution that supports merchants by providing service for processing payment card transactions. |
| **Payment Gateway** | A system that provides electronic commerce services to merchants in support of an Acquirer, and interfaces with the Acquirer to support the authorization and capture of transactions. |
| **Brand** | A franchiser of payment systems and/or instruments. |
| **Certificate Authority (CA)** | An agent of one or more payment card brands that provides for the creation and distribution of electronic certificates for cardholders, merchants, and Payment Gateways. |
| **Payment card brand's financial network** | The existing private network operated by a payment card brand that links Acquirers and Issuers. |

**Table 2: SET Entities**

The responsibilities of these entities (other than Brands) are further described in "Architecture" on page 38.

*Continued on next page*

# Overview, continued

**End entities**

Three of the entities – Cardholder, Merchant, and Payment Gateway – are designated as end entities, sometimes abbreviated EE.

**Entity interaction**

Figure 1 depicts the interaction of the SET entities via SET messages.

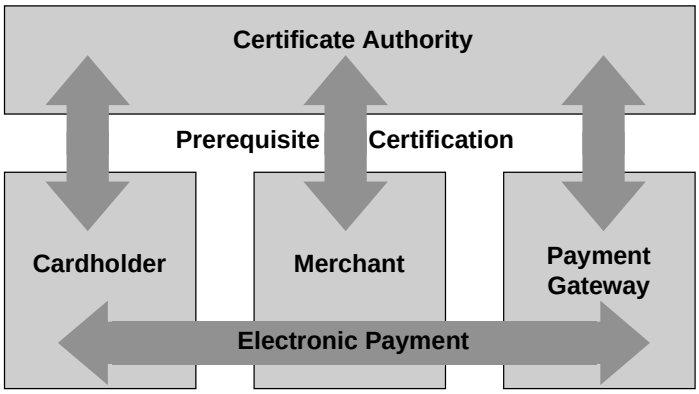


**Figure 1: Entity Interaction via SET Messages**

# Electronic Shopping

### Opportunities and challenges

The electronic commerce environment will provide new opportunities for merchants to conduct business due to increased exposure and increased access by consumers to information about their products and services. Consumers will be able to shop, access information, and pay for goods and services electronically. Greater convenience is likely to lead both to more purchases and to greater use of payment cards.

A standard protocol for electronic commerce has distinct political advantages as well. Existing solutions are either domain-specific or country-limited. Because SET clearly defines the application use of cryptography, more governments can exclude it from current export/import restrictions, allowing a large, standard distribution of one payment protocol.

With these new opportunities, new challenges will need to be addressed in order to facilitate secure payment processing for electronic shopping.

Currently, shoppers hesitate to send their account number and expiration dates over electronic networks. They are concerned that:

- their transmissions may be intercepted and read by unauthorized parties;
- fraudulent charges will appear on their statements; and
- people pretending to be merchants will accept their orders, but never deliver the products or services purchased.

Merchants and financial institutions are concerned that:

- electronic fraud will significantly increase the cost of processing transactions, and
- a maze of software will be developed to prevent fraud and will not support or interact with their current payment systems.

### Phases

Electronic shopping will typically include the phases shown in Figure 2. The order of the phases is determined by the implementation.

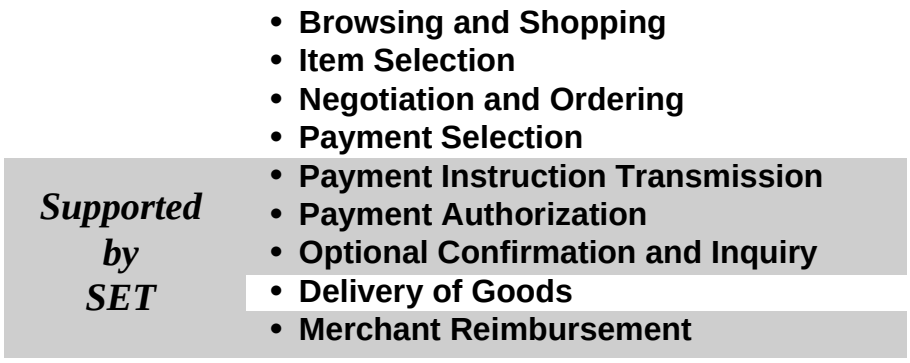


**Figure 2: Phases of Electronic Shopping**

*Continued on next page*

# Electronic Shopping, continued

## Processing

Table 3 describes the phases of the electronic shopping model. Interactions between the customer and the merchant can occur in either an interactive environment, such as the World Wide Web, or through non-interactive means such as electronic ~~or postal~~ mail exchanges. SET pertains to the phases that are shaded, and only in those instances in which a payment card is selected as the means of payment.

| Phase | Description |
|---|---|
| 1 | The cardholder browses for items – tangible goods, electronic media (for example, information, software, etc.), or services – described in a variety of media, such as:<br>• an on-line catalog on a merchant's World Wide Web page;<br>• a CD-ROM catalog supplied by the merchant; or<br>• a paper catalog. |
| 2 | The cardholder selects items to be purchased from a merchant. |
| 3 | The cardholder is presented with an order form containing the list of items, their prices, and a total price including shipping, handling, and taxes.<br>This order form may be delivered electronically from the merchant's server or created on the cardholder's computer by electronic shopping software.<br>Note: Some on-line merchants may also support the ability for a cardholder to negotiate for the price of items (such as by presenting frequent shopper identification or information about a competitor's pricing). |
| 4 | The cardholder selects the means of payment, including:<br>• a payment instrument (such as a specific payment card),<br>• a payment mechanism (such as SET), and<br>• in some cases, additional information - for example, to define installment payments.<br>Although SET processing normally begins after the means of payment has been selected, a SET implementation may include payment selection. |
| 5 | The Cardholder software sends the merchant a completed order along with a means of payment.<br>In SET, the order and the payment instructions are digitally signed by those cardholders who possess certificates. |
| 6 | The merchant checks inventory to determine if the goods and services ordered by the customer are in stock or need to be backordered. If only part of the order is currently in stock, the merchant may decide to handle the order as a split shipment. |

**Table 3: Phases of Electronic Shopping**

*Continued on next page*

# Electronic Shopping, continued

**Processing** (continued)

| Phase | Description |
|---|---|
| 7 | The merchant requests payment authorization. In SET, the authorization is obtained from the cardholder's financial institution via the Payment Gateway. The response includes an indication of whether the authorization request has been approved or declined. (The request for payment shown in Step 10 may be combined with this step.) |
| 8 | Either, both, or neither of these may occur:<br>• If authorization succeeds, the Merchant may send confirmation of the order ~~out of band to SET~~.<br>• The cardholder may query the status of the order. |
| 9 | The merchant delivers the goods or performs the services ordered. The delay between authorization and shipment (which shall must precede capture) can legitimately be several days. If goods are not available for immediate delivery, the shipment is held up until the order can be fulfilled. |
| 10 | The merchant ~~requests~~ submits a capture request to the Acquirer in order to obtain payment. For transactions authorized using SET, the Merchant may request payment from the cardholder's financial institution:<br>• using SET, via the Payment Gateway; or<br>• using existing connections to the Acquirer.<br>(This step may be combined with Step 7, the request for payment authorization.) |
| 11 | Funds are transferred from the shopper's payment card account to the merchant's account. |
| 12 | If a credit is to be issued to a customer, such as when the goods are returned or defective, the merchant sends a message to the Acquirer requesting that a credit be issued to the cardholder's account. For transactions authorized using SET, the Merchant may request the credit:<br>• using SET, via the Payment Gateway; or<br>• using existing connections to the Acquirer. |

**Table 3: Phases of Electronic Shopping,** continued

# Section 2
# ~~Environment~~ Processing Overview

## Introduction

**Similarity to mail order/ telephone order**

Before the advent of electronic commerce, payment card transactions typically followed one of two patterns:

| | |
|---|---|
| card present | Customer physically presents a payment card to the merchant.<br>Electronic processing of the payment begins with the Merchant or the Acquirer. |
| Mail Order/Telephone Order (MOTO) | Order and payment information is transmitted to the merchant either by mail or by telephone.<br>Electronic processing of the payment begins with the Merchant or the Acquirer. |

The processing of transactions using SET generally follows that of the MOTO environment except that:

| | |
|---|---|
| SET | Order and payment information is transmitted electronically.<br>Electronic processing of the payment begins with the Cardholder rather than the Merchant or the Acquirer. |

**Table 4: Comparison of Payment Card Environments**

*Continued on next page*

# **Introduction,** continued

**SET and MOTO** Figure 3 illustrates how SET and MOTO complement one another.

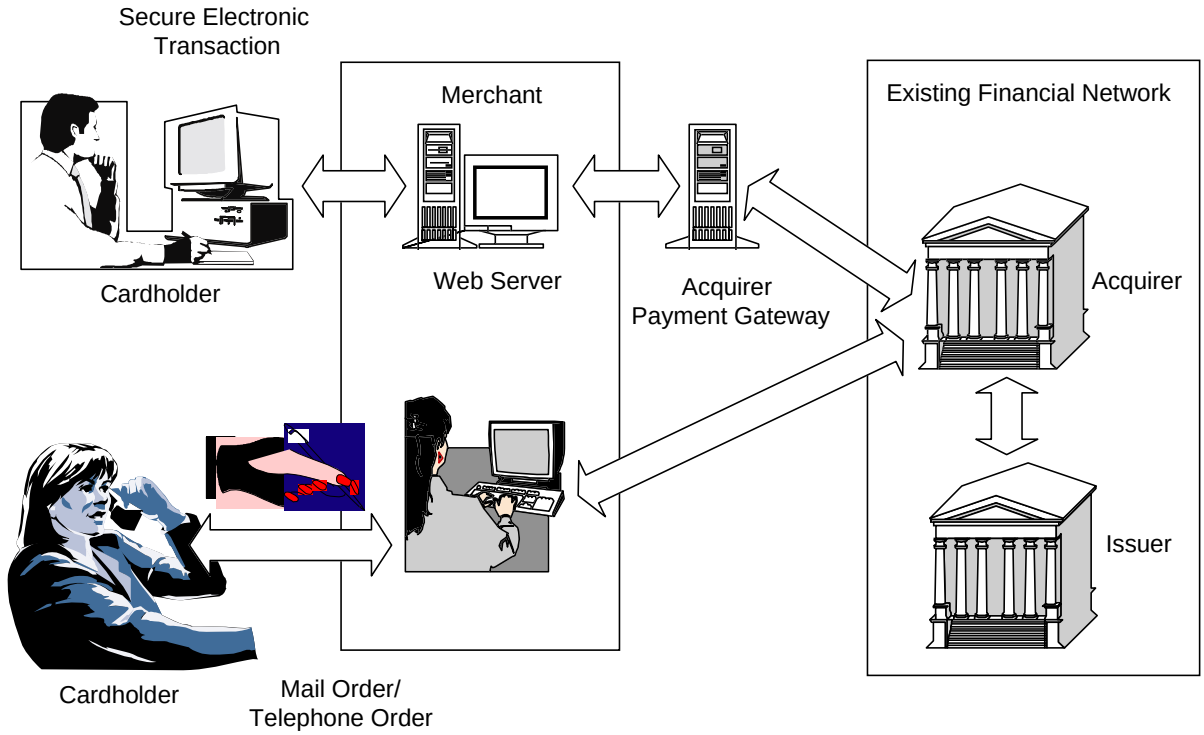


**Figure 3: SET / MOTO Comparison**

*Continued on next page*

# Introduction, continued

## Processing

Table 5 is a description of a simplified MOTO processing model. There are many variations on this processing model, but the table represents a typical exchange. See Table 3 on page 10 for a similar description of electronic processing.

| Phase | Description |
|---|---|
| 1 | The shopper (cardholder) receives a list of the goods and services offered by a merchant, often in a paper catalog or other direct marketing mailing. |
| 2 | The cardholder selects items to be purchased from this list. |
| 3 | The cardholder either:<br>• prepares an order form, including the means of payment, and sends it to the merchant, or<br>• provides the order and payment information to the merchant by telephone. |
| 4 | The merchant checks inventory to determine if the goods and services ordered by the customer are in stock or need to be backordered. If only part of the order is currently in stock, the merchant may decide to handle the order as a split shipment. |
| 5 | The merchant sends an authorization request to its financial institution (Acquirer). The Acquirer incorporates the authorization data into a request that is sent via a payment network for processing by the financial institution (Issuer) that issued the payment card to the cardholder. (An Acquirer may allow a merchant to combine the authorization message with the capture message shown in Step 9.) |
| 6 | The Issuer responds to the Acquirer via the payment card network with an authorization response. The response includes an indication of whether the authorization request has been approved or declined. The Acquirer responds to the merchant with the outcome. |
| 7 | The cardholder may query the status of the order. |

*Continued on next page*

# Introduction, continued

**Processing** (continued)

| Phase | Description |
|---|---|
| 8 | The merchant delivers the goods or performs the services ordered. The delay between authorization and shipment (which ~~shall~~ must precede capture) can legitimately be several days. Many MOTO merchants are not able to check inventory before authorization. If goods are not available for immediate delivery, the shipment is held up until the order can be fulfilled. |
| 9 | The merchant submits a capture request to the Acquirer in order to obtain payment. This request is sent through the payment card network to the Issuer. (An Acquirer may allow a merchant to combine the capture message with the authorization message shown in Step 5.) |
| 10 | Funds are transferred from the shopper's payment card account to the merchant's account. |
| 11 | If a credit is to be issued to a customer, such as when the goods are returned or defective, the merchant sends a message to the Acquirer requesting that a credit be issued to the cardholder's account. |

**Table 5: Phases of MOTO Shopping**

# Section 3
# Business Flows

## Overview

**Purpose**

This section provides a high-level description of typical business flows relevant to SET.

**Introduction**

A SET purchasing transaction generally follows the MOTO process described in Section 2. The purpose of SET is to allow a similar exchange to take place electronically in a manner that ensures the security of the shopper's payment card account information.

There are a number of different ways a purchase transaction can progress. This section describes several business flow variations.

**Transaction variations**

The purchase transaction may vary depending on the shopper's preferences and the merchant's business situation. For example:

- The shopper may want to pay in installments.
- The order may be for tangible goods and the merchant may be out of stock on one or more of the items ordered, but able to ship the rest.
- The order may be for non-tangible goods, such as a video clip that can be delivered electronically – in which case the merchant can immediately process both the authorization and the capture request.

# Basic Business Functions

**Basic business functions**

SET is designed to support all these basic functions.

| | |
|---|---|
| Payment Instructions **(PI)** | The shopper, usually while filling out the order form, indicates how payment is to be made. Typically this will be by specifying a payment card brand and account number with expiration date. The SET protocol allows the shopper's payment card account number and expiration date to be encrypted and included with the order automatically.<br>Payment Instructions may include some variations. SET allows the shopper to request recurring and installment payments, if they are offered by the merchant. |
| Authorization Request **(AuthReq)** | Before a merchant fills a payment card order, payment must be authorized by the Issuer of the payment card. The Issuer verifies that the account number is valid and that the purchase is within the credit limit or available funds of the account. |
| Payment Capture **(CapReq)** | Once the goods are shipped or the services are performed, the Merchant sends a request to be paid from the shopper's account. Payment capture can be handled in a number of ways. Sometimes a capture request is sent along with the authorization request. Often merchants send several requests at the same time; this is known as batch processing and is described further on page 32. |
| Subsequent Authorization **(subsequentAuthInd)** | If a transaction cannot be completed as authorized – for example, if part of the order is out of stock – a subsequent authorization indicator is used to tell the system that there is a business need for another authorization using the same account information as is contained in the original **AuthReq**. In this case the Payment Gateway will return an **AuthToken** that may be used when requesting authorization for the remaining parts of the order. |
| Authorization Reversal **(AuthRevReq)** | If a mistake is made in the **AuthReq** message or the amount of the authorization needs to be changed – for example if some of the goods ordered need to be back-ordered – the Merchant software may send an authorization reversal for all or part of the original authorization. |
| Credit Request **(CredReq)** | If the shopper cancels the order or returns the goods to the merchant, the Merchant software sends a credit request so that a credit may be posted to the shopper's account. |

**Table 6: Basic Business Functions**

# Transaction Identification Alternatives

**Alternatives to account numbers**

SET Cardholder payment system messages are sent to the Merchant but include the Cardholder account number encrypted in such a way that the Merchant cannot read it. The Merchant passes the encrypted data to the Payment Gateway, which decrypts it to determine the account number, so that the account number can be used in transactions sent to non-SET systems (for example, for clearing).

When issuing Merchant certificates, the Acquirer sets a flag that indicates whether the Merchant may receive the cardholder account number as part of a response. If the Acquirer does not return the account number, it needs to ensure that the Merchant has an alternative means of identifying the transaction for business processing of non-SET messages from the Issuer (such as a request for copy or charge back).

Note: These messages may be identified by the Issuer using the Acquirer Reference Number assigned by the Acquirer, which has no equivalent in SET.

Table 7 lists a number of fields other than the account number that can be used for transaction identification. Each implementation will determine the field(s) to use.

| | Field | Included in data structure | Definition |
|---|---|---|---|
| unique by transaction | **xid** | **TransIDs** **MessageWrapper** **TransStain** | 20-byte number that uniquely identifies the transaction, including all authorization~~, and clearing~~ capture, credit, and reversal messages for a single order |
| | **lid-M** | **TransIDs** **MessageWrapper** (also **PInitReq**) | 1- to 20-byte local identifier assigned to the transaction by the Merchant software.<br>Depending on the implementation, this may be a tracking number assigned by staff operating the system or an internal number used solely by the Merchant software. |
| | **merOrderNum** | **SaleDetail** | 1- to 25-byte merchant order number |
| unique by authorization | **paySysID** | **TransIDs** (optional) | 1- to 64-byte payment system transaction identifier |
| | **authRRPID** | **RRTags**, among others | A statistically unique 20-byte number that ~~uniquely~~ identifies ~~a request/response pair —~~ a single authorization ~~or clearing message~~ |
| unique by request/response pair | **rrpid** | **MessageWrapper** and **RRTags**, among others | A statistically unique 20-byte number that ~~uniquely~~ identifies a request/response pair ~~— a single authorization or clearing message~~ |

**Table 7: Transaction Identification Data**

*Continued on next page*

## Transaction Identification Alternatives, continued

**Not unique by transaction**

**PaySysID** and **AuthRRPID** are unique by authorization request. Therefore for a transaction with split shipments or recurring payments, these fields will have multiple values.

**RRPID** is unique by request/response pair. Therefore a transaction will have many **RRPID**s – one for each authorization, capture, credit, reversal, etc.

# Typical Business Scenarios

## Overview

This section illustrates a range of typical business scenarios that are enabled by SET processing according to the specific circumstances of a purchase.

Overall business flows are illustrated in Figure 4 on page 22. After that, descriptions of the following scenarios are included:

- Authorize now and capture later (the most typical scenario)
- Authorize and capture now: Sale Request
- Split shipment
- Installment and recurring payments
- Credit for an old transaction

*Continued on next page*

# Typical Business Scenarios, continued

## Business flows

Figure 4 is a state diagram that illustrates SET business flow messages. It shows, at a high level, the transitions from shopping to ordering and processing of the order, with the processed state shown in two variations:

- *sale processed* in the case of an order that is authorized and captured at the same time, and
- *captured* for an order that is authorized now and captured later.

The figure also shows the processing of a credit from both of these states.

The message pairs are implicit in this diagram; for example, **AuthReq** represents both the authorization request and the response message.

In this scenario, there are transitions from one state to another – for example, from the *ordered* state to the *sale processed* state. Once the **PReq** message is processed, any transition that follows it can be reversed, with the effect of returning to the previous state. For example, when receiving an order, the merchant submits an authorization request; a subsequent authorization reversal request would take the transaction back to the *ordered* state. There is one exception: A partial authorization reversal (to specify a new amount) leaves the transaction in the *authorized* state.

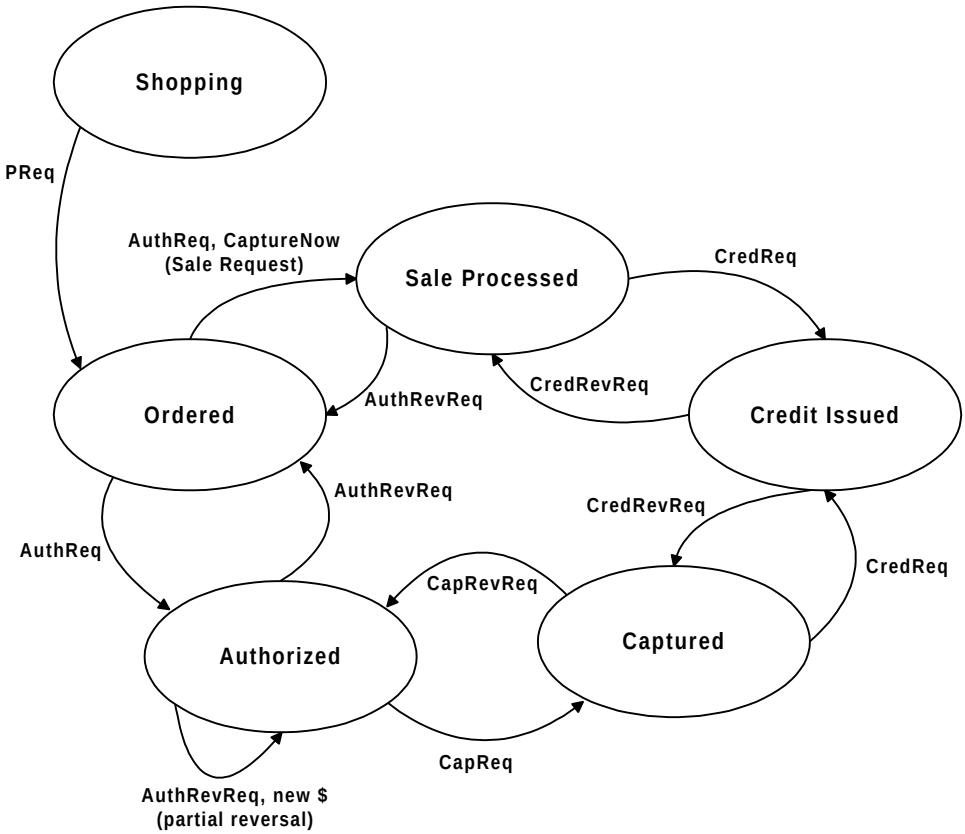


**Figure 4: Business Flows**

*Continued on next page*

# Typical Business Scenarios, continued

### Authorize now and capture later

The most typical on-line purchase is one in which the merchant is ready to authorize the transaction now, but wants to submit the capture request later. For example, many merchants prefer to submit their capture requests in batches at the end of the business day.

| | |
|---|---|
| Purchase Request **(PReq)** | After the shopper creates an order, the Cardholder software sends the Purchase Request (**PReq**) to the Merchant. This message and its response encompass the actual payment between the Cardholder and the Merchant, and take the cardholder from the *shopping* state to the *ordered* state. The **PReq** includes:<br>• the Order Instruction (OI) from the Cardholder for the Merchant, and<br>• the Payment Instruction (PI) from the Cardholder, encrypted and tunneled through the Merchant to the Payment Gateway. |
| Purchase Response **(PRes)** | The Merchant may sent the Purchase Response to the Cardholder immediately or at any time later in the protocol. The information returned will depend on the processing that has occurred when the **PReq** is returned – for example, order received, transaction authorized, or transaction captured. |
| Authorization Request **(AuthReq)** | The Merchant sends an Authorization Request to the Payment Gateway, but does not set the **CaptureNow** flag to TRUE, as a capture request will be processed later. The **AuthReq** indicates whether the merchant expects to do another authorization for a split shipment, recurring payment, or installment payment (discussed later in this section). |
| Authorization Reversal Request **(AuthRevReq)** | If a full authorization reversal is needed, it will return the transaction to the *ordered* state.<br>A partial authorization reversal may be used to change the amount after the authorization, leaving the transaction in the *authorized* state. For example, the amount might be changed if the merchant checks inventory and finds the entire order cannot be shipped together.<br>Note: Some payment brands do not support partial authorization reversals; in this case, the Payment Gateway indicates "success", but does not actually send a message to the financial network.<br>*(table continues)* |

**Table 8: Authorize Now and Capture Later**

*Continued on next page*

# Typical Business Scenarios, continued

**Authorize now and capture later** (continued)

| | |
|---|---|
| Capture Request **(CapReq)** | The merchant now has a commitment for payment from the Issuer, but will need to process the capture request in order to be paid. The capture request may include multiple capture items. It includes a capture token if one is provided in the authorization response. |
| Credit Request **(CredReq)** | Later, the cardholder may request (or the merchant may decide to issue) a credit for the order – for example, if the cardholder returns the order because it was damaged in shipment. In this case a credit request is processed, moving the transaction from the *sale processed* state to the *credit issued* state.<br>Unlike a capture reversal, a credit request ~~is processed after an order is completed and shipped, and~~ results in a credit on the cardholder's statement. |

**Table 8: Authorize Now and Capture Later,** continued

*Continued on next page*

# Typical Business Scenarios, continued

### Authorize and capture now: Sale Request

A Sale Request is used:

- when the Merchant knows the item ordered is in stock and can be shipped right away, if the authorization is approved; or
- for purchase of non-tangible goods available electronically – such as video clips, encyclopedia pages, and software programs – for which there is no question of inventory, so the order can be fulfilled immediately.

| | |
|---|---|
| Purchase Request and Response **(PReq** and **PRes)** | As in Table 8 on page 23. |
| Authorization Request **(AuthReq)** | SET allows a merchant to process the transaction as a single message by setting the **CaptureNow** flag in the Authorization Request to TRUE. This indicates that if the transaction is authorized, the capture should be done now, as well. In effect, it is a combined authorization and capture ~~clearing~~. |
| Authorization Response **(AuthRes)** | When the Payment Gateway processes the request, there is a transition to the *sale processed* state. From a financial perspective, the *sale processed* state is equivalent to the *captured* state discussed on page 22. |
| Authorization Reversal Request **(AuthRevReq)** | If the amount of the transaction is in error, a full authorization reversal with the **CaptureNow** flag set is performed. Unlike the authorize now and capture later scenario, there are no partial reversals. |
| No Capture Request (**CapReq**) is submitted, as the capture was accomplished in the **AuthReq**. | |
| Credit Request **(CredReq)** | As in Table 8 on page 23. |

**Table 9: Authorize and Capture Now: Sale Request**

*Continued on next page*

# Typical Business Scenarios, continued

## Split shipment

When the merchant cannot fulfill the entire order, the items in stock are shipped and the remaining items are back-ordered. Processing varies depending on whether the need for the split shipment is known at the time of authorization, as described in Table 10.

In either case, an Authorization Token (**AuthToken**) is used to enable subsequent authorizations. **AuthToken** serves the same purpose as the Payment Instruction, except that it originates with the Payment Gateway and is a means of allowing one additional authorization.

| | *If the need for a split shipment is known at the time of authorization – for example, when inventory information is available:* | *If the need for a split shipment is determined after the initial authorization:* |
|---|---|---|
| Purchase Request and Response **(PReq** and **PRes)** | As in Table 8 on page 23. | |
| Authorization Request **(AuthReq)** | The Merchant sets **SubsequentAuthInd** in the initial Authorization Request to indicate a business need for a subsequent authorization. | The Merchant submits a normal Authorization Request:<br>• without **SubsequentAuthInd** (since the need for a subsequent authorization is not known), and<br>• without **CaptureNow** (which, as described on page 25, is used only when the Merchant is sure there will be no need for a split shipment). |
| Authorization Response **(AuthRes)** | The Payment Gateway returns an Authorization Token (**AuthToken**) in the Authorization Response. | The Payment Gateway returns a normal Authorization Response (without **AuthToken**). |
| Authorization Reversal Request **(AuthRevReq)** | | Once the need for a split shipment is determined, the Merchant sends a partial Authorization Reversal Request~~; it includes a capture token to be used for the subsequent authorization and capture~~ with **SubsequentAuthInd** set. |
| Authorization Reversal Response **(AuthRevRes)** | | The Payment Gateway returns an **AuthToken** in the Authorization Reversal Response. |

*Continued on next page*

# Typical Business Scenarios, continued

**Split shipment** (continued)

| | *If the need for a split shipment is known at the time of authorization – for example, when inventory information is available:* | *If the need for a split shipment is determined after the initial authorization:* |
|---|---|---|
| Authorization Request **(AuthReq)** | When the remainder of the order is ready, the Merchant submits another **AuthReq** including the **AuthToken** returned in the **AuthRes** or **AuthRevRes**.<br>If the order must be further split, the Merchant sets the **SubsequentAuthInd** in the new **AuthReq** to obtain an **AuthToken** for one additional authorization. This process can be repeated as many times as necessary. A new **AuthToken** is required for each subsequent authorization. | |
| Capture Request **(CapReq)** | The capture request for each partial shipment is processed normally. | |
| Credit Request **(CredReq)** | Again, if there is a need to return money to the cardholder, a credit request moves the transaction into the *credit issued* state. | |

**Table 10: Split Shipment**

*Continued on next page*

# Typical Business Scenarios, continued

### Installment and recurring payments

The merchant may offer customers the option of paying in installments – for example, three monthly payments. Or, the merchant may offer to process payments on a regular basis – for example, an Internet service provider may offer to bill the cardholder's account for the monthly service charge with no action needed by the cardholder.

| | |
|---|---|
| Purchase Request **(PReq)** | The Merchant presents the installment or recurring payment option, which is then indicated by the cardholder in the **PReq** message. Usually, the payment instruction from the cardholder may only be used for one authorization request; thus, it is necessary for the shopper to indicate explicitly that the Merchant will need multiple authorizations. |
| Authorization Request **(AuthReq)** | ~~The Merchant sets the subsequent authorization indicator to alert the system that there is a business need for subsequent authorization.~~ The Merchant passes the installment or recurring payment data from the Cardholder to the Payment Gateway. |
| Authorization Response **(AuthRes)** | The Payment Gateway returns an **AuthToken** in the Authorization Response, which will allow one additional authorization. As each **AuthReq** is processed, the Payment Gateway includes an **AuthToken** for the next authorization – until the authorization for the final installment is processed, when no **AuthToken** is returned. |

### Credit for an old transaction

Individual Acquirers will establish recommended times for data to be retained by their merchants. However, a cardholder may request a credit after all data relating to the original transaction has been purged from the Merchant's logs. SET supports the processing of a credit when the Merchant no longer has the information about the original transaction – in this case, an operator will need to manually enter the credit data.

### Credit to a different account

SET also supports the processing of a credit to a different account than that used to pay for the order – for example, if a cardholder returns a gift and requests a credit to their account, rather than to the account of the person who purchased the gift. In this case, an operator will need to manually enter the account information.

# Section 4
# Capture Processing

## Overview

### Purpose

This section provides a high-level description of the optional capture processing models relevant to SET.

### Organization

This section includes the following topics:

- Overview of Capture Processing
- Capture Processing Guidelines
- Batch Processing Overview
- Merchant Batch Processing
- Payment Gateway Batch Processing

### Terminology

The following terms are used to refer to the process of requesting payment from the cardholder's financial institution.

| | |
|---|---|
| capture | the exchange of messages between the Merchant and the Acquirer |
| clearing | the exchange of messages between the Acquirer and the Issuer over a financial network |
| settlement | the transfer of funds between the Issuer and the Acquirer |

### Scope

SET provides a mechanism for capture processing as well as for reporting on clearing and settlement activities.

# Overview of Capture Processing

## Introduction

SET capture processing recognizes the following options:

| | |
|---|---|
| Connectivity | • a Payment Gateway connected to an Acquirer host,<br>• a Payment Gateway connected to an intermediate capture system, and<br>• processing performed out-of-band to SET. |
| Accounting | • Merchant/Acquirer accounting through batches, and<br>• Merchant/Acquirer accounting through out-of-band mechanisms. |
| Batch control | • batches controlled by the Merchant, and<br>• batches controlled by the Acquirer or Payment Gateway |
| Credits | • credits for the full amount of the transaction,<br>• credits for a partial amount of the transaction,<br>• credits processed after transaction data has aged off logs, and<br>• credits processed for a different account number. |

## Out-of-band capture requests

For merchants processing capture requests out-of-band to SET, the Authorization Response (**AuthRes**) must include all data necessary to clear the transaction at the best available interchange rate, based on the characteristics of the authorization.

In particular, if the merchant does not receive the cardholder account information in the **AuthRes**, the Acquirer must provide a mechanism to add that information to capture requests that are received out-of-band to SET. See Table 7 on page 18 for possible transaction identification data.

## Merchant/ Acquirer accounting

SET provides explicit support for Merchant/Acquirer accounting through the use of capture batches to combine transactions for reconciliation and reporting. Other mechanisms must be supported out-of-band to SET or through message extensions.

# Capture Processing Guidelines

| | |
|---|---|
| **Ship before capture** | In general, the merchant software should not submit an item into capture until the ordered goods have been shipped. There are exceptions: Payment brand rules may permit capture before shipment of goods such as a custom-built computer. |
| **Capture all items** | The merchant software should ensure that all authorized items are either submitted for capture or reversed. If the software has on-line access to order status so that it can determine when an order has shipped or been canceled, it can perform this processing automatically. If it does not have such access, it must depend on manual input from a user to determine when items should be captured or reversed. |
| **Capture amount** | The capture amount may be different than the authorization amount. Payment brand and Acquirer rules will determine the allowable difference. |
| **Capture request analysis** | The merchant can submit multiple items for a single payment brand in a capture request. The Payment Gateway will analyze each item and will accept or reject each item. The items that are accepted will be submitted by the Payment Gateway to the Acquirer for processing. The merchant must determine the action to take on any item that is rejected. |
| **Reversals correct errors** | The merchant must submit a **CapRevReq** or **CredRevReq** to correct processing errors for a capture or credit request.<br><br>Note: There are no partial capture or credit reversals; the amount of a reversal is always the same as the amount of the corresponding request. This applies even if the capture was accomplished via **AuthReq** with **CaptureNow**. In this case the Merchant submits **AuthRevReq** (with **CaptureNow**) for the full amount. |

# Batch Processing Overview

**Identifying batches**

The Merchant, Payment Gateway, or Acquirer assigns each SET transaction to a specific capture batch and also:

- assigns an integer to identify the batch, and
- optionally assigns a unique integer to identify each item within the batch.

**Access to batches**

A capture batch may be opened by the Merchant, Acquirer, or Payment Gateway.

If a batch is opened by the Acquirer or Payment Gateway, it shall be closed only by that Acquirer or Payment Gateway.

If a batch is opened by the Merchant, it may be closed by the Merchant, Acquirer, or Payment Gateway.

**Batch contents**

A batch contains capture items and credit items, which are added using **CapReq** or **AuthReq** with **CaptureNow** and **CredReq** respectively. Items are removed from the batch using **CapRevReq**, **AuthRevReq** with **CaptureNow**, and **CredRevReq**.

In rare circumstances, an item may be reversed after the batch has been closed, in which case the reversed item will appear in another batch with a negative amount.

The merchant can remove all items from an open batch by issuing a purge operation using **BatchAdminReq**.

**Reasons to use a batch**

A capture batch provides a convenient mechanism to group transactions, such as to group:

- all transactions for a period of time;
- all transactions for a specific payment brand;
- items for accounting purposes; and
- items for reporting and reconciliation purposes.

*Continued on next page*

# Batch Processing Overview, continued

### Merchant inquiries

The Merchant can use **BatchAdminReq** to inquire of the Acquirer or Payment Gateway to determine the status of:

- a batch and the items within a batch, and
- the transmission of batch information from the Payment Gateway to the first non-SET system.

The Merchant can send or receive batch totals and transaction detail to (or from) the Acquirer or Payment Gateway.

### Batch transaction detail

The Merchant can use **BatchAdminReq** to request transaction detail. The Payment Gateway can also return transaction detail in a Capture Response (**CapRes**) message.

When the transaction detail is generated by the Payment Gateway, it will contain an entry for each item in the batch that:

- has been captured using **AuthReq** with **CaptureNow** set to TRUE and has not subsequently been reversed using **AuthRevReq**;
- has been captured using **CapReq** and has not subsequently been reversed using **CapRevReq**;
- has been credited using **CredReq** and has not subsequently been reversed using **CredRevReq**;
- has been reversed using **CapRevReq** or **CredRevReq** after the item has already been submitted for clearing.

Note: A negative amount is returned for an item that has been reversed after clearing.

### Batch balancing

Batch balancing may be performed by the Merchant after requesting batch detail from the Payment Gateway, or by the Payment Gateway on receipt of batch detail from the Merchant.

Batch balancing is performed ~~by the merchant~~ by adding the transaction amounts ~~receipts at the point of service~~ and comparing that total to the batch total(s) ~~at the host~~. If the amounts are the same, the batch balances. If the amount differs, ~~the merchant may be able to examine each transaction through a look-up based on account number, transaction ID, amount, or various combinations of those elements.~~ the individual transactions are examined to identify the source of the discrepancy.

~~*Note: Topics Terminal Data Capture and Host Data Capture were deleted completely.*~~

# Merchant Batch Processing

**Processing models**

A Merchant can submit items for capture using:

- **AuthReq** with **CaptureNow** set to TRUE;
- **CapReq** for capture items; and
- **CredReq** for credit items.

**Batch administration**

A merchant can use **BatchAdminReq** to administer the batch or request information about it. The functions available are:

| | |
|---|---|
| open | The Merchant uses this function to open a new batch. If the merchant is assigning batch numbers, the value is specified; otherwise, the Payment Gateway will assign the batch number and return the value in the response. |
| purge | The Merchant uses this function to purge all items in an open batch. Most often this function is used when the Merchant is unable to reconcile batch totals. |
| close | The Merchant uses this function to close an open batch. Once the batch has been closed no items can be added to it, so the Merchant should ensure that there are no outstanding requests for items in the batch. (Subsequent requests for the same transactions – such as the authorization of a recurring payment – may, of course, be put into subsequent batches.) |
| request summary ~~detail~~ information | The Merchant uses this function to request transaction amount totals for all items in the batch. |
| request transaction detail | The Merchant uses this function to request transaction detail for each item in the batch. |

# Payment Gateway Batch Processing

**Processing models**

A Payment Gateway shall process items submitted for capture by a Merchant using:

- **AuthReq** with **CaptureNow** set to TRUE;
- **CapReq** for capture items; and
- **CredReq** for credit items.

**Authorize and capture now**

When the Merchant sends an **AuthReq** with the **CaptureNow** flag set to TRUE, the Payment Gateway shall either:

- submit a full financial transaction (one that both authorizes and settles) to the Acquirer host for processing through a financial network; or
- submit an authorization request to the Acquirer host for processing through a financial network; if the authorization request is approved, the Payment Gateway shall continue to process the transaction as though a **CapReq** had been submitted by the Merchant for that transaction.

**Authorize now and capture later**

When the Merchant sends a **CapReq** or **CredReq** message, the Payment Gateway shall validate each item in the request to ensure that it has been authorized through the Payment Gateway and that the contents of the item are valid.

For those items that are valid, the Payment Gateway shall process each item so that it can be cleared through the Acquirer host. Depending on the processing model of the Acquirer, the items may be transmitted immediately or stored in a transaction database for later transmission. If the items are stored in a transaction database, they are sent to the Acquirer host after the batch is closed.

The Payment Gateway sends the corresponding **CapRes** or **CredRes** as soon as the items in the request have been validated.

*Continued on next page*

# Payment Gateway Batch Processing, continued

**Use of capture token**

The Payment Gateway may return a **CapToken** to the Merchant on an authorization response. The contents of the token are defined by each Payment Gateway vendor based on its own processing requirements. It will contain the account information and authorization-related data necessary to process capture and credit requests.

Many Payment Gateway implementations will maintain a transaction database that contains information about approved and captured items. In these situations, the **CapToken** will not be necessary for normal transaction processing. However, the Payment Gateway vendor may still choose to return a **CapToken** to support processing of credit requests that occur long after the original capture item is processed.

SET processing rules require the Payment Gateway to validate that the Merchant submits the correct **CapToken**. If the Payment Gateway does not require the token to process the transaction, it can avoid the processing overhead of decrypting the token by comparing a hash of the encrypted token against a hash that is stored in the transaction database when the token is created.

**Data augmentation**

The Payment Gateway must ensure that the Acquirer host has all data elements necessary to clear a transaction. If the Acquirer host does not maintain its own transaction database, the Payment Gateway can access this data by:

- storing the data elements in a transaction database;
- obtaining the information in the capture and credit requests from the Merchant; or
- obtaining the information from a **CapToken**.

**Batch administration**

The Payment Gateway shall process batch administration requests from the Merchant by performing the following actions:

| | |
|---|---|
| open | The Payment Gateway will open a new batch. If the Merchant is not assigning batch numbers, the Payment Gateway will assign the batch number and return the value in the response. |
| purge | The Payment Gateway will remove all items related to the batch from its transaction database or will instruct the Acquirer host to purge the batch. |
| close | The Payment Gateway will close the batch. If the Payment Gateway has been storing the items in a transaction database, it will transmit the items to the Acquirer host. |
| request summary detail | The Payment Gateway will return transaction amount totals for all items in the batch. |
| request transaction detail | The Payment Gateway will return transaction detail for each item in the batch. |

# Chapter 2
# System Architecture

## Overview

### Introduction

Chapter 2 provides an overview of the system architecture.

### Organization

Chapter 2 includes the following sections:

| Section | Title | Contents | Page |
|---|---|---|---|
| 1 | System Overview | Provides a high-level overview of the SET architecture. | 38 |
| 2 | Security Services | Describes the security features provided by SET and the certificates and certificate controls provided to implement them. | 45 |

# Section 1
# System Overview

## Architecture

**Protection of information**

The architecture of SET is designed to protect the transmission of financial information involved with a payment transaction between a cardholder, a merchant, and an Acquirer. It does not impose requirements on the transmission of the transaction's order information. Vendors developing shopping and ordering applications and protocols are strongly encouraged to protect this order information.

**Tamper resistant hardware**

*Tamper resistant* means the device itself is protected against intrusion. ISO 9564-1:1991 calls it a "physically secure device" and defines it as follows:

> "A physically secure device is a hardware device which when operated in its intended manner and environment cannot be successfully penetrated to disclose all or part of any cryptographic key or PIN resident within the device.
>
> Penetration of the device when operated in its intended manner and environment shall cause the automatic and immediate erasure of all PINs, cryptographic keys and all useful residue of PINs and keys contained within the device.
>
> A device shall only be operated as a physically secure device when it can be assured that the device's internal operation has not been modified to allow penetration (e.g. the insertion within the device of an active or passive 'tapping' mechanism)."

*Continued on next page*

# Architecture, continued

### SET Cardholder

The SET Cardholder is represented in SET by a computer. This provides the cardholder with the flexibility to shop and conduct negotiations with Merchant systems offering items for sale. The computer may support all phases of the electronic shopping model described on page 10. In supporting SET, the computer has the functionality to support the payment process.

### Cardholder interfaces

The Cardholder software's primary interface in SET is to Merchant systems. This interface supports the Cardholder's portion of the payment protocol, which enables the user to initiate payment, perform inquiries, and receive order acknowledgment and status.

The Cardholder software also has an indirect interface to the Acquirer through the Merchant system. This interface shall support encrypted data fields that are sent via the Merchant to the Acquirer, but can only be decrypted by the Payment Gateway. This enables the Acquirer to mediate interactions between the Cardholder and Merchant, and by so doing to provide security services to the cardholder. These security services ensure that the cardholder is dealing with a valid, payment-card-approved merchant.

Depending on the policies established by the payment card brand, the Cardholder software may also interface with a Cardholder CA (CCA) to request and renew public-key certificates that support electronic commerce security functions.

### Cardholder functions

Cardholder software shall support:

- security services – integrity, authentication, and certificate management as prescribed by SET, and
- communications functions.

It may also support shopping and payment selection.

Performing cryptographic functions in hardware cryptographic modules is recommended, but not required. Secret-key generation and storage using tamper resistant hardware cryptographic modules such as smart cards is encouraged.

*Continued on next page*

# Architecture, continued

**Merchant interfaces**

SET Merchant software provides a convenient interface to the Cardholder for the support of electronic payments. In addition, the Merchant interfaces with the Acquirer using the payment protocol to receive authorization and capture services for electronic payment transactions. The Merchant shall interface with the Merchant CA (MCA) to request and renew public-key certificates that support electronic commerce security functions.

**Merchant functions**

Merchant software shall support:

- SET protocols for the authorization of electronic commerce transactions initiated by the Cardholder;
- security services: integrity, authentication, and certificate management; and
- the shopping, payment selection, and communications functions.

It is expected that the Merchant system will also support captures.

Performing cryptographic functions in hardware cryptographic modules is strongly recommended, but not required. Secret key generation and storage using tamper resistant hardware cryptographic modules ~~such as smart cards~~ is strongly encouraged. Payment card brand requirements for a specific implementation and environment in which the merchant server may operate will dictate requirements for the use of hardware cryptographic support.

*Continued on next page*

# Architecture, continued

**Issuer**

An Issuer is the financial institution that establishes an account for a cardholder and issues the payment card. The Issuer guarantees payment for authorized transactions using the payment card.

The processing and interface to the Issuer is out-of-band from the perspective of SET.

**Acquirer**

An Acquirer is the financial institution (or its agent) that supports merchant activity through account relationships with merchants.

The Acquirer is responsible for gathering financial data related to payment card transactions in order to obtain authorization for payment from the cardholder's Issuer.

**Payment Gateway**

The Payment Gateway system is operated on behalf of the Acquirer to provide electronic commerce services to merchants in support of the Acquirer.

The Payment Gateway shall support:

- interface with the payment card brand's financial network to support the authorization and capture of transactions;
- interface with the Payment Gateway CA (PCA) to request and renew public-key certificates to support the electronic commerce security functions; and
- distribution of Certificate Revocation Lists (CRLs) and Brand CRL Identifiers (BCIs).

The Payment Gateway's interface to the payment card brand's financial network is largely unchanged from the interface supporting Acquirers today.

Cryptographic functions shall be performed in hardware cryptographic modules. Secret key generation and storage shall use tamper resistant hardware cryptographic modules.

**Third party processor**

In some environments, Issuers and Acquirers may choose to assign the processing of payment card transactions to third-party processors. SET does not distinguish between the financial institution and the processor of the transactions.

*Continued on next page*

# Architecture, continued

### Certificate Authority

The architecture of SET defines a trusted hierarchy of Certificate Authority (CA) systems that begins with a Root CA (RCA), then a brand-specific CA (BCA) and an optional Geopolitical CA (GCA). At the bottom level, one or more trusted CAs support the issuance and renewal of public-key certificates for cardholders, merchants, and Acquirers. For example, Cardholder Certificate Authorities (CCAs) interface with Issuers to authenticate requests for Cardholder certificates.

CA certificates are issued by the superior CA in the SET hierarchy, as described in Table 11.

| These CAs: | ...issue certificates for these CAs: |
|---|---|
| Root CA | Brand CA |
| Brand CA | Geopolitical CA<br>if no Geopolitical CA:<br>• Cardholder CA<br>• Merchant CA<br>• Payment Gateway CA |
| Geopolitical CA (if one exists for the Cardholder, Merchant, or Payment Gateway CA's area) | Cardholder CA<br>Merchant CA<br>Payment Gateway CA |

**Table 11: Certificate Issuance**

### Certificate Authority functions

Cryptographic functions shall be performed in hardware cryptographic modules. Secret key generation and storage shall use tamper resistant hardware cryptographic modules. Certificate management shall be performed in a secure physical environment compliant with payment card brand standards.

Part II, starting on page **Error! Bookmark not defined.**, provides a detailed explanation of certificates and certificate formats, certificate issuance and renewal, CRLs, and other certificate management functions. See also "Certificates" on page 47 and "CRLs and Brand CRL Identifiers" on page 51.

### Payment card brand's financial network

The payment card brand's financial network is the existing private network through which Acquirers obtain authorization for payment from Issuers. (VisaNet and Banknet are examples of these types of networks.) These networks are protected by each payment card brand and provide messaging interfaces (such as ISO 8583 formatted messages).

*Continued on next page*

# Architecture, continued

### Transport mechanisms

Two classes of transport mechanisms are recognized: interactive and non-interactive. The World Wide Web is an interactive mechanism, and electronic or postal mail are non-interactive mechanisms.

The SET specification does not define how a SET message is transported between entities. SET messages may be transported using any mechanism agreed upon by the sender and receiver.

It is expected that transport standards will be developed to address the issue of ~~interoperable~~ communication interoperability between SET applications. Until such standards are available, SETCo provides an interim standard in the *SET External Interface Guide*. (See "Related documentation" in the Preface.)

*Continued on next page*

# Architecture, continued

## Data storage

The processing descriptions later in *SET Book 2: Programmer's Guide* presume the following logical distinctions in data storage. (The physical implementation of these databases is beyond the scope of SET.)

| | |
|---|---|
| message database | The message database includes:<br>• unsigned and unencrypted data structures, and<br>• if a message is idempotent, the complete signed and/or encrypted message.<br>Items remain in the message database while messages are being exchanged (for example, a Cardholder's certificate request, including **CardCInitReq**, **RegFormReq**, and **CertReq**); once the series of messages is complete, the database entries have served their usefulness.<br>Only parts of messages may actually be stored in the message database. It is the application's responsibility to be able to retrieve all relevant data. In particular, things like *BrandCRLIdentifier* do not need to be stored in the message database. |
| transaction database | Includes significant data for a transaction (that is, for all messages linked by an **XID**). Both current state and significant previous states are stored.<br>The transaction database lives much longer than the message database; the Merchant may have to retain purchase information for months in able to process credits correctly. However, transaction information will not be stored by SET software indefinitely. Operational guidelines of payment card brands and Acquirers will specify minimum time periods that information must be stored on Merchant and Payment Gateway systems. |
| secure data storage | Applies to Cardholder applications only: Includes data whose confidentiality and security requires particular care, such as payment card numbers, expiration dates, and private keys. For further detail, see "Secure Data Storage" on page 107. |
| trusted cache | Includes certificates, Certificate Revocation Lists (CRLs), and Brand CRL Identifiers (BCIs) that have been validated and have not expired, along with their Thumbprints. (See "Thumbprints" on page 68.) |
| untrusted cache | Includes certificates, CRLs, and BCIs that have not been validated. |

# Section 2
# Security Services

## Overview

**Purpose**

This section provides a brief summary of fundamental security services provided in the architecture of SET and the certificates used to implement them.

**Organization**

This section includes the following topics:

- Services
- Certificates
- CRLs and Brand CRL Identifiers

# Services

**Integrity**

SET provides integrity to ensure that a message was not modified in transit by using:

- one-way cryptographic hashing algorithms,
- digital signatures, and
- ~~a linkage mechanism for verifying that a message contains a reference to another message by verifying an embedded link using a~~ (when necessary for additional integrity) one-way cryptographic hashing algorithms to cryptographically link one message to another.

**Authentication**

SET provides authentication of a message's origin by using digital signature verification algorithms when signature certificates are available.

**Confidentiality**

SET provides confidentiality by using both asymmetric and symmetric-key algorithms to protect financial information from eavesdroppers.

As an option, confidential Acquirer-to-cardholder messages are provided. This feature is intended to allow Issuers to communicate back to cardholders about the reason that a transaction is being declined or to request that the cardholder call the Issuer.

**Caveat**

~~SET does not provide non-repudiation. It is the intent to permit non-repudiation via rules and policies of individual payment card brand implementations.~~ Non-repudiation is a legal concept indicating whether a party has a right to challenge another party's claim of not having participated in a transaction. The provisions and technical requirements to achieve non-repudiation may vary depending on the jurisdiction.

SET has not been analyzed to determine if its digitally-signed messages meet the legal definition of non-repudiation. The rules and regulations of each payment card brand will determine when a party has a right to deny participation in a SET transaction.

# Certificates

## SET certificates

SET uses X.509 version 3 certificates to support public keys for signature and encryption. These certificates include a public key together with the means of authentication of that key.

## Purpose of certificates

The fundamental purpose of a certificate is to bind a public key to a uniquely identified entity. It does so as follows:

- An entity creates a unique key pair including a private key and a public key that are mathematically linked, and sends it to a CA along with a certificate request.
- After authentication, the CA creates a certificate containing the entity's identification and its public key and digitally signs it.
- It is the responsibility of the cardholder, merchant, financial institution, or CA to maintain exclusive control of the private key.

A digital signature cryptographically binds the signed data with the private key. Since the private key is mathematically linked to the public key of the key pair, the digital signature has the effect of binding the public key to the data as well.

However, anyone can generate a public/private key pair, so it is essential that some mechanism be established that binds the public key to the entity in a trustworthy manner.

Since a fraudulent CA could be set up to create certificates that would contain information nearly identical to that contained in a valid certificate, the signature of the CA itself shall be certified as authentic by a higher level CA. The only exception to this requirement is the industry Root CA; it is the only implicitly trusted CA.

*Continued on next page*

# Certificates, continued

**Cardholder certificates - function and content**

A cardholder's signature certificate implicitly binds the public key to the cardholder's primary account number (PAN), but the PAN is effectively obfuscated by using a blinding technique so that ~~only the CCA, the cardholder, and the Issuer know the account number~~ it cannot be determined from the certificate alone. The cardholder passes the account number and a secret value to the Acquirer, so that the Acquirer can then verify the card number against the blinded account information contained in the cardholder's certificate. In order to protect the cardholder's confidentiality, the cardholder's name is not included in the certificate. In effect, the blinded account information is a pseudonym of the cardholder.

~~One function of the Acquirer is to ensure that the private key used to sign a payment is, in fact, associated with the right payment card account. To avoid revealing the cardholder's PAN to third parties, the number is hidden using a keyed-hashing mechanism as a blinding function. The result of this function is what is stored in a cardholder certificate.~~

**Cardholder certificates - optional**

SET allows cardholders without signature certificates to conduct SET transactions. This is an interim option intended for use only in situations where the Issuer does not provide certificate services. ~~Acquirers~~ Payment brands may choose whether or not to support this option.

A flag in the Payment Gateway certificate indicates support for transactions in which the cardholder has no certificate. Cardholder software and Payment Gateway software shall use that flag to ensure that certificates are included in transactions when necessary.

Brands that initially support cardholders without certificates may remove such support by reissuing Payment Gateway certificates.

If a user has obtained a cardholder certificate for a payment card account, the Cardholder software ~~should~~ shall perform only signed transactions for that account.

Support for cardholders with certificates is mandatory: Cardholder, Merchant, and Payment Gateway software shall fully support cardholder certificates and transactions based on them.

*Continued on next page*

# Certificates, continued

### Merchant certificates

At least two key pairs are required for a Merchant to participate in SET transactions:

- A signature pair that is used to sign and verify messages provided to the Cardholder and Payment Gateway; and
- A key-encryption pair that is used to protect messages generated by the Payment Gateway.

A merchant may have additional sets of encryption and signature key pairs because of physical implementation, security concerns, Acquirer policy, or a variety of other reasons. For example, a merchant that operates multiple servers may elect to have a separate set of encryption and signature key pairs for each server. In addition, new key pairs shall be generated periodically.

The number of certificates needed by a merchant is a function of the number of that merchant's encryption and signature key pairs, the number of Payment Gateways that interface with the merchant, and the number of brands accepted by the merchant.

In the simplest case, the merchant will interface with a single Payment Gateway to process all brands. However, a merchant may have relationships with multiple Acquirers. For example, a single Acquirer may not process all the brands the merchant accepts, or the merchant may do business in multiple national markets (and currencies) and have corresponding Acquirer relationships. In addition, Acquirers may choose to operate multiple Payment Gateways for load balancing and redundancy.

### Merchant access to account information

SET allows the Acquirer to return cardholder payment information to the merchant, encrypted under the merchant's key. This capability is designated by an indicator in the merchant's certificate. This option is intended to allow merchants to use out-of-band clearing mechanisms and to support legacy systems that depend on the availability of the account information.

*Continued on next page*

# Certificates, continued

### Payment Gateway certificates

Two key pairs are required at the Payment Gateway:

- A signature pair that is used to sign and verify messages provided to the ~~Cardholder and~~ Merchant; and
- A key-encryption pair that is used to protect payment instructions generated by the cardholder and messages generated by the merchant.

The number of certificates required by the Payment Gateway reflects the number of brands it handles.

### Certificate chain validation

Certificates shall be validated through a hierarchy of trust. Each certificate is linked to the signature certificate of the certificate issuing entity. Certificates are validated by following the trust hierarchy to the Root CA. The path through which the certificates are validated is called the certificate chain.

The validation of each certificate shall be enforced at all levels of the chain. For example, a cardholder shall validate the merchant, Merchant CA, Geopolitical CA (if any), Brand CA, and Root CA certificates. The validation process may stop at a level that has been previously validated. A detailed description is provided in "Certificate Chain Validation" on page 123.

### Summary of certificate types

Table 12 lists the certificates defined by SET:

| Certificate Types | Message Signing | Key Encryption | Certificate and/or CRL Signing |
|---|---|---|---|
| Root CA | | | X |
| Brand CA | | | X |
| Geopolitical CA | X | | X |
| Payment Gateway CA | X | X | X |
| Merchant CA | X | X | X |
| Cardholder CA | X | X | X |
| Payment Gateway | X | X | |
| Merchant | X | X | |
| Cardholder | X | | |

**Table 12: Summary of Certificate Types**

# CRLs and Brand CRL Identifiers

### Certificate Revocation List (CRL)

A certificate may need to be revoked or canceled for a number of reasons: for example, due to a real or suspected compromise of the private key, a change in the identification information contained in the certificate, or termination of use.

Each CA, with the exception of the MCA and CCA, shall generate, maintain, and distribute a Certificate Revocation List (CRL) that lists certificates that it issued that have been revoked.

Table 13 lists the SET entities that may generate a CRL and the reasons for which they would do so.

~~Root CA — unscheduled replacement of a Root certificate or BCA certificate~~

~~Brand CAs — unscheduled replacement or termination of a CA certificate issued by the BCA~~

~~Geopolitical CAs — unscheduled replacement or termination of a CCA, MCA, or PCA certificate issued by the BCA or GCA~~

~~Payment Gateway CAs — unscheduled replacement or termination of a Payment Gateway certificate issued by the PCA~~

| This entity: | ...shall generate a CRL in the event of unscheduled replacement or termination of a certificate that it issued to: |
|---|---|
| Root CA | Root CA<br>Brand CA |
| ~~Any~~ Brand CA | Geopolitical CA<br>CCA, MCA, or PCA |
| Geopolitical CA | CCA, MCA, or PCA |
| Payment Gateway CA | Payment Gateway |

**Table 13: Entities That Generate CRLs**

Note: Cardholder and Merchant certificates are canceled rather than revoked; that is, they do not appear on any CRL, as there are other means of determining that they are no longer valid.

### Brand CRL Identifier (BCI)

Each brand is responsible for managing Certificate Revocation Lists (CRLs) within its own domain. The SET architecture introduces the concept of a Brand CRL Identifier (BCI). A BCI is digitally signed by the brand and used to identify the SET CRLs that the Cardholder, Merchant, Payment Gateway, and CA systems need to reference when validating certificates as part of signature verification.

### Further detail

Additional information about CRLs and BCIs is available in Part II starting at page **Error! Bookmark not defined.**.

# Chapter 3
# Technical Requirements

## Overview

### Introduction

Chapter 3 summarizes other design considerations that affect the overall technical requirements for SET.

### Organization

Chapter 3 includes the following sections:

| Section | Title | Contents | Page |
|---|---|---|---|
| 1 | Security | Summarizes the primary security considerations for SET. | 53 |
| 2 | Adaptability | Summarizes the implications on the design of supporting different environments with respect to cardholder certificates. | 59 |
| 3 | Interoperability | Summarizes the general message formats and encapsulation methods. | 60 |

# Section 1
# Security

## Overview

**Introduction**

The intent of SET is to address certain security issues related to three-party payment mechanisms conducted over the Internet.

**Organization**

This section includes the following topics:

- Integrity
- Authentication
- Confidentiality

# Integrity

## Definition

Data integrity is the assurance that the data received is the data that was sent.

The sender generates an integrity value based on the data to be transmitted, then transmits both the data and the integrity value to the receiver. The receiver validates the integrity value, thus verifying that the data has not been altered during transmission.

## Hash functions

Data integrity is supported by the use of hash functions. A hash function is applied to the appropriate data to produce a statistically unique integrity value called the *hash value*. Hash functions by themselves do not guarantee absolute data integrity. To provide this guarantee, part of what is hashed must be a secret key.

Hash functions are different from symmetric-key algorithms and have the following properties:

- A hash function is a public algorithm.
- A hash function is one-way; that is, given the hash value, it is not possible to recreate the original data. (If the hash function is not cryptographically secure, it may be possible to predict the input from the output – but many possible inputs could have the same output. For a cryptographically secure hash algorithm, such as those used in SET, it is not computationally feasible to recreate the original data.)
- The hash value is computed in such a manner that it is not feasible to identify other data that will hash to the same value.

## Digital signature

A digital signature is defined as data appended to, or a cryptographic transformation of, a data unit that allows a recipient of the data to prove the source and integrity of the data, and thereby protects against forgery.

In SET, a digital signature is a hash value encrypted using the private key of the sender. The hash value provides integrity of the data within the message; if the payment data is modified, the hash value will be different, and that difference can be detected when the receiver re-computes the hash. The hash is encrypted to ensure that a third party cannot change the hash, since encryption of the new hash value would not be possible without the private encryption key.

# Authentication

**Definition**

Authentication provides assurance that the data received was sent by the party who claims to have sent it.

The sender uses digital signatures and public-key certificates to prepare the data to be sent. The receiver verifies the digital signatures and public-key certificates, thus authenticating the sender.

**Authenticating public keys**

SET requires public/private key pairs for Payment Gateways, Merchants, and CAs, and supports them as a recommended option for Cardholders. The public keys are distributed via certificates signed by authorized CAs.

Cardholders, Merchants, and Payment Gateways shall authenticate the public keys of the CAs and the Root Keys using mechanisms provided in SET. See "Root key distribution and authentication" on page **Error! Bookmark not defined.** in Part II.

**Entity authentication**

Digital signatures require a trusted third party to vouch for the authenticity of the public key used to verify the signature. The process dictates that the trusted third party, a CA, provides an electronic certificate that vouches for the fact that a public key is "owned" by a certain entity. This electronic certificate (itself digitally signed by the CA) is stored by the entity in its computer, and accompanies signed messages sent to other entities. The receiver's system uses the certificate and certificate chain to verify the sender's public key. At that point the receiver is sure that:

- the original data was not altered (data integrity);
- the message could only have been signed by the holder of that private key (entity authentication); and
- a trusted third party has vouched for the fact that the signer is in fact the holder of that key pair.

The uniqueness of the digital signature and the underlying hash value coupled with the strength of the public key certificate provide an acceptable level of assurance to authenticate the sender and to verify that the sender originated the signed data.

*Continued on next page*

# Authentication, continued

### Cardholder authentication

The Cardholder certificate issued by the CCA is evidence that the Cardholder's public key has been tied to the account number. This mechanism will reduce the incidence of fraud and therefore the overall cost of payment processing.

Merchants and Acquirers shall verify that a cardholder is using a valid account number by verifying the signatures on the Cardholder certificate and by validating the certificate's chain.

### Merchant authentication

The Acquirer shall authenticate the merchant's certificate request and, if appropriate, issue a certificate through its MCA. The Merchant certificate provides verification of an agreement between the merchant and the Acquirer. In essence, the certificate is an "electronic decal," similar to the brand decal in the merchant's window.

Cardholders and Payment Gateways shall authenticate Merchants by verifying the signatures on the Merchant certificate and by validating the certificate's chain.

### Payment Gateway authentication

Payment card brands shall authenticate the Acquirer's certificate request and, if appropriate, issue a Payment Gateway certificate through the brand's PCA.

Since the Cardholder uses the Payment Gateway's public key for encrypting the symmetric key used to encrypt the payment instruction, the Cardholder (as well as the Merchant) must authenticate the Payment Gateway. The Merchant provides the Cardholder with the Payment Gateway's encryption certificate. Cardholders and Merchants shall authenticate Payment Gateways by verifying the signatures on the Payment Gateway certificate and by validating the certificate's chain.

# Confidentiality

### Definition

Data confidentiality is the protection of sensitive and personal information from unintentional and intentional attacks and disclosure.

Securing such data in uncontrolled environments, such as unsecured networks, requires data encryption and associated key management.

### Confidentiality in SET

SET uses both asymmetric and symmetric-key algorithms in conjunction with a digital envelope to provide data confidentiality. Refer to *SET Book 1: Business Description* for an overview of this technique.

### Protecting private keys

Public-key signature mechanisms are critically dependent on the security of the corresponding private keys. Developers shall pay particular attention to the methods used to store the private keys:

- Private keys shall be protected through encryption or tamper resistant mechanisms.
- Payment Gateways and Certificate Authorities shall use tamper resistant hardware cryptographic modules to perform cryptographic functions and to generate and store secret keys.
- Merchant and Cardholder applications should also employ hardware cryptographic modules to perform cryptographic functions and to generate and store secret keys.

### Trusted cache

Certificates, CRLs, and BCIs will be accessed frequently when processing SET messages. Thus, the processing of successive SET messages may be optimized by maintaining a local trusted cache of frequently accessed certificates, CRLs, and BCIs and their Thumbprints. (See "Thumbprints" on page 68.)

- Each SET application shall fully validate certificates, CRLs, and BCIs before adding them to the application's trusted cache.
- Each Cardholder and Merchant system supporting SET shall enforce a policy to protect its trusted cache from unauthorized access or modification.

Each certificate, CRL, and BCI shall either be authenticated and added to the trusted cache or discarded at the conclusion of processing the message that contained it.

*Continued on next page*

# Confidentiality, continued

### Protecting account information

SET offers an option that permits the Payment Gateway to provide cardholder account information to the merchant, encrypted under the Merchant's public key. When this option is used, care shall be taken to ensure the security of the payment information as it resides on the merchant's systems:

- Merchant applications shall store payment information in encrypted form.
- Merchants should store payment information off-line, or behind a firewall or similar mechanism.

### Payment data

SET is responsible for the confidentiality of payment data that it needs to manage. Where non-payment data confidentiality is needed, it is provided in the protocol messages by including a reference to the data rather than the data itself. For example, SET does not exchange the Order Description (**OD**), but includes a hash of the **OD** in the Purchase Request (**PReq**). The following assumptions apply:

- The bit stream for the **OD** and purchase amount at the Merchant is identical to the bit stream for the **OD** and purchase amount at the Cardholder.
- The Cardholder and Merchant software shall agree on the representation of this data before SET is invoked.

# Section 2
# Adaptability

## Variations

**Purpose**

This section illustrates how SET has been designed to be adaptable to different business models and operational environments, such as support for cardholders without certificates. Several appendices provide more information about this topic:

- Appendix D: SET Fields
- Appendix E: Field Support Requirements
- Appendix S: Implementation Variations

**Use of cardholder certificates**

The cardholder's signature certificate provides authentication and integrity of data sent to the Merchant and to the Payment Gateway. SET supports environments in which cardholder signature certificates are required, and also environments in which they are optional. A payment card brand determines whether or not its application of SET requires cardholder signature certificates.

**Certificate-required environments**

In environments in which certificates are required, all messages from the cardholder that require authentication and integrity shall be signed with a signature authenticated by the cardholder certificate. There are protocol initiation requests that do not include such signatures, since no significant protocol failures would result from their abuse. All other messages are signed~~, and the recipients of these messages are assured receipt of the corresponding certificates by the protocol~~.

**Certificate-optional environments**

When a cardholder does not have a signature certificate, no digital signature is generated. Instead, the Cardholder software generates a hash of the data and inserts the hash into the digital envelope to ensure ~~the integrity of its contents~~ that the data in the message corresponds to the digital envelope. However, unlike a signed message, the hash does not protect against substitution of both the digital envelope and the message.

# Section 3
# Interoperability

## Introduction

**Organization**

This section includes the following topics:

- General Message Formats
- MessageWrapper
- Backward Compatibility
- System Clock Differences
- Extension Mechanism for SET Messages
- PKCS #7 Formats
- Transaction Validation by Non-SET Systems
- Optional Fields
- Language
- Date Fields
- Amount Fields

# Initiation

### SET environments

It is anticipated that SET applications will operate in one of two environments:

- interactive – in this environment, the entities communicate in "real-time" with small time delays between the exchange of messages (such as the World Wide Web); or
- non-interactive – in this environment, the entities communicate with large time delays between the exchange of messages (such as electronic mail).

### SET initiation process

In an interactive environment, it is expected that a "SET initiation process" takes place that triggers the SET protocol. This process allows the Cardholder and Merchant to exchange certain information required for SET. Such information includes (but is not limited to):

- the brand the cardholder has selected,
- the order description, and
- the purchase amount.

It is expected that standards will be developed to address how this information is exchanged and how the SET protocol is initiated. Until such standards are available, SETCo provides an interim standard in the *SET External Interface Guide*. (See "Related documentation" in the Preface.)

# Message Formats

**Overview**

SET messages shall be formatted using non-proprietary techniques, permitting communication over a variety of interactive and non-interactive mechanisms (as discussed in "Transport mechanisms" on page 43). Wherever possible, external standards are employed to enable the protocol to be easily implemented and to ensure that interoperability among implementations is possible. Appendix A lists the specific version of each external standard on which SET is based.

**Cryptographic treatments**

Cryptographic treatments are constrained to ensure that only as much cryptography is employed as is required by the security needs of the payment card transaction.

To promote interoperability and the ability to upgrade, SET uses the Public Key Cryptography Standards (PKCS) to represent the cryptographic parameters and message encapsulation

**Notation and encoding**

SET messages are defined using the ISO/IEC and ITU-T Abstract Syntax Notation (ASN.1) standard and shall be encoded using the Distinguished Encoding Rules (DER). This permits unambiguous encoding through a standard that is well understood and widely accepted.

**Encoding alternate character sets**

Unicode and the Basic Multilingual Plane are synonymous. SET supports BMPString for all character strings where data can be displayed to a user; SET uses VisibleString in cases where the character representation is limited to 7-bit ASCII.

*Continued on next page*

# Message Formats, continued

### ASN.1/DER encoded messages

The ASN.1 notation provides a clear, unambiguous definition of the content of messages; DER provides an encoding that is precise and that ensures a single format for encoded data. Such precision and uniqueness is critical to being able to support operations involving hashes and signatures.

SET ASN.1 definitions include a collection of intrinsic types that are used to define data fields and messages but depend on additional restrictions and constraints. These shall be checked by the application software. For example:

- ASN.1 type *IA5String* is used to define several data fields that contain character string data (such as **MerOrderNum**). The permitted alphabet for values of the *IA5String* type is sometimes referred to as the ASCII character set.
- Size constraints on the fields are imposed (for example, **MerOrderNum** may not exceed 25 bytes) and shall be checked by all SET software.

Commercial ASN.1 code generators are available that enable software developers to generate and receive SET messages with only modest programming effort beyond providing the ASN.1 specification itself to such tools.

# MessageWrapper

**Purpose**

The **MessageWrapper** is the top level ASN.1~~/DER data structure~~ type in the SET protocol. Every SET message contains a ~~cleartext~~ **MessageWrapper**, which contains the following components:

- a clear text **MessageHeader**,
- a **Message**, and
- optionally, message extensions.

The **MessageWrapper** presents information to the receiver of a message at the very start of message processing that can be used directly by the receiver without first performing cryptographic processing. The **MessageWrapper** identifies the type of SET **Message** and provides unique identifiers that are sufficient for the receiver to detect duplicate and unexpected messages.

**Processing**

All SET-related processing begins with the **MessageWrapper**.

The **MessageHeader** shall be decoded before **Message** processing. The **TransIDs** and **RRPID** fields have been placed in the ~~**MessageWrapper**~~ **MessageHeader** to permit early duplicate detection; these fields are repeated within the **Message**, so that the integrity of this data can be protected within the body of the message by cryptographic enhancements.

At the time the ~~**MessageWrapper**~~ **MessageHeader** is decoded, decoding of the **Message** component may ~~not~~ be ~~processed~~ deferred; ~~but~~ however, its type can be determined from the ~~DER type~~ ASN.1 tag field ~~of~~ **Message**. After initial ~~**MessageWrapper**~~ **MessageHeader** processing is performed:

- the **Message** is decoded (if that has been deferred),
- the **Message** is decrypted and/or its signature is verified, as appropriate, then
- the content of the **Message** is decoded to yield the data that is processed individually for each message type.

# Backward Compatibility

**Application requirements**

In order for SET to be successful, new versions of SET must be able to interoperate with prior versions. In general, applications shall interoperate with the current version ~~revision~~ of SET and the immediate prior version (and the revisions of each). That is, an application that supports Version 2 of SET (when it is published) shall be able to send and receive Version 1 messages. A future version of SET may require compatibility with more than one prior version. Compatibility requirements will be explicitly stated in each version/revision that is published.

SET messages shall use the highest version/revision that both sender and receiver support.

**Checking the version**

To determine the version of the message, the software shall check **MessageHeader.version** and **MessageHeader.revision**.

**Responding to older version messages**

An application that can process a message from a previous version shall respond (if appropriate) using messages and formats from the received version.

An application that receives a message with a version number that is lower than it can process (such as a Version ~~1~~ 3 application receiving a Version ~~0~~ 1 message) shall reject the message by responding with an **Error** message containing an **ErrorCode** of *versionTooOld*.

**Software upgrade prompts**

An application that receives an **ErrorCode** of *versionTooOld* should display a message with information about how to upgrade to the latest version of the software. Cardholder software vendors in particular should include such a feature.

**Responding to newer version messages**

An application that receives a message with a version number that is higher than it can process (such as a Version 1 application receiving a Version ~~2~~ 3 message) shall reject the message by responding with an **Error** message with an **ErrorCode** of *versionTooNew*.

If possible, an application that receives an **Error** message with an **ErrorCode** of *versionTooNew* should try re-sending the message with a lower version of SET.

# System Clock Differences

**Clocks must match**

*Note to reviewers: This section will be reviewed to determine what changes are needed for the following situations:*
*1) message wrapper processing,*
*2) certificate chain validation, and*
*3) receipt and processing of initial certificates (may be different than 2 above)*

In order to process messages correctly, the system clocks of the sender and receiver must match reasonably well. Factors that can result in either system reporting an inaccurate time include:

- The user failed to set the clock to the correct local date and time.
- The user indicated an incorrect local time zone.
- The battery protecting the clock in the event of power failures has failed.
- The clock has drifted significantly since it was set.

**Checking the date and time**

Each implementation of SET will determine the time variation that it will accept. For example, a system may accept messages that report to have been generated up to 48 hours in the past or 12 hours in the future. If a message is received that is outside of this range, the application shall reject the message by responding with an **Error** message containing an **ErrorCode** of *messageTooOld* or *messageTooNew*.

**Clock change prompts**

An application that receives an **ErrorCode** of *messageTooOld* or *messageTooNew* should display a message with the current system date and time as well as the date and time reported by the other system. If possible, the application should provide a user interface feature (such as a button) that when activated changes the system clock to match the time provided by the remote system. Cardholder software vendors in particular should include such a feature.

*Note to reviewers: this will be updated to address situations where the CA accepts a message in the "near future" resulting in a not-yet-valid certificate being returned to the end entity.*

# Extension Mechanism for SET Messages

### Why extensions may be necessary

This version of SET was intentionally limited to the minimum functionality necessary to support cardholders and merchants doing business on the Internet. Consequently, some business functions are not included in the definition of SET payment messages. Furthermore, it is unlikely that SET could ever be robust enough to cover the business practices of every national market and every Acquirer. Therefore, it is necessary to provide a mechanism to extend SET payment messages.

An example of a business function that is not supported by the SET messages is Japanese Payment Options. Issuers in Japan have options for payment that are selected by the consumer at the time of the purchase. Since there is no place in the SET message to carry this information, an extension to the protocol is necessary.

### The extension mechanism

SET messages are extended in the same way that X.509 certificates are extended. Specifically:

- An extensions field is provided that contains a sequence of extension data.
- The extension data indicates the type of extension and the criticality of the extension.

See Appendix H: "Extension Mechanism for SET Messages" for details.

# Thumbprints

**Purpose**

Thumbprints are hashes of certificates, CRLs, or BCIs. They have several uses in SET:

- to minimize certificates, CRLs, and BCIs exchanged,
- to help ensure that an unsigned message was not altered,
- to indicate specific certificates included in a message, and
- to indicate a certificate, CRL, or BCI that caused processing to fail.

**Thumbprint generation**

A Thumbprint is computed by performing the SHA-1 hash of one of the following DER-encoded ASN.1 structures:

- **UnsignedCertificate**
- **UnsignedCertificateRevocationList**
- **UnsignedBrandCRLIdentifier**

The hash is computed over the tag-length-value of the encoded structure. The Thumbprint is the same hash that is used to sign or verify a certificate, CRL, or BCI.

*Continued on next page*

# Thumbprints, continued

### Minimizing certificates, CRLs, and BCIs exchanged

In order to support the security requirements of SET, public-key certificates, CRLs, and BCIs shall be carried in the protocol. Since these data structures are large, the Thumbprint mechanism is provided to reduce required traffic.

An end entity may include Thumbprints in a message as a compact way to identify the certificates, CRLs, and BCI that it is holding. The recipient of a message that contains Thumbprints may check the Thumbprints and include in the response only certificates, CRLs, or the BCI that the end entity does not have but will need for the transaction. (This process is described in more detail below.)

Since Thumbprints are very small compared to the certificates, CRLs, and BCIs that they represent, much overhead is avoided.

### Sending entity

~~Thumbprints are sent by an entity in a SET request message and can always be ignored by the corresponding recipient.~~

~~If~~ An end entity ~~would normally need a certificate or CRL from another SET entity, it~~ may ~~send the remote entity~~ include in many SET request messages the Thumbprints of the certificates, CRLs, and BCIs that it possesses and that it expects to be related to the transaction.

SET software shall not send Thumbprints for all certificates, CRLs, and BCIs currently existing in its cache, but only for those that are pertinent to a particular request/response message pair. For example, ~~merchant~~ Cardholder software shall not send the Thumbprints for other ~~cardholders~~ merchants or for other brands.

Thumbprints may be listed in any order.

### Receiving entity

The recipient of a SET request message shall ensure that the requester possesses every certificate, CRL, and BCI needed to complete the processing of the response message and to create subsequent request messages.

The system responding to a message that contains Thumbprints may either:

- ~~The responding entity should omit from its response message any certificates and CRLs for which it has received Thumbprints.~~ check the Thumbprints and include in the response only certificates, CRLs, or the BCI that the requester does not have but will need for the transaction; or
- ignore the Thumbprints and send every certificate, CRL, and BCI that the requester will need.

(If the requester does not include Thumbprints, the responder must always include all needed certificates, CRLs, and BCI.)

*Continued on next page*

# Thumbprints, continued

**Integrity of unsigned message**

For most SET messages, signature verification and cryptographic hashing provide assurance that the request was not altered. For certain unsigned messages, these methods are not available. Instead, the requester sends Thumbprints in the request message, then compares them to Thumbprints received in the response to obtain some assurance of integrity.

**Indicating certificates in request or response**

Some SET messages include one or more specific Thumbprints that identify specific certificates included in the message:

| When: | the Thumbprint(s) of | is/are included in: |
|---|---|---|
| a key-encryption certificate is included in a response | the key-encryption certificate | the response |
| a key-encryption certificate is being renewed | the certificate being renewed | the request |
| certificates are issued (new or renewal) | the new or renewed certificates | the response |

**Indicating invalid certificate, CRL, or BCI**

When message processing fails because of a certificate, CRL, or BCI, its thumbprint is included in the **Error** message. If more than one of these data structures is invalid, the **Error** message will indicate the first one processed (because processing stops once an **Error** message is generated).

# PKCS #7 Types

**Purpose**

To ensure interoperability and the ability to upgrade, the Public-Key Cryptography Standards (PKCS) #7, Cryptographic Message Syntax Standard Version 1.6, is used as the basis for SET cryptographic encapsulation methods. Review the PKCS #7 documentation cited in "Related documentation" in the Preface.

~~Benefits~~

~~PKCS #7 formats are used to represent the enveloped data in SET messages. ASN.1 and its encoding rules, a set of international standards, are used throughout the PKCS #7 specification. By using ASN.1 to define the SET messages, one format is used throughout the entire SET specification.~~

**PKCS #7 types**

SET uses the following PKCS #7 ASN.1 types:

- *SignedData*, for digitally signed data,
- *EnvelopedData*, for data encrypted with public keys,
- *EncryptedData*, for data encrypted with symmetric keys,
- *DigestedData*, for hashed (or linked) data.

**Implicit certificates and CRLs**

Each signed message contains all certificates and CRLs necessary for the receiver to verify the message signature. Certificates and CRLs are included in the *Certificates* and CRLs fields of the *SignedData* type. ~~When a key-exchange certificate is included, its Thumbprint is also included within the message.~~ (Because BCIs are a SET innovation, no place is allowed for them in PKCS #7. Instead, they are carried in SET fields.)

*Continued on next page*

# PKCS #7 Types, continued

### *SignedData*

The *SignedData* type from PKCS #7 is shown below to aid in understanding the signature process.

Multiple occurrences of *SignerInfos* are permitted within *SignedData*; however, in SET a *SignedData* message is signed by no more than two ~~parties~~ keys (both belonging to the same entity).

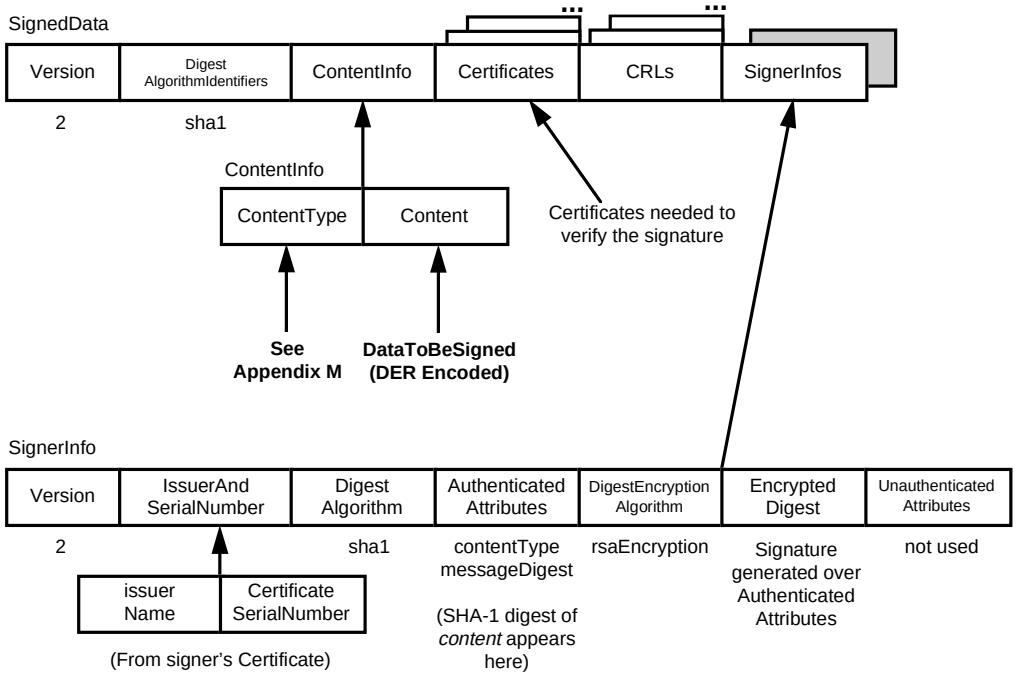


**Figure 5: *SignedData***

### Content

SET includes two operators for type *SignedData*:

| | | | Page |
|---|---|---|---|
| *Signature* | *S ()* | *content* always present | 151 |
| *SignatureOnly* | *SO ()* | *content* always absent | 159 |

### Further detail

Appendix M: "ContentTypes" provides a table of SET messages (or components of messages) with their *content* and *contentType* values used in *SignedData*.

*Continued on next page*

## PKCS #7 Types, continued

**Sample code: *SignedData***

The following sample ASN.1 code shows how to compose *SignedData* using data structure *dataTBS* as the data to be signed using the *Signature* operator. The ASN.1 type of *dataTBS* is identified by *id-set-content-dataTBS.*

```
signedData  SignedData ::= {
     sdVersion  2,
     digestAlgorithms {
          { algorithm   id-sha1,
          parameters  NULL
          }
     },
     contentInfo {
          contentType  "id-set-content-dataTBS",
          content      "dataTBS"
     },
     certificates { ... },
     crls { ... },
     signerInfos {
          {  siVersion  2,
               issuerAndSerialNumber {
                    issuer        "Certificate.issuer",
                    serialNumber  "Certificate.serialNumber"
               },
               digestAlgorithm {
                    algorithm   id-sha1,
                    parameters  NULL
               },
               authenticatedAttributes {
                    { type    contentType,
                    value  "id-set-content-dataTBS"
                    },
                    { type    messageDigest,
                    value  "Digest of dataTBS"
                    }
               },
               digestEncryptionAlgorithm {
                    algorithm   id-rsaEncryption,
                    parameters  NULL
               }   encryptedDigest "Signed authenticatedAttributes"
          }
     }
}
```

*Continued on next page*

# PKCS #7 Types, continued

### Authenticated attributes

SET PKCS #7 *SignedData* always includes two authenticated attributes: *contentType* and *messageDigest*. The attributes may occur in either order; the verifying party must preserve the order while validating the message.

| | |
|---|---|
| contentType | The type of *content* being signed, and therefore protected by the signature. |
| messageDigest | A digest of the *content*. |

Object identifiers have been defined to uniquely identify each SET ASN.1 type that can appear in *SignedData*. See Appendix M: "Content Types."

### Example

Consider the signature on an ASN.1 type named *dataTBS*. The SHA-1 hash of the DER-encoding of this type is computed. An authenticated attributes data structure is created by placing the object identifier *id-set-content-dataTBS* into a *contentType* attribute and the digest of *dataTBS* into a *messageDigest* attribute as shown in the following table.

| | |
|---|---|
| *contentType* | id-set-content-dataTBS |
| *messageDigest* | SHA-1(*dataTBS*) |

The SHA-1 hash of the DER-encoding of the **AttributeSeq** identified by *authenticatedAttributes* (that is, excluding the outer tag [2] and its length) ~~of this data structure~~ is computed and the result encrypted using the signer's private key; it is this encrypted digest that is placed in the *EncryptedDigest* field of the *SignedData* structure.

The object identifier *id-set-content-dataTBS* identifies the content.

*Continued on next page*

# PKCS #7 Types, continued

## EnvelopedData

The *EnvelopedData* type from PKCS #7 is shown below to aid in defining understanding the process of encryption with public keys. Multiple occurrences of *RecipientInfos* are permitted within PKCS #7 *EnvelopedData*; however, only one *RecipientInfo* is used in SET messages.

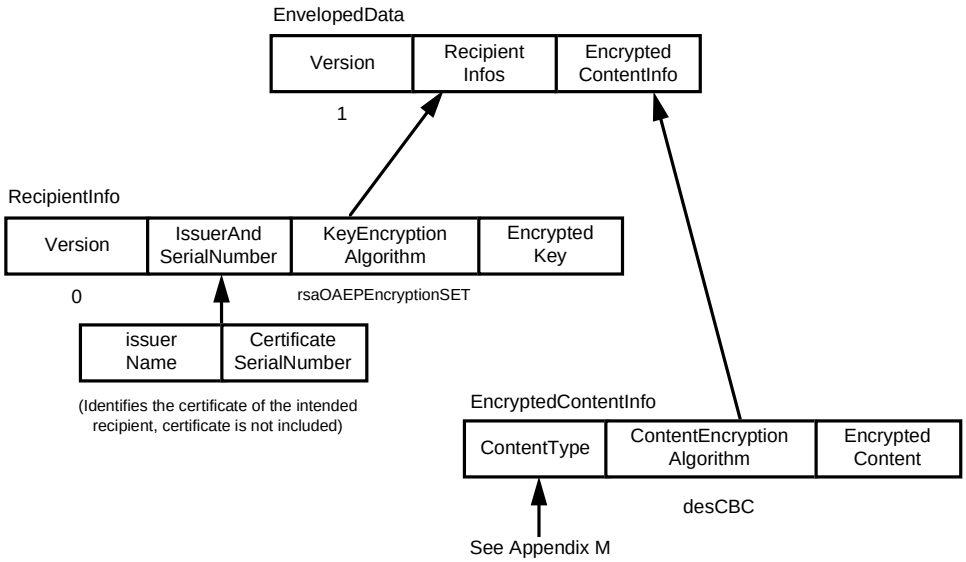


**Figure 6: *EnvelopedData***

## Further detail

Appendix M: "ContentTypes" provides a table of SET messages (or components of messages) with their *content* and *contentType* values used in *EnvelopedData*.

*Continued on next page*

# PKCS #7 Types, continued

**Sample code:**
***EnvelopedData***

The following sample ASN.1 code shows how to compose *EnvelopedData* using data structure *dataTBE* as the data to be enveloped (whose ASN.1 type is identified by *id-set-content-dataTBE)* and optional data structure *extraData*. The *symmetric key* in this sample is a DES key, a known, shared secret between the sender and the receiver.

```
envelopedData  EnvelopedData ::= {
    edVersion  1,
    recipientInfos {
        {  riVersion  0,
            issuerAndSerialNumber {
                issuer        "Certificate.issuer",
                serialNumber  "Certificate.serialNumber"
            },
            keyEncryptionAlgorithm {
                algorithm   rsaOAEPEncryptionSET,
                parameters  NULL
            },
            encryptedKey  "RSA encrypted DES key, extraData"
        }
    },
    encryptedContentInfo {
        contentType  "id-set-content-dataTBE",
        contentEncryptionAlgorithm {
            algorithm   id-desCBC,
            parameters  cbc8Parameter
        },
        encryptedContent  "dataTBE encrypted with DES symmetric key"
    }
}
```

*Continued on next page*

# PKCS #7 Types, continued

**EncryptedData**

The *EncryptedData* construct from PKCS #7 is shown below to aid in defining the process of encryption with symmetric keys.

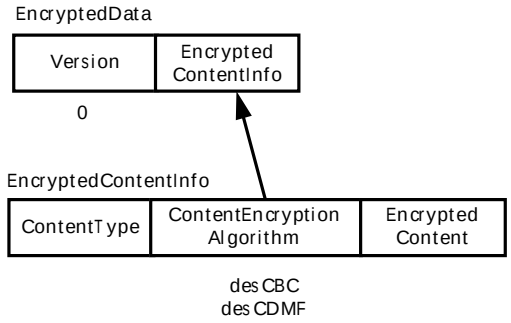


**Figure 7: *EncryptedData***

**Sample code: *EncryptedData***

The following ASN.1 sample code shows how to compose *EncryptedData* using data structure *dataTBE* as the data to be encrypted, whose ASN.1 type is identified by *id-set-content-dataTBE*. The *symmetric key* in this sample is a DES key, a known, shared secret between the sender and the receiver.

```
encryptedData  EncryptedData ::= {
    version  0,
    encryptedContentInfo {
        contentType  "id-set-content-dataTBE",
        contentEncryptionAlgorithm {
            algorithm   id-desCBC,                    -- or id-desCDMF
            parameters  cbc8Parameter
        },
        encryptedContent  "dataTBE encrypted with DES symmetric key"
    }
}
```

**Further detail**

Appendix M: "ContentTypes" provides a table of SET messages (or components of messages) with their *content* and *contentType* values used in *EnvelopedData*.

*Continued on next page*

# PKCS #7 Types, continued

**DigestedData**

The *DigestedData* construct from PKCS #7 is shown below to aid in defining the hashing process.

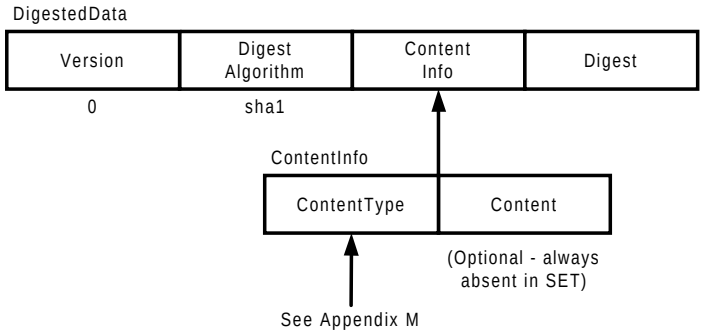


**Figure 8: *DigestedData***

**Sample code: *DigestedData***

The following ASN.1 sample code shows how to compose *DigestedData* as a detached digest (without content) using data structure *dataTBH* as the data to be digested (or hashed), whose ASN.1 type is identified by *id-set-content-dataTBH*.

```
digestedData DigestedData ::= {
    ddVersion  0,
    digestAlgorithm {
       algorithm   id-sha1,
       parameters  NULL
    },
    contentInfo {
       contentType  "id-set-content-dataTBH"
    },
    digest  "SHA-1 hash of the dataTBH"
}
```

**Further detail**

Appendix M: "ContentTypes" provides a table of SET messages (or components of messages) with their *content* and *contentType* values used in *DigestedData*.

# Transaction Validation by Non-SET Systems

**Explanation**

Evidence that a cardholder has participated in a SET transaction is provided in two ways:

- SET participants (Merchant and Payment Gateway) receive the cardholder's digital signature and certificate, which serve as evidence.
- The Issuer does not receive the digital signature and certificate; they are dropped when the Payment Gateway processes the transaction and formats it for use by legacy payment systems. Instead, the Issuer can verify a hash of data known only to the Cardholder and (via the CCA) to the Issuer.

This hash, called the Transaction Stain (**TransStain**), is developed and used as follows:

- During certificate registration, the cardholder sends a secret value, **CardSecret**, to the CCA. ~~This secret value is combined with another secret value generated by the CA to create the secret.~~
- The Cardholder and CCA remember **CardSecret**. Because the CCA is operated on behalf of the Issuer, this value is also available to the Issuer.
- When creating each Payment Request (**PReq**), Cardholder software generates **TransStain** as a hash of the globally unique transaction identifier, **XID**, and **CardSecret**, and includes it in the Payment Instructions. Because the hash includes **XID**, the value changes for every transaction.
- The Payment Gateway sends both **TransStain** and **XID** to the Issuer.

**Cardholders without certificates**

If the Cardholder does not have a certificate, the value of **CardSecret** is zero.

# Optional Fields

## Overview

When a field is marked OPTIONAL in the ASN.1, that field may or may not appear in individual messages. Whether the field appears in a given instance of the message is described in *SET Book 3: Formal Protocol Definition* and in the processing steps in Parts II and III of this book. Whether an application shall (or may) include support for the field is defined in Appendix E: "Field Support Requirements."

# Language

**Specifying Language**

The value of **Language** shall be one of the following, and SET software shall be able to process all of the following:

- a two-character value specified in ISO 639;
- a two-character value specified in ISO 639, a hyphen, and a two-character value specified in ISO 3166; or
- a value registered with IANA in accordance with RFC 1766.

When an application supports multiple variants of a language, one variant shall be designated the primary variant. If an unsupported variant of a language is requested, the primary variant shall be used.

# Date Fields

**Date field format**

Dates in SET are ~~typically~~ indicated in the form of a string representing the calendar date and ~~UTC~~ Greenwich Mean Time, in the format:

YYYYMMDDHHMM~~[~~SS[.f[f[f]]]~~]~~Z GeneralizedTime

YYMMDDHHMM~~[~~SS[.f[f[f]]]~~]~~Z UTCTime

where Z is a literal upper-case letter Z.

That is, the string should consist of:

- a two- or four-digit representation of the year that includes the century (GeneralizedTime uses four digits; UTCTime uses two),
- a two-digit representation of the month,
- a two-digit representation of the day in the month,
- a two-digit representation of the hour (on a 24-hour clock),
- a two-digit representation of the minutes after the hour,
- ~~an optional~~ a two-digit representation of the seconds after the minute,
- an optional representation of fractional seconds, indicated by a decimal point followed by one to three digits, and
- a literal upper-case letter Z.

No separators are used aside from the decimal point in the optional representations of fractional seconds.

Examples:

| | |
|---|---|
| 19960223210600Z | February 23, 1996, 9:06 p.m. |
| 19960223210630Z | February 23, 1996, 9:06:30 p.m. |
| 19960223210630.123Z | February 23, 1996, 9:06:30:123 p.m. |

Midnight shall be represented in the form: YYYYMMDD000000Z, where YYYYMMDD represents the day ~~following~~ that begins with the midnight in question.

Following are examples of invalid representations:

| | |
|---|---|
| 19920520240000Z | invalid - midnight represented incorrectly |
| 19920622123421.0Z | invalid - spurious trailing zeros |

# Amount Fields

### Amount format

Amounts in the SET payment messages are expressed in terms of three fields: *currency*, *amount*, and *amtExp10*.

| Field | Definition |
|---|---|
| *currency* | The value shall be ~~a numeric ASCII string specifying~~ the three-digit ISO 4217 currency code. For example, a payment denominated in U.S. currency will have a currency value of 840. The values shall be between 1 and 999 inclusive. |
| *amount* | The value shall be ~~a numeric ASCII string representing~~ the amount of the payment, specified in terms of the minor unit of the stated currency. The value shall be a non-negative integer. |
| *amtExp10* | The value shall be a ~~numeric ASCII string~~ number representing an exponent base 10 such that<br>`amount * (10 ** amtExp10)`<br>shall be the value of the amount in the ~~minor~~ major unit of the currency ~~specified in ISO 4217~~. The value may be either a negative or positive integer, but is usually between -3 and 0. |

### Example

In order to represent US $2.50 in the **PurchAmt** field, the values for *currency*, *amount*, and *amtExp10* fields are 840, 250, and -2, respectively.

# Chapter 4
# System Concepts

## Overview

### Introduction

Chapter 4 summarizes other important system concepts pertinent to understanding the architecture of SET.

### Organization

Chapter 4 includes the following sections:

| Section | Title | Contents | Page |
|---|---|---|---|
| 1 | Cryptography | Highlights the specific cryptographic algorithms and features. | 85 |
| 2 | Notation and Definitions | Summarizes the notation and conventions used throughout the remainder of this Programmer's Guide. | 93 |
| 3 | Other Features | Describes other features of the design of SET. | 98 |

# Section 1
# Cryptography

## Cryptographic Features

**Default algorithms**

The default algorithms defined for SET are:

| type | algorithm | usage |
|---|---|---|
| asymmetric | RSA with SHA-1 | message signing<br>certificate and CRL signing<br>BCI signing |
| asymmetric | RSA (OAEP) | data encryption |
| symmetric | DES-CBC | data encryption |
| hashing | SHA-1 | message digests and keyed hash |
| keyed hash | HMAC-SHA-1 | Transaction Stain (see page 79)<br>Unique Cardholder ID in Cardholder certificate (see page **Error! Bookmark not defined.** in Part II) |

**Table 14: Default Algorithms**

*Continued on next page*

# Cryptographic Features, continued

### Asymmetric key sizes

| Entity | Message Signing | Key Encryption | Certificate Signing | CRL Signing (also used by the BCA to sign BCI) |
|---|---|---|---|---|
| Cardholder | 1024 | | | |
| Merchant | 1024 | 1024 | | |
| Payment Gateway | 1024 | 1024 | | |
| Cardholder CA | 1024 | 1024 | 1024 | |
| Merchant CA | 1024 | 1024 | 1024 | |
| Payment Gateway CA | 1024 | 1024 | 1024 | 1024 |
| Geopolitical CA | | | 1024 | 1024 |
| Brand CA | | | 1024 | 1024 |
| Root CA | | | 2048 | 2048 |

**Table 15: Asymmetric Key Sizes**

Note: These key sizes may change in future versions of SET; however, because of import and export restrictions, SET applications must ~~shall~~ hard-code these sizes.

*Continued on next page*

# Cryptographic Features, continued

### DES

The Data Encryption Standard (DES-CBC) is the default symmetric-key algorithm used in SET to protect sensitive financial data, such as the payment instruction (**PI**). Originally published in 1977 for use by the United States government to protect valuable and sensitive – but unclassified – data, DES was subsequently adopted by the American National Standards Institute (ANSI) as the Data Encryption Algorithm (DEA).

DES specifies a cryptographic algorithm to encrypt and decrypt 64-bit blocks of data under the control of a unique key. The algorithm is defined in Federal Information Processing Standard (FIPS) 46-2, published by the U.S. National Institute of Standards and Technology (NIST). SET uses the Cipher Block Chaining (CBC) mode of DES, as defined in FIPS 81. The key is 8 bytes long, with each byte having a parity bit in position 0, yielding an effective key length of 56 bits. The standard padding rule shall be used with the DES-CBC mode as described below.

### SET DES-CBC Padding Rule

The SET padding rule for DES-CBC requires that a padding string always be appended to the final plaintext block being encrypted. This final block may be a complete data block, or a partial data block whose length is not an integral multiple of the block length. A padding string is used in SET regardless of whether the final block is a partial or complete data block.

The padding string appended to the final data block makes its length an integral multiple of eight octets. If *BL* represents the length in octets of the final data block, then the padding string consists of 8 - ( || *BL* || mod 8 ) octets. Each octet in the padding string has as its value 8 - ( || *BL* || mod 8 ).

When the length of the padding string is a single octet, the value of that octet is 01. When the length of the string is two octets, the value of the two octets is 02, and the padding string used is '0202'. When the length is three, the value is 03, and the padding string is '030303', and so on.

*Continued on next page*

# Cryptographic Features, continued

**CDMF**

Commercial Data Masking Facility (DES-CDMF) is an alternate symmetric-key algorithm, used in SET as one of the choices to protect Acquirer-to-Cardholder and CA-to-Cardholder messages.

CDMF is a scrambling technique that relies on DES as the underlying cryptographic algorithm, but weakens the overall cryptographic operation by defining a key-transformation method that produces the equivalent of a 40-bit DES key instead of the 56-bit key length required for full strength DES. Since the CDMF algorithm is not as resistant to key exhaustion as DES, CDMF provides a form of data masking rather than data encryption.

The CDMF key transmitted in the SET protocol is the key before being transformed for use in a DES encryption/decryption engine. In other words, a CDMF key is passed just like a normal DES key.

For further information about CDMF, see "Related documentation" in the Preface.

**Hashing algorithm**

Secure Hash Algorithm (SHA-1) is the default hashing algorithm used in SET, including the hashes used in signatures. All references to hash algorithms shall be interpreted as using the SHA-1 hash algorithm defined in FIPS 180-1.

**Keyed hash algorithm**

HMAC-SHA-1 (or simply HMAC) is the default keyed hash algorithm used in SET. It is specified in RFC 2104.

For further information about HMAC, see "Related documentation" in the Preface.

**Digital envelope**

A digital envelope is a generic cryptographic technique to encrypt data and send the encryption key along with the data. Generally, a symmetric-key algorithm is used to encrypt the data and an asymmetric algorithm is used to encrypt the symmetric encryption key.

**OAEP**

SET uses the Bellare-Rogaway Optimal Asymmetric Encryption Padding (OAEP) method in conjunction with its cryptographic encapsulation operators. In addition, SET uses the hashed data technique developed by Matyas and Johnson as an enhancement to the basic Bellare-Rogaway construction. Although OAEP is not directly related to the digital enveloping process, SET toolkits and applications shall apply OAEP prior to encrypting the DES key and optional data using the public key of the receiver.

# Other Cryptographic Implications

**Randomness**

An area of special consideration for developers of SET toolkits and applications is the implementation of random number generation used for keys and nonces. Although a precise definition of randomness is outside the scope of the SET specification, developers of products need to be cognizant of the importance of this aspect in their implementation. Poor key generation and seeding methods due to using weak random numbers are common downfalls of cryptographic implementations. The reader is encouraged to use the recommendations provided in *RFC 1750, Randomness Recommendations for Security*, D. Eastlake, S. Crocker, J. Schiller, December 1994.

For cryptographic purposes, once a strong seed is collected, it shall either be used one time only or it shall be used exclusively in a cryptographically secure random number generator. Also, each instance of random number generation algorithm shall have its own independent key-generation seed.

**Statistically unique field values**

SET defines several field values as "statistically unique." This means that statistically, the odds are extremely small that the same value will be randomly generated twice. The following are among the statistically unique fields used by SET: **XID**, **RRPID**, **EXNonce**, **NonceCCA**, and **ODSalt**.

**Nonce, salt, or freshness challenge**

SET defines several fields as *nonces*, *salts*, or *freshness challenges* to defeat playback attacks. The sending entity shall generate a ~~random~~ statistically unique value and insert this value into the message. The recipient of the message shall copy this value into the corresponding response message. The sending entity will compare the two values to ensure they are the same.

**Algorithm independence**

Although this version of the SET specification is explicit about the cryptographic algorithms that shall be supported by Cardholder, Merchant, and Payment Gateway systems, the protocol's cryptographic encapsulation operators have been designed to be algorithm independent. All ASN.1 algorithm information object sets are coded with the extension marker (…) to allow additional algorithm objects to be added to future versions of the specification, while remaining backward compatible with this version of SET.

*Continued on next page*

# Other Cryptographic Implications, continued

### Hardware tokens

~~Depending on the policies established by the Acquirer and the brand, hardware tokens may also be used by systems supporting SET.~~ A hardware token is defined as a hardware cryptographic module that does not allow disclosure of the private key.

Regarding performing cryptographic functions in hardware tokens:

- CAs shall use hardware tokens for all private-key operations.
- Payment Gateways shall ~~support the~~ use ~~of~~ hardware tokens for all private-key operations~~; their use may be mandated by Acquirer or brand policy~~.
- Merchant software should support the use of hardware tokens; their use may be mandated by Acquirer or payment card brand policy.
- Cardholder software may support the use of hardware tokens.

For more information on hardware tokens, see "Tamper resistant hardware" on page 38.

# Cryptographic Optimization

## Reuse of symmetric DES keys

The computational overhead of generating and processing RSA envelopes can be lessened in certain circumstances by reusing the DES key used to encrypt information. Specifically, when the Merchant encrypts SET messages for transmission to the Payment Gateway or when the Payment Gateway encrypts SET messages for transmission to the Merchant, they may reuse symmetric DES keys with the following restrictions:

- A particular DES key may be reused only between a specific Merchant and Payment Gateway. (For this purpose, the Merchant and Payment Gateway are each defined by a single SET key-encryption certificate.) A key shall not be shared with multiple sources or destinations.
- A particular DES key shall not be used for longer than 24 hours nor more than 1,000 messages. (Keys must be changed at least this frequently to avoid creating an attractive target for cryptographic analysis.)
- A particular DES key shall not be used for both sending and receiving SET messages; that is, each key may be used in only one direction. (This ensures that different random number generators are used for each direction of communication, and reduces the usefulness of analyzing the merchant-to-gateway communications.)

*Continued on next page*

# Cryptographic Optimization, continued

**Optimizing Enc- and EncB-protected messages**

The **Enc** and **EncB** operators are used to encrypt several SET messages transmitted between the Merchant and Payment Gateway. There is an opportunity to optimize the computational cost of the RSA envelope for these messages. This technique is recommended for Merchant and Payment Gateway implementations.

Note: This optimization technique does not apply to the **EncX** and **EncBX** operators, which carry additional data in the RSA envelope; nor to the Cardholder, who does not send multiple messages to the same Payment Gateway.

To create a message, a Merchant or Payment Gateway:

- may optionally remember and reuse DES keys as discussed in "Reuse of symmetric DES keys" above;
- may optionally remember the RSA envelopes created for **Enc** and **EncB** encryptions and associated with remembered DES keys; and
- may optionally reuse previous DES keys and matching RSA envelopes to prepare a new **Enc**-protected or **EncB**-protected message for encryption.~~,~~

To process a message, a Merchant or Payment Gateway:

- may optionally remember DES keys and matching RSA envelopes recovered from incoming messages, and
- may compare an incoming RSA envelope against cached copies of envelopes from previous messages from the same entity. If an incoming envelope is the same as the envelope of a previous message, then the DES key of the previous message can be used to decrypt the DES portion of the current message. This saves the computational cost of decrypting the RSA envelope.

# Section 2
# Notation and Definitions

## Overview

**Purpose**

This section provides a high-level overview of the fundamental cryptographic treatments that are used to describe the payment and certificate processing flows in this Programmer's Guide.

# Notation

**Purpose**

The remainder of this book makes use of the abstract notation described in Table 16.

| Concept | Notation | Definition | |
|---|---|---|---|
| Tuple | **{A, B, C}** | A grouping of zero or more data elements. These represent *documents* or *messages*, terms occasionally used interchangeably with "tuple." Tuples are denoted by *identifiers:* alphanumeric symbols.<br>This notation means "the tuple containing ***A***, ***B***, and ***C***," which may, themselves, be tuples. | |
| Component | **T = {A, B, C}** | A tuple may be given a name as shown, in which case ***T.A***, ***T.B***, and ***T.C*** refer to the respective *components* of ***T***. | |
| Ordered concatenation | **A \| B \| C** | An explicit, *ordered concatenation* of items ***A***, ***B***, and ***C***. | |
| Optional | **[A]** | Item ***A*** is *optional*. | Any other nesting of these brackets is permissible. |
| Selection | **<A, B, C>** | Exactly one of ***A***, ***B***, and ***C*** must appear. | |
| Optional selection | **[<A, B, C>]** | The *selection* is *optional*; that is, that either nothing or exactly one of ***A***, ***B***, and ***C*** may appear. | |
| Multiple instances | **{A +}** | A tuple containing *one or more instances* of ***A*.** | |
| | **{A *}** | A tuple containing *zero or more instances* of ***A***. | |
| | **{[A] +}** | A tuple containing:<br>• *one or more instances* of **A**<br>• in an *ordered array*<br>• where *each instance* of **A** *is optional* (that is, may be NULL). | |
| Exclusive-or | ⊕ | A bit-wise *exclusive-or* (XOR) operation. | |

**Table 16: Notation**

# Notation for Cryptographic Treatments

## Caveat

The following tables introduce the notations for the hashing, signature, encryption, and encapsulation cryptographic treatments which are used throughout the remainder of this Programmer's Guide. For additional information refer to *SET Book 3: Formal Protocol Definition* and "Cryptographic Processing" on page 144.

## Hashing

Table 17 summarizes the notation corresponding to the hashing and hashed-based operators used by SET.

| Notation | Operator | Description |
|---|---|---|
| **HMAC ( *t, k* )** | *Keyed-hash* | 160-bit keyed-hash of tuple ***t*** using key ***k*** based on HMAC-SHA-1<br>**HMAC ( t, k ) =**<br>**SHA-1 (( k ⊕ opad ) \| SHA-1 (( k ⊕ ipad ) \| t ))**<br>where:<br>• **ipad** is the byte 0x36 repeated 64 times; and<br>• **opad** is the byte 0x5C repeated 64 times. |
| **DD ( *t* )** | *DetachedDigest* | 160-bit SHA-1 hash of tuple t. Corresponds to PKCS #7 DigestedData. |
| **L ( *t1, t2* )** | *Linkage* | A ~~reference, pointer, or~~ link to ***t2*** is included with ***t1***; equivalent to the tuple **{ t1, H ( t2 ) }** |

**Table 17: Notation for Hashing and Hash-Based Operators**

## Signature

Table 18 summarizes the notation corresponding to the signature operators used by SET.

| Notation | Operator | Description |
|---|---|---|
| **S ( *s, t* )** | *Signature* | The signature of entity ***s*** on tuple ***t***, including the plaintext of ***t***.<br>Corresponds to PKCS #7 *SignedData*. |
| **SO ( *s, t* )** | *Signature Only* | The signature of entity ***s*** on tuple ***t***, but not including the plaintext of ***t***.<br>Corresponds to PKCS #7 *external signature SignedData*. |

**Table 18: Notation for Signature Operators**

*Continued on next page*

# Notation for Cryptographic Treatments, continued

## Encryption

Table 19 summarizes the notation corresponding to the encryption operators used by SET.

| Notation | Operator | Description |
|---|---|---|
| **E ( *r*, *t* )** | *Asymmetric Encryption* | Corresponds to the standard PKCS #7 *EnvelopedData* with ***t*** encrypted with fresh symmetric key ***k*** and **OAEP ( *k* )** encrypted using the public key of entity ***r***. |
| **EX ( *r*, *t*, *p* )** | *Extra Asymmetric Encryption* | This is like **E** except that ***t*** and ***p*** are the two parts of a message; ***t*** is the tuple subjected to ordinary, symmetric encryption, and ***p*** is a *parameter* subject to "extra" processing. The ***t*** slot is called the *ordinary slot* of **EX**, and the ***p*** slot (which holds **OAEP ({ *k*, *p* })** ) is called the *extra slot* of **EX**. **EX** does not link ***t*** and ***p*** together; the operators derived from **EX** provide the linkage. |
| **EXL ( *r*, *t*, *p* )** | *Extra Asymmetric Encryption with Linkage* | This is like **EX** except that ***t*** is linked to ***p*** and this linkage is subjected to ordinary, symmetric encryption; equivalent to **EX ( r, L ( *t*, *p* ), *p* } )** |
| **EH ( *r*, *t* )** | *Asymmetric Encryption with Integrity* | This is like **E** except that the PKCS #7 envelope contains **OAEP ( { k, H ( *t* ) } )** for a guarantee of integrity when signature is not available. Processing software shall rehash ***t*** and check for match against the **H ( *t* )** in the PKCS #7 envelope. |
| **EXH ( *r*, *t*, *p* )** | *Extra Asymmetric Encryption with Linkage and Integrity* | This is like **EX** except with **OAEP ( { *k*, H ( *t* ), *p* } )** in the PKCS #7 envelope and with the requirement that processing software check **H ( *t* )**, as with **EH.** |
| **EK ( *k*, *t* )** | *Symmetric Encryption* | The symmetric encryption of tuple ***t*** using secret key ***k***.<br>Corresponds to an instance of PKCS #7 *EncryptedData*. |

**Table 19: Notation for Encryption Operators**

*Continued on next page*

# Notation for Cryptographic Treatments, continued

## Encapsulation

Table 20 summarizes the notation corresponding to the encapsulation operators used by SET. These operators combine signature and encryption operators and are used on most messages, facilitating security analysis of this protocol.

| Notation | Operator | Description |
|---|---|---|
| **Enc ( *s*, *r*, *t* )** | *Simple Encapsulation with Signature* | Signed, then encrypted message.<br>**E ( *r*, S ( *s*, *t* ) )**<br>Corresponds to an instance of PKCS #7 *SignedData* encapsulated in *EnvelopedData*. |
| **EncK ( *k*, *s*, *t* )** | *Simple Encapsulation with Signature and a Provided Key* | Signed messages encrypted with a known, secret key.<br>**EK ( *k*, S ( *s*, *t* ) )**<br>Corresponds to an instance of PKCS #7 *SignedData* encapsulated in *EncryptedData*. |
| **EncX ( *s*, *r*, *t*, *p* )** | *Extra Encapsulation with Signature* | Two-part messages encrypted with the first part of the message in the ordinary (symmetric encryption) slot of **E** and the second part of the message in the extra (OAEP) slot of **E**.<br>**EX ( *r*, { *t*, SO ( *s*, { *t*, *p* } ) }, *p* )** |
| **EncB ( *s*, *r*, *t*, *b* )** | *Simple Encapsulation with Signature and Baggage* | Signed, encrypted messages with external baggage.<br>**{ Enc ( *s*, *r*, L ( *t*, *b* ) ), *b* }** |
| **EncBX ( *s*, *r*, *t*, *b*, *p* )** | *Extra Encapsulation with Signature and Baggage* | Signed, **E**-encrypted, two-part messages with baggage.<br>**{ EncX ( *s*, *r*, L ( *t*, *b* ), *p* ), *b* }** |

**Table 20: Notation for Encapsulation Operators**

# Section 3
# Other Features

## Idempotency

**Definition**

When an operation can be executed any number of times, with no harm done, it is said to be *idempotent*. From the SET perspective, idempotency is a property of how a receiver responds to a message.

- Any request in SET for which a response is not received shall be sent again, since it is impossible for the sender to know whether it was the request or the response that was lost. The re-transmitted request shall be bit-wise identical to the original request message.
- Any entity receiving an idempotent request that it has already processed shall re-transmit the original response message.

In general, a duplicate message is not an error condition. (However, see "Response to attacks" on 101.)

**Rationale**

The SET protocol is designed to work in environments where message delivery is not guaranteed. If a SET application does not receive a response in a reasonable period of time (as defined by the application or possibly in response to a user query), it sends the message again. When the receiving SET application determines that it has previously processed the same message, it retrieves the previous response and sends it again.

Not all SET messages require idempotency. The Inquiry Request, for example, has been designed to be sent at any time so it is not necessary for a Merchant to store every inquiry request to determine if a duplicate is received; it simply returns the current status of the transaction in the Inquiry Response. On the other hand, the Purchase Request does require idempotency.

A summary of per-message idempotency requirements is provided in Appendix C: "SET Messages."

*Continued on next page*

# Idempotency, continued

**Retention of messages**

SET applications shall retain copies of idempotent request and response messages that have been processed.

- A requester shall retain a bit-wise identical copy of a request message until a response has been received or the application abandons processing of the message.
- A responder shall retain a bit-wise identical copy of a response message – as well as the request to which it is responding – for a reasonable period of time after the most recent transmission of the response. The length of time is determined by the application developer based on the operating environment.

| | |
|---|---|
| **interactive** (such as World Wide Web) | A reasonable retention period will be measured in hours; for example, the application could allow the installation configuration to specify a retention period of one to four hours. |
| **non-interactive** (such as electronic mail) | A reasonable retention period will be measured in days; for example, the application could allow the installation configuration to specify a retention period of one to fourteen days. |

If the retention period can be configured at the installation, the application shall enforce a minimum retention period based on the operating environment.

Note that the retention time for idempotent messages should be no less than the timeout period for rejecting messages as too old or too new.

**Description**

SET applications shall guarantee idempotency of the protocol by examining ~~transaction (**XID**) and~~ request/response pair (**RRPID**) identifiers. The applications must distinguish between those requests that are bit-wise identical (idempotent) and those that are either processing errors or attempts at fraud.

For example, a Payment Gateway will ~~reject attempts to replay authorization requests from merchants. It will detect these attempts by examining the **RRPID** of the authorization request and **XID** of the embedded payment instruction, separately signed (or hashed) and encrypted by the cardholder.~~ examine incoming Authorization Requests to detect duplicate **XID/RRPID** pairs and either send an identical response message (for bit-wise identical requests) or reject an attempt by the merchant to reuse Payment Instructions for a second Authorization Request.

*Continued on next page*

# **Idempotency,** continued

**Detecting idempotent requests**

An application may use any method to detect idempotent requests.

One possibility is to store the **RRPID** and SHA-1 hash of all messages. When a duplicate **RRPID** is detected, the hash of the message can be generated and compared against the stored value with a match indicating an idempotent request. Using this approach, the application does not have to keep a complete copy of each incoming request; however, it shall be capable of generating bit-wise identical responses.

**Duplicate responses**

If multiple responses to an idempotent request are received, the recipient can ignore all such responses after the first one.

*Continued on next page*

# Idempotency, continued

**Response to attacks**

A SET application is not required to respond to messages when it detects that it is being subjected to a malicious flooding or spamming attack involving one or more idempotent SET messages types. Application developers can establish their own criteria to detect such malicious attacks. For example, an application might consider more than ten repetitions in less than a minute to be a malicious attack.

**Scenarios**

Descriptions of likely scenarios involving idempotent messages follow. These scenarios depict two transmissions of the same request, but depending on the conditions at the time of the failure, the request may be repeated many times. Combinations of these scenarios are also possible.

**Delayed request or response**

An idempotent request is sent, but delivery of either the request or the response is delayed so the receiver does not receive a timely response. The request is transmitted again. The receiving system processes the first request, then re-transmits that response when it receives the second request.


**Lost request**

An idempotent request is sent, but the delivery of the message does not occur because of a network failure. The request is transmitted again.


**Lost response**

An idempotent request is sent and responded to, but the delivery of the response does not occur because of a network failure. The request is transmitted again. The receiving system processes the first request, then re-transmits that response when it receives the second request.

# Special Fields

### XID

**XID** is intended as a statistically unique identifier assigned to a payment transaction so that all messages of the transaction can be related to one another. It is a 20-byte string.

**XID** does not change during the life of a transaction. The only exception is for a credit on a transaction that has aged off of the merchants logs; in that case:

- The merchant generates a new **XID** (because the original **XID** is unknown).
- The merchant sets the value of **AuthRRPID** to zero (because the original **AuthRRPID** is unknown).

### XID example

The example below demonstrates the correct **XID** and **AuthRRPID** throughout a transaction which includes a credit request.

| | |
|---|---|
| 1. | Cardholder sends **PReq** (**XID** = 5) to Merchant. |
| 2. | Merchant sends **AuthReq** (**XID** = 5, **AuthRRPID** = 10) to Payment Gateway. |
| 3. | Payment Gateway sends **AuthRes** (**AuthRRPID** = 10) to Merchant. |
| 4. | Merchant sends **PRes** to Cardholder. |
| 5. | Merchant sends **CapReq** with one item (**XID** = 5, **AuthRRPID** = 10). |
| 6. | Payment Gateway sends **CapRes** (**XID** = 5, **AuthRRPID** = 10). |

Later the Merchant submits a credit:

| | *If the original transaction is still available in the Merchant's log, the credit request includes the original* **XID** *and* **AuthRRPID***:* | *If the original transaction has aged off the Merchant's log, the credit request includes a new* **XID** *and* **AuthRRPID***:* |
|---|---|---|
| 7. | Merchant sends **CredReq** (**XID** = 5, **AuthRRPID** = 10). | Merchant sends **CredReq** (**XID** = 20, **AuthRRPID** = 0). |
| 8. | Payment Gateway sends **CredRes**. | |

*Continued on next page*

# Special Fields, continued

### BrandID

**BrandID** is an important field used in both the payment and certificate management protocol messages. It has two components:

| | |
|---|---|
| *brand name* | the brand of the payment card |
| *product* | optional: the type of product within the brand |

When *product* is included, it is separated from *brand name* by a colon (:) as follows:

brand name[:product]

In messages, **BrandID** shall be encoded using VisibleString if possible.

In certificates, **BrandID** shall be encoded using PrintableString if possible.

The separator (colon) between *brand name* and *product* is encoded as:

- 0x3A if **BrandID** is a VisibleString or PrintableString; or
- 0x003A if **BrandID** is a BMPString.

# Root Public Key Distribution

**Significance of Root certificate**

The security of the SET system depends ultimately on the authenticity of the certificates used in the system. These certificates are verified by checking a chain of certificates, with the final certificate in the chain being a single system-wide entity. Only through trust in the Root certificate will trust in the SET system be maintained.

**Initial distribution of Root key**

The Root public key is initially distributed as a certificate with the SET software. This certificate shall also contain a hash of the next Root public key. The initial distribution of the Root certificate shall be self-signed and ~~shall~~ may be verified by an out-of-band mechanism (as described in "Root key distribution and authentication" in Part II on page **Error! Bookmark not defined.**). The chaining process for the Root certificates is based on hash values rather than the distinguished name and serial number of the previous Root certificate.

If a Root key/certificate is not represented by the hash within the previous certificate, it shall be treated like the initial Root certificate and requires out-of-band verification.

**Root key update**

The Root key may be updated implicitly using the SET protocol; that is, in the course of receiving ordinary SET transaction messages, you may receive a new Root key when necessary. This is described in detail in "Root Certificate Update" in Part II on page **Error! Bookmark not defined.**.

# Off-line Certificates

**Certificate provision off-line**

In the case of orders that are created off-line, such as those envisioned with CD-ROM shopping, abbreviated protocols may be used that omit the initialization phase between the Cardholder and Merchant. During this phase, the Merchant determines which certificates the Cardholder already possesses and sends the Cardholder any missing certificates. With abbreviated protocols expected in off-line shopping, these certificates will be delivered off-line (for example, in the CD-ROM catalog).

# Cert-PE

**Definition**

**Cert-PE** is the certificate generated by the PCA that binds the Payment Gateway to the proposed encryption public key provided in a certificate request (**CertReq**) message. **Cert-PE** is used by other SET entities as follows:

- The Payment Gateway sends **Cert-PE** to the Merchant in any response message, if the Thumbprints sent in the Merchant's request indicate that a new **Cert-PE** is needed.
- The Merchant sends **Cert-PE** to the Cardholder on behalf of the Payment Gateway.
- The Cardholder uses **Cert-PE** to encrypt the Payment Instructions, regardless of whether the cardholder has a certificate or not.

~~The **certThumbs** will include the Thumbprint corresponding to **Cert-PE**.~~ Although **Cert-PE** is not referenced explicitly in any SET message, it is an optional certificate that may be included in the PKCS #7 *SignedData* block of ~~the corresponding certificate response (**CertRes**)~~ any SET message when it is necessary to transmit the certificate. The corresponding Thumbprint, **PEThumb**, appears in the message to indicate which of the included certificates is **Cert-PE**.

# Secure Data Storage

**Data to store securely**

Certain data requires extra protection and shall be stored in secure data storage, including:

| | Cardholder | Merchant | Payment Gateway | CA |
|---|---|---|---|---|
| private keys | **x** | **x** | **x**[1] | **x**[1] |
| **CardSecret**[2] | **x** | | | **x** |
| payment card number and expiration date | **x** | **x**[3] | **x** | **x** |
| **AcqBackKeyData** | | | **x** | |

**Notes**

1. Secret key generation and storage shall use tamper resistant hardware cryptographic modules. See pages 41 and 42.
2. **CardSecret**, the shared secret between Cardholder and Issuer, is described in "Transaction Validation by Non-SET Systems" on page 79.
3. Merchants receive payment card information only if *MerAuthFlag* in the *MerchantData* private extension of the Merchant certificate is TRUE. See "MerchantData Private Extension" on page **Error! Bookmark not defined.** in Part II.

# Chapter 5
# ~~Section 4~~
# Processing

## Overview

### Purpose

This chapter describes step-by-step processing of common cryptographic treatments, as well as other common processing used by the payment and certificate management protocol descriptions in this Programmer's Guide.

### Organization

This chapter includes two sections:

| Section | Contents | Page |
|---|---|---|
| Non-Cryptographic Processing | Provides processing descriptions for sending and receiving a message, creating and processing Thumbprints, comparing BrandIDs, certificate chain validation, and SET error processing. | 109 |
| Cryptographic Processing | Describes processing for cryptographic treatments and operators. | 144 |

### Guideline

In general, be strict generating; be permissive receiving. That is, generate messages that precisely match your understanding of the protocol, but when processing inbound messages, avoid imposing unnecessary restrictions.

For example, one application may strip trailing spaces while another does not. Regardless of the way your application handles trailing spaces, be prepared to deal with messages from applications that have made a different choice.

Note: Any data that is subject to authentication (hashing or signing) is not flexible and the recipient's copy must be the same as the sender's copy.

# Section 1
# Non-Cryptographic Processing

## Overview

### Organization

This section includes the processing descriptions listed below.

The description of cryptographic processing begins on page 144.

# Send Message

**Create MessageWrapper**

This procedure represents the standard processing required each time a message is sent. SET applications shall implement this procedure, or functionally equivalent procedures, for all messages sent.

| Step | Action |
|---|---|
| 1 | Receive as input: |

| | |
|---|---|
| ***recip*** | the recipient of the message |
| ***msg*** | the SET **Message** |
| ***ext*** | an instance of *MsgExtensions* (optional) |
| ***rrpid*** | the **RRPID** included in the message (optional: may not be available when sending **Error** message) |
| ***lid-C*** | Cardholder's local ID (optional) |
| ***lid-M*** | Merchant's local ID (optional) |
| ***xID*** | globally unique ID (optional) |

*Continued on next page*

# **Send Message,** continued

**Create MessageWrapper** (continued)

| Step | Action |
|---|---|
| 2 | Construct *messageIDs*:<br>*lid-C* — lid-C if present<br>*lid-M* — lid-M if present<br>*xID* — xID if present |
| 3 | Construct *MessageHeader*:<br>*version* — 1<br>*revision* — 0<br>*date* — the current date and time (see "System Clock Differences" on page 66)<br>*messageIDs* — result of Step 2<br>*rrpid* — ***rrpid***<br>*swIdent* — vendor software identification |
| 4 | Construct *MessageWrapper*:<br>*messageHeader* — the result of Step 3<br>*message* — ***msg***<br>*mwExtensions* — ***ext*** |
| 5 | If the message requires idempotency processing (as described in "Idempotency" on page 98), save the result of Step 4. |
| 6 | Pass the message from Step 4 to the transport mechanism for delivery to ***recip***. Depending on the transport mechanism, the message may be further wrapped (for example, with a MIME or HTTP header). |

*Continued on next page*

# Send Message, continued

### MessageWrapper data

| | |
|---|---|
| **MessageWrapper** | **{MessageHeader, Message, [MWExtensions]}** |
| **MessageHeader** | **{Version, Revision, Date, [MessageIDs], [RRPID], SWIdent}** |
| **Message** | **<**<br>**PInitReq, PInitRes,**<br>**PReq, PRes,**<br>**InqReq, InqRes,**<br>**AuthReq, AuthRes,**<br>**AuthRevReq, AuthRevRes,**<br>**CapReq, CapRes,**<br>**CapRevReq, CapRevRes,**<br>**CredReq, CredRes,**<br>**CredRevReq, CredRevRes,**<br>**PCertReq, PCertRes,**<br>**BatchAdminReq, BatchAdminRes,**<br>**CardCInitReq, CardCInitRes,**<br>**Me-AqCInitReq, Me-AqCInitRes,**<br>**RegFormReq, RegFormRes,**<br>**CertReq, CertRes,**<br>**CertInqReq, CertInqRes,**<br>**Error**<br>**>** |
| **MWExtensions** | *Appropriate where:*<br>• *the data in the extension is general purpose information about SET messages, or*<br>• *the contents of the message are encrypted and the extension contains non-financial data that does not require confidentiality.*<br>*Note: The message wrapper is not encrypted so this extension must not contain confidential information. Also, there is no implicit integrity checking on the contents; it is the responsibility of the extension definition to include integrity checking if it is necessary.* |

**Table 21: MessageWrapper Data**

*Continued on next page*

## Send Message, continued

**MessageWrapper data** (continued)

| | |
|---|---|
| **Version** | *Version of SET message* |
| **Revision** | *Minor revision of SET message* |
| **Date** | *Date and time of message generation* |
| **MessageIDs** | **{[LID-C], [LID-M], [XID]}** |
| **RRPID** | *Request/response pair ID for this cycle* |
| **SWIdent** | *String identifying the software (vendor and version) initiating the request.* |
| **LID-C** | *Local ID; convenience label generated by and for Cardholder system* |
| **LID-M** | *Local ID; convenience label generated by and for Merchant system* |
| **XID** | *Globally unique ID generated by Merchant in* **PInitRes** *or by Cardholder in* **PReq** |

**Table 21: MessageWrapper Data,** continued

# Receive Message

~~Receiving entity responsibilities~~

~~The receiving entity shall ensure that the message contents have been properly formatted and encapsulated based on the message type. Additional data such as certificates, CRLs, and BCIs shall be extracted from the message to authenticate any digital signatures applied by the sending entity. The receiving entity's system cache should be updated to reflect these new certificates, CRLs, and BCIs.~~

### Process MessageWrapper

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***msgWrpr*** — an instance of *MessageWrapper* (received from the transport layer ~~Depending on the transport mechanism,~~ with the ~~a~~ transport wrapper ~~may need to be~~ removed) |
| 2 | If the message is too big to be processed, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *messageTooBig* |
| 3 | DER decode ***msgWrpr*.messageHeader**. If decoding fails, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *badMessageHeader*<br>Note: The application will be unable to populate **MessageWrapper** fields in the **Error** message. |
| 4 | Validate the following contents of ***msgWrpr*.messageHeader**:<br>*version* — 1<br>*revision* — 0<br>*date* — a date and time within the range supported by the application<br>If errors are encountered during the validation process, invoke "Create **Error** Message" on page 137 with the following input based on the field that failed:<br>***errorCode*** — *version* or *revision* — *versionTooOld* or *versionTooNew*<br>***errorCode*** — *date* — *messageTooOld or messageTooNew*<br>For further information, see "Backward Compatibility" on page 65 and "System Clock Differences" on page 66. |

*Continued on next page*

# Receive Message, continued

**Process MessageWrapper** (continued)

| Step | Action |
|---|---|
| 5 | Determine the type of **Message**. (Depending on the application's implementation of ASN.1, this may require DER decoding of ***msgWrpr*.message** as described in Step 10.) If the message type is not supported by the application, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** = *messageNotSupported*<br>(See Appendix C: "SET Messages" for a description of mandatory and optional messages.) |
| 6 | If the message type does not require idempotency processing as defined in Appendix C: "SET Messages," continue with Step 10. |
| 7 | If the message is a response:<br>• Determine if a request for the same **RRPID** has been transmitted. If not, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** = *unknownRRPID*<br>• Determine if this is the first response received. If a response for the same **RRPID** has been successfully processed, stop processing the message.<br>• Continue with Step 10. |

*Continued on next page*

# Receive Message, continued

**Process MessageWrapper** (continued)

| Step | Action |
|---|---|
| 8 | Determine if a previous request has been processed with the same **RRPID**. If not, continue with Step 10. |
| 9 | Compare the new request to the previously-processed request with the same **RRPID**. See "Detecting idempotent requests" on page 100 for additional information.<br>If the requests are bit-wise identical, send a bit-wise identical copy of the prior response and stop processing the message.<br>If the request is not a duplicate, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *idempotencyFailure* |
| 10 | DER decode ***msgWrpr*.message**. If decoding fails, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *decodingFailure* |
| 11 | If ***msgWrpr*.mwExtensions** contains a critical message extension that the application does not recognize, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *unrecognizedExtension*<br>***errorOID*** — the *extnID* field of the extension |
| 12 | Purge the untrusted cache.<br>Periodically, purge trusted cache of expired certificates, CRLs, and BCIs. |
| 13 | Process the message by invoking the procedure indicated in Table 22 on page 118 with the following input:<br>***hdr*** — ***msgWrpr*.messageHeader**<br>***msg*** — ***msgWrpr*.message**<br>***ext*** — ***msgWrpr*.mwExtensions** |

*Continued on next page*

# Receive Message, continued

### Processing by Message

| To process **Message**: | ...see Page: |
|---|---|
| **PInitReq** | **Error! Bookmark not defined.** |
| **PInitRes** | **Error! Bookmark not defined.** |
| **PReq** | **Error! Bookmark not defined.** |
| **PRes** | **Error! Bookmark not defined.** |
| **InqReq** | **Error! Bookmark not defined.** |
| **InqRes** | **Error! Bookmark not defined.** |
| **AuthReq** | **Error! Bookmark not defined.** |
| **AuthRes** | **Error! Bookmark not defined.** |
| **AuthRevReq** | **Error! Bookmark not defined.** |
| **AuthRevRes** | **Error! Bookmark not defined.** |
| **CapReq** | **Error! Bookmark not defined.** |
| **CapRes** | **Error! Bookmark not defined.** |
| **CapRevReq** | **Error! Bookmark not defined.** |
| **CapRevRes** | **Error! Bookmark not defined.** |
| **CredReq** | **Error! Bookmark not defined.** |
| **CredRes** | **Error! Bookmark not defined.** |
| **CredRevReq** | **Error! Bookmark not defined.** |
| **CredRevRes** | **Error! Bookmark not defined.** |
| **PCertReq** | **Error! Bookmark not defined.** |
| **PCertRes** | **Error! Bookmark not defined.** |
| **BatchAdminReq** | **Error! Bookmark not defined.** |
| **BatchAdminRes** | **Error! Bookmark not defined.** |

**Table 22: Processing by Message**

# Thumbprints

## Overview

As described on page 68, Thumbprints have several uses in SET. This section provides procedures for some of the uses as follows:

| for this use: | procedure included here: | procedure elsewhere: |
|---|---|---|
| to minimize certificates, CRLs, and BCIs exchanged | "Create set of Thumbprints for request"<br>"Process set of Thumbprints in request" | |
| to help ensure that an unsigned message was not altered | | The Thumbprints are copied from the request to the response as described in the processing steps for the specific message in Part II and Part III. |
| to indicate specific certificates included in a message | "Process single Thumbprint" | Instructions for including the Thumbprint in the message are included in the processing steps for the specific message in Part II and Part III. |
| to indicate a certificate, CRL, or BCI that caused processing to fail | | "Certificate Chain Validation" on page 123 |

*Continued on next page*

# Thumbprints, continued

### Create set of Thumbprints for request

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***brand*** — the brand of the transaction<br>***bin*** — the BIN of the transaction<br>This procedure uses the following internal variables:<br>***thumbList*** — Thumbprints (of certificates, CRLs, and BCIs) to be included in the request message |
| 2 | Initialize ***thumbList*** so that it contains zero entries. |
| 3 | For each certificate in the trusted cache ~~that is pertinent for processing the response message and for validating the certificate chain~~, examine the *CertificateType* extension. If the certificate type is:<br>*rca* — Add the Thumbprint of the certificate to ***thumbList***.<br>*bca* or *gca* — If *brand name* in the *subject* Name matches ***brand***, add the Thumbprint of the certificate to ***thumbList***.<br>*mer* — If:<br>• this is a cardholder application, and<br>• the merchant name in the *MerchantData* extension matches the merchant's name, and<br>• the *brand name* in the *subject* Name matches ***brand***,<br>then add to ***thumbList*** the Thumbprint of the certificate and of any certificate in its chain below the brand CA.<br>*pgwy* — If:<br>• the *brandID* in the *subject* Name matches ***brand***, and<br>• the BIN in the *subject* Name matches ***bin*** (if provided),<br>then add to ***thumbList*** the Thumbprint of the certificate and of any certificate in its chain below the brand CA. |
| 4 | For each CRL in the trusted cache ~~that is pertinent for processing the response message and for validating the certificate chain~~ whose *brandID* in the Issuer field matches ***brand***, add its Thumbprint to ***thumbList***. |
| 5 | For each BrandCRLIdentifier in the trusted cache ~~that is pertinent for processing the response message and for validating the certificate chain~~ whose *brandID* matches ***brand***, add its Thumbprint to ***thumbList***. |
| 6 | Return ***thumbList***. |

# Comparing BrandIDs

### Comparison of BrandIDs

**BrandID** contains a *brand name* and an optional *product*. Instances of **BrandID** appear both in message fields and in certificate *subject* fields.

While processing messages, SET applications must compare two instances of **BrandID** to determine if they match. This comparison is complicated by two factors:

- the presence of the *product* component is optional; and
- the value can be encoded as either VisibleString or BMPString in SET messages; and
- the value can be encoded as either PrintableString or BMPString in SET certificates.

The procedure to compare two instances of **BrandID** is provided below.

Note: Requirements for encoding of **BrandID** are given on page 103.

### Compare BrandIDs

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***hier*** — Boolean: TRUE if there is a hierarchical relationship between ***brand1*** and ***brand2*** (for example, if one **BrandID** belongs to a CA certificate and the other to an end entity certificate)<br>***brand1*** — if ***hier*** is TRUE, the **BrandID** that is higher in the hierarchy; otherwise, either of the two instances of **BrandID** to be compared<br>***brand2*** — if ***hier*** is TRUE, the **BrandID** that is lower in the hierarchy; otherwise, either of the two instances of **BrandID** to be compared |
| 2 | If one of ***brand1*** and ***brand2*** uses BMPString and the other does not (as indicated by the DER tag of each), continue with Step 3. Otherwise, convert ***brand1*** and ***brand2*** to BMPString.<br>*Note to reviewers: this will be updated to describe how to do the conversion to BMPString.* |
| 3 | Compare ***brand1***.*brand* to ***brand2***.*brand*. (As used here, "*brand*" indicates the beginning of **BrandID**, up to but not including the colon.) If they do not match, then compare fails and processing stops. |
| 4 | If both ***brand1***.*product* and ***brand2***.*product* exist, compare them. If they do not match, then compare fails and processing stops. ("*product*" indicates that part of **BrandID** that follows the colon, if present.) |
| 5 | If ***hier*** is FALSE, continue with Step 7. |
| 6 | If ***brand1***.*product* exists but ***brand2***.*product* does not, then compare fails and processing stops. |
| 7 | The comparison of the two instances of **BrandID** is acceptable. |

# Certificate Chain Validation

**Purpose**

Each SET application shall fully validate each certificate, CRL, and BCI prior to adding it to the application's trusted cache or using it in SET processing.

**Certificate chain definition**

Each certificate is linked to the signature certificate of the signing CA. The path through which the certificates are validated is called the "certificate chain."

The SET certificate chain is comprised of the set of certificates from the end entity to the Root certificate, plus all of the Root's predecessors back to the initial Root certificate.

**Overview**

The validation of the certificate chain requires that:

- each certificate in the path – from the end entity certificate through the initial Root certificate – is validated, and
- each certificate correctly maps to the CA that issued the certificate.

**Processing**

Validation requirements shall be enforced for all levels of the chain. For example, a Cardholder application shall validate the Merchant, Merchant CA, Geopolitical CA (if applicable), Brand CA, and Root CA certificates ~~and related payment card brands~~.

The validation process includes:

- Brand CRL Identifier (BCI) processing
- Certificate Revocation List (CRL) processing
- X.509 certificate validation
- SET certificate validation

In practice, it is assumed that the validation process will stop at a level that has been previously validated.

All SET software shall validate certificate dates as part of the certificate chain validation process. SET software shall provide a warning mechanism for expiring certificates and shall prevent their attempted use after expiration.

**Year 2000**

For dates and times, X.509 certificates use UTCTime, which has a two-digit year. SET relies on the consensus in the X.509 community (at the time the specification was published) that the two-digit year in dates for certificates and CRLs specifies a year between 1950 and 2049.

*Continued on next page*

# Certificate Chain Validation, continued

**X.509 requirements**

SET certificate chain validation is performed according to the processing requirements specified in Section 12.4.3 of Amendment 1 to X.509 as well as the additional SET requirements specified below.

**Comparing EE certificate to signing CA certificate**

In addition to the certificate chain processing requirements of X.509, the following SET constraints on the certificate chain shall be met:

- The *authorityCertIssuer* and *authorityCertSerialNumber* fields in the *authorityKeyIdentifier* extension of the subordinate certificate shall match the *issuer* Name and *serialNumber* fields of the signing CA certificate.
- The *validity* dates in the subordinate certificate and in its *privateKeyUsagePeriod* extension shall be within the *validity* dates of the signing CA certificate.
- The *notBefore validity* date in the subordinate certificate shall be within the *validity* dates in the *privateKeyUsagePeriod* extension of the signing CA certificate.
- The *organizationName* of the *subject* Name of each certificate shall meet the criteria described in "Comparing BrandIDs" on page 121.
- The *CertPolicyId* of the *certificatePolicies* extension of the certificates shall be related as described in "Certificate generation" on page **Error! Bookmark not defined.** in Part II.
- The signature verifies (that is, the *issuer* Name of the subordinate certificate matches the *subject* Name of the signing CA certificate).

**End entity certificate validation**

In addition to the certificate chain processing requirements of X.509, the following SET requirements for end entity certificates shall be validated:

- The *cA* field of the *basicConstraints* extension is FALSE, indicating end entity.

**All certificate validation**

In addition to the certificate chain processing requirements of X.509, the following SET requirements for ~~CA~~ all certificates shall be validated:

- The *KeyUsage* field of the *keyUsage* extension is valid for the intended purpose.
- The *certificateType* private extension corresponds with the context in which the certificate is being used.
- All required extensions are present. See "End Entity Certificate Extension" on page **Error! Bookmark not defined.** and "CA Certificate Extensions" starting on page **Error! Bookmark not defined.** in Part II.

*Continued on next page*

# Certificate Chain Validation, continued

### Diagram of certificate comparison

Figure 9 provides a logical view of the certificate data elements, with an emphasis on the data elements used for certificate chain validation. Arrows indicate some of the fields which are compared.

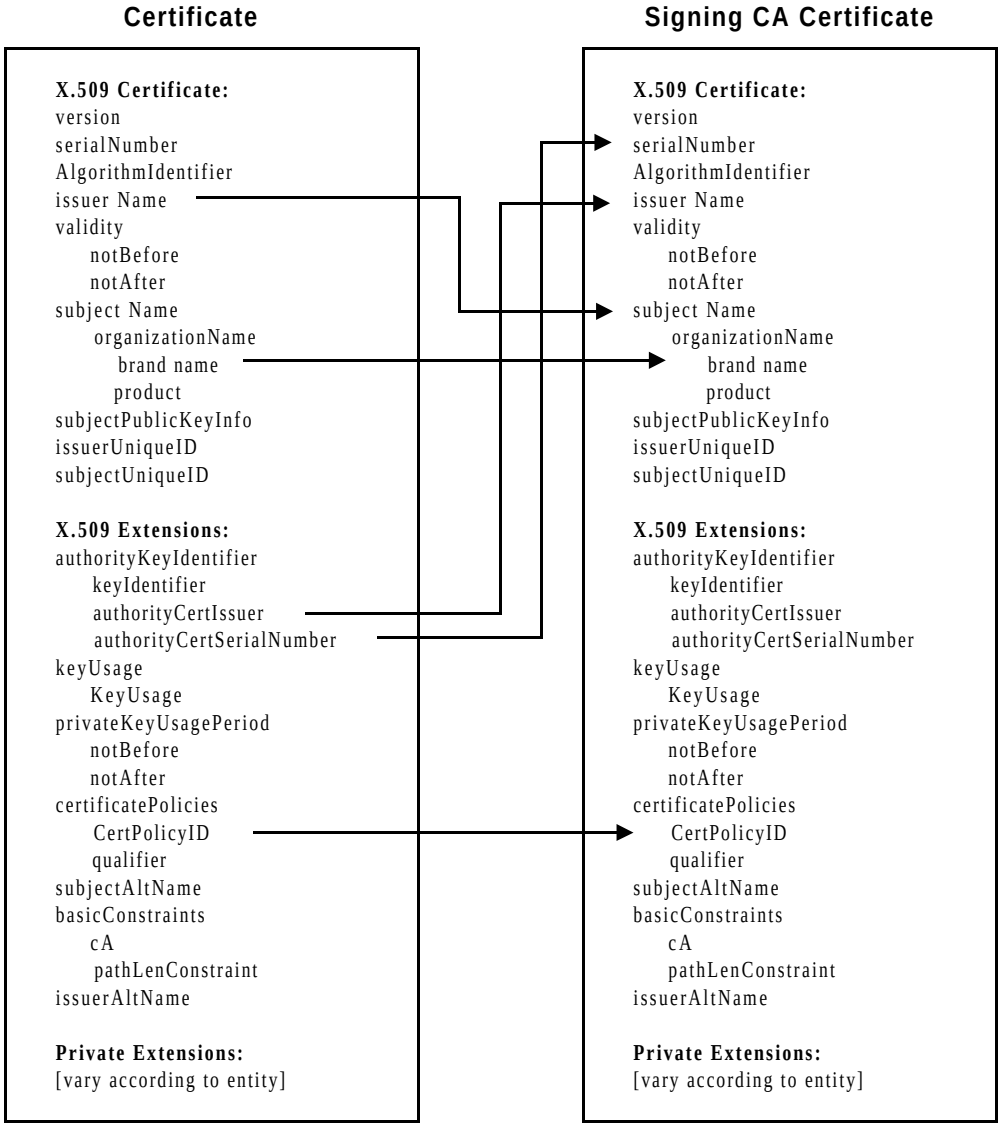


**Figure 9: Certificate Comparison**

*Continued on next page*

# Certificate Chain Validation, continued

### Verify BCI

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***newBci*** — a new *brandCRLIdentifier* (optional)<br>***brand*** — the payment card brand whose BCI is being verified<br>This procedure uses the following internal variables:<br>***bci*** — the current BCI |
| 2 | Retrieve the BCI for ***brand*** from the trusted cache and designate it as ***bci***. |
| 3 | If ***newBci*** is specified, continue with Step 4. Otherwise, continue with Step 12. |
| 4 | If ***newbci.***sequenceNum is greater than ***bci.***sequenceNum, continue with Step 5. Otherwise, remove ***newBci*** from the untrusted cache and continue with Step 12. |
| 5 | Validate the following contents of ***newBci***:<br>*algorithm* / *algorithm* = sha1-with-rsa-signature<br>*parameters* = NULL<br>If errors are encountered during the validation process, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** = *unsupportedAlgorithm*<br>***errorThumb*** = SHA-1 hash of ***newbci.***toBeSigned<br>***errorOID*** = the *algorithm* field |
| 6 | Locate the Brand CA CRL signing certificate.<br>• Search the trusted cache for a certificate whose:<br>  ◦ *certificateType* is *bca*, and<br>  ◦ *KeyUsage* includes *crlSign*.<br>• If no certificate was found, search the untrusted cache for a certificate matching those criteria. If found, invoke "Verify Certificate" on page 131 with the following input:<br>***cert*** — the certificate from the untrusted cache<br>If no certificate was found, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** = *missingCertificateCRLorBCI*<br>***errorThumb*** = SHA-1 hash of ***newbci.***toBeSigned |

*Continued on next page*

# Certificate Chain Validation, continued

**Verify BCI** (continued)

| Step | Action |
|---|---|
| 7 | Decrypt ***newbci***.*signature* using the public key from the certificate found in Step 5. |
| 8 | Compute the SHA-1 hash of ***newbci***.*toBeSigned*. |
| 9 | Compare the result of Step 7 to the result of Step 8. If the values are not the same, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *invalidSignature*<br>***errorThumb*** — the result of Step 8 |
| 10 | If the present date and time are not between ***newBci***.*notBefore* and ***newBci***.*notAfter*, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *invalidCertificateCRLorBCI*<br>***errorThumb*** — the result of Step 8 |
| 11 | Delete ***bci*** and its thumbprint from the trusted cache. Move ***newBci*** and its Thumbprint to the trusted cache.<br>Designate ***newBci*** as ***bci***.<br>Continue with Step 13. |

*Continued on next page*

# Certificate Chain Validation, continued

**Verify BCI** (continued)

| Step | Action |
|---|---|
| 12 | If the present date and time are outside the range of ***bci.****notBefore* and ***bci.****notAfter*, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *invalidCertificateCRLorBCI*<br>***errorThumb*** — the Thumbprint of ***bci*** |
| 13 | For each CRL named on the BCI:<br>• Search the trusted cache for a CRL whose<br>  • *issuer* Name matches ***bci****.crlIdentifierSeq.issuerName* and<br>  • *CRLNumber* extension matches ***bci****.crlIdentifierSeq.crlNumber*.<br>• If no CRL was found, search the untrusted cache for a CRL matching those criteria. If found, invoke "Verify CRL" on page 129 with the following input:<br>***newCrl*** — the CRL found in the untrusted cache<br>***bci*** — ***bci***<br>• If found and if the present date and time are outside the range of **thisUpdate** and **nextUpdate** of the CRL, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *expiredCertificateCRLorBCI*<br>***errorThumb*** — the Thumbprint of the CRL<br>• If found and if ***newCrl.****crlNumber* is greater than or equal to *crlNumber* of the same *CRLIdentifier:*<br>  • delete ***crl*** (if found in Step 2) from the trusted cache, and<br>  • move ***newCrl*** and its Thumbprint to the trusted cache.<br>If any CRL listed on the BCI was not found in either the trusted or untrusted cache, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *missingCertificateCRLorBCI*<br>***errorThumb*** — the Thumbprint of the BCI |

*Continued on next page*

# Certificate Chain Validation, continued

### Verify CRL

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***newCrl*** — an instance of *CRL*<br>***bci*** — the current BCI for this brand<br>This procedure uses the following internal variables:<br>***crl*** — the current CRL |
| 2 | Search the trusted cache for a CRL whose *issuer* is ***newCrl****.issuer*. If found:<br>• designate it as ***crl***, and<br>• if ***newCrl****.crlNumber* is less than or equal to ***crl.****crlNumber*, remove ***newCrl*** from the untrusted cache and return. |
| 3 | Validate the following contents of ***newCrl***:<br>*algorithm*: *algorithm* = sha1-with-rsa-signature; *parameters* = NULL<br>If errors are encountered during the validation process, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** = *unsupportedAlgorithm*<br>***errorThumb*** = SHA-1 hash of ***newCrl****.toBeSigned*<br>***errorOID*** = the *algorithm* field |

*Continued on next page*

# Certificate Chain Validation, continued

**Verify CRL** (continued)

| Step | Action |
|---|---|
| 4 | Locate the CRL signing certificate.<br>• Search the trusted cache for a certificate whose:<br>  • *subject* Name matches ***newCrl****.issuer*,<br>  • *issuer* Name and *serialNumber* match the values of *authorityCertIssuer* and *authorityCertSerialNumber* in the **AuthorityKeyIdentifier** extension of ***newCrl***,<br>  • *basicConstraints.cA* is TRUE, and<br>  • *KeyUsage* includes *crlSign*.<br>• If no certificate was found, search the untrusted cache for a certificate matching those criteria. If found, invoke "Verify Certificate" on page 131 with the following input:<br>***cert*** — the certificate from the untrusted cache<br>If no certificate was found, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *missingCertificateCRLorBCI*<br>***errorThumb*** — SHA-1 hash of ***newCrl****.toBeSigned* |
| 5 | Decrypt ***newCrl****.signature* using the public key from the certificate found in Step 4. |
| 6 | Compute the SHA-1 hash of ***newCrl****.toBeSigned*. |
| 7 | Compare the results of Step 5 to the results of Step 6.<br>• If the values match:<br>  • Delete ***crl*** and its Thumbprint from the trusted cache.<br>  • Move ***newCRL*** and its Thumbprint to the trusted cache.<br>• If the values are not the same, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *invalidSignature*<br>***errorThumb*** — the result of Step 6 |

*Continued on next page*

# Certificate Chain Validation, continued

### Checking certificates against a CRL

A given certificate shall be deemed to be included on a CRL if:

| this value in the certificate: | matches this value in the CRL: |
|---|---|
| *issuer* | *issuer* |
| *serialNumber* | *revokedCertificates.CertificateSerialNumber* |

### Verify certificate

| Step | Action |
|---|---|
| 1 | Receive as input:<br>**cert** — an instance of *Certificate* |
| 2 | Validate the certificate according to the rules specified in Section 12.4.3 of Amendment 1 to X.509 and using the additional SET chain validation steps specified starting on page 123. When the validation requires another certificate:<br>• Search the trusted cache for a certificate whose:<br>  • *subject* Name matches ***cert.****issuer*, and<br>  • *serialNumber* matches ***cert****.AuthorityKeyIdentifier.authorityCertSerialNumber*<br>• If no certificate was found, search the untrusted cache for a certificate matching these criteria. If found, invoke "Verify Certificate" on page 131 with the following input:<br>**cert** — the certificate from the untrusted cache<br>The following input is provided to the X.509 process:<br>*trusted public key* — The set shall contain the key described in Appendix R. Applications may support other certification hierarchies as well, adding those keys to this set.<br>*initial-policy-set* — The set shall contain *id-set-policy-root* and may contain other policies.<br>*initial-explicit-policy* — The value of this indicator is determined by local policy.<br>*initial-policy-mapping-inhibit* — The value of this indicator is determined by local policy.<br>Verify that the certificate extensions **KeyUsage**, **CertificatePolicies**, **PrivateKeyUsage**, and **AuthorityKeyIdentifier** are being used in accordance with X.509.<br>*(continues)* |

*Continued on next page*

# Certificate Chain Validation, continued

**Verify certificate** (continued)

| Step | Action |
|---|---|
| 2 (cont) | If errors are encountered, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — As specified in Table 23 on page 133.<br>***errorThumb*** — The Thumbprint of the certificate that failed. If the failure results from comparing an entity certificate to that of the issuing CA, the Thumbprint of the entity certificate shall be used. (For *missingCertificateCRLorBCI*, you may include the Thumbprint of the certificate that was being validated when the error was encountered.)<br>For a signature validation failure, populate ***errorThumb*** with the locally generated digest.<br>***errorOID*** — If the error resulted from evaluating an extension, the object identifier of that extension; otherwise, this field does not appear. |
| 3 | If no certificate was found in Step 2, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *missingCertificateCRLorBCI*<br>***errorThumb*** — SHA-1 hash of ~~***newCrl**.toBeSigned*~~ the certificate being validated when a new certificate was required |
| 4 | If no errors were encountered in Step 2, move the certificate to the trusted cache. |

*Continued on next page*

# Certificate Chain Validation, continued

**Certificate error processing**

Table 23 defines the value of **ErrorCode** for input to SET error processing resulting from certificate chain validation failures.

| Use this value: | ...if any of these failures occurs: |
|---|---|
| *invalidCertificateCRLorBCI* | The *notBefore* field has a value that is later than the current date and time.<br>The certificate *subject* and certificate *issuer* names do not chain correctly.<br>The certificate's *notBefore* date is not within the *PrivateKeyUsagePeriod* of its CA certificate.<br>The certificate's *validity* period is not within the *validity* period of its CA certificate.<br>The validation fails for any of the certificate extensions *AuthorityKeyIdentifier*, *BasicConstraints*, *CertificatePolicies*, *CertificateType*, *KeyUsage*, and *PrivateKeyUsagePeriod*.<br>The certificate does not contain required extensions.<br>The *policy* qualifier of *CertificatePolicies* does not inherit *AdditionalPolicy.policyOID* or *AdditionalPolicy.policyAddedBy*.<br>*CertificatePolicies* contains *AdditionalPolicy.policyQualifier* with fields that are not inherited.<br>The **BrandID**s do not chain correctly.<br>The *thisUpdate* field has a value that is later than the current date and time. |
| *expiredCertificateCRLorBCI* | The *notAfter* field has a value that is earlier than the current date and time.<br>The *nextUpdate* field has a value that is earlier than the current date and time. |
| *revokedCertificateCRLorBCI* | The certificate has been revoked. |
| *missingCertificateCRLorBCI* | A certificate with a subject name matching the issuer name of a certificate to be validated is not found in the trusted cache or in the message being processed.<br>The value of *CRLNumber* is less than that specified in the BCI. |
| *signatureFailure* | The signature does not verify. |
| *unrecognizedExtension* | The certificate, CRL, or BCI contains a critical extension that the application does not recognize. |

**Table 23: Enumerated Values for ErrorCode in Certificate Chain Validation**

# SET Error Processing

## Introduction

From the perspective of a SET participant, SET flow always occurs in message pairs. Each message transmitted by a requester is answered by a responder. The error flow (unlike all the other flows) is defined with respect to senders and receivers because it is used when any participant cannot reliably identify an incoming message.

**Error** indicates that a receiver rejects a message because it fails format or content verification tests; that is, the message is corrupted or unintelligible. The receiver sends **Error** (rather than, for example, a negative response code) when the receiver cannot trust the fields of an incoming message. In general, **Error** shall be used only:

- to respond to the direct sender of the message, and
- when it is not possible to clearly isolate the error to an incorrect value of a field.

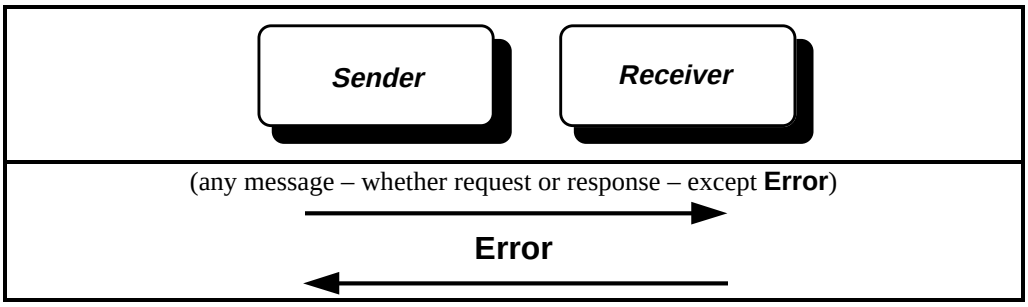


**Figure 10: Error Message Flow**

*Continued on next page*

# SET Error Processing, continued

**When NOT to send Error**

An **Error** message shall not be used:

| | |
|---|---|
| for normal business results | Normal business results such as a declined authorization are indicated by explicit codes in standard SET response messages. |
| in response to an **Error** message | A valid SET message contains:<br>• a tag of 0x30 (a SEQUENCE) and a length for the ~~entire message (~~**~~MessageWrapper~~** **MessageHeader**~~ plus ~~**~~Message~~** body).~~ ~~The~~ **MessageWrapper**, which contains:<br>– a tag of 0x30 (a SEQUENCE) followed by the length and content of the **MessageHeader**,<br>– a tag of 0xA0 (a context specific tag of [0]) followed by the tag for the type of message then the length and content of the **Message**, and<br>– optionally, a tag of 0xA1 (a context specific tag of [1]) followed by the length and content of any message extensions.<br>If the tag for the type of message is 0xBF8767 (a context specific tag of 999), which indicates an **Error** message, a SET application shall never send a response, even if the message appears to be malformed. This is to prevent loops where one **Error** message triggers another. |
| in response to excessive duplicate messages | Enough information appears in cleartext in the ~~message wrapper~~ **MessageHeader** that an application can detect whether a message is a retransmission or not. The receiver's reaction to a duplicate message depends on:<br>• the idempotency property of the message type,<br>• the number of duplicated messages,<br>• the source of the duplicate message, and<br>• the frequency of duplicate messages.<br>If a system suspects that it is being subjected to a flooding or spamming attack, duplicate messages may be ignored. |
| in response to excessive unusable messages from a single sender | To mitigate the effects of denial-of-service attacks, software may limit the number of **Error** messages that are sent. For example, software may elect to send only one **Error** message per day to a given sender. |
| in response to a non-SET message | Any messages that do not appear to be SET messages should be ignored. |

*Continued on next page*

# SET Error Processing, continued

### When to send Error

| | |
|---|---|
| in response to SET request message | Merchant, Payment Gateway, and CA software should send an **Error** message when encountering a low-level processing error on a SET request message. |
| in response to SET response message | Cardholder, Merchant, and Payment Gateway software should send an **Error** message when encountering a low-level processing error on a SET response message.<br><br>The **Error** message should be sent to a *diagnostic log* port if one has been defined for the system that sent the response. (Defining a diagnostic log port allows a separate system to receive **Error** messages so that the primary system may be devoted to processing SET message pairs.) Applications should avoid sending **Error** messages to the same port as ~~request~~ SET messages; however, if no diagnostic log port is available, the application may send one **Error** message per day to the ~~request~~ normal port. (The diagnostic log port is discussed further in the *SET External Interface Guide*. See "Related documentation" in the Preface.) |

### Error message in response to response

The SET protocol is based on request/response pairs. The **Error** message does not conform to this paradigm, since it may be a response to either a request or a response. The former case poses no difficulty. However, in the latter case, difficulties may arise if the underlying transport is based on a request/response paradigm, as in a World Wide Web browser. In this case, the **Error** message may be sent as a request message, and the SET protocol will not require a response message; as a result, the underlying protocol may time out. It is recognized that the operational constraints of a World Wide Web browser may require user permission for an **Error** message to be sent. This is acceptable, but not encouraged.

*Continued on next page*

# SET Error Processing, continued

**Create Error message**

When an application encounters a SET error, it shall create an **Error** message as described below.

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***errorCode*** — a value indicating the error that was detected<br>***errorOID*** — an object identifier related to the error that was detected (optional)<br>***errorThumb*** — a thumbprint related to the error that was detected (optional) |
| 2 | Construct *ErrorMsg* as either:<br>*messageHeader* — the header of the offending message<br>*badWrapper* — the entire offending message, up to the size restriction of 20,000 bytes (optional)<br>The choice of whether to copy only the header (***messageHeader***) or the entire message (***badWrapper***) is left to each implementation. Providing ***badWrapper*** gives the system that receives the message the most possible information. |
| 3 | Construct *ErrorTBS*:<br>*errorCode* — ***errorCode***<br>*errorNonce* — a fresh nonce<br>*errorOID* — ***errorOID***<br>*errorThumb* — ***errorThumb***<br>*errorMsg* — result of Step 2 |
| 4 | If a signature certificate is available, invoke "Compose SignedData (S)" on page 153 with the following input:<br>***s*** — ~~EE's~~ responder's signature certificate<br>***t*** — result of Step 3<br>***type*** — *id-set-content-ErrorTBS* |

*Continued on next page*

# SET Error Processing, continued

**Create Error message** (continued)

| Step | Action |
|---|---|
| 5 | Invoke "Send Message" on page 110 with the following input: |

| | |
|---|---|
| ***recip*** | the entity that sent the offending message |
| ***msg*** | result of Step 4 (or Step 3, if Step 4 was not performed) |
| ***ext*** | any message extension(s) required to support additional business functions (optional) |
| ***rrpid*** | the **RRPID** from the **MessageHeader** of the offending message if available (optional) |
| ***lid-C*** | the **lid-C** from the **MessageHeader** of the offending message if available (optional) |
| ***lid-M*** | the **lid-M** from the **MessageHeader** of the offending message if available (optional) |
| ***xID*** | the **xID** from the **MessageHeader** of the offending message if available (optional) |

| | |
|---|---|
| 6 | Abort processing the rest of the message. |

*Continued on next page*

# SET Error Processing, continued

**Error message data**

The following fields are defined for the **Error** message:

| | |
|---|---|
| **Error** | **< SignedError, UnsignedError >** |
| **SignedError** | **S(EE, ErrorTBS)** |
| **UnsignedError** | **ErrorTBS**<br>*The unsigned version of* **Error** *shall only be used if the entity does not have a valid signature certificate or is temporarily unable to generate signatures (such as when there is a cryptographic hardware failure).* |
| **ErrorTBS** | **{ErrorCode, ErrorNonce, [ErrorOID], [ErrorThumb], ErrorMsg}** |
| **ErrorCode** | *Enumerated code.* |
| **ErrorNonce** | *A nonce to ensure the signature is generated over unpredictable data.* |
| **ErrorOID** | *The object identifier of an object (extension, content type, attribute, etc.) that caused the error.* |
| **ErrorThumb** | *The thumbprint of the certificate, CRL or BrandCRLIdentifier that caused the error.* |
| **ErrorMsg** | **<MessageHeader, BadWrapper>** |
| **MessageHeader** | *The message header of the message that produced the error.* |
| **BadWrapper** | *The message wrapper of the message that produced the error (up to 20,000 bytes).* |

**Table 24: Error Message Data**

*Continued on next page*

# SET Error Processing, continued

**ErrorCode**

The following values are defined for **ErrorCode**.

| | |
|---|---|
| unspecifiedFailure | *The reason for the failure does not appear elsewhere in this list.* |
| messageNotSupported | *This valid message type is not supported by the recipient.* |
| decodingFailure | *An error was encountered during the DER decoding process on the message.* |
| invalidCertificateCRLorBCI | *A certificate, CRL, or BCI necessary to process this message was not valid (for a reason not specified elsewhere in this table). The* **ErrorThumb** *field identifies the invalid certificate, CRL, or BCI. Additional detail about this* **ErrorCode** *appears in* Table 23 *on page 133.* |
| expiredCertificateCRLorBCI | *A certificate, CRL, or BCI necessary to process this message has expired. The* **ErrorThumb** *field identifies the invalid certificate, CRL, or BCI. Additional detail about this* **ErrorCode** *appears in* Table 23 *on page 133.* |
| revokedCertificateCRLorBCI | *A certificate, CRL, or BCI necessary to process this message has been revoked. The* **ErrorThumb** *field identifies the invalid certificate, CRL, or BCI.* |
| missingCertificateCRLorBCI | *A certificate, CRL, or BCI necessary to process this message is not available in the recipient's ~~certificate~~ trusted cache and was not included in the message. Additional detail about this* **ErrorCode** *appears in* Table 23 *on page 133.* |
| signatureFailure | *The digital signature of the message could not be verified.* |
| badMessageHeader | *The message header cannot be processed.* |
| wrapperMsgMismatch | *The contents of the message wrapper are inconsistent with the internal content of the message, for example, the* **RRPID** *does not match.* |
| versionTooOld | *The version number of the message is too old for the recipient to process.* |
| versionTooNew | *The version number of the message is too new for the recipient to process.* |

**Table 25: Enumerated Values for ErrorCode**

*Continued on next page*

# SET Error Processing, continued

**ErrorCode** (continued)

| | |
|---|---|
| unrecognizedExtension | *The message or a certificate contains a critical extension that the recipient cannot process. The* **ErrorOID** *field identifies the extension. If the extension appears in a certificate, the* **ErrorThumb** *field identifies the certificate.* |
| messageTooBig | *The message is too big for the recipient to process.* |
| signatureRequired | *The unsigned version of this message is not valid.* |
| messageTooOld | *The date of the message is too ~~new~~ old for the recipient to process.* |
| messageTooNew | *The date of the message is too new for the recipient to process.* |
| thumbsMismatch | *Thumbprints sent in an unsigned request did not match those returned to the requester checking for substitution attack.* |
| unknownRRPID | *An unknown* **RRPID** *was received.* |
| unknownXID | *An unknown* **XID** *was received.* |
| unknownLID~~XID~~ | *An unknown local identifier was received.* |
| challengeMismatch | *A challenge sent in a request did not match the challenge in the response.* |

**Table 25: Enumerated Values for ErrorCode,** continued

See also "Table 23" on page 133.

**Process Error message**

In general, processing of a SET **Error** message is at the discretion of the application and outside the scope of SET. However, a few **ErrorCode**s warrant special processing:

| **ErrorCode** | Processing |
|---|---|
| versionTooOld or versionTooNew | See "Backward Compatibility" on page 65. |
| messageTooOld or messageTooNew | See "System Clock Differences" on page 66. |

*Continued on next page*

# SET Error Processing, continued

**Future values for ErrorCode**

The following error conditions were identified after the ASN.1 for version 1.0 was completed. They are currently defined as constants mapping to *unspecifiedFailure*. In a future version of the ASN.1, these values will be added to the ENUMERATED **ErrorCode**. Application developers are encouraged to use these symbolic names in place of *unspecifiedFailure*.

| | |
|---|---|
| badOAEPBlock | *The OAEP block is not correctly formatted or was encrypted with the wrong public key.* |
| baggageLinkageFailure | *The linkage between the baggage and the message could not be verified.* |
| decryptionFailure | *The message could not be successfully decrypted.* |
| idempotencyFailure | *An idempotent request containing identical identifiers as a previous request is not bit-wise identical to the prior message.* |
| keyUnavailable | *A cryptographic key necessary to process this message is unavailable.* |
| typeMismatch | *The object identifier of contentType of a structure ~~could not be verified~~ did not match the value expected.* |
| unsupportedAlgorithm | *The hashing or encryption algorithm is not supported.* |
| unsupportedBrand | *A request was made for a brand ID that is not supported.* |
| requestTypeMismatch | *The* **RequestType** *received in a response message does not match the* **RequestType** *in the corresponding request.* |
| unrecognizedField | *The application has received a field that it cannot process.* |
| missingData | *An optional field was omitted from the message, but the processing conditions require it to be present.* |
| requestResponseMismatch | *Date of response does not match that of request.* |

**Table 26: Future Enumerated Values for ErrorCode**

# Section 2
# Cryptographic Processing

## Overview

### Organization

This section describes processing for the following treatments and operators:

# Keyed-Hash

### *HMAC*

The keyed-hash operator, *HMAC(t, k)*, corresponds to the 160-bit HMAC-SHA-1 hash of *t* using the secret *k*. This function is used as the blinding function to protect the account number in the Cardholder certificate and to create transaction stains.

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***t*** — the content to be hashed<br>***k*** — a secret key for cryptographic enhancement of ***t*** |
| 2 | Create a buffer containing 64 bytes with 0x36 repeated 64 times. |
| 3 | Create a buffer containing 64 bytes with 0x5c repeated 64 times. |
| 4 | Append zeros to the end of ***k*** to create a 64-byte buffer (for example, if ***k*** is of length 20 bytes, append 44 bytes of 0x00). |
| 5 | Compute bit-wise exclusive-or of the result of Step 4 and the result of Step 2. |
| 6 | Append ***t*** to the result of Step 5. |
| 7 | Compute the SHA-1 hash of the result of Step 6. |
| 8 | Compute bit-wise exclusive-or of the result of Step 4 and the result of Step 3. |
| 9 | Append the result of Step 7 to the result of Step 8. |
| 10 | Compute the SHA-1 hash of the result of Step 9. |
| 11 | Return the result of Step 10. |

# DetachedDigest

**DD**

The *DetachedDigestedData* operator *DD(t)* corresponds to a 160-bit SHA-1 hash of *t* embedded in a PKCS *DigestedData* sequence. *t* is not included in the *content* component of *ContentInfo*.

Each type of content digested in SET is identified by a unique object identifier in the *contentType* component of *ContentInfo*.

**Compose *DetachedDigest***

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***t*** — the content to be digested<br>***type*** — an object identifier for the content of ***t*** |
| 2 | Compute the SHA-1 hash of ***t***, including the tag and length octets. |
| 3 | Construct and return *DigestedData*:<br>*ddVersion* = 0<br>*digestAlgorithm*: *algorithm* = id-sha1; *parameters* = NULL<br>*contentInfo*: *contentType* = ***type***<br>*digest* = the result of Step 2 |

*Continued on next page*

# **DetachedDigest,** continued

**Verify**
***DetachedDigest***

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***t*** — the content to be verified<br>***d*** — an instance of *DigestedData*<br>***type*** — an object identifier for the content that was digested |
| 2 | Validate the following contents of ***d***:<br>*ddVersion* = 0<br>*digestAlgorithm*.*algorithm* = id-sha1<br>*digestAlgorithm*.*parameters* = NULL<br>*contentInfo*.*contentType* = ***type***<br>If errors are encountered during the validation process, invoke "Create **Error** Message" on page 137 with the following input based on the field that failed:<br>***errorCode***: *ddVersion* → *decodingFailure*; *digestAlgorithm* → *unsupportedAlgorithm*; *contentType* → *typeMismatch*<br>***errorOID***: for *digestAlgorithm*, the *algorithm* field |
| 3 | Compute the SHA-1 hash of the complete DER representation of ***t***, including the tag and length octets. |
| 4 | Compare the results of Step 3 to ***d***.*digest*. If the comparison fails, return a status of *failure*. |
| 5 | Return a status of success and the following:<br>***type*** = ***d***.*contentType* |

*Continued on next page*

# DetachedDigest, continued

**Sample code: *DD***

The following ASN.1 sample code shows how *DigestedData* is constructed as the result of *DD(t)*.

```
detachedDigest  DigestedData ::= {
    ddVersion  0,
    digestAlgorithm {
       algorithm   id-sha1,
       parameters  NULL
    },
    contentInfo {
       contentType  type
    },
    digest  "SHA-1 hash of t"
}
```

# Linkage

***L***

The linkage operator, *L(t1, t2)*, corresponds to a sequence of *t1* and a PKCS #7 *DigestedData* component represented by *DD(t2)*. It links *t1* to *t2*, but does not include the content of *t2*.

**Compose *Linkage***

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***t1*** — the content that is linked to ***t2***<br>***t2*** — the content whose digest is concatenated to ***t1***<br>***type*** — an object identifier for the content of ***t2*** |
| 2 | Invoke "Compose *DetachedDigest*" on page 146 with the following input:<br>***t*** — ***t2***<br>***type*** — ***type*** |
| 3 | Append the result of Step 2 to ***t1***. |
| 4 | Return the result of Step 3. |

**Verify *Linkage***

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***d*** — a structure containing:<br>*t1* — the content that is linked to ***t2***<br>*t2* — an instance of *DigestedData*<br>***t2*** — the content whose digest is concatenated to ***t1***<br>***type*** — an object identifier for the content of ***t2*** |
| 2 | Invoke "Verify *DetachedDigest*" on page 147 with the following input:<br>***t*** — ***t2***<br>***d*** — ***d***.*t2*<br>***type*** — ***type***<br>If the result is *failure*, return a status of *failure*. |
| 3 | Return a status of *success* and the following:<br>***type*** — the value of *type* returned in Step 2 |

*Continued on next page*

## Linkage, continued

**Sample code: *L*** The following ASN.1 sample code shows how to link two values.

```
linkage Linkage ::= {
    t1      t1,
    t2  {
        ddVersion  0,
        digestAlgorithm {
            algorithm   id-sha1,
            parameters  NULL
        },
        contentInfo {
            contentType  type
        },
        digest  "SHA-1 hash of t2"
    }
}
```

# Signature

***S***

The *Signature* operator, *S(s, t)*, corresponds to PKCS #7 *SignedData*. *SignedData* uses the private key of *s* to sign *t* and includes *t* in the content of the *SignedData* type. The digital signature method employed by SET uses an encrypted hash. A digital signature operation is performed by encrypting the SHA-1 hash of *t*, with the RSA private key of signer *s*.

**Authenticated attributes**

SET PKCS #7 *SignedData* digital signature operations are always performed on values that are the DER representations of values of ASN.1 types. *SignedData* signature operations are never performed on arbitrary octet strings, such as ASCII text files or random strings with no consistent internal structure, so the PKCS #7 *data* content type is never used. Instead one of the SET-specific content object identifiers is used. In such situations, when the content type *data* is not used , PKCS #7 requires that at least two authenticated attributes be included in the actual content that is signed. The parameterized types, *S{}* and *SO{}*, both represent *SignedData* in SET, and both require authenticated attributes.

Two attributes, *contentType* and *messageDigest*, are always included in the *authenticatedAttributes* signed in SET. For *SignedData*, a message digest results from the application of the PKCS #7 message-digesting process to some SET ASN.1 type, the content to be signed. For SET *SignedData*, the content to be signed is always ~~the complete DER representation, including the tag and length octets, of~~ two authenticated attributes tightly coupled with the *content* component of *ContentInfo*.

Note: The message digest generated for *authenticatedAttributes* is computed over the inner *AttributeSeq* type and does not include the outer tag [2] and its length.

The initial input to the message-digesting process is the DER representation of the *content* component of the *ContentInfo* sequence. *ContentInfo* binds a *contentType* component object identifier to the type in its *content* component. In SET, each *SignedData* content type is uniquely named by an object identifier. Since this value is not protected directly against a substitution attack, it is also included in the *authenticatedAttributes*.

- The *contentType* attribute shall specify an object identifier that matches the value in the *contentType* component of the *ContentInfo* sequence.
- The *messageDigest* attribute contains the value of the digested *content* component of *ContentInfo*.

*Continued on next page*

# Signature, continued

**Number of signers**

The definition of the *SignerInfos* sequence in PKCS #7 allows any number of signers to be included in the collection, providing one *SignerInfo* per signer.

SET PKCS #7 *SignedData* requires one signer for all messages except **CertReq** and **CertInqReq**, which require two signers when used for certificate renewal. It is constrained to permit only one or two signers, so that the general processing requirements of PKCS #7 are simplified in SET.

In the *SignerInfo* component of *SignerInfos*, both the *authenticatedAttributes* and the *unauthenticatedAttributes* components are specified as optional. In SET:

- The *authenticatedAttributes* component is always present, and it is this value that is signed. ~~shall be included in the message-digesting process.~~
- The *unauthenticatedAttributes* component of the *SignerInfo* sequence is always absent, and never appears in an encoding of a value of *SignedData*.

*Continued on next page*

# Signature, continued

**Compose**
***SignedData (S)***

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***s*** — the signature certificate of the signer (except during certificate registration, as described in Step 7)<br>***s2*** — a second signature certificate (optional)<br>***t*** — the content to be signed<br>***type*** — an object identifier for the content of ***t***<br>***certs*** — additional certificate(s) to be included in message (optional) (that is, additional to those required to validate the signature: key-encryption certificates and newly generated certificates)<br>***crls*** — CRLs to be included in message (optional) |
| 2 | Compute SHA-1 hash of ***t***. |
| 3 | Construct *authenticatedAttributes* with two entries:<br>*type* — *contentType*<br>*value* — ***type***<br>*type* — *messageDigest*<br>*value* — the result of Step 2 |
| 4 | Compute SHA-1 hash of the contents of the DER-encoding of the **AttributeSeq** identified by *authenticatedAttributes* generated in Step 3. Note: Do not include the outer tag [2] and its length in the hash. |

*Continued on next page*

# Signature, continued

**Compose SignedData (S)** (continued)

<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>5</td><td>Sign the result of Step 4 using the private key corresponding to s (or s2 for a second signature).</td></tr>
<tr><td>6</td><td>Verify the result of Step 5. (The validation method is at the discretion of the application developer.) If the verification fails, abort processing.<br><br>Note: In a fully debugged system, this is an indication that the signature generation process is under attack to try to determine the private key.</td></tr>
<tr><td>7</td><td>Construct SignerInfo:
<table>
<tr><td>siVersion</td><td colspan="2">2</td></tr>
<tr><td>issuerAndSerialNumber</td><td colspan="2">the issuer and serialNumber fields of s (or s2 for a second signature)<br><br>During certificate registration, signatures are generated without a certificate to authenticate the public key. In this event, encode issuerAndSerialNumber as follows:
<table>
<tr><td>issuer</td><td>RDNSequence</td><td>a SEQUENCE with zero items</td></tr>
<tr><td>serialNumber</td><td colspan="2">0</td></tr>
</table>
See “Sample code: signature without certificate” on page 158.</td></tr>
<tr><td rowspan="2">digestAlgorithm</td><td>algorithm</td><td>id-sha1</td></tr>
<tr><td>parameters</td><td>NULL</td></tr>
<tr><td>authenticatedAttributes</td><td colspan="2">the result of Step 3</td></tr>
<tr><td rowspan="2">digestEncryptionAlgorithm</td><td>algorithm</td><td>id-rsaEncryption</td></tr>
<tr><td>parameters</td><td>NULL</td></tr>
<tr><td>encryptedDigest</td><td colspan="2">the result of Step 5</td></tr>
</table></td></tr>
<tr><td>8</td><td>If s2 is specified, repeat Steps 5, 6 and 7 for s2.</td></tr>
<tr><td>9</td><td>If certs is not provided, initialize certs so that it contains zero entries.<br><br>Add the following to certs:<ul><li>the certificate chain of any certificate in certs;</li><li>s and its certificate chain;</li><li>if s2 is specified, s2 and its certificate chain;</li><li>all root certificates that are previous generations of any root certificates in certs (to the generation of the root certificate specified in Appendix R).</li></ul></td></tr>
</table>

*Continued on next page*

# **Signature,** continued

**Compose SignedData (S)** (continued)

| Step | Action |
|---|---|
| 10 | If ***crls*** is not provided, initialize ***crls*** so that it contains zero entries.<br>If this is a response message containing a BrandCRLIdentifier, add all certificate revocation lists contained on it to ***crls***. |
| 11 | If this is a response message and the request contained Thumbprints, optionally remove any entries from ***certs*** and ***crls*** whose Thumbprint appears in the request. |
| 12 | Construct and return *SignedData*: |

| | | |
|---|---|---|
| *sdVersion* | 2 | |
| *digestAlgorithms* | *algorithm* | id-sha1 |
| | *parameters* | NULL |
| *contentInfo* | *contentType* | ***type*** |
| | *content* | ***t*** |
| *certificates* | ***certs*** | |
| *crls* | ***crls*** | |
| *signerInfos* | the result of Step 7 (with two entries if ***s2*** is provided; otherwise, one entry) | |

*Continued on next page*

# **Signature,** continued

**Verify**
***SignedData (S)***

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***d*** — an instance of *SignedData*<br>***type*** — an object identifier for the content that was signed<br>***unauthOK*** — a flag indicating whether an unauthorized signature is permissible (optional) |
| 2 | Invoke "Verify *SignedData (SO)*" on page 160 with the following input:<br>***t*** — ***d****.contentInfo.content*<br>***d*** — ***d***<br>***type*** — ***type***<br>***unauthOK*** — ***unauthOK*** |
| 3 | Return the following:<br>***t*** — ***d****.contentInfo.content*<br>***type*** — the value of *type* returned in Step 2<br>***si*** — the value of *si* returned in Step 2 |

*Continued on next page*

## Signature, continued

**Sample code: *S*** The following ASN.1 sample code shows how *SignedData* is constructed for the signature operator ***S*** *(**s**, **t**)*. (A separate sample shows how *issuerAndSerialNumber* is constructed when a signature is generated without a certificate.)

```
signature  SignedData ::= {
    sdVersion  2,
    digestAlgorithms {
       { algorithm   id-sha1,
         parameters  NULL
       }
    },
    contentInfo {
       contentType  type,
       content      t
    },
    certificates { ... },
    crls { ... },
    signerInfos {
       {  siVersion  2,
          issuerAndSerialNumber {
             issuer        s.issuer,
             serialNumber  s.serialNumber
          },
          digestAlgorithm {
             algorithm   id-sha1,
             parameters  NULL
          },
          authenticatedAttributes {
             { type   contentType,
               value  type
             },
             { type   messageDigest,
               value  "SHA-1 hash of t"
             }
          },
          digestEncryptionAlgorithm {
             algorithm   id-rsaEncryption,
             parameters  NULL
          }
          encryptedDigest "Signed authenticatedAttributes"
       }
    }
}
```

*Continued on next page*

## Signature, continued

**Sample code: signature without certificate**

The following ASN.1 sample code shows how *issuerAndSerialNumber* is constructed when a signature is generated without a certificate to authenticate the public key. The DER encoding is shown as comments.

```
noCertificate   IssuerAndSerialNumber ::= {        -- 30 05
      issuer rdnSequence : {},                      -- 30 00
      serialNumber         0                        -- 02 01 00
}
```

# Signature Only

***SO***

The signature only operator, *SO(s, t)*, corresponds to PKCS #7 *external signature SignedData*. External signature *SignedData* uses the private key of *s* to sign *t* and does not include *t* in the *content* of the *SignedData* type.

See "Signature" on page 151 for information that applies to both *SignedData* types, *S{}* and *SO{}*.

**Compose *SignedData (SO)***

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***s*** — the signature certificate of the signer (except during certificate registration, as described in Step 7)<br>***s2*** — a second signature certificate (optional)<br>***t*** — the content to be signed<br>***type*** — an object identifier for the content of ***t***<br>***certs*** — additional certificate(s) to be included in message (optional)<br>***crls*** — CRLs to be included in message (optional) |
| 2 to 11 | Perform Steps 2 through 11 of "Compose *SignedData (S)*" on page 153. |
| 12 | Construct and return *SignedData*:<br>*sdVersion*: 2<br>*digestAlgorithms*: *algorithm*: id-sha1; *parameters*: NULL<br>*contentInfo*: *contentType*: ***type***<br>*certificates*: ***certs***<br>*crls*: ***crls***<br>*signerInfos*: the result of Step 7 (with two entries if ***s2*** is provided; otherwise, one entry) |

*Continued on next page*

# Signature Only, continued

**Verify**
***SignedData (SO)***

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***t*** — the content that was signed<br>***d*** — an instance of *SignedData*<br>***type*** — an object identifier for the content that was signed<br>***unauthOK*** — flag indicating whether an unauthenticated signature is valid (optional) |
| 2 | Validate the following contents of ***d***:<br>*sdVersion* — 2<br>*digestAlgorithms* — *algorithm* — id-sha1<br>*parameters* — NULL<br>*contentInfo* — *contentType* — ***type***<br>If errors are encountered during the validation process, invoke "Create **Error** Message" on page 137 with the following input based on the field that failed:<br>***errorCode*** — *sdVersion* — *decodingFailure*<br>*digestAlgorithm* — *unsupportedAlgorithm*<br>*contentType* — *typeMismatch*<br>***errorOID*** — for *digestAlgorithm*, the *algorithm* field |
| 3 | Create an untrusted cache and populate it with ***d***.*certs* and ***d***.*crls*. |
| 4 | If ***type*** appears in Table 27 on page 163, extract **bci** from ***t*** as specified in the second column of the table. |
| 5 | Invoke "Verify BCI" on page 126 with the following input:<br>***newBci*** — result of Step 4, if any<br>***brand*** — the brand field of ***d***.*SignerInfo.issuerAndSerialNumber.issuer.organizationName* |

*Continued on next page*

# Signature Only, continued

**Verify SignedData (SO)** (continued)

<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>6</td><td>Validate the following contents of d.signerInfos:
<table>
<tr><td>siVersion</td><td colspan="2">2</td></tr>
<tr><td rowspan="2">digestAlgorithm</td><td>algorithm</td><td>id-sha1</td></tr>
<tr><td>parameters</td><td>NULL</td></tr>
<tr><td rowspan="2">digestEncryption Algorithm</td><td>algorithm</td><td>id-rsaEncryption</td></tr>
<tr><td>parameters</td><td>NULL</td></tr>
</table>
If errors are encountered during the validation process, invoke "Create Error Message" on page 137 with the following input based on the field that failed:
<table>
<tr><td rowspan="3">errorCode</td><td>siVersion</td><td>decodingFailure</td></tr>
<tr><td>digestAlgorithm</td><td>unsupportedAlgorithm</td></tr>
<tr><td>digestEncryption Algorithm</td><td>unsupportedAlgorithm</td></tr>
<tr><td>errorOID</td><td colspan="2">for digestAlgorithm or digestEncryptionAlgorithm, the algorithm field</td></tr>
</table></td></tr>
<tr><td>7</td><td>Compute SHA-1 hash of t.</td></tr>
<tr><td>8</td><td>Validate the following contents of d.signerInfos.authenticatedAttributes:
<table>
<tr><td>type</td><td>contains an entry for contentType and an entry for messageDigest</td></tr>
<tr><td>value for contentType</td><td>d.contentInfo.contentType</td></tr>
<tr><td>value for messageDigest</td><td>result of Step 7</td></tr>
</table>
If errors are encountered during the validation process, invoke "Create Error Message" on page 137 with the following input:
<table>
<tr><td>errorCode</td><td>invalidSignature</td></tr>
</table></td></tr>
</table>

*Continued on next page*

# Signature Only, continued

**Verify SignedData (SO)** (continued)

| Step | Action |
|---|---|
| 9 | If ***d***.*signerInfos.issuerAndSerialNumber* indicates that a signature was performed without a certificate to authenticate the public key:<br>• If ***unauthOK*** is TRUE and this is the first signature without a certificate, save this instance of ***d***.*signerInfos* to return and skip to Step 14.<br>• Otherwise, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *invalidSignature* |
| 10 | Locate the signature certificate:<br>a. Search the trusted cache for a certificate whose *issuer* and *serialNumber* matches ***d***.*signerInfos.issuerAndSerialNumber*.<br>b. If no certificate was found, search the untrusted cache for a certificate matching the same criteria. If found, invoke "Verify Certificate" on page 131 with the following input:<br>***cert*** — the certificate from the untrusted cache<br>If no certificate was found, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *missingCertificateCRLorBCI* |
| 11 | Compute SHA-1 hash of the contents of the DER-encoding of the **AttributeSeq** identified by ***d***.*signerInfos.authenticatedAttributes*.<br>Note: Do not include the outer tag [2] and its length in the hash. |
| 12 | Decrypt ***d***.*signerInfos.encryptedDigest* using the public key of the certificate obtained in Step 10. |
| 13 | Compare the results of Step 11 to the results of Step 12. If the values are different, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *invalidSignature* |
| 14 | If ***unauthOK*** is TRUE and if there is a second instance of ***d***.*signerInfos*, repeat Steps 6 through 13. |
| 15 | Return the following:<br>***type*** — ***d***.*contentInfo.contentType*<br>***si*** — any unvalidated *signerInfo* from Step 9 |

*Continued on next page*

## Signature Only, continued

**BCI location**

The location of the Brand CRL Identifier depends on the *contentType* of the message as defined in Table 27.

| Content type | Location of BCI in ***t*** |
|---|---|
| id-set-content-AuthResTBS | AuthResTBS.t1.brandCRLIdentifier |
| id-set-content-AuthResTBSX | AuthResTBSX.authResTBS.t1.brandCRLIdentifier |
| id-set-content-AuthRevResData | AuthRevResData.brandCRLIdentifier |
| id-set-content-AuthRevResTBS | AuthRevResTBS.t1.brandCRLIdentifier |
| id-set-content-BCIDistributionTBS | BCIDistribution.BCIDistributionTBS. BrandCRLIdentifier |
| id-set-content-CapResData | CapResData.brandCRLIdentifier |
| id-set-content-CapRevResData | CapRevResData.brandCRLIdentifier |
| id-set-content-CardCInitResTBS | CardCInitResTBS.brandCRLIdentifier |
| id-set-content-CertResData | CertResData.brandCRLIdentifier |
| id-set-content-CredResData | CredResData.brandCRLIdentifier |
| id-set-content-CredRevResData | CredRevResData.brandCRLIdentifier |
| id-set-content-Me-AqCInitResTBS | Me-AqCInitResTBS.brandCRLIdentifier |
| id-set-content-PCertResTBS | PCertResTBS.brandCRLIdentifierSeq. brandCRLIdentifier |
| id-set-content-PInitResData | PInitResData.brandCRLIdentifier |
| id-set-content-PResData | PResData.brandCRLIdentifier |
| id-set-content-RegFormResTBS | RegFormResTBS.brandCRLIdentifier |

**Table 27: *BrandCRLIdentifier* Location**

*Continued on next page*

## Signature Only, continued

**Sample code: *SO***

The following ASN.1 sample code shows how *SignedData* is constructed for the signature only operator, *SO(s, t)*.

```
signature  SignedData ::= {
    sdVersion  2,
    digestAlgorithms {
       { algorithm   id-sha1,
         parameters  NULL
       }
    },
    contentInfo {
       contentType  type,
    },
    certificates { ... },
    crls { ... },
    signerInfos {
       {  siVersion  2,
          issuerAndSerialNumber {
             issuer        s.issuer,
             serialNumber  s.serialNumber
          },
          digestAlgorithm {
             algorithm   id-sha1,
             parameters  NULL
          },
          authenticatedAttributes {
             { type   contentType,
               value  type
             },
             { type   messageDigest,
               value  "SHA-1 hash of t"
             }
          },
          digestEncryptionAlgorithm {
             algorithm   id-rsaEncryption,
             parameters  NULL
          }
          encryptedDigest "Signed authenticatedAttributes"
       }
    }
}
```

# Optimal Asymmetric Encryption Padding

**Purpose**

The purpose of OAEP is to ensure that individual pieces of a message cannot be extracted from a PKCS #7 block. There are cryptoanalytic techniques that make some bits of a message easier to determine than others. OAEP randomly distributes the bits of a PKCS #7 block, making each bit equally difficult to extract.

**Algorithm description**

The *E*, *EX*, *EXL*, *EH*, and *EXH* encryption primitives combine RSA encryption and OAEP. SET uses OAEP to provide "extra" protection of the account information associated with the Cardholder, Merchant, or Acquirer in the digital envelope.

This section provides a brief description of how to implement OAEP to support its "extra encryption" and "extra decryption" operators as they are used in SET. The reader is encouraged to supplement this description with the OAEP information provided in *SET Book 3: Formal Protocol Definition.*

*Continued on next page*

# Optimal Asymmetric Encryption Padding, continued

**Create OAEP block**

~~SET "extra encryption"~~ The creation of an OAEP block involves the following processing steps:

| Step | Action |
|---|---|
| 1 | Receive as input:<br>*BC* — Block contents byte<br>*DEK* — DES encryption key<br>*X* — "Extra encrypted" data (optional; included for values of *BC* other than 0x00 or 0x80)<br>*HD* — Integrity hash (optional; included for values of *BC* greater than or equal to 0x80) |
| 2 | Form the Actual Data Block, *ADB*, based on the value of *BC* as follows:<br>**Value of BC** → **Formation of ADB**<br>0x00 → *DEK*<br>> 0x00 and < 0x80 → *DEK* \| *X*<br>0x80 → *DEK* \| *HD*<br>> 0x80 → *DEK* \| *HD* \| *X* |
| 3 | Compute the Data Block, *DB*, as follows:<br>*BT* — 0x03<br>*V* — 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 (7 bytes of zero)<br>*DB* — *BT* \| *BC* \| *V* \| *ADB* |
| 4 | Pad the result of Step 3 with trailing zeros to form a string with a total length of 111 bytes. |
| 5 | Generate a random 16-byte string *E-Salt*, and compute *H1(E-Salt)* as follows:<br>SHA-1(*E-Salt* \| 0) \| SHA-1(*E-Salt* \| 1) \| SHA-1(*E-Salt* \| 2) \| SHA-1(*E-Salt* \| 3) \| SHA-1(*E-Salt* \| 4) \| SHA-1(*E-Salt* \| 5).<br>Truncate the string to 111 bytes by discarding the nine trailing bytes. |
| 6 | Perform an exclusive-or on the results of Step 4 and the results of Step 5. |
| 7 | Compute the SHA-1 hash of the results of Step 6. Discard the leading four bytes to form a sixteen-byte string. |
| 8 | Perform an exclusive-or on *E-Salt* (from Step 5) and the results of Step 7. |
| 9 | Generate a single-byte random value between 0x01 and 0x7F. |
| 10 | Concatenate the results of Step 9, Step 6 and Step 8. |
| 11 | Return the result of Step 10. |

*Continued on next page*

# Optimal Asymmetric Encryption Padding, continued

**Process OAEP block**

~~SET "extra decryption"~~ Receipt of an OAEP block requires the following steps:

| Step | Action |
|---|---|
| 1 | Receive the OAEP block as input. |
| 2 | Extract values from the results of Step 1 as follows:<br>*I* — byte 1<br>*A* — bytes 2 through 112<br>*B* — bytes 113 through 128 |
| 3 | Verify that *I* is ~~neither 0x00 nor 0x80~~ in the range 0x01 to 0x7F. If this verification fails, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *badOAEPBlock* |
| 4 | Compute the SHA-1 hash of *A*. Discard the leading four bytes to form a sixteen-byte string. |
| 5 | Perform an exclusive-or on *B* and the results of Step 3 to produce *E-Salt*. |
| 6 | Compute *H1(E-Salt)* as follows:<br>SHA-1(*E-Salt* \| 0) \| SHA-1(*E-Salt* \| 1) \| SHA-1(*E-Salt* \| 2) \| SHA-1(*E-Salt* \| 3) \| SHA-1(*E-Salt* \| 4) \| SHA-1(*E-Salt* \| 5).<br>Truncate the string to 111 bytes by discarding the nine trailing bytes. |
| 7 | Perform an exclusive-or on *A* and the results of Step 6. |
| 8 | Extract values from the results of Step 7 as follows:<br>*BT* — byte 1<br>*BC* — byte 2<br>*V* — bytes 3 through 9<br>*ADB* — bytes 10 through 111 |
| 9 | Verify that *BT* contains a value of 0x03, *BC* contains a value either in the range 0x00 to 0x05 or in the range 0x80 to 0x85, and that *V* is zero. If any of these verifications fail, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *badOAEPBlock* |

*Continued on next page*

# Optimal Asymmetric Encryption Padding, continued

**Process OAEP block** (continued)

| Step | Action |
|---|---|
| 10 | Extract values from *ADB*, based on the value of *BC* as follows:<br>• Extract the first eight bytes into *DEK*.<br>• If *BC* is greater than or equal to 0x80, extract the next 20 bytes into *HD*.<br>• If *BC* is in the range 0x01 to 0x05 or in the range 0x81 to x85, extract the appropriate value of X as indicated in the following table. |
| 11 | Return the following: |

| **Value of BC** | **Content of X** |
|---|---|
| 0x01 or 0x81 | **PANData** |
| 0x02 or 0x82 | **PANData0** |
| 0x03 or 0x83 | **PANToken** |
| 0x04 or 0x84 | **PANOnly** |
| 0x05 or 0x85 | **AcctData** |

| | |
|---|---|
| *BC* | Block Contents byte |
| *DEK* | DES encryption key |
| *X* | "extra encrypted" data (optional; included for values of *BC* other than 0x00 or 0x80) |
| *HD* | integrity hash (optional; included for values of *BC* greater than or equal to 0x80) |

*Continued on next page*

# Optimal Asymmetric Encryption Padding, continued

**Processing**

Figure 11 illustrates the processing flow for OAEP as it is used in SET.

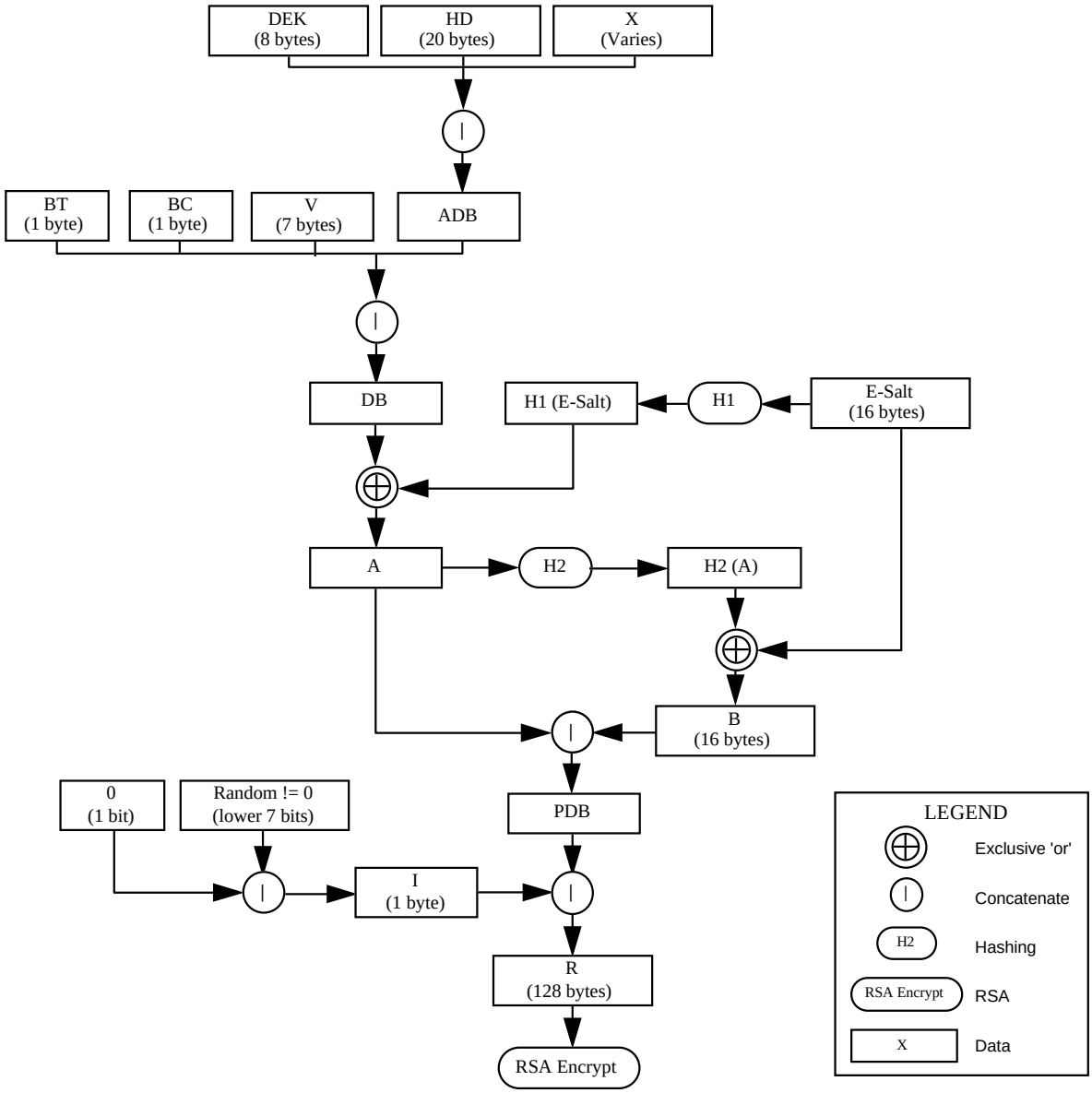


**Figure 11: OAEP Processing Flow**

*Continued on next page*

# Optimal Asymmetric Encryption Padding, continued

**Encoding of DB**

Data present in data block (**DB**) fields is not formatted with the usual DER encoding method, in order to save space. The format used for the **DB** is defined here.

For all of the definitions, all fields shall be present.

Only fields from the ASN.1 definition ~~atomic (in the sense of ASN.1) data elements~~ are present in **DB**. Each element is encoded within **DB** in the canonical form used by DER-encoding, but without tag and length octets. When transferring data from DER-encoded format to **DB**, add pad characters ~~(0x00)~~ to the end of the data; when transferring from **DB** to DER-encoded format, strip all pad characters from the end of the data.

To understand the format of a **DB** field, examine the ASN.1 used to define the field for signature purposes. ~~Determine the matching DER-encoded wire format, and store the field in *DB* accordingly.~~ Determine the corresponding ASN.1 type, and store the field in **DB** according to the following table, which summarizes the DER format of field types used in SET extra-encrypted data:

| ASN.1 Type | DB Encoding |
|---|---|
| VisibleString | ASCII string, first character in lowest-numbered position, padded with blanks (0x20). |
| NumericString | ASCII string, first character in lowest-numbered position, padded with blanks (0x20). |
| OCTET STRING | Binary byte string in lowest-numbered position, padded with bytes of zero (0x00). |

**For more information**

Additional information about the encoding of specific extra-encryption data formats in the OAEP block is found in SET Book 3.

# EnvelopedData

**Compose *EnvelopedData***

The processing steps that follow are shared by **E**, **EX**, **EXL**, **EH**, and **EXH**, all of which produce a PKCS #7 *EnvelopedData* block.

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***r*** — the key encryption certificate of the recipient<br>***t*** — the content to be encrypted<br>***p*** — the parameter receiving extra encryption protection (optional)<br>***link*** — Boolean value indicating if linkage is required (default FALSE)<br>***h*** — Boolean value indicating if hash is required (default FALSE)<br>***type-t*** — an object identifier for the content of ***t***<br>***type-p*** — an object identifier for the content of ***p*** (optional) |
| 2 | If ***p*** is not provided, continue with Step 5.<br>If ***link*** is FALSE, continue with Step 4.<br>Generate a fresh nonce and put it into the **EXNonce** field of ***p***. |
| 3 | To link tuple ***t*** with parameter ***p***, invoke "Compose *Linkage*" on page 149 with the following input:<br>***t1*** — ***t***<br>***t2*** — the result of Step 2<br>***type*** — ***type-p*** |
| 4 | Format ***p*** for insertion into the "extra encrypted data" portion of OAEP. |
| 5 | If ***p*** is provided and ***link*** is TRUE, select the result from Step 3; otherwise, select ***t***. |
| 6 | Generate a fresh symmetric DES key (or select a key in accordance with "Reuse of symmetric DES keys " on page 91). |
| 7 | Generate an eight-byte DES-CBC initialization vector. |
| 8 | Encrypt the result from Step 5 with the DES key from Step 6 and the initialization vector from Step 7 using DES-CBC mode following the standard padding rule described on page 87. |
| 9 | If ***h*** is TRUE, compute SHA-1 hash of the result of Step 5. |

*Continued on next page*

# **EnvelopedData,** continued

**Compose EnvelopedData** (continued)

<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>10</td><td>Determine the value of BC as follows:<br>a. Initialize the value to 0x00.<br>b. If type-p is provided, add the value from the following table based on type-p:
<table>
<tr><td>0x01</td><td>id-set-content-PANData</td></tr>
<tr><td>0x03</td><td>id-set-content-PANToken</td></tr>
<tr><td>0x04</td><td>id-set-content-PANOnly</td></tr>
</table>
Otherwise, if p is provided, add the value from the following table based on the contents of p.
<table>
<tr><td>0x02</td><td>PANData0</td></tr>
<tr><td>0x05</td><td>AcctData</td></tr>
</table>
c. If h is TRUE, add 0x80</td></tr>
<tr><td>11</td><td>Invoke “Create OAEP block” on page 166 with the following input:
<table>
<tr><td>BC</td><td>the result of Step 10</td></tr>
<tr><td>DEK</td><td>the result of Step 6</td></tr>
<tr><td>X</td><td>the result of Step 4 (if p is provided)</td></tr>
<tr><td>HD</td><td>the result of Step 9 (if h is TRUE)</td></tr>
</table></td></tr>
<tr><td>12</td><td>Encrypt the result from Step 11 using the public key from r.</td></tr>
<tr><td>13</td><td>Construct RecipientInfo:
<table>
<tr><td>riVersion</td><td colspan="2">0</td></tr>
<tr><td>issuerAndSerialNumber</td><td colspan="2">the issuer and serialNumber fields of r</td></tr>
<tr><td rowspan="2">keyEncryptionAlgorithm</td><td>algorithm</td><td>rsaOAEPEncryptionSET</td></tr>
<tr><td>parameters</td><td>NULL</td></tr>
<tr><td>encryptedKey</td><td colspan="2">the result of Step 12</td></tr>
</table></td></tr>
</table>

*Continued on next page*

# **EnvelopedData,** continued

**Compose EnvelopedData** (continued)

<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>14</td><td>Construct EncryptedContentInfo:
<table>
<tr><td>contentType</td><td colspan="2">type-t</td></tr>
<tr><td rowspan="2">contentEncryption Algorithm</td><td>algorithm</td><td>id-desCBC</td></tr>
<tr><td>parameters</td><td>the result of Step 7</td></tr>
<tr><td>encryptedContent</td><td colspan="2">the result of Step 8</td></tr>
</table></td></tr>
<tr><td>15</td><td>Construct and return EnvelopedData:
<table>
<tr><td>edVersion</td><td>1</td></tr>
<tr><td>recipientInfos</td><td>the result of Step 13</td></tr>
<tr><td>encryptedContentInfo</td><td>the result of Step 14</td></tr>
</table></td></tr>
</table>

*Continued on next page*

# **EnvelopedData,** continued

**Verify *EnvelopedData***

The processing steps that follow are shared by **E**, **EX**, **EXL**, **EH**, and **EXH**, all of which produce a PKCS #7 *EnvelopedData* block.

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***d*** — an instance of *EnvelopedData*<br>***type-t*** — an object identifier for the content that was encrypted<br>***type-p*** — an object identifier for the parameter that received extra encryption protection (optional) |
| 2 | Validate the following contents of ***d***:<br>*edVersion* = 1<br>If errors are encountered during the validation process, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** = *decodingFailure* |
| 3 | Validate the following contents of ***d***.*recipientInfos*:<br>*riVersion* = 0<br>*keyEncryptionAlgorithm*.*algorithm* = rsaOAEPEncryptionSET<br>*keyEncryptionAlgorithm*.*parameters* = NULL<br>If errors are encountered during the validation process, invoke "Create **Error** Message" on page 137 with the following input based on the field that failed:<br>***errorCode*** = *riVersion* → *decodingFailure*<br>***errorCode*** = *keyEncryptionAlgorithm* → *unsupportedAlgorithm*<br>***errorOID*** = for *keyEncryptionAlgorithm*, the *algorithm* field |

*Continued on next page*

# **EnvelopedData,** continued

**Verify EnvelopedData** (continued)

| Step | Action |
|---|---|
| 4 | Validate the following contents of ***d****.encryptedContentInfo*:<br>*contentType* — ***type-t***<br>*contentEncryptionAlgorithm* — algorithm — id-desCBC<br>If errors are encountered during the validation process, invoke "Create **Error** Message" on page 137 with the following input based on the field that failed:<br>***errorCode*** — *contentType* — *typeMismatch*<br>*algorithm* — *unsupportedAlgorithm*<br>***errorOID*** — the *algorithm* field |
| 5 | Locate the key pair for the certificate identified by ***d****.recipientInfos.issuerAndSerialNumber* and use the private key to decrypt ***d****.recipientInfos.encryptedKey*. If the private key is no longer available, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *keyUnavailable*<br>***errorThumb*** — the Thumbprint of the certificate (if available) |
| 6 | Invoke "Process OAEP block" on page 167 with the result of Step 5 as input. |
| 7 | Using *DEK* from the result of Step 6 and the DES-CBC initialization vector in ***d****.encryptedContentInfo.contentEncryptionAlgorithm.parameters*, decrypt ***d****.encryptedContentInfo.encryptedContent* using DES-CBC mode. Validate and discard the padding. If the padding is not valid, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *decryptionFailure* |
| 8 | If *BC* from the result of Step 6 is greater than or equal to 0x80, compute the SHA-1 hash of the result of Step 7. Compare this hash to *HD* from the result of Step 6. If the values are not the same, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *decryptionFailure* |

*Continued on next page*

# **EnvelopedData,** continued

**Verify EnvelopedData** (continued)

| Step | Action |
|---|---|
| 9 | If *BC* from the result of Step 6 is not 0x00 or 0x80 and the result of Step 7 is an instance of *DigestedData*, invoke "Verify *Linkage*" on page 149 with the following input:<br>***d*** — the result of Step 7<br>***t2*** — the value of *X* from the result of Step 6<br>***type*** — ***type-p***<br>If the result is *failure*, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *decryptionFailure* |
| 10 | Extract *t* from the result of Step 7 as follows:<br>• If *BC* from the result of Step 6 is 0x00 or 0x80, the result of Step 7;<br>• otherwise, extract *t1* from the *Linkage* in the result of Step 7. |
| 11 | Return the following:<br>***t*** — the result of Step 10<br>***p*** — *X* from the result of Step 6 (optional; only if ***p*** is returned in Step 6)<br>***type-t*** — ***d****.encryptedContentInfo.contentType*<br>***type-p*** — *type* from the result of Step 9 (optional; only if ***p*** is returned in Step 6) |

# Asymmetric Encryption

***E***

The asymmetric encryption operator, *E(r, t)*, corresponds to PKCS #7 *EnvelopedData* of tuple *t* encrypted for entity *r*. This operator consists of applying fast, symmetric, bulk encryption to *t* using a secret key and then encrypting that secret key with the recipient's public key.

OAEP is used to obfuscate the contents of the PKCS #7 envelope.

### Compose *E*

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***r*** — the key encryption certificate of the recipient<br>***t*** — the content to be encrypted<br>***type*** — an object identifier for the content of ***t*** |
| 2 | Invoke "Compose *EnvelopedData*" on page 171 with the following input:<br>***r*** — ***r***<br>***t*** — ***t***<br>***link*** — FALSE<br>***h*** — FALSE<br>***type-t*** — ***type*** |
| 3 | Return the result of Step 2. |

### Verify *E*

Invoke "Verify *EnvelopedData*" on page 174.

*Continued on next page*

## Asymmetric Encryption, continued

**Sample code: *E*** The following ASN.1 sample code shows how *EnvelopedData* is constructed as the result of *E(r, t)*.

```
envelopedData  EnvelopedData ::= {
    edVersion  1,
    recipientInfos {
        {  riVersion  0,
            issuerAndSerialNumber {
                issuer         r.issuer,
                serialNumber   r.serialNumber
            },
            keyEncryptionAlgorithm {
                algorithm    rsaOAEPEncryptionSET,
                parameters   NULL
            },
            encryptedKey   "RSA encrypted OAEP block"
        }
    },
    encryptedContentInfo {
        contentType   type,
        contentEncryptionAlgorithm {
            algorithm    id-desCBC,
            parameters   cbc8Parameter
        },
        encryptedContent   "DES encrypted t"
    }
}
```

# Extra Asymmetric Encryption with Linkage

***EXL***

The "extra" asymmetric encryption with linkage operator, *EXL(r, t, p)*, consists of applying fast, symmetric, bulk encryption to *t* after linking it to *p* and applying a separate "extra" process to *p*, as follows:

- Generate a fresh 20-byte nonce (**EXNonce**) to foil dictionary attacks on *p*.
- Link *t* and *p*.
- Encrypt the linkage.
- Create an OAEP block including *p*.
- Encrypt the OAEP block.
- Construct a data structure that includes the encrypted linkage and the encrypted OAEP block.

~~In SET's implementation of this operator, which is achieved by putting *p* is put inside the PKCS #7 envelope and *t* is linked to *p* prior to encrypting its contents. The secret key and parameter *p* are encrypted with the recipient's public key. OAEP is used to obfuscate the contents of the RSA envelope.~~

## Compose *EXL*

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***r*** — the key encryption certificate of the recipient<br>***t*** — the content to be encrypted<br>***p*** — the parameter receiving extra encryption protection<br>***type-t*** — an object identifier for the content of ***t***<br>***type-p*** — an object identifier for the content of ***p*** |
| 2 | Invoke "Compose *EnvelopedData*" on page 171 with the following input:<br>***r*** = ***r***<br>***t*** = ***t***<br>***p*** = ***p***<br>***link*** = TRUE<br>***h*** = FALSE<br>***type-t*** = ***type-t***<br>***type-p*** = ***type-p*** |
| 3 | Return the result of Step 2. |

## Verify *EXL*

Invoke "Verify *EnvelopedData*" on page 174.

*Continued on next page*

## Extra Asymmetric Encryption with Linkage, continued

**Sample code: *EXL***

The following ASN.1 sample code shows how *EnvelopedData* is constructed as the result of *EXL(r, t, p)*.

```
linkage DigestedData ::= {
    ddVersion  0,
    digestAlgorithm {
        algorithm   id-sha1,
        parameters  NULL
    },
    contentInfo {
        contentType  type-p
    },
    digest  "SHA-1 hash of p"
}

dataTBE  SEQUENCE ::= {
    t  t,
    p  linkage
}

exEnvelopedData  EnvelopedData ::= {
    edVersion  1,
    recipientInfos {
        {  riVersion  0,
            issuerAndSerialNumber {
                issuer        r.issuer,
                serialNumber  r.serialNumber
            },
            keyEncryptionAlgorithm {
                algorithm   rsaOAEPEncryptionSET,
                parameters  NULL
            },
            encryptedKey  "RSA encrypted OAEP block"
        }
    },
    encryptedContentInfo {
        contentType  type-t,
        contentEncryptionAlgorithm {
            algorithm   id-desCBC,
            parameters  cbc8Parameter
        },
        encryptedContent  "DES encrypted dataTBE"
    }
}
```

# Asymmetric Encryption with Integrity

### *EH*

The integrity encryption operator, *EH(r, t)*, is similar to *E*, except that the PKCS #7 envelope includes a hash of *t*. It consists of applying fast, symmetric, bulk encryption to *t* using a secret key and then encrypting that secret key and the hash with the recipient's public key.

OAEP is used to obfuscate the contents of the RSA envelope.

### Compose *EH*

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***r*** — the key encryption certificate of the recipient<br>***t*** — the content to be encrypted<br>***type*** — an object identifier for the content of ***t*** |
| 2 | Invoke "Compose *EnvelopedData*" on page 171 with the following input:<br>***r*** — ***r***<br>***t*** — ***t***<br>***link*** — FALSE<br>***h*** — TRUE<br>***type-t*** — ***type*** |
| 3 | Return the result of Step 2. |

### Verify *EH*

Invoke "Verify *EnvelopedData*" on page 174.

*Continued on next page*

# Asymmetric Encryption with Integrity, continued

**Sample code:**
***EH***

The following ASN.1 sample code shows how *EnvelopedData* is constructed as the result of *EH(r, t)*.

```
ehEnvelopedData EnvelopedData ::= {
    edVersion  1,
    recipientInfos {
        {  riVersion  0,
            issuerAndSerialNumber {
                issuer         r.issuer,
                serialNumber   r.serialNumber
            },
            keyEncryptionAlgorithm {
                algorithm   rsaOAEPEncryptionSET,
                parameters  NULL
            },
            encryptedKey  "RSA encrypted OAEP block"
        }
    },
    encryptedContentInfo {
        contentType  type,
        contentEncryptionAlgorithm {
            algorithm   id-desCBC,
            parameters  cbc8Parameter
        },
        encryptedContent  "DES symmetric key encrypted t"
    }
}
```

# Extra Asymmetric Encryption with Integrity

### *EXH*

The "extra" asymmetric encryption with integrity operator, *EXH(r, t, p)*, is similar to *EX*, except that the PKCS #7 envelope also includes a hash of *t*. It consists of applying fast, symmetric, bulk encryption to *t* after linking it to *p* and applying a separate "extra" process to *p*, as follows: ~~In SET's implementation of this operator, *p* is put inside the PKCS #7 envelope and *t* is linked to *p* prior to bulk encrypting its contents. A fresh 20-byte nonce (**EXNonce**) is also included to foil dictionary attacks on *p*. The secret key, the hash of *{t, p}*, and parameter *p* are encrypted with the recipient's public key.~~

~~OAEP is used to obfuscate the contents of the PKCS #7 envelope.~~

- Generate a fresh 20-byte nonce (**EXNonce**) to foil dictionary attacks on *p*.
- Link *t* and *p*.
- Encrypt the linkage.
- Create an OAEP block including *p* and a hash of the linkage.
- Encrypt the OAEP block.
- Construct a data structure that includes the encrypted linkage and the encrypted OAEP block.

### Compose *EXH*

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***r*** — the key encryption certificate of the recipient<br>***t*** — the content to be encrypted<br>***p*** — the parameter receiving extra encryption protection<br>***type-t*** — an object identifier for the content of ***t***<br>***type-p*** — an object identifier for the content of ***p*** |
| 2 | Invoke "Compose *EnvelopedData*" on page 171 with the following input:<br>***r*** — ***r***<br>***t*** — ***t***<br>***p*** — ***p***<br>***link*** — TRUE<br>***h*** — TRUE<br>***type-t*** — ***type-t***<br>***type-p*** — ***type-p*** |
| 3 | Return the result of Step 2. |

### Verify *EXH*

Invoke "Verify *EnvelopedData*" on page 174.

*Continued on next page*

## Extra Asymmetric Encryption with Integrity, continued

**Sample code:**
***EXH***

The following ASN.1 sample code shows how *EnvelopedData* is constructed as the result of *EXH(r, t, p)*.

```
linkage DigestedData ::= {
    ddVersion  0,
    digestAlgorithm {
        algorithm   id-sha1,
        parameters  NULL
    },
    contentInfo {
        contentType  "type-p"
    },
    digest  "SHA-1 hash of p"
}

dataTBE  SEQUENCE ::= {
    t  t,
    p  linkage
}

exhEnvelopedData  EnvelopedData ::= {
    edVersion  1,
    recipientInfos {
        {  riVersion  0,
            issuerAndSerialNumber {
                issuer         r.issuer,
                serialNumber   r.serialNumber
            },
            keyEncryptionAlgorithm {
                algorithm   rsaOAEPEncryptionSET,
                parameters  NULL
            },
            encryptedKey  "RSA encrypted OAEP block"
        }
    },
    encryptedContentInfo {
        contentType  type-t,
        contentEncryptionAlgorithm {
            algorithm   id-desCBC,
            parameters  cbc8Parameter
        },
        encryptedContent  "DES encrypted dataTBE"
    }
}
```

# Symmetric Encryption

***EK***

The symmetric encryption operator, *EK(k, t)*, encrypts *t* with the provided key *k*. Either the DES or CDMF algorithm may be used.

### Compose *EK*

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***k*** — a symmetric encryption key<br>***t*** — the content to be encrypted<br>***type*** — an object identifier for the content of ***t***<br>***aid*** — an object identifier for the algorithm that will be used for cryptographic processing<br>The following algorithm identifiers are permitted:<br>• id-desCBC<br>• id-desCDMF |
| 2 | If ***aid*** is *id-desCDMF*, transform ***k*** according to the CDMF requirements. |
| 3 | Generate an eight-byte DES-CBC initialization vector. |
| 4 | Encrypt ***t*** with ***k*** and the result of Step 3 using DES-CBC mode following the standard padding rule described on page 87. |
| 5 | Construct *EncryptedContentInfo*:<br>*contentType* — ***type***<br>*contentEncryptionAlgorithm* — *algorithm*: ***aid***; *parameters*: the result of Step 3<br>*encryptedContent* — the result of Step 4 |
| 6 | Construct *EncryptedData*:<br>version — 0<br>*encryptedContentInfo* — the result of Step 5 |
| 7 | Return the result from Step 6. |

*Continued on next page*

# Symmetric Encryption, continued

### Verify *EK*

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***k*** — a symmetric encryption key<br>***d*** — an instance of *EncryptedData*<br>***type*** — an object identifier for the content that was encrypted |
| 2 | Validate the following contents of *EncryptedData*:<br>version = 0<br>If errors are encountered during the validation process, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** = *decodingFailure* |
| 3 | Validate the following contents of ***d***.*encryptedContentInfo*:<br>*contentType* = ***type***<br>*contentEncryptionAlgorithm*.*algorithm* = id-desCBC or id-desCDMF<br>If errors are encountered during the validation process, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** = *algorithm* / *unsupportedAlgorithm*<br>***errorCode*** = *contentType* / *typeMismatch*<br>***errorOID*** = for *algorithm*, the *algorithm* field |
| 4 | If ***d***.*encryptedContentInfo.contentEncryptionAlgorithm.algorithm* is *id-desCDMF*:<br>• If the application does not support CDMF, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** = *unsupportedAlgorithm*<br>***errorOID*** = the value of *algorithm*<br>• otherwise, transform ***k*** according to the CDMF requirements. |

*Continued on next page*

# Symmetric Encryption, continued

**Verify EK** (continued)

| Step | Action |
|---|---|
| 5 | Using the result of Step 4 and the initialization vector in ***d***.*encryptedContentInfo.contentEncryptionAlgorithm.parameters*, decrypt ***d***.*encryptedContentInfo.encryptedContent* using DES-CBC mode. Validate and discard the padding. If the padding is not valid, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *decryptionFailure* |
| 6 | Return the following:<br>***t*** — the result of Step 5<br>***type*** — ***d***.*encryptedContentInfo.contentType* |

*Continued on next page*

## Symmetric Encryption, continued

**Sample code: *EK***

The following ASN.1 sample code shows how *EncryptedData* is constructed as the result of *EK(k, t)*.

```
ekEncryption  EncryptedData ::= {
    version  0,
    encryptedContentInfo {
        contentType  type,
        contentEncryptionAlgorithm {
            algorithm  aid,
            parameters  cbc8Parameter
        },
        encryptedContent  "Symmetric key k encrypted t"
    }
}
```

# Simple Encapsulation with Signature

***Enc***

The simple encapsulation with signature operator, *Enc(s, r, t)*, implements signed then encrypted messages. It corresponds to an instance of PKCS #7 *SignedData* encapsulated in *EnvelopedData*.

### Compose *Enc*

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***s*** — the signature certificate of the signer<br>***s2*** — a second signature certificate (optional)<br>***r*** — the key encryption certificate of the recipient<br>***t*** — the content to be encapsulated<br>***type-t*** — an object identifier for the content of ***t***<br>***type-s*** — an object identifier for the signed content of ***t***<br>***certs*** — additional certificate(s) to be included in message (optional) |
| 2 | Invoke "Compose *SignedData (S)*" on page 153 with the following input:<br>***s*** → ***s***<br>***s2*** → ***s2***<br>***t*** → ***t***<br>***type*** → ***type-s***<br>***certs*** → ***certs*** |
| 3 | Invoke "Compose *E*" on page 177 with the following input:<br>***r*** → ***r***<br>***t*** → the result of Step 2<br>***type*** → ***type-t*** |
| 4 | Return the result of Step 3. |

*Continued on next page*

# Simple Encapsulation with Signature, continued

### Verify *Enc*

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***d*** — an instance of *EnvelopedData*<br>***type-t*** — an object identifier for the content that was encapsulated<br>***type-s*** — an object identifier for the signed content<br>***unauthOK*** — flag indicating whether an unauthenticated signature is valid (optional) |
| 2 | Invoke "Verify *EnvelopedData*" on page 174 with the following input:<br>***d*** — ***d***<br>***type-t*** — ***type-t*** |
| 3 | Invoke "Verify *SignedData (S)*" on page 156 with the following input:<br>***d*** — *t* from the result of Step 2<br>***type*** — ***type-s***<br>***unauthOK*** — ***unauthOK*** |
| 4 | Return the following:<br>***t*** — *t* from the result of Step 3<br>***si*** — *si* from the result of Step 3<br>***type-t*** — *type-t* from the result of Step 2<br>***type-s*** — *type* from the result of Step 3 |

*Continued on next page*

# Simple Encapsulation with Signature, continued

**Sample code: *Enc***

The following ASN.1 sample code shows how *SignedData* and *EnvelopedData* are constructed as the result of *Enc(s, r, t)*.

```
encSignature  SignedData ::= {
    sdVersion  2,
    digestAlgorithms {
       { algorithm   id-sha1,
         parameters  NULL
       }
    },
    contentInfo {
       contentType  type-t,
       content      t
    },
    certificates { ... },
    crls { ... },
    signerInfos {
       {  siVersion  2,
          issuerAndSerialNumber {
             issuer        s.issuer,
             serialNumber  s.serialNumber
          },
          digestAlgorithm {
             algorithm   id-sha1,
             parameters  NULL
          },
          authenticatedAttributes {
             { type   contentType,
               value  type-t
             },
             { type   messageDigest,
               value  "SHA-1 hash of t"
             }
          },
          digestEncryptionAlgorithm {
             algorithm   id-rsaEncryption,
             parameters  NULL
          }
          encryptedDigest "Signed authenticatedAttributes"
       }
    }
}
```

*Continued on next page*

## Simple Encapsulation with Signature, continued

**Sample code: Enc** (continued)

```
encEnvelopedData  EnvelopedData ::= {
    edVersion  1,
    recipientInfos {
        {  riVersion  0,
            issuerAndSerialNumber {
                issuer         r.issuer,
                serialNumber   r.serialNumber
            },
            keyEncryptionAlgorithm {
                algorithm   rsaOAEPEncryptionSET,
                parameters  NULL
            },
            encryptedKey  "RSA encrypted OAEP block"
        }
    },
    encryptedContentInfo {
        contentType  type-s,
        contentEncryptionAlgorithm {
            algorithm   id-desCBC,
            parameters  cbc8Parameter
        },
        encryptedContent  "DES encrypted encSignature"
    }
}
```

# Simple Encapsulation with Signature and Provided Key

***EncK***

The simple encapsulation with signature and provided key operator, *EncK(k, s, t)*, implements signed messages encrypted with a known, shared, secret key provided by the sender of a prior message.

### Compose *EncK*

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***k*** — a symmetric encryption key<br>***s*** — the signature certificate of the signer<br>***t*** — the content to be encapsulated<br>***type-t*** — an object identifier for the content of ***t***<br>***type-s*** — an object identifier for the signed content of ***t***<br>***aid*** — an object identifier for the algorithm that will be used for cryptographic processing<br>***certs*** — additional certificate(s) to be included in message (optional) |
| 2 | Invoke "Compose *SignedData (S)*" on page 153 with the following input:<br>***s*** → ***s***<br>***t*** → ***t***<br>***type*** → ***type-s***<br>***certs*** → ***certs*** |
| 3 | Invoke "Compose *EK*" on page 187 with the following input:<br>***k*** → ***k***<br>***t*** → the result of Step 2<br>***type*** → ***type-t***<br>***aid*** → ***aid*** |
| 4 | Return the result of Step 3. |

*Continued on next page*

# Simple Encapsulation with Signature and Provided Key, continued

### Verify *EncK*

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***k*** — a symmetric encryption key<br>***d*** — an instance of *EncryptedData*<br>***type-t*** — an object identifier for the content that was encapsulated<br>***type-s*** — an object identifier for the signed content |
| 2 | Invoke "Verify *EK*" on page 188 with the following input:<br>***k*** — a symmetric encryption key<br>***d*** — ***d***<br>***type*** — ***type-t*** |
| 3 | Invoke "Verify *SignedData (S)*" on page 156 with the following input:<br>***d*** — *t* from the result of Step 2<br>***type*** — ***type-s*** |
| 4 | Return the following:<br>***t*** — *t* from the result of Step 3<br>***type-t*** — *type* from the result of Step 2<br>***type-s*** — *type* from the result of Step 3 |

*Continued on next page*

## Simple Encapsulation with Signature and Provided Key, continued

**Sample code: *EncK***

The following ASN.1 sample code shows how *SignedData* and *EncryptedData* are constructed as the result of *EncK(k, s, r, t)*.

```
enckSignature  SignedData ::= {
    sdVersion  2,
    digestAlgorithms {
       { algorithm   id-sha1,
         parameters  NULL
       }
    },
    contentInfo {
       contentType  type-t,
       content      t
    },
    certificates { ... },
    crls { ... },
    signerInfos {
       {  siVersion  2,
          issuerAndSerialNumber {
             issuer        s.issuer,
             serialNumber  s.serialNumber
          },
          digestAlgorithm {
             algorithm   id-sha1,
             parameters  NULL
          },
          authenticatedAttributes {
             { type   contentType,
               value  type-t
             },
             { type   messageDigest,
               value  "SHA-1 hash of t"
             }
          },
          digestEncryptionAlgorithm {
             algorithm   id-rsaEncryption,
             parameters  NULL
          }
          encryptedDigest "Signed authenticatedAttributes"
       }
    }
}
```

*Continued on next page*

## Simple Encapsulation with Signature and Provided Key, continued

**Sample code: EncK** (continued)

```
enckEncryptedData EncryptedData ::= {
    version  0,
    encryptedContentInfo {
        contentType  type-s,
        contentEncryptionAlgorithm {
            algorithm  aid,
            parameters  cbc8Parameter
        },
        encryptedContent  "Symmetric key k encrypted enckSignature"
    }
}
```

# Extra Encapsulation with Signature

***EncX***

The extra encapsulation with signature operator, *EncX(s, r, t, p)*, implements two-part signed messages encrypted with extra encryption.

### Compose *EncX*

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***s*** — the signature certificate of the signer<br>***s2*** — a second signature certificate (optional)<br>***r*** — the key encryption certificate of the recipient<br>***t*** — the content to be encapsulated<br>***p*** — the parameter receiving extra encryption protection<br>***type-t*** — an object identifier for the content of ***t***<br>***type-s*** — an object identifier for the signed content of ***t***<br>***type-p*** — an object identifier for the content of ***p***<br>***certs*** — additional certificate(s) to be included in message (optional) |
| 2 | Generate a fresh nonce and put it into the **EXNonce** field of ***p***. Append ***p*** to ***t***. |
| 3 | Invoke "Compose *SignedData (SO)*" on page 159 with the following input:<br>***s*** — ***s***<br>***s2*** — ***s2***<br>***t*** — the result of Step 2<br>***type*** — ***type-s***<br>***certs*** — ***certs*** |
| 4 | Append the results of Step 3 to ***t***. |
| 5 | Invoke "Compose ~~EX~~ *EnvelopedData*" on page 171 with the following input:<br>***r*** — ***r***<br>***t*** — the result of Step 4<br>***p*** — ***p***<br>***link*** — FALSE<br>***h*** — FALSE<br>***type-t*** — ***type-t***<br>***type-p*** — ***type-p*** |
| 6 | Return the result of Step 5. |

*Continued on next page*

# Extra Encapsulation with Signature, continued

### Verify *EncX*

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***d*** — an instance of *EnvelopedData*<br>***type-t*** — an object identifier for the content that was encapsulated<br>***type-s*** — an object identifier for the signed content<br>***type-p*** — an object identifier for the parameter receiving extra encryption protection<br>***unauthOK*** — flag indicating whether an unauthenticated signature is valid (optional) |
| 2 | Invoke "Verify *EnvelopedData*" on page 174 with the following input:<br>***d*** — ***d***.*t*<br>***type-t*** — ***type-t***<br>***type-p*** — ***type-p***<br>Designate the value of ***t*** returned as ***m***. Note: ***m*** has the following format:<br>***m*** — a structure containing:<br>*t* — arbitrary data<br>*s* — an instance of *SignedData* |
| 3 | Append ***p*** from the results of Step 2 to ***m***.*t*. |
| 4 | Invoke "Verify *SignedData (SO)*" on page 156 with the following input:<br>***t*** — the result of Step 3<br>***d*** — ***dm***.*s*<br>***type*** — ***type-s***<br>***unauthOK*** — ***unauthOK*** |
| 5 | Return the following:<br>***t*** — ***m***.*t*<br>***p*** — *p* from the result of Step 2<br>***si*** — *si* from the result of Step 4<br>***type-t*** — *type-t* from the result of Step 2<br>***type-s*** — *type* from the result of Step 4<br>***type-p*** — *type-p* from the result of Step 2 |

*Continued on next page*

## Extra Encapsulation with Signature, continued

**Sample code:**
***EncX***

The following ASN.1 sample code shows how *EnvelopedData* is constructed as the result of *EncX(s, r, t, p).*

```
dataTBS  SEQUENCE ::= {
    t  t,
    p  p
}

encxSignatureOnly  SignedData ::= {
    sdVersion  2,
    digestAlgorithms {
        { algorithm   id-sha1,
          parameters  NULL
        }
    },
    contentInfo {
        contentType  type-t
    },
    certificates { ... },
    crls { ... },
    signerInfos {
        {  siVersion  2,
           issuerAndSerialNumber {
              issuer        s.issuer,
              serialNumber  s.serialNumber
           },
           digestAlgorithm {
              algorithm   id-sha1,
              parameters  NULL
           },
           authenticatedAttributes {
              { type   contentType,
                value  type-t
              },
              { type   messageDigest,
                value  "SHA-1 hash of dataTBS"
              }
           },
           digestEncryptionAlgorithm {
              algorithm   id-rsaEncryption,
              parameters  NULL
           }
           encryptedDigest "Signed authenticatedAttributes"
        }
    }
}
```

*Continued on next page*

## Extra Encapsulation with Signature, continued

**Sample code: EncX** (continued)

```
dataTBE  SEQUENCE ::= {
    t  t,
    s  encxSignatureOnly
}

encXEnvelopedData  EnvelopedData ::= {
    edVersion  1,
    recipientInfos {
        {  riVersion  0,
            issuerAndSerialNumber {
                issuer         r.issuer,
                serialNumber   r.serialNumber
            },
            keyEncryptionAlgorithm {
                algorithm   rsaOAEPEncryptionSET,
                parameters  NULL
            },
            encryptedKey  "RSA encrypted OAEP block"
        }
    },
    encryptedContentInfo {
        contentType  type-s,
        contentEncryptionAlgorithm {
            algorithm   id-desCBC,
            parameters  cbc8Parameter
        },
        encryptedContent  "DES encrypted dataTBE"
    }
}
```

# Simple Encapsulation with Signature and Baggage

***EncB***

The simple encapsulation with signature and baggage operator, *EncB(s, r, t, b)*, implements signed, encrypted messages with external baggage.

### Compose *EncB*

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***s*** — the signature certificate of the signer<br>***r*** — the key encryption certificate of the recipient<br>***t*** — the content to be encapsulated<br>***b*** — the baggage<br>***type-t*** — an object identifier for the content of ***t***<br>***type-s*** — an object identifier for the signed content of ***t***<br>***type-b*** — an object identifier for the content of ***b***<br>***certs*** — additional certificate(s) to be included in message (optional) |
| 2 | To link tuple ***t*** with ***b***, invoke "Compose *Linkage*" on page 149 with the following input:<br>***t1*** — ***t***<br>***t2*** — ***b***<br>***type*** — ***type-b*** |
| 3 | Invoke "Compose *Enc*" on page 191 with the following input:<br>***s*** — ***s***<br>***r*** — ***r***<br>***t*** — the result of Step 2<br>***type-t*** — ***type-t***<br>***type-s*** — ***type-s***<br>***certs*** — ***certs*** |
| 4 | Append ***b*** to the results of Step 3. |
| 5 | Return the result of Step 4. |

*Continued on next page*

# Simple Encapsulation with Signature and Baggage, continued

### Verify *EncB*

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***d*** — a structure containing:<br>*t* — an instance of *EnvelopedData*<br>*b* — baggage<br>***type-t*** — an object identifier for the content that was encapsulated<br>***type-s*** — an object identifier for the signed content<br>***type-b*** — an object identifier for the content of the baggage |
| 2 | Invoke "Verify *Enc*" on page 192 with the following input:<br>***d*** — ***d***.*t*<br>***type-t*** — ***type-t***<br>***type-s*** — ***type-s*** |
| 3 | Invoke "Verify *Linkage*" on page 149 with the following input:<br>***d*** — *t* from the result of Step 2<br>***t2*** — ***d***.*b*<br>***type*** — ***type-b***<br>If the result is *failure*, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *baggageLinkageFailure* |
| 4 | One of the results of Step 2 is *t*, a linkage. Extract *t1* from *t*. |
| 5 | Return the following:<br>***t*** — the result of Step 4<br>***b*** — ***d***.*b*<br>***type-t*** — *type-t* from the result of Step 2<br>***type-s*** — *type-s* from the result of Step 2<br>***type-b*** — *type* from the result of Step 3 |

*Continued on next page*

# Simple Encapsulation with Signature and Baggage, continued

**Sample code:**
***EncB***

The following ASN.1 sample code shows how *result* is constructed as the result of *EncB(s, r, t, b)*.

```
detachedDigestedBaggage  DigestedData ::= {
    ddVersion  0,
    digestAlgorithm {
       algorithm   id-sha1,
       parameters  NULL
    },
    contentInfo {
       contentType  type-b,
    },
    digest  "SHA-1 hash of b"
}
```

*Continued on next page*

## Simple Encapsulation with Signature and Baggage, continued

**Sample code: EncB** (continued)

```
dataTBS  SEQUENCE ::= {
    t  t,
    b  detachedDigestedBaggage
}

encbSignature  SignedData ::= {
    sdVersion  2,
    digestAlgorithms {
       { algorithm   id-sha1,
         parameters  NULL
       }
    },
    contentInfo {
       contentType  type-t,
       content      dataTBS
    },
    certificates { ... },
    crls { ... },
    signerInfos {
       {  siVersion  2,
          issuerAndSerialNumber {
             issuer        s.issuer,
             serialNumber  s.serialNumber
          },
          digestAlgorithm {
             algorithm   id-sha1,
             parameters  NULL
          },
          authenticatedAttributes {
             { type   contentType,
               value  type-t
             },
             { type   messageDigest,
               value  "SHA-1 hash of dataTBS"
             }
          },
          digestEncryptionAlgorithm {
             algorithm   id-rsaEncryption,
             parameters  NULL
          }
          encryptedDigest "Signed authenticatedAttributes"
       }
    }
}
```

*Continued on next page*

## Simple Encapsulation with Signature and Baggage, continued

**Sample code: EncB** (continued)

```
encbEnvelopedData  EnvelopedData ::= {
    edVersion  1,
    recipientInfos {
        {  riVersion  0,
            issuerAndSerialNumber {
                issuer         r.issuer,
                serialNumber   r.serialNumber
            },
            keyEncryptionAlgorithm {
                algorithm   rsaOAEPEncryptionSET,
                parameters  NULL
            },
            encryptedKey  "RSA encrypted OAEP block"
        }
    },
    encryptedContentInfo {
        contentType  type-s,
        contentEncryptionAlgorithm {
            algorithm   id-desCBC,
            parameters  cbc8Parameter
        },
        encryptedContent  "DES encrypted encbSignature"
    }
}

encbResult SEQUENCE ::= {
    envelope  encbEnvelopedData,
    baggage   b
}
```

# Extra Encapsulation with Signature and Baggage

***EncBX***

The extra encapsulation with signature and baggage operator, *EncBX(s, r, t, b, p)*, implements two-part signed messages encrypted with extra encryption and with external baggage.

### Compose EncBX

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***s*** — the signature certificate of the signer<br>***r*** — the key encryption certificate of the recipient<br>***t*** — the content to be encapsulated<br>***b*** — the baggage<br>***p*** — the parameter receiving extra encryption protection<br>***type-t*** — an object identifier for the content of ***t***<br>***type-s*** — an object identifier for the signed content of ***t***<br>***type-p*** — an object identifier for the parameter receiving extra encryption protection<br>***type-b*** — an object identifier for the content of ***b***<br>***certs*** — additional certificate(s) to be included in message (optional) |
| 2 | To link tuple ***t*** with ***b***, invoke "Compose *Linkage*" on page 149 with the following input:<br>***t1*** — ***t***<br>***t2*** — ***b***<br>***type*** — ***type-b*** |

*Continued on next page*

# Extra Encapsulation with Signature and Baggage, continued

**Compose EncBX** (continued)

| Step | Action |
|---|---|
| 3 | Invoke "Compose *EncX*" on page 199 with the following input:<br>***s*** = ***s***<br>***r*** = ***r***<br>***t*** = the result of Step 2<br>***p*** = ***p***<br>***type-t*** = ***type-t***<br>***type-s*** = ***type-s***<br>***type-p*** = ***type-p***<br>***certs*** = ***certs*** |
| 4 | Append ***b*** to the results of Step 3. |
| 5 | Return the result of Step 4. |

*Continued on next page*

# Extra Encapsulation with Signature and Baggage, continued

### Verify EncBX

| Step | Action |
|---|---|
| 1 | Receive as input:<br>***d*** — a structure containing:<br>*t* — an instance of *EnvelopedData*<br>*b* — baggage<br>***type-t*** — an object identifier for the content that was encapsulated<br>***type-s*** — an object identifier for the signed content<br>***type-p*** — an object identifier for the parameter that received extra encryption protection<br>***type-b*** — an object identifier for the content of the baggage |
| 2 | Invoke "Verify *EncX*" on page 201 with the following input:<br>***d*** — ***d***.*t*<br>***type-t*** — ***type-t***<br>***type-s*** — ***type-s***<br>***type-p*** — ***type-p*** |
| 3 | Invoke "Verify *Linkage*" on page 149 with the following input:<br>***d*** — *t* from the result of Step 2<br>***t2*** — ***d***.*b*<br>***type*** — ***type-b***<br>If the result is *failure*, invoke "Create **Error** Message" on page 137 with the following input:<br>***errorCode*** — *baggageLinkageFailure* |
| 4 | One of the results of Step 2 is *t*, a linkage. Extract *t1* from *t*. |
| 5 | Return the following:<br>***t*** — the result of Step 4<br>***b*** — ***d***.*b*<br>***p*** — *p* from the result of Step 2<br>***type-t*** — *type-t* from the result of Step 2<br>***type-s*** — *type-s* from the result of Step 2<br>***type-p*** — *type-p* from the result of Step 2<br>***type-b*** — *type* from the result of Step 3 |

*Continued on next page*

# Extra Encapsulation with Signature and Baggage, continued

**Sample code:**
***EncBX***

The following ASN.1 sample code shows how *result* is constructed as the result of *EncBX(s, r, t, b, p).*

```
detachedDigestedBaggage  DigestedData ::= {
    ddVersion  0,
    digestAlgorithm {
       algorithm   id-sha1,
       parameters  NULL
    },
    contentInfo {
       contentType  type-b,
    },
    digest  "SHA-1 hash of b"
}

linkedData  SEQUENCE ::= {
    t  t,
    b  detachedDigestedBaggage
}

dataTBS  SEQUENCE ::= {
    t   linkedData,
    p   p
}
```

*Continued on next page*

## Extra Encapsulation with Signature and Baggage, continued

**Sample code: EncBX** (continued)

```
encbxSignatureOnly  SignedData ::= {
    sdVersion  2,
    digestAlgorithms {
        { algorithm   id-sha1,
          parameters  NULL
        }
    },
    contentInfo {
        contentType  type-t
    },
    certificates { ... },
    crls { ... },
    signerInfos {
        {  siVersion  2,
            issuerAndSerialNumber {
                issuer         s.issuer,
                serialNumber   s.serialNumber
            },
            digestAlgorithm {
                algorithm   id-sha1,
                parameters  NULL
            },
            authenticatedAttributes {
                { type   contentType,
                  value  type-t
                },
                { type   messageDigest,
                  value  "SHA-1 hash of dataTBS"
                }
            },
            digestEncryptionAlgorithm {
                algorithm   id-rsaEncryption,
                parameters  NULL
            }
            encryptedDigest "Signed authenticatedAttributes"
        }
    }
}
```

*Continued on next page*

## Extra Encapsulation with Signature and Baggage, continued

**Sample code: EncBX** (continued)

```
dataTBE  SEQUENCE ::= {
    t   linkedData,
    s   encbxSignatureOnly
}

encbxEnvelopedData  EnvelopedData ::= {
    edVersion  1,
    recipientInfos {
        {  riVersion  0,
            issuerAndSerialNumber {
                issuer         r.issuer,
                serialNumber   r.serialNumber
            },
            keyEncryptionAlgorithm {
                algorithm   rsaOAEPEncryptionSET,
                parameters  NULL
            },
            encryptedKey  "RSA encrypted OAEP block"
        }
    },
    encryptedContentInfo {
        contentType  type-s,
        contentEncryptionAlgorithm {
            algorithm   id-desCBC,
            parameters  cbc8Parameter
        },
        encryptedContent  "DES encrypted dataTBE"
    }

encbxResult SEQUENCE ::= {
    envelope  encbxEnvelopedData,
    baggage   b
}
```