Cincom

Cincom Smalltalk™

Security Guide

P46-0143-03

SIMPLIFICATION THROUGH INNOVATION®

Part Number: P46-0143-03

Software Release 7.6

Cincom Systems, Inc.

55 Merchant Street

Cincinnati, Ohio 45246

Phone: (513) 612-2300

Fax: (513) 612-2000

World Wide Web: http://www.cincom.com

Contents

About This Book vii

Overview ... vii
- Audience ... vii
- Organization ... vii

Conventions ... viii
- Typographic Conventions ... viii
- Special Symbols ... ix
- Mouse Buttons and Menus ... ix

Getting Help ... x
- Commercial Licensees ... x
- Non-Commercial Licensees ... xi

Additional Sources of Information ... xii
- Books on Computer Security ... xii
- Specifications ... xii

Chapter 1 Introduction to Security

Overview ... 1-1
- Symmetric-key Encryption ... 1-2
- Public-key Cryptography ... 1-2

Loading Security Components ... 1-2

Importing Security Components into a Name Space ... 1-3

Chapter 2 Hashes and Message Digests

Overview ... 2-1

Hash Algorithms ... 2-1
- MD5 ... 2-2
- SHA ... 2-2
- SHA-256 ... 2-2

Working with Hashes ... 2-2
- Hash Methods ... 2-2
- Hashing a Data Collection ... 2-3
- Hashing a Complete Data Stream ... 2-4

Hashing a Data Stream Incrementally 2-4
Comparing to a Known Hash Value 2-5
HMAC Message Digests 2-6
Key Selection 2-6
Generating an HMAC 2-6
Validating a Message using HMAC 2-7

Chapter 3 Random Number Generators

DSSRandom 3-2
Initializing a DSSRandom Generator 3-2
Constraining the range of values 3-3
Using Autogenerated Seeds 3-3
Reusing the Generator 3-4
Reseeding a Generator 3-5
Default Generator 3-6
Reseeding the Default Generator 3-7
Generating a Good Random 3-8
Selecting a Seed 3-8
Primality Testing 3-9
Configuring Miller-Rabin Testing 3-9
Configuring Prime Sieve Testing 3-10

Chapter 4 Symmetric-key Cryptography

Overview 4-1
Generating Keys 4-1
Symmetric-key Cipher General API 4-3
Instance Creation 4-3
Encryption/Decryption Messages 4-3
Block Ciphers 4-4
Block Cipher Implementations 4-4
AES 4-4
Blowfish 4-5
DES 4-5
General API 4-5
Cipher Modes 4-5
Padding Options 4-7
Triple-key EDE Encryption 4-8
Instance Creation Methods 4-8
Encrypting and Decrypting with Block Ciphers 4-9
Providing Custom Padding 4-10
Stream Ciphers 4-11
ARC4 4-11

Password-based Encryption and Authentication 4-12
Loading PKCS Support 4-12
PKCS Encryption 4-12
Message Authentication 4-14

Chapter 5 Public Key Cryptography

Digital signatures 5-2
Generating Keys 5-2
Exporting and Importing Keys 5-3
RSA 5-3
Creating a Digital Envelope 5-4
Digitally Signing with RSA 5-5
Using a Custom Hash 5-5
DSA 5-7
Generating DSA key pairs 5-7
Digitally Signing with DSA 5-7
Verifying a DSA signature 5-8
Diffie-Hellman key agreement 5-8
Basic DH Shared Secret Generation 5-8
Using the Shared Secret for Encryption 5-9
Using pre-defined parameter values 5-10
Controlling Parameter Generation 5-11

Chapter 6 Secure Socket Layer

Limitations 6-1
Usage 6-2
SSL Exceptions 6-4
Handshake and Certificates 6-4
Certificate Subject Validation 6-7
Client Authentication 6-8
Certificates for Signing 6-8
Diffie-Hellman Key Exchange 6-9
Anonymous Handshake 6-10
Session Renegotiation 6-11
Session Resumption 6-11

Chapter 7 ASN.1

Overview 7-1
ASN.1 7-1
ASN.1 Fundamental Types 7-2
ASN.1 Modules 7-3

ASN.1's Constructed Types 7-3
ASN.1's OID 7-3
Type Definition in ASN.1 7-4
ASN.1 Constraints 7-5
ASN.1 Type Tags 7-5
ASN.1 Encoding Rules 7-7
Packaging 7-8
Design of the ASN.1 Implementation 7-8
Outline 7-9
The ASN.1 Type System 7-9
The Types 7-9
The Type Extension Machinery 7-10
The Constraint Specification Machinery 7-11
The Mapping of Smalltalk Classes to the ASN.1 Type System 7-11
One-To-One Base Type Mappings 7-11
Many-To-One Base Type Mappings, Encoding Policies, and Type Wrappers 7-12
Constructed Type Mappings, Structs, and User-Defined Mappings ... 7-13
Imported Type Mappings 7-13
SMINode 7-14
The Encoding Rules 7-14
Using the ASN.1 Implementation 7-16
Getting the Encoded Bytes 7-16
Type-Agnostic Marshaling 7-18
Defining ASN.1 Types 7-19
Module Creation 7-19
Type Definitions 7-19
Type-In Hand Marshaling 7-22
Debugging Tips and Error Types 7-22
Known Limitations 7-22

Index **Index-1**

About This Book

Overview

This document describes the VisualWorks libraries and frameworks for employing encryption and related security features in Smalltalk applications.

Audience

This document is intended for experienced developers, who already know VisualWorks and Smalltalk development. It further assumes familiarity with the requirements for secure communications. It is beyond the scope of this document to fully describe the issues around and solutions to problems of security, for which you are referred to the many publications that can cover this topic in a comprehensive manner.

It is assumed that you have at least a beginning knowledge of programming in a Smalltalk environment, though not necessarily with VisualWorks. For introductory level information, the on-line *VisualWorks Tutorial* (available at http://www.cincom.com/smalltalk/tutorial), and the *Application Developer's Guide*.

Organization

This document begins with a general, and brief, introduction to software security and the problems it address. The subsequent chapters introduce specific support for security technologies within VisualWorks and their use.

The chapters are:

Chapter 1, "Introduction to Security." A very basic introduction to the issues of software security addressed by the VisualWorks libraries, types of security supported, and how to load security support.

Chapter 2, "Hashes and Message Digests." Describes hashing algorithms in general, and the algorithms implemented in VisualWorks and their use, particularly for creating message digests.

Chapter 3, “Random Number Generators.” Describes the role of pseudo-random number generators in providing software security, the generators provided by VisualWorks, and how to use them to maximize their effectiveness for secure applications.

Chapter 4, “Symmetric-key Cryptography.” Describes the support in VisualWorks for symmetric-key ciphers, including how to generate a key and the algorithms for using a key to encrypt and decrypt data.

Chapter 5, “Public Key Cryptography.” Describes the support in VisualWorks for public-key encryption, including how to generate keys, and their use for encryping and/or digitally signing data.

Chapter 6, “Secure Socket Layer.” Describes the implementation and use of Netscape’s SSL 3.0 implementation within VisualWorks.

Chapter 7, “ASN.1.” Describes ASN.1 and the design of the VisualWorks ASN.1 implementation.

Conventions

We have followed a variety of conventions, which are standard in the VisualWorks documentation.

Typographic Conventions

The following fonts are used to indicate special terms:

Example	Description
template	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
`cover.doc`	Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).
`filename.xwd`	Indicates a variable element for which you must substitute a value.
windowSpec	Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks graphical user interface.
Edit menu	Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.

Special Symbols

This book uses the following symbols to designate certain items or relationships:

Examples	Description
File → New	Indicates the name of an item (New) on a menu (File).
<Return> key <Select> button <Operate> menu	Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.
<Control>-<g>	Indicates two keys that must be pressed simultaneously.
<Escape> <c>	Indicates two keys that must be pressed sequentially.
Integer>>asCharacter	Indicates an instance method defined in a class.
Float class>>pi	Indicates a class method defined in a class.

Mouse Buttons and Menus

VisualWorks supports a one-, two-, or three-button mouse common on various platforms. Smalltalk traditionally expects a three-button mouse, where the buttons are denoted by the logical names <Select>, <Operate>, and <Window>:

<Select> button	*Select* (or choose) a window location or a menu item, position the text cursor, or highlight text.
<Operate> button	Bring up a menu of *operations* that are appropriate for the current view or selection. The menu that is displayed is referred to as the *<Operate> menu*.
<Window> button	Bring up the menu of actions that can be performed on any VisualWorks *window* (except dialogs), such as **move** and **close**. The menu that is displayed is referred to as the *<Window> menu*.

These buttons correspond to the following mouse buttons or combinations:

	3-Button	2-Button	1-Button
<Select>	Left button	Left button	Button

	3-Button	2-Button	1-Button
<Operate>	Right button	Right button	<Option>+<Select>
<Window>	Middle button	<Ctrl> + <Select>	<Command>+<Select>

Getting Help

There are many sources of technical help available to users of VisualWorks. Cincom technical support options are available to users who have purchased a commercial license. Public support options are available to both commercial and non-commercial license holders.

Commercial Licensees

If, after reading the documentation, you find that you need additional help, you can contact Cincom Technical Support. Cincom provides all customers with help on product installation. For other problems there are several service plans available. For more information, send email to supportweb@cincom.com.

Before Contacting Technical Support

When you need to contact a technical support representative, please be prepared to provide the following information:

- The *version id,* which indicates the version of the product you are using. Choose **Help → About VisualWorks** in the VisualWorks main window. The version number can be found in the resulting dialog under **Version Id:**.
- Any modifications (*patch files*) distributed by Cincom that you have imported into the standard image. Choose **Help → About VisualWorks** in the VisualWorks main window. All installed patches can be found in the resulting dialog under **Patches:**.
- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select **copy stack** in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to technical support.

Contacting Technical Support

Cincom Technical Support provides assistance by:

Electronic Mail

To get technical assistance on VisualWorks products, send email to:

supportweb@cincom.com.

Web

In addition to product and company information, technical support information is available on the Cincom website:

http://supportweb.cincom.com

Telephone

Within North America, you can call Cincom Technical Support at (800) 727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time.

Outside North America, you must contact the local authorized reseller of Cincom products to find out the telephone numbers and hours for technical support.

Non-Commercial Licensees

VisualWorks Non-Commercial is provided “as is,” without any technical support from Cincom. There are, however, on-line sources of help available on VisualWorks and its add-on components. Be assured, you are *not* alone. Many of these resources are valuable to commercial licensees as well.

The University of Illinois at Urbana-Champaign very kindly provides several resources on VisualWorks and Smalltalk:

- A mailing list for users of VisualWorks Non-Commercial, which serves a growing community of VisualWorks Non-Commercial users. To subscribe or unsubscribe, send a message to:

  vwnc-request@cs.uiuc.edu

  with the SUBJECT of "subscribe" or "unsubscribe".

- An excellent Smalltalk archive is maintained by faculty and students at UIUC, who are long-time Smalltalk users and leading lights in the Smalltalk community, at:

  http://st-www.cs.uiuc.edu/

- A Wiki (a user-editable web site) for discussing any and all things VisualWorks related at:

  http://wiki.cs.uiuc.edu/VisualWorks

- A variety of tutorials and other materials specifically on VisualWorks at:

  http://wiki.cs.uiuc.edu/VisualWorks/Tutorials+and+courses

The Usenet Smalltalk news group, comp.lang.smalltalk, carries on active discussions about Smalltalk and VisualWorks, and is a good source for advice.

Additional Sources of Information

This is but one manual in the VisualWorks library. The Cincom Smalltalk publications website:

http://www.cincomsmalltalk.com/documentation

is a resource for the most up to date versions of VisualWorks manuals and additional information pertaining to Cincom Smalltalk.

Books on Computer Security

The following is a short list of books that the development team has found helpful in understanding the issues of security and cryptography in the course of developing these libraries.

Burnett, Steve and Paine, Stephen. *RSA Security's Offical Guide to Cryptography.* McGraw-Hill Osborne Media (March 29, 2001).

Menezes, Alfred J., van Oorschot, Paul C., and Vanstone, Scott A. *Handbook Of Applied Cryptography*. CRC Press (October 16, 1996)

Rescorla, Eric. SSL and TLS: *Designing and Building Secure Systems*.

Schneier, Bruce. *Applied Cryptography*. Addison-Wesley Pub Co. (October 13, 2000)

Ferguson, Niels and Schneier, Bruce. *Practical Cryptography*. John Wiley & Sons (March 28, 2003)

Specifications

Specific definitions of various security measures are contained in several specifications, which are available on the web. The following is a sample:

Secure Random Numbers	RFC 1750
Secure Hashes & MACS	
MD5	RFC 1321
SHA	FIPS180-1
SHA-1,256,384,512	FIPS180-2
HMAC	RFC 2104, FIPS198a
Encryption Algorithms	
Symmetric Key Algorithms	

DES	FIPS46-3
Blowfish	http://www.counterpane.com/blowfish.html]
AES	FIPS197
Public Key Alorithms	
Diffie-Hellman	RFC2631
DSA	FIPS186-2
RSA	PKCS#1, RFC 2437
Keys & Certificates	
X509	Algorithms RFC3279, Certificates RFC 3280, Attribute Certificates RFC 3281
Password Based Cryptography	PKCS#5, RFC 2898
Private-Key Information Syntax	PKCS#8
Personal Information Exchange Syntax	PKCS#12
Secure Protocols	
SSL & TLS	RFC2246
HTTPS	RFC2818

1

Introduction to Security

Overview

Providing security features for computing systems has been a standard practice. For example, an operating system typically protects access to the system, and restricts access to parts of the system and its files, by granting permissions via a password scheme. With the advent of the internet, firewall mechanisms have been used to limit access to systems.

Increasingly, software developers are required to add security features to their software. Not all data can be kept behind “locked doors,” but must be held in areas that are subject to public access, or transported across the internet, and so susceptible to access by unauthorized users.

Cryptography goes a long way toward meeting the needs of securing access to private data, and a wide variety of mechanisms have been provided by software developers. This document describes those mechanisms that have been implemented within VisualWorks, and gives guidelines for employing them within your Smalltalk application.

Cryptography provides a way to convert intelligible data into unintelligible gibberish, and then convert it back to the original intelligible data. Simple versions of cryptography have been used for centuries to send secret messages. Computers have facilitated the task of breaking such codes, but at the same time provide the ability to create codes that are virtually uncrackable by the same or improved computers.

Encryption algorithms work based on a key, which is used to convert data into apparently random bits. Either the same key (symmetric-key encryption) or another key (asymmetric-key encryption) is used to restore the encrypted data to its original form.

Used in combination, possibly together with hashing algorithms, provides the several styles of encryption security in current use.

Symmetric-key Encryption

With symmetric-key cryptography, the same key is used for encrypting and decrypting the data, just like a standard code book. Accordingly, the key must be known to all persons who are authorized to decode and view the data, whether this is one or several people. The supported symmetric-key ciphers are described in Chapter 4, “Symmetric-key Cryptography.”

Keeping the key secret from unauthorized users, yet known to authorized users, is its own security issue. There are a variety of schemes for sharing symmetric keys, descriptions of which a outside the scope of this document, with the exception of public-key cryptographic methods

Public-key Cryptography

Public, or asymmetric, key algorithms provide a solution to the key sharing problem of symmetric keys. These algorithms use a pair of keys, one public and the other private, that are used together in several ways. For example, RSA is used to encrypt a symmetric session key, DSA uses keys exclusively for digitally signing a document and verifying the signature, and Diffie-Hellman uses the keys to agree on a session key. These methods are described in Chapter 5, “Public Key Cryptography.”

Loading Security Components

Security support is provided in a collection of parcels. In general, each parcel contains support for a single security feature, such as a cipher algorithm or hash. Using the Parcel Manager, they are listed in the Security folder. The parcels are as follows.

Symmetric key ciphers:

- AES - Advanced Encryption Standard, the National Institute of Standards (NIST) block cipher
- ARC4 - “alleged RC4,” a popular public domain algorithm that is claimed by its original presenter to be the RC4 algorithm, which is proprietary to RSA Security
- Blowfish - a symmetric block cipher designed in 1993 by Bruce Schneier as a fast, free alternative to existing encryption algorithms

- DES - Digital Encryption Standard (DES) was the most common encryption during the 1980s

Public key ciphers:

- DSA - Digital Signing Algorithm, a public key algorithm that does not encrypt data but is used only for digital signatures
- DH - the Diffie-Hellman key agreement algorithm variant described in RFC 2631
- RSA - the RSA encryption algorithm

Hashes:

- MD5 - the MD5 hashing algorithm
- SHA - Secure Hash Algorithm
- SHA-256 - 256-bit SHA
- HMAC - Hashed Message Authentication Checksum

Secure Sockets:

- SSL - Secure Socket Layer support

A few other parcels are loaded by these as prerequisites.

Importing Security Components into a Name Space

All Security support classes are defined in the Smalltalk.Security name space. Application code that uses these facilities should include this name space by a general import.

You can import the Security name space either into your own name space, making these resources available to every class in that name space, or into individual classes that need to use Security features.

To import the Security name space, include this line in the **imports:** line in either the class or the name space definition:

```
private Security.*
```

For example, you may have a network client application that uses many or all of the Security support classes. A class definition, with the Security.* import, might look something like this:

```
Smalltalk.MyNamespace defineClass: #MyNetClientApp
    superclass: #{Core.Object}
    indexedType: #none
    private: false
    instanceVariableNames: 'user mailAddress proxy'
    classInstanceVariableNames: ''
    imports: 'private Security.*'
    category: 'Tools-Mail'
```

Because your application is defined in your own name space, and possibly only few of your classes require access to the security classes, it would be inappropriate to import Security into your name space. If, on the other hand, your application security code were defined in its own sub-name space, then importing Security to that name space might be appropriate.

2

Hashes and Message Digests

Overview

Secure hash algorithms are used in several ways in cryptographic security. For example, a hash to verify file integrity, by storing a hash of the file and periodically rehashing the file and comparing the digests; a different hash indicates that the file has changed. Hashes are also commonly used to generate a message digest for message authentication. Finally, hashes are frequently used as part of the combining of several random data elements to generate the seed for a (pseudo) random number generator.

In this chapter we describe the hash algorithms implemented in VisualWorks and their use. Also, we cover the HMAC message authentication.

Hash Algorithms

VisualWorks provides implementations of the popular MD5 and SHA hash algorithms, in the classes MD5, SHA (implementing SHA-1) and SHA256 (implementing 256-bit SHA).

Most of the functionality of these classes is provided by their two superclasses, MessageDigest and Hash. MessageDigest in particular provides the services protocol that you most typically use to produce a message digest using a hash algorithm. Hash then provides the basic services, the lower-level implementations used by the services protocol, that are basic to MD5 and SHA.

MD5

Class MD5 implements the MD5 algorithm, which produces a 16-byte (128-bit) message digest, and is reasonably fast and secure. While there are some known internal weaknesses, and so a potential that it will someday be broken, it remains secure, and so is generally safe to use.

SHA

Class SHA implements the SHA-1 algorithm, which produces a 160-bit message digest. It is a stronger algorithm than MD5, and is highly recommended in the cryptography community.

SHA-256

Class SHA-256 implements the SHA-256 hashing algorithm. Compared to SHA-1, it further reduces the chance of collisions by extending the size of the digest to 256 bits. It is, however, significantly slower.

Working with Hashes

There is generally little to do with hashes except to create them and to compare them.

Hash Methods

The following methods are defined in the **services** protocol.

hash: *aByteArray*
Returns a digest of the entire *aByteArray*.

hash: *aByteArray* **from:** *start* **to:** *end*
Returns a digest of the segment of *aByteArray* specified by the *start* and *end* integer values, indicating byte or character locations.

hashFrom: *aReadStream*
Returns a digest of the entire contents of *aReadStream*, which must be a byte stream, not a character stream.

hashNext: *anInteger* **from:** *aReadStream*
Returns a digest of the next *anInteger* bytes from *aReadStream*.

For convenience, these are defined as class methods as well as instance methods.

The above service messages invoke more basic (lower level) service methods. These methods are useful in more complex circumstances.

updateFrom: *aReadStream*
Updates the hash function block from all the available data on *aReadStream*.

updateWith: *aByteArray*
Updates the hash function block from all of the data in *aByteArray*.

updateWith: *aByteArray* **from:** *start* **to:** *end*
Updates the hash function block from the range of data in *aByteArray* starting at *start* up to and including *end*.

updateWithNext: *count* **from:** *aReadStream*
Updates the hash function block from the next *count* bytes of data available on *aReadStream*.

Unlike the hash methods, the update methods do not return a digest. To create the digest, you need to send a digest message to the hash algorithm once the data is accumulated:

digest
Returns a digest generated from the hash function's data block.

You can also reuse the hash class instance to generate a new digest, but you need to reset it first:

reset
Resets the hash function to its initial state.

The use of several of these messages is described in "Hashing a Data Stream Incrementally" below.

Hashing a Data Collection

The simple case of generating a digest of on a body of data assumes that all of the data is available at once. This data, represented as a `ByteArray`, is provided as an argument to one of the hash messages, and sent to an instance of a hash algorithm class.

The simplest message is `hash:`, which simply hands the data to the generator:

```
MD5 new hash: 'This is a test' asByteArray.
```

Since `hash:` is also defined as a class method, this can be shortened to the following, using `hash:` as an instance creator:

```
MD5 hash: 'This is a test' asByteArray
```

The hash: message is appropriate for generating a hash from any ByteArray. For example, you might have the contents of a file in a ByteArray, in which case you can generate a hash for the file as follows:

```
ba := '..\readme.txt' asFilename readStream binary contents.
MD5 hash: ba.
```

It is necessary for the read stream to be set to binary, for the reasons described for hashFrom: below.

If you have reason to hash a segment of a ByteArray, use hash:from:to:. For example, you could generate the hash of two characters:

```
MD5 hash: 'This is a test' asByteArray from: 6 to: 7.
```

Hashing a Complete Data Stream

To digest the complete contents of a data stream, send a hashFrom: message, with the stream as argument. The complete contents of the stream must be available for these methods to work. To generate the digest, send a hashFrom: message to the generator with the read stream as argument:

```
MD5 hashFrom: '..\readme.txt' asFilename readStream binary
```

The binary message ensures that character encoding does not affect the result, and also produces a ByteArray as the result, as required by the message. The pattern is the same for a binary file:

```
MD5 hashFrom: '\program files\PINs\PINs.exe' asFilename readStream binary
```

You can also generate a hash from a specified number of bytes from the stream.

```
MD5 hashNext: 64
    from: '\program files\PINs\PINs.exe' asFilename readStream binary
```

You can set the position in the stream as usual. For example, to skip some initial bytes you can do:

```
rs := '\program files\PINs\PINs.exe' asFilename readStream binary.
rs next: 16.
MD5 hashNext: 64 from: rs.
```

Hashing a Data Stream Incrementally

In some circumstances, a data stream is not available all at once, but is received in blocks. To generate a digest of the contents of such a stream, you generate it partially, then progressively update the digest until the entire stream has been received and hashed. The basic services are required to do this.

The specific requirements will differ based on the source of the stream data, which might involve a high-level connection protocol FTP or HTTP, or a low-level raw socket connection. Here we demonstrate the general approach, simulating an incremental hash using a reading data from several files.

```
hash := MD5 new.
hash updateFrom: '..\doc\WalkThrough.pdf' asFilename readStream binary.
hash updateFrom: '..\examples\Adapt4.pcl' asFilename readStream binary.
hash updateFrom: '..\readme.txt' asFilename readStream binary.
^hash digest.
```

This example accumulates the hash data from three sources, by successive update messages. When all of the data is accumulated, the digest is finally generated by sending a digest message to the hash instance.

The alternative update messages can be used for accumulating the block from a ByteArray, or from segments of a ByteArray or Stream. We will adapt this example later, for seeding random number generators.

Comparing to a Known Hash Value

Hashes are typically used for verification of data integrity. Consequently, a hash needs to be generated, stored, and then used for comparison with a newly generated digest.

The message asHexString, which is added to ByteArray by the security base parcel, provides the necessary conversion to a usable string. The resulting string can be stored for later comparison.

To verify the integrity of the data, such as a file, for which there is a known hash (in this case an MD5 hash), simply generate the digest and sent asHexString to the result, and then check for equality with the known hash value string:

```
(MD5 hashFrom: '\program files\PINs\PINs.exe' asFilename
    readStream binary ) asHexString =
        'E83A28DCB4F653F3189B520F16025C7B'
```

HMAC Message Digests

MACs (Message Authentication Codes) are a mechanism for validating the integrity of a message using a shared secret key. The message originator calculates a value from the message and a shared key, and sends that value along with the message. The message receiver also calculates a value from the message and key, and compares the values. If the values are the same, the message is authenticated. HMAC is a specific hash-based MAC algorithm.

Except for specifying the key, which is required, the protocol for generating a HMAC digest is the same as for hashes.

HMAC can be used with any hashing algorithm. In VisualWorks, HMAC is implemented to use either MD5 or SHA-1.

Key Selection

The HMAC specification recommends a random (or cryptographically secure pseudo-random) key or length between the output length and the block length of the hash algorithm employed.

- MD5 - between 16 bytes and 64 bytes, inclusive
- SHA-1 - between 20 bytes and 64 bytes, inclusive

If the key is longer than the maximum, the algorithm calls for hashing it, which reduces its length to the minimum.

For directions on generating a cryptographically secure random value, see “Generating a Good Random”.

Generating an HMAC

Generating the HMAC for a message is nearly as simple as generating the hash, except that you need a secret key shared between the message originator and its validator. In this example we simply us a DSSRandom:

```
secretKey := DSSRandom default next.
```

Then, the HMAC instance is created. Helper methods are provided for the common hash algorithms MD5 and SHA-1:

```
hmac := HMAC MD5.
```

To use SHA-1, this would be:

```
hmac := HMAC SHA.
```

You can use any hash algorithm, however, such as SHA-256 by sending the hash: instance creation method instead, with an instance of the algorithm as the argument:

```
hmac := HMAC hash: SHA256 new
```

Then we set the key, which must be a ByteArray. For the example, we simply take the next DSSRandom value:

```
hmac setKey: secretKey asByteArray.
```

We could combine the instance creation and random value setting using the alternate instance creation methods, MD5: and SHA:, for example:

```
hmac := HMAC MD5: DSSRandom default next asByteArray.
```

Finally, we generate the hash of the message, using any of the hashing methods described earlier:

```
validationCode := hmac hash: 'this is a test' asByteArray.
```

The returned value is a ByteArray. In general, it is more useful to provide the HMAC value as a string, which can be produced using the asHexString message:

```
(mac hash: 'this is a test' asByteArray) asHexString.
```

Validating a Message using HMAC

The message recipient will receive both the original message and the HMAC value. To validate the message, the recipient regenerates the HMAC value from the message. This requires that the recipient also has the shared secret key, which is generally shared using a separate, and secure, method.

The validation is then accomplished by comparing the regenerated validation code and comparing it to the value provided with the message. For example:

```
hmac := HMAC MD5.
hmac setKey: secretKey  asByteArray.
(hmac hash: 'this is a test' asByteArray) asHexString =
    '559B7BCC7523C7ACDD8D4265529F5140'
```

If the message is valid, this will evaluate to true.

3

Random Number Generators

Many operations involved in providing cryptographic security rely on using good, secure, random values. For example, generating secure keys for symmetric and public key ciphers relies on a computational process requiring seeding of high quality random number generators (RNG) or pseudo random number generator (PRNG). The quality of the generators and their seeding has a significant and direct impact on the security of generated keys.

VisualWorks provides a several PRNGs, but not all of them are satisfactory for use in security applications (refer to the description of Random in the *Application Developer's Guide*). `DSSRandom`, which implements the algorithm specified in the DSS standard (FIPS 186-2), does meet the standards for random number generation for secure applications. Its implementation conforms to the `Random` protocol by responding to the standard `#seed:` and `#next` messages. It extends the API to allow more flexibility and control over the range of values generated.

Note: Note well that proper seeding of the random generators is a critical requirement for any application with serious security requirements. Relying on any kind of computed seeding, *including the default seeding used in this framework*, is generally considered to be a serious security risk, and should not be relied upon in applications where security is a serious concern.

This chapter describes the use of `DSSRandom`, including suggestions for its seeding, that will help you maximize the security provided by this RNG. It also describes how to test a generated value for primality, for cases when a random prime is required.

DSSRandom

DSSRandom provides an implementation of the random number generator for the Digital Signature Algorithm (DSA). This is a cryptographically strong PRNG, suitable for use in secure applications. Note, however, that it still needs to be used carefully to ensure security.

Note: The PRNG algorithm implemented in DSSRandom conforms with the requirements of FIPS 186-2, Appendix 3, as revised in the change notice October 5, 2001. Accordingly, the number of signatures generated using a key pair does not need to be restricted, as described in the change notice.

Initializing a DSSRandom Generator

The DSSRandom is intended to be seeded with a value. Consequently, unlike Random, the seed cannot be left implicit.

Several instance creation methods are defined for DSSRandom, providing options for setting the seed and other parameters.

b: *seedBitSize*
Specifies the bit size of the seed, which is then computationally generated.

q: *upperBound* **b:** *seedBitSize*
Specifies the upper bound of pseudo-random integers to be generated, and seeds the generator with a computed value of size *seedBitSize*.

q: *upperBound* **seed:** *seedInteger*
Specifies the upper bound of pseudo-random integers to be generated, and seeds the generator with the specified seed value.

seed: *seedInteger*
Seeds the generator with the specified seed value.

You can also create an instance of DSSRandom by sending new, and then set the parameters using the equivalent instance initialization messages.

Selecting a good seed and upper bound values can be complicated, and is discussed in the following section, Generating a Good Random.

For simple cases, including testing during development, instead of generating your own seed value, the b: message generates a seed integer with the specified bit size:

```
rand := DSSRandom b: 160.
rand next.
```

The bit size must be at least 160, as required for DSS compliance.

The seed is generated from various system state data. While this may be good enough for many purposes, you need to ensure that it is adequate for your security needs, or provide a better seed.

Constraining the range of values

The q:b: and q:seed: messages constrain the range of integers within which random values can occur. If a values is specified, only random values lower than that will be generated. For example, setting the upper bound to 5 constrains the values returned to be from 0 to 4.

```
rand := DSSRandom q: 5 b: 160.
rand next.
```

The upper bound, if set at all, is typically much larger than that. The DSS standard expects it to be a large prime number, and it is typically the same as the *q* value used in generating DSA or DH keys.

Using Autogenerated Seeds

The b: and q:b: initialization and instance creation methods invoke the automatic seed generator. A good seed requires that the generator itself be seeded with random data. Without that random data, the generated seed may not be adeqate for a secure application. Accordingly, you should use caution when relying on this seed generator.

To ensure a good seed, wherever available the seed generator uses randomness sources provided by the operating system. Specifically it attempts to access the **CryptGenRandom** facility on MS Windows platforms and the **/dev/urandom** on Unix platforms (see DSSRandom class>>osGeneratedSeed). If it succeeds, the generated seed is reasonably assured to be good.

If the OS randomness source cannot be accessed, the seed generator resorts to its internal algorithm for generating a seed. Because the “random” data used by the generator in this case is not good enough for DSS security, it raises an AutogeneratedSeed warning. The purpose of the warning is to alert the user that the quality of the seeding may not satisfy application security requirements, in terms of the DSS standard.

The warning can be disruptive if the OS facilities fail, which is its purpose. To deal with the warning, developers have the following options:

- Seed the generator explicitly before using it (see class comments). This circumvents the autogenerated seed mechanism.
- Handle the AutogeneratedSeed warning. This is a proceedable warning, and so you can capture the warning and continue operation.
- Turn the warning off globally with:

```
DSSRandom seedWarning: false.
```

Note that turning off or proceeding the warning does not solve the issue of using a low quality seed, and so the security of the application is compromised. However, for some applications this might be acceptable.

Reusing the Generator

The strength of a secure PRNG resides in its capability to generate extremely long, unpredictable sequences of random numbers, given proper, truly random, seeding. Given the difficulty of ensuring quality seeding, it is desirable to keep and reuse a PRNG, rather than creating and seeding a new PRNG after getting only a few random numbers out of a previous one.

To take full advantage of the strength of DSSRandom, you should create an instance with a good seed, as described above, and keep it as a live object for use in generating later random values. This can be done in several ways, but a simple way would be to assign a DSSRandom to a shared variable, then reference it each time a new random is required.

For example, this shared variable definition defines MyPRNG with an instance of DSSRandom which, once initialized, can be referenced each time a new random is needed:

```
Security.DSSRandom defineSharedVariable: #MyPRNG
    private: false
    constant: false
    category: 'generators'
    initializer: 'DSSRandom seed: (SHA hash: ''this is a test'')
        asLargePositiveInteger'
```

Clearly the seeding needs to be better, and so you would probably use an initialization method instead of an initializer expression, as described in the Application Developer’s Guide (refer to chapter 3, “Syntax,” subsection “Shared Variables”). Also, as described below (see “Reseeding a Generator”), you should implement a scheme for reseeding the generator upon image startup.

Reseeding a Generator

At times you should reseed a PRNG generator. For instance:

- Every PRNG repeats itself eventually, so if you use the same generator for many numbers, it should periodically be reseeded.
- If you do not save your image at each shut down, then you should reseed the generator so that it does not repeat a sequence each time the image is launched.

To reseed the generator, send a seed: message to the generator, with the new seed integer value as argument. The seed can be provided from a variety of sources. Many UNIX and Linux systems run a **/dev/urandom** daemon. Seed data can be collected from it by evaluating this expression:

```
MyPRNG seed:
    ('/dev/urandom' asFilename readStream binary; next: 20)
      asLargePositiveInteger
```

On any system, you can write a file with the next random and then read it back to reseed the generator. For example, to write the file, evaluate:

```
'seed' asFilename writeStream binary;
    nextPutAll: (MyPRNG next changeClassTo: ByteArray);
    close
```

and to read it back evaluate:

```
MyPRNG seed: 'seed' asFilename readStream binary contents
    asLargePositiveInteger
```

Using the file approach, the file should be written before every image shutdown to ensure maximum randomness of the system state data that is collected, and then used to reseed the generator at system startup. A simple way to update the seed is to create a Subsystem subclass, and define setUp and tearDown methods. For example, create a subclass of UserApplication (a subclass of Subsystem) called MyPRNGSubsystem.

To use the **urandom** daemon you only need the setUp method:

```
setUp
    DSSRandom.MyPRNG seed:
        ('/dev/urandom' asFilename readStream binary; next: 20)
            asLargePositiveInteger
```

Then save the image. The next time you launch the image, the generator will be seeded from **urandom**.

For the file approach, you need both setUp and tearDown methods:

```
setUp
    DSSRandom.MyPRNG seed: 'seed' asFilename readStream binary
        contents asLargePositiveInteger

tearDown
    'seed' asFilename writeStream binary;
        nextPutAll: (DSSRandom.MyPRNG default next
            changeClassTo: ByteArray);
        close
```

Save the image, and evaluate

```
MyPRNGSubsystem activate
```

to activate the subsystem. When you exit, the seed file should be written. The setUp method is executed and reseeds the generator when you next launch the image.

Exclude the seed file from any backup routines, since you should not restore it. Restoring the file will cause a repetition of the sequence starting with that seed value.

Default Generator

It is common to create a PRNG, use it to generate one or a few values, and then destroy it. While this is fine in some cases, it is not a good use of a cryptographically strong generator, such as DSSRandom, in security applications. Creating a new generator instance for each required random reduces the security provided to the quality of the seed value, and generating a good seed can be a time consuming operation.

To facilitate better reuse of a DSSRandom, and so to make better use of its capabilities, DSSRandom holds a default instance in its default class instance variable. To access the instance, send a default message to the class. Generally this would be used to get the next random value from the generator:

```
DSSRandom default next
```

When it is first invoked, because there is no default instance (it is lazy initialized), one is created and stored to the class instance variable. Subsequent evaluations of this expression continue to use the same generator instance, making full use of its strength.

The generator is flushed at each image launch, to prevent restarting with the same seed. To ensure a good seeding of the generator, you should employ a seeding strategy such as described in the next section, "Reseeding a Generator".

Note that the automatic seeding method used for creating the default generator expects that the image has been running for a while and that it has been used heavily in an unpredictable manner. The seeding method distills the seed out of the more volatile parameters of ObjectMemory, therefore it is desirable to let object memory drift away from its initial startup state as much as possible. Since this cannot be counted on in a security sensitive production system, it is highly recommended that you use a more reliable, external seeding instead (see "Generating a Good Random" below).

Reseeding the Default Generator

To reseed the default generator, send a resetDefault or a resetDefaultFrom: message to the class.

The quality of the seed used by resetDefault assumes that the image has been up for a significant period of time and that it has been used heavily in an unpredictable manner, as mentioned above.

The resetDefaultFrom: message allows you to specify a seed source. You can use methods similar to those described for DSSRandom instances generally, although the parameter is slightly different.

To use the value provided by the **`/dev/urandom`** daemon, set the seed by evaluating this expression:

```
DSSRandom resetDefaultFrom:
    ('/dev/urandom' asFilename readStream binary; next: 20) readStream
```

To use the seed file approach, you can write the file with by evaluating:

```
'seed' asFilename writeStream binary;
    nextPutAll: (DSSRandom default next changeClassTo: ByteArray);
    close
```

Then read it back by evaluating:

```
DSSRandom resetDefaultFrom: 'seed' asFilename readStream binary
```

Using the file approach, the file should be written before every image shutdown, and then used to reseed the generator at system startup.

You can define a Subsystem to set the seed upon startup, as described above (see “Reseeding a Generator”).

Generating a Good Random

Having a good PRNG, such as DSSRandom, is important, but not necessarily sufficient. The PRNG algorithm implemented in DSSRandom is secure, but can still produce good random numbers only if used properly.

Selecting a Seed

While a good PRNG produces a good, unpredictable series of values, its value in providing security in an application depends on the security of its initial seed value.

In practical terms, here is the problem you need to overcome. Your adversary will know your PRNG algorithm, or even the precise generator (that is, DSSRandom). If your seed can also be discovered, the series of numbers generated can be reproduced. Security is effectively broken if that occurs. The solution is to use a good seed value that cannot practically be discovered.

The general approach to finding such a value is to collect data from several sources and combine them using a hash function, such as MD5 or SHA-1. The data sources should themselves produce random data.

There are several sources of random data available:

- Commercial or free sources, such as Random.org, LavaRand, or HotBits
- Collect time between keystrokes as the user types a block of text, or between mouse move events as the user moves the mouse, using a millisecond clock.
- Collect data on volatile system state, such as memory usage, while a process is running.

Refer to the available literature for additional and more specific suggestions.

When collecting data from user or system activity, not all of the data collected is equally valuable. For example, while Time microsecondClock may return 64 bits of data, only 24 bits or less is sufficiently volatile to useful; the rest can be easily guessed or determined.

For DSSRandom, because the seed must be at least 160 bits, a SHA-1 digest of a collection of data is appropriate. Assuming data collected from three sources, you can hash the data as follows:

```
hash := SHA new.
hash updateFrom: hotBitsData.
hash updateFrom: keyboardEntryData.
hash updateFrom: systemMemoryStateData.
^hash digest.
```

You will need to create the data collection procedures and ensure that they return suitably random values.

Primality Testing

In some contexts a random value is expected also to be a prime number. For these contexts, VisualWorks provides two primality test classes: MillerRabin and PrimeSieve.

The Miller-Rabin primality test is a statistical test, returning a “probably prime” result. By iterating the test several times, the probability of being really prime is increased. Testing in this way is much faster than doing exhaustive verification for large numbers.

While Miller-Rabin is pretty fast, it can be sped up by first passing the candidate prime through a prime sieve. This tests the candidate number for divisibility by some number of known primes, such as primes less than 100. This eliminates a large number of candidates. Candidates that pass through the sieve are then subjected to additional testing, such as Miller-Rabin.

Configuring Miller-Rabin Testing

MillerRabin requires a random number generator, which it uses for generating values to use in testing, and a number of iterations to perform in verifying primality. Instead of specifying a random number generator, you can specify an upper bound for values to check for primality, and a default random number generator is created. These four instance creation methods provide these requirements:

max: *upperBound*

Creates an instance with a default instance of DSSRandom to generate values no larger than *upperBound*, and set to run 50 iterations.

random: *aPRNG*

Creates an instance with *aPRNG* as the random number generator, and set to run 50 iterations.

t: *iterations* **max:** *upperBound*

Creates an instance with a default instance of DSSRandom to generate values no larger than *upperBound*, and set to run *iterations* iterations.

t: *iterations* **random:** *aPRNG*

Creates an instance with *aPRNG* as the random number generator, and set to run *iterations* iterations.

The *upperBound* value will typically be used in the context of DSA or Diffie-Hellman key generation, in which case it will be the same as the *q* value specified in those contexts.

The default value for t: is 50, and is typically adequate. The value can be increased for added testing, or reduced for increased speed.

A simple instance can be created by specifying a simple PRNG, such as:

```
mr := MillerRabin random: (DSSRandom b: 160).
```

Once the tester is created, send it a value: message with the value to be tested for primality. The return value is a Boolean, either true if the value tests as prime, or false otherwise. To generate a random prime value, you can combine testing with random number generating, for example:

```
mr := MillerRabin random: (DSSRandom b: 160).
gen := DSSRandom seed: aSeedValue.
[ candidate:= gen next.
    mr value: candidate ] whileFalse.
^ candidate
```

The value retuned is a generated random number that the test verifies as prime.

Configuring Prime Sieve Testing

Testing successive randomly generated values in search of a large prime can take a significant amount of time. PrimeSieve implements a technique for speeding up the process by quickly eliminating values that are divisible by a known set of prime numbers. Candidate values that pass through the sieve are then subjected to additional testing, such as that provided by a MillerRabin test instance.

An instance of PrimeSieve is created with either of these methods:

on: *aPrimalityTest*

Creates a PrimeSieve test that submits values that pass through the sieve to further testing by *aPrimalityTest*.

on: *aPrimalityTest* **boundedBy:** *upperBound*

Same as on: but rejects (fails) all values greater than *upperBound*.

Because the only other primality test provided is MillerRabin, an instance of that class would be used to create the PrimeSieve instance. For example:

```
ps := PrimeSieve on: ( MillerRabin random: ( DSSRandom b: 160 ) ).
```

As for Miller-Rabin testing, the tese is performed by sending a value: message to the tester with the value to be tested. Again, included with a PRNG for generating a large prime, this can be done as follows:

```
mr := MillerRabin random: (DSSRandom b: 160).
ps := PrimeSieve on: mr.
gen := DSSRandom seed: aSeedValue.
[ candidate:= gen next.
    ps value: candidate ] whileFalse.
^ candidate
```

By default, PrimeSieve tests against the primes under 100, which are held in the Primes shared variable.

4

Symmetric-key Cryptography

Overview

Symmetric-key cryptography uses the same key value for both encrypting and decrypting data.

There are two types of symmetric-key ciphers: block and stream. Block ciphers encrypt a full, fixed-size block at a time. Since the block is fixed-size, a final block must be padded if it is smaller than that size. A Stream cipher encrypts one byte at a time.

Implemented block ciphers are:

- AES
- Blowfish
- DES

Implemented stream ciphers are:

- ARC4 (“alleged” RC4)

Generating Keys

A random number is usually used as one-time session keys for the encryption of communication channels. The problem is getting a good random number for the key.

One possibility is to acquire a random number from a source, such as random.org or lavarand.com. Such sources provide a source of random numbers, and can be useful if you need one infrequently.

Any cryptographically secure random number generator (RNG), such as the one provided by the DSSRandom class, is a good source of symmetric encryption keys, provided the generator is properly seeded and properly used. When a key is needed just extract required number of bytes from the generator and use that as a key.

Note, however, that even using a RNG to generate session keys, it is easy to generate keys that are relatively easy to discover. RFC 1750 discusses many of the issues and provides some recommendations, as do also several books and articles on security. Because of the complicated issues, it is generally recommended that you seek assistance from an encryption expert, if security is a serious requirement.

One major concern is that the RNG be seeded with sufficient unguessable material. Using the time, for example, is usually insufficient because it, at best, has very few unguessable bits of information. At least two, and usually several, sources of seeding material should be collected and then mixed. Browse the DSSRandom class method systemStateSeed for one example. You might also use a hashing algorithm, such as MD5, for the mixing.

Also, once created and seeded, recreating a generator circumvents the randomness built into it, again compromising security. It is recommended that you create a secure RNG, and store it in a shared variable, and reuse it whenever you need another random number. For example, in MySecureApp class, define an RNGInstance shared variable holding a DSSRandom instance:

```
Smalltalk.MySecureApp defineSharedVariable: #RNGInstance
    private: false
    constant: false
    category: 'RNG'
    initializer: 'Security.DSSRandom seed:
        Security.DSSRandom systemStateSeed'
```

(In the initialization, you will want to seed the RNG appropriately, and possibly set the number of bits returned. Refer to the DSSRandom class comment for details.)

Then, initialize the shared variable (**Method → Shared Variable → Initialize**). Now, each time a new random is required, retrieve one as usual:

```
MySecureApp.RNGInstance next.
```

Doing this makes good use of the randomizing quality of the generator, enhancing the security of the key.

Arbitrary random keys are often impractical, however, when you will need to use the key at a later time to decrypt the encrypted data. The alternatives are to try to remember the random sequences of bytes, which is difficult, or to record the key somewhere, which is a security risk. Various key derivation schemes are employed to help with that, such as the scheme based on textual passwords described in PKSC#5/RFC#2898. Security of such keys is of course lower but, still yields much better results then trying to force random passwords on users.

Symmetric-key Cipher General API

Class SymmetricCipher is an abstract class for both block and stream symmetric-key ciphers, but provides the general API for these ciphers.

Instance Creation

Because all ciphers work with a key, the basic instance creation message is:

key: *aByteArray*

Creates a new instance of the cipher class and sets its key to *aByteArray*. The key must be the right size, as determined by the specific cipher subclass.

For example, to create a bare instance of the AES cipher, get a key as a ByteArray and send key: to the class (note that the key is an obviously poor choice):

```
| key |
key := #[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ].
^AES key: key.
```

An alternative is to send new to the class, then set the key using the setKey: instance method. Setting the key in this way is necessary when using some of the instance creation methods defined in subclasses (for example, see the BlockCipher instance creation methods):

```
| key |
key := #[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ].
^AES new setKey: key.
```

Encryption/Decryption Messages

The standard encryption and decryption messages are defined in SymmetricCipher, although several are reimplemented in subclasses.

decrypt: *aByteArray*
Decrypts *aByteArray*, returning a copy containing the decrypted bytes, leaving the original ByteArray unchanged.

decryptInPlace: *aByteArray*
Decrypts *aByteArray* in the same ByteArray instead of a copy.

decryptInPlace: *aByteArray* **from:** *start* **to:** *end*
Decrypts a range of bytes in *aByteArray*, starting at *start* byte. Returns the index of the last data byte decrypted.

encrypt: *aByteArray*
Encrypts *aByteArray*, returning a copy containing the encrypted bytes, leaving the original ByteArray unchanged.

encryptInPlace: *aByteArray*
Encrypts *aByteArray* in the same ByteArray instead of a copy.

encryptInPlace: *aByteArray* **from:** *start* **to:** *end*
Encrypts a range of bytes in *aByteArray*, starting at *start* byte. Returns the index of the last data byte encrypted.

The methods encrypt:, decrypt:, encryptInPlace:, and decryptInPlace: are the most usual methods to use. encryptInPlace:from:to: and decryptInPlace:from:to: are seldom used, but would be useful, for example, in encrypting a section of larger message.

Because the various encryption mechanisms have specific requirements (such as for block alignment and padding), refer to the discussions of the particular ciphers below for examples.

Block Ciphers

Block ciphers operate on blocks of data, with the size of the block varying depending on the encryption mechanism. To provide a common basic API, block ciphers are implemented as subclasses of BlockCipher.

Block Cipher Implementations

VisualWorks provides implementations of three standard block ciphers.

AES

Advanced Encryption Standard (AES) is a the National Institute of Standards (NIST) cipher (FIPS 197) that was introduced in October, 2000, based on the Rijndael algorithm. It is intended as a replacement for the aging DES standard (FIPS 46). AES encrypts 16-byte blocks. Its key is either 16, 24, or 32 bytes long.

Blowfish

Blowfish is a symmetric block cipher designed in 1993 by Bruce Schneier as a fast, free alternative to existing encryption algorithms. Blowfish was designed to work as a drop-in replacement for DES and IDEA encryption. Blowfish encrypts 8-byte blocks. It takes a variable-length key, from 32 bits to 448 bits, making it ideal for both domestic and exportable use.

DES

Digital Encryption Standard (DES) was the most common encryption during the 1980s, but with the increasing speed of computers and discovery of weaknesses in the algorithm has been increasingly supplanted by newer encryption schemes, such as AES and Blowfish. Nonetheless, it remains a common encryption scheme. DES encrypts 8-byte blocks. It takes a 56-bit (7-byte) key.

The algorithm itself is specified with 8 byte key input, with every eighth bit used for key integrity checking. Those bits are eventually dropped to produce the actual 7-byte key. Our implementation follows the suite so the byte array argument of setKey: should be 8 bytes long.

Note: It is generally recommend to avoid DES for new applications today. It is used mostly for backward compatibility with legacy protocols and applications.

General API

Class BlockCipher provides some general API to all block cipher subclasses. The instance creation selector names indicate the cipher mode, the padding option and whether to use Triple-key EDE encryption.

Cipher Modes

Cipher modes describe ways in which an encryption algorithm is applied to produce the cipher text. The various modes are necessary in order to increase the difficulty of breaking an encryption.

CBC

Cipher block chaining. A feedback mode in which each block of plain text is XORed with the previous encrypted block, and the result is encrypted. The first block is XORed with an initialization vector.

ECB

Electronic code book. Each block is encrypted individually. This mode provides no security beyond that of the underlying cipher. Repeated blocks of plain text produce identical cipher text.

CFB

Cipher feedback. The previous block of cipher text is encrypted then XORed with the current block of plain text. This mode is as secure as the underlying encryption. CFB mode can use feedback less than a full block. The first block is XORed with an initialization vector.

OFB

Output feedback. Similar to CFB, except that the quantity XORed with the plan text is generated independently of both the cipher text and the plain text.

The selection of a cipher modes is very important, and the importance grows with the amount of data being encrypted. The more ciphertext the attacker collects the easier it is to crypto-analyze it.

ECB is rarely used because it is the weakest encryption mode. ECB mode exposes too much of the larger structure of the plaintext; identical chunks of plaintext produce identical chunks of ciphertext. Accordingly, ECB is not recommended unless you are encrypting very small amount of data, 10-20 bytes at most, and never when encrypting text.

CBC is the usual choice, because it hides such repetitions in the plaintext. CFB and OFB have additional interesting properties that might be desirable in certain applications (for example, they do not require the data to be padded to the block size).

All the non-ECB modes require an initialization vector (IV), which is a block worth of data that is used to start the mode. Like the key, the same IV must be used to decrypt encrypted data, but unlike the key the IV does not need to be kept secret. Often the IV is simply prepended to the ciphertext.

The following is an example of using CBC mode:

```
plaintext := 'This is the end ...' asByteArray.
key := #[1 2 3 4 5 6 7 8].
iv := #[8 7 6 5 4 3 2 1].
alice := DES newBP_CBC.
alice setKey: key;
    setIV: iv copy.
ciphertext := alice encrypt: plaintext.

bob := DES newBP_CBC.
bob setKey: key;
    setIV: iv copy.
(bob decrypt: ciphertext) asString
```

Modes are implemented as wrappers. They support the same protocols as ciphers do, so they can be used inter-changeably; you can use a cipher in some mode anywhere where you can use bare cipher. ECB mode is represented by a bare cipher instance, that is, there is no wrapper for it. You can mix and match ciphers and modes (and paddings) arbitrarily, although some modes do not need padding (CFB, OFB). (Note that padding must always be the outermost wrapper).

This example illustrates using the CFB mode:

```
(CipherFeedback on: Blowfish new)
    setKey: #[1 2 3 4];
    setIV: #[8 7 6 5 4 3 2 1] copy;
    encrypt: 'hello' asByteArray
```

Padding Options

Block ciphers specify a block size, and encrypt/decrypt only complete blocks. Frequently, however, the data to be encrypted is not block aligned, and so must be padded to make it aligned.

VisualWorks provides two mechanisms for padding a plaintext: Block Padding (BP) and Ciphertext Stealing (CS), and instance creation methods to create appropriate cipher instances, as described above. Block Padding employs a popular scheme of padding the last block with a byte value equal to the number of padding bytes, so when encrypting that many bytes are removed. Ciphertext Stealing uses a technique that allows the ciphertext to be the same length as the plaintext.

Block padding is provided by wrapping the cipher instance in an instance of a subclass of BlockCipherPadding. The wrappers provide and extended API for encrypting/decrypting to take advantage of the padding strategy.

BP

Pads the last block. If the data isn't block aligned, padding is used to fill the block. If the data is block aligned, a full block of padding is appended. Padding is a byte value equal to the number of pad bytes. The cipher instance is wrapped in a BlockPadding.

CS

Ciphertext stealing. This is a padding method that allows returning a cipher the same size as the unpadded plain text. The cipher instance is wrapped in a CBCCiphertextStealing or ECBCiphertextStealing instance.

Unpadded

If one of the padding, BP or CS, instance creation method is not selected, padding must be provided by the application, unless the data is ensured to be block aligned.

Triple-key EDE Encryption

Another encryption option is triple-key encryption. Encryption is done in three passes using three keys. The first pass encrypts using the first key, the second pass decrypts the result using the second key, and the third pass encrypts that result using the third key. Accordingly, this pattern is referred to as 3EDE.

The main purpose of 3EDE is to strengthen the underlying cipher by effectively extending its key. For example, DES 3EDE has a key size of 21 bytes and appears to be significantly harder to break than a single DES key. 3EDE has been used mostly to strengthen DES when there were no good alternatives. Presently, AES is generally preferred because it allows both longer keys and is significantly faster than DES.

Instance Creation Methods

The following instance creation methods create block ciphers with the indicated cipher mode and padding option. These messages are sent to the appropriate encryption class (DES, AES

new3EDE_CBC
Creates a new block cipher using triple EDE encryption in outer CBC mode, without padding.

newBP_3EDE_CBC
Creates a new block cipher using triple EDE encryption in outer CBC mode, with block padding.

newBP_CBC
Creates a new block cipher in CBC mode, with block padding.

newBP_ECB
Creates a new block cipher in ECB mode, with block padding.

newCBC
Creates a new block cipher in CBC mode, without block padding.

newCFB
Creates a new block cipher in CFB mode, without block padding.

newCS_3EDE_CBC
Creates a new block cipher using triple EDE encryption in outer CBC mode, with ciphertext stealing.

newCS_CBC
Creates a new block cipher in CBC mode, using ciphertext stealing.

newCS_ECB
Creates a new block cipher in ECB mode, using ciphertext stealing.

newECB
Creates a new block cipher in ECB mode, without block padding.

newOFB
Creates a new block cipher in OFB mode, without block padding.

Besides instance creation methods, BlockCipher provides a few instance methods.

Since the block size varies with the encryption method, this method is useful:

blockSize
Returns the block size, as specified by the specific cipher class.

The following methods are reimplemented in BlockCipher to account for block requirements.

decryptInPlace: *aByteArray* **from:** *start* **to:** *end*
Decrypts a range of bytes in *aByteArray*, starting at *start* byte. Blocks are decrypted up to and including the block containing *end*. The range is expected to be a multiple of blockSize. Returns the index of the last data byte.

encryptInPlace: *aByteArray* **from:** *start* **to:** *end*
Encrypts a range of bytes in *aByteArray*, starting at *start* byte. Blocks are encrypted up to and including the block containing *end*. The range is expected to be a multiple of blockSize. Returns the index of the last data byte.

Encrypting and Decrypting with Block Ciphers

An instance of a block cipher class, such as AES, can be created using any of the BlockCipher instance creation messages. Creating with block padding is generally the simplest and most common approach. Then set the key value (this is a poor key choice, used for illustration only).

```
| text key cipher ciphertext |
text := (Filename named: '..\readme.txt') asLogicalFileSpecification
      readStream contents asByteArray.
key := #[1 2 3 4 5 6 7 8 9 0 11 12 13 14 15 16 ].
cipher := AES newBP_ECB.
cipher setKey: key.
```

Then, to encrypt the text, send either an encrypt: or an encryptInPlace: message to the cipher with the text as argument. To encrypt the text in a new ByteArray, leaving the original unchanged, use encrypt::

```
ciphertext := cipher encrypt: text
```

To replace the text in the original ByteArray with the encrypted text, use encryptInPlace::

```
cipher encryptInPlace: text
```

Decrypting is done the same way, only using either decrypt: or decryptInPlace:. For example, given a ciphertext:

```
| key cipher plaintext |
key := #[1 2 3 4 5 6 7 8 9 0 11 12 13 14 15 16 ].
cipher := AES newBP_ECB.
cipher setKey: key.
^plaintext := cipher decrypt: ciphertext
```

Providing Custom Padding

Instead of using one of the provided padding methods, you can provide your own. The main problem is to know what to remove when decrypting, to ensure that the plaintext is restored properly.

A simple technique is to pad with a byte value that you know will not occur in the plaintext. For example, if the plaintext contains only printable text characters, you can be sure that byte value 0 does not occur. Use that value as your padding character, and then strip it upon decrypting.

```
| plaintext paddedtext ciphertext decipheredtext key cipher size newsize |

    plaintext := 'This is only a test.' asByteArray.
    size := plaintext size.

    "configure cipher"
    key := #[ 1 2 3 4 ].
    cipher := Blowfish newECB.
    cipher setKey: key.

    "create new ByteArray to padded size with padding char"
    newsize := ( size // cipher blockSize ) + 1 * cipher blockSize.
    paddedtext := ByteArray new: newsize withAll: 0.

    "create padded plaintext and encrypt"
    paddedtext replaceFrom: 1 to: size with: plaintext startingAt: 1.
    ciphertext := cipher encrypt: paddedtext.

    "decrypt and strip padding"
    decipheredtext := cipher decrypt: ciphertext.
    decipheredtext := decipheredtext copyUpTo: 0.
```

Stream Ciphers

Stream ciphers operate on a single byte at a time. Using a key as seed, a RNG creates a key stream, which is then used to encrypt. From the key and the RNG, the same key stream is generated to decrypt the stream.

ARC4

The RC4 algorithm is a trade secret of RSA Security. ARC4, or “alleged RC4,” is in the public domain, originally presented anonymously as the RC4 algorithm (hence “alleged”).

ARC4 uses a variable length key, up to 256 bytes. Beyond that, additional bytes are ignored. The algorithm generates a *key stream* from the key, which is then used for encryption/decryption by XORing a byte on the stream with a byte from the key. The stream must be restarted for decryption.

Typically, a stream cipher key will be used only once to encrypt and decrypt data. Reusing the key compromises the security of the cipher, and reusing the key for different data completely destroys security of any stream cipher, because of the basic nature of XOR:

```
(a xor b) xor (a xor c) = (b xor c)
```

So, given two pieces of data that have been encrypted with the same key, XORing them together effectively removes encryption provided by the key, and leaves a simple XOR of the two plaintexts, which is fairly simple to break. Therefore even though a stream cipher like ARC4 is a very attractive tool for its ease of use and speed, one must be very careful when using it.

To encrypt a plaintext, create the cipher, set the key, and send a `encrypt:` message to the cipher with the text as argument:

```
| plaintext key cipher ciphertext |
plaintext := 'This is only a test.' asByteArray.

key := #[1 2 3 4 5 6 7 8 9 0 11 12 13 14 15 16 ].

cipher := ARC4 key: key.
ciphertext := cipher encrypt: plaintext
```

Because encryption/decryption is symmetric, using by XORing text with the key stream, the stream must be reset, which is done by regenerating the key stream by creating a new instance of the cipher. Given the key, the process is the same, only sending `decrypt:` instead of `encrypt:`.

```
| plaintext key cipher ciphertext |
ciphertext := aCipherText.

key := #[1 2 3 4 5 6 7 8 9 0 11 12 13 14 15 16 ].

cipher := ARC4 key: key.
plaintext := cipher decrypt: ciphertext
```

Password-based Encryption and Authentication

In many applications use passwords as part of their security system. Passwords alone are vulnerable to a variety of attacks, because they are usually chosen from a relatively small space. Also, passwords are not usually directly applicable to the usual cryptographic systems. So, some further processing of passwords is necessary to make the usable and to provide the desired security.

PKCS #5 is the RSA recommendation for a password-based encryption standard. This recommendation is implemented in VisualWorks in the PBC class.

PKCS5 combines “salt” and interaction count methods with passwords to form the basis of password-based key generation. The salt effectively prevents precomputation of the encryption key, and so prevents a dictionary attack. The iteration count specifies the number of times the key generation function is applied, and is usually at least 1000. While the additional cost of generating an individual key is minimal, the cost for mounting an attack is very high.

The PKCS #5 recommendation applies to both message encryption and message authentication. The VisualWorksimplementation includes both encryption and message authentication, and implements both version 1 and version 2 of the recommendation. Version 1 is recommended only for compatibility with old applications; version 2 is recommended for all new applications.

Loading PKCS Support

To load PKCS support, load the PKCS5 parcel, in the **Security** group in the Parcel Manager.

PKCS Encryption

PKCS #5 recommendations apply to symmetric key encryption schemes. VisualWorks implements several encryption schemes for password-based encryption, making it simple to use these facilities.

In general, you create an instance of PBC configured for the desired encryption scheme. Then, set the necessary parameters and request the encryption.

The following instance creation methods create a PBC instance with specific encryption specifications:

pbes2WithHMAC_SHA1AndDES_CBC
Creates a PBC instance configured to encrypt according to PKCS #5 version 2 (PBES2), using HMAC-SHA1 for random number generation, and encrypting using DES with cipher block chaining.

pbes2WithHMAC_SHA1AndDES_EDE3_CBC
Creates a PBC instance configured to encrypt according to PKCS #5 version 2 (PBES2), using HMAC-SHA1 for random number generation, and encrypting using DES with cipher block chaining and triple key encryption.

pbeWithMD5AndDES_CBC
Creates a PBC instance configured to encrypt according to PKCS #5 version 1.5 (PBES1), MD5 for key generation, and encrypting using DES with cipher block chaining. This is recommended only for compatibility with existing applications.

pbeWithSHA1AndDES_CBC
Creates a PBC instance configured to encrypt according to PKCS #5 version 1.5 (PBES1), SHA-1 for key generation, and encrypting using DES with cipher block chaining. This is recommended only for compatibility with existing applications.

Send one of these messages to the PBC class to create and instance of the encryptor. For example:

```
pbc := PBC pbes2WithHMAC_SHA1AndDES_EDE3_CBC.
```

Unless changed, the encryption uses the default 1000 iterations. To change the number of iterations, send a count: message to the generator:

```
pbc count: 1500
```

The minimum iteration count is recommended to be 1000.

To encrypt a message, send one of these messages to the generator:

encrypt: *msgByteArray* **password:** *pwdByteArray*
Return a ciphertext of a *msgByteArray* encrypted using *pwdByteArray* and a default salt.

encrypt: *msgByteArray* **password:** *pwdByteArray* **salt:** *saltByteArray*
Return a ciphertext of a *msgByteArray* encrypted using *pwdByteArray* and *saltByteArray*.

For example:

```
pwd := ‘a well-kept secret’ asByteArray.
salt := DSSRandom default next asByteArray.
msg := 'Message in a Bottle' asByteArray.

ciphertext := pbc encrypt: msg password: pwd salt: salt.
```

To decrypt a message, send this message to a PBC instance with the same configuration:

decrypt: *cipherByteArray* **password:** *pwdByteArray* **salt:** *saltByteArray*
Return a plaintext of *cipherByteArray*.

For example:

```
pbc := PBC pbes2WithHMAC_SHA1AndDES_EDE3_CBC.
pwd := 'a well-kept secret' asByteArray.
salt := DSSRandom default next asByteArray.

message := pbc decrypt: ciphertext password: pwd salt: salt.
```

Message Authentication

PBC also implements the PKCS #5 recommendation for message authentication. VisualWorks provides an implementation of the recommended schemes.

The authentication scheme is based on a password, a salt, and an iteration count, from which are produced a key. The key is then used to generate the digest.

As for password-based encryption, you create a correctly configured instance of PBC, set the necessary parameters, then create the signature. The message authentication key is derived from the password and other information.

One instance creation method is available for creating and configuring the PBC instance:

pbmac1WithHMAC_SHA1AndHMAC_SHA1
Creates a PBC instance configured to use HMAC for both key generation and digest generation.

Given the PBC instance, there are three messages for producing the message authentication code.

sign: *msgByteArray* **password:** *pwdByteArray*
Returns an association with a generated salt as key and the message signature as value. The salt is generated deterministically as specified in PKCS #5 version 2.

sign: *msgByteArray* **password:** *pwdByteArray* **salt:** *saltByteArray*
  Returns a message signature generated from *msgByteArray*, *pwdByteArray* and *saltByteArray*. The derived key length is determined by the key length requirement of the underlying hash function.

sign: *msgByteArray* **password:** *pwdByteArray* **salt:** *saltByteArray* **keyLength:** *length*
  Returns a message signature generated from *msgByteArray*, *pwdByteArray* and *saltByteArray*. The key length is the desired length of the derived key.

To verify a message from its signature, use the following corresponding messages:

verify: *sigByteArray* **of:** *msgByteArray* **password:** *pwdByteArray*
  Returns a Boolean indicating whether *sigByteArray* matches the signature generated from *msgByteArray* and *pwdByteArray* with a deterministically computed salt.

verify: *sigByteArray* **of:** *msgByteArray* **password:** *pwdByteArray* **salt:** *saltByteArray*
  Returns a Boolean indicating whether *sigByteArray* matches the signature generated from *msgByteArray*, *pwdByteArray* and *saltByteArray*.

verify: *sigByteArray* **of:** *msgByteArray* **password:** *pwdByteArray* **salt:** *saltByteArray* **keyLength:** *length*
  Returns a Boolean indicating whether *sigByteArray* matches the signature generated from *msgByteArray*, *pwdByteArray* and *saltByteArray*, with the derived key of size *length*.

For example, using the first method,

```
pwd := 'my secret password' asByteArray.
msg := 'Message in a bottle'.
pbc := PBC pbmac1WithHMAC_SHA1AndHMAC_SHA1.

saltAndSignature := pbc sign: msg password: pwd.
```

Because this approach generates the salt, you may need to be able to retrieve both the salt and the signature:

```
salt := saltAndSignature key.
signature := saltAndSignature value.
```

To verify a signature you have the option of either generating a salt, or using the previously generated salt. You must have the password. The first for the first approach, evaluate:

```
pbc verify: signature of: msg password: pwd
```

If the salt is provided along with the signature and message, you can use:

```
pbc verify: signature of: msg password: pwd salt: salt.
```

Rather than use the deterministically computed salt, you can provide your own, in which case it must be known to the verifier. For example:

```
pwd := 'my secret password' asByteArray.
salt := DSSRandom default next asByteArray.
msg := 'Message in a bottle'.
pbc := PBC pbmac1WithHMAC_SHA1AndHMAC_SHA1.

signature := pbc sign: msg password: pwd salt: salt.
```

Provided the message, signature and salt, and knowing the password, the receiver can verify the messages by evaluating:

```
pbc := PBC pbmac1WithHMAC_SHA1AndHMAC_SHA1.

verify: signature of: message password: password salt: salt
```

5

Public Key Cryptography

While symmetric-key encryption can keep your data safe, if you need to share the key there is the risk of the key falling into the wrong hands, thus again compromising security. Various measures can be taken to protect a key that you distribute, but they can be difficult to implement.

Public-key, or asymmetric-key, cryptography uses two keys, a private and a public key, to partially solve the key-distribution problem. One key is used to encrypt data, and the other is used to decrypt it. By distributing the public key but keeping the private key private, a message encrypted with the public can be sent, and only the holder of the private key can read it. In this way, the public key can be freely distributed, without concern about it falling into the wrong hands.

In the other direction, a message encrypted with the private key but decrypted with the public key, security concerns are the same as for symmetric key; anyone who has the public key can decrypt the messages. However, in this direction, public-key encryption has use in digitally signing a document, since only the private-key holder could encrypt data that can be decrypted with the public key. This is discussed more in "Digital signatures" below and in the sections for each cipher.

Because public-key encryption algorithms are much slower than symmetric-key algorithms, public-key cryptography is not generally used for bulk data. Instead, bulk encryption is done using symmetric-key encryption, and public-key mechanisms are used to form a digital envelope that wraps the bulk data (possibly encrypted) together with an encrypted session key, where the session key is the key for decrypting the bulk data.

Digital signatures

A digital signature employs public-key cryptography to validate the source of a document, providing both authentication and nonrepudiation protection; the presence of data encrypted using a private key verifies that it was generated by the hold of that key. The verification authenticates the data as from the claimed source, but also makes it very difficult for the source to deny having originated it.

Digitally signing does not require encryption. DSA, for example, does not encrypt anything, but can be used to sign either encrypted or unencrypted data.

Generally speaking, a digital signature involves a fixed-size digest, consisting of the data, a private key, and possibly some other data, which are then combined in some way, usually a hash, that can be used to verify that the data was unchanged and originated from the putative source. The details differ for the various mechanisms, and will each be described more fully under the sections RSA, DSA, and DH.

Generating Keys

Public/private key pairs are generated by subclasses of KeyGenerator. Key generator classes are implemented for RSA and DSA. The output from these generators are subclasses of EncryptionKey specific to the algorithm.

To generate keys, create an instance of the appropriate key generator class by sending a variant of keySize:random:primalityTest: to the class. The minimal instance creation requires specifying the key size, for example:

```
    rsaKeyGen := RSAKeyGenerator keySize: 1024.
```

where the argument is the key size in bits.

In this form, a default random number generator and primality test is used. Note that this form is only suitable for use during development or for personal use, and that in a production application you should instantiate a good random number generator and seed it with a good, random seed value. (Refer to Chapter 3, “Random Number Generators.”)

The above method uses default seeding for the DSSRandom PRNG. To provide your own seed value, you need to explicitly provide a random number generator, and possibly primality testing, by sending either keySize:random: or keySize:random:primalityTest:. For example:

```
rsaKeyGen := RSAKeyGenerator keySize: 1024
    random: ( DSSRandom seed: aSeedValue )
    primalityTest: ( MillerRabin random: ( DSSRandom b: 160 ) )
```

Given a generator, generate the keys by sending a privateKey or publicKey message to the generator:

```
privateKey := rsaKeyGen privateKey.
publicKey := rsakeyGen publicKey.
```

Either key can be requested first, upon which the calculation is performed. Both keys are then cached in the generator for access using these accessor methods.

Because the generator caches the keys, to generate a new set of keys with the same generator instance, send #flush to the generator to flush the generated keys and parameters. Then request the new keys:

```
rsaKeyGen flush.
newPublicKey := rsaKeyGen publicKey.
newPrivateKey := rsaKeyGen privateKey.
```

Generating keys is a computationally demanding process and the time it takes is proportional to the size of the keys being generated. To facilitate user feedback during generation, the generator signals various object events through the various stages of the process. The class method #eventsTriggered lists the kinds of events that are signaled.

Exporting and Importing Keys

There are currently no automatic general mechanisms for exporting or importing keys between VisualWorks and non-VisualWorks environments. Rather, given the component values for a key, you can create an instance of that key by providing the values to the relevant instance creation method, such as for DSAPublicKey or DSAPrivateKey. Browse the instance creation methods for these classes for the required parameters. Instance variable names follow the conventions in the relevant specifications.

RSA

RSA is a public-key algorithm for encrypting data. Because the RSA algorithm is quite slow compared with symmetric key encryption algorithms, RSA is seldom used to encrypt bulk data. Instead, RSA is used to encrypt a session key, which is then used by a symmetric algorithm to decrypt data. The encrypted data and encrypted session key comprise a “digital envelope.”

Basic encryption and decryption involved creating an instance of RSA and setting its key, then sending an encrypt: or decrypt: message to the RSA instance with the data as argument. The data must be a ByteArray.

For keeping data secret, the public key is used to encrypt:

```
rsa := RSA new.
rsa publicKey: anRSAPublicKey.
anEncryptedByteArray := rsa encrypt: aByteArray
```

Then the data can only be decrypted by the holder of the private key:

```
rsa := RSA new.
rsa privateKey: anRSAPrivateKey.
aByteArray := rsa decrypt: anEncryptedByteArray
```

Creating a Digital Envelope

A digital envelope consists of two pieces of information: the data encrypted with a symmetric cipher and a session key, and an encrypted version of the session key. A new session key should be generated for each exchange.

```
| text sessionKey cipher ciphertext rsaCipher cipherkey |
text := (Filename named: '..\readme.txt') asLogicalFileSpecification
    readStream contents asByteArray.
sessionKey := #[1 2 3 4 5 6 7 8 9 0 11 12 13 14 15 16 ].
cipher := AES newBP_ECB.
cipher setKey: sessionKey.
ciphertext := cipher encrypt: text.
rsaCipher := RSA new publicKey: aPublicKey.
cipherkey := rsaCipher encrypt: sessionKey.
envelope := (Array with: ciphertext with: cipherkey).
```

How the two parts, the encrypted data and encrypted session key, are stored or transmitted as a unit is up to you. For the example, an Array is convenient, which is held in the workspace variable envelope.

Upon receiving this message, the recipient uses the matching private key to recover the session key and decrypt the data:

```
| sessionKey cipher plaintext rsaCipher |
rsaCipher := RSA new privateKey: aPrivateKey.
sessionKey := rsaCipher decrypt: (envelope at: 2).
cipher := AES newBP_ECB.
cipher setKey: sessionKey.
^plaintext := ( cipher decrypt: (envelope at: 1)) asByteString
```

Digitally Signing with RSA

A message encrypted with a private key constitutes a “digital signature,” because only the holder of that private key could have produced that encrypted message (assuming the key has been kept secure).

Because RSA is a fairly slow algorithm, you seldom encrypt an entire message to form the digital signature. Instead, you encrypt a digest of the message, which is a relatively small, fixed sized hash of the message. Class RSA provides a simple API for generating creating and verifying a signature for a message.

The RSA signature consists of the encrypted hash. To verify the signature, it is first decrypted, leaving the digest of the message. Then the message is itself digested, and the two digests are compared. If they are identical, then the signature is verified.

To sign a message, which must be a ByteArray, create an instance of RSA, assign the private key, and set the hash algorithm. Two methods, useMD5 and useSHA, specify the hash algorithm used as MD5 or SHA-1, respectively. Then send a sign: message to the RSA instance with the text to sign as argument:

```
| text rsaCipher |
text := (Filename named: '..\readme.txt') asLogicalFileSpecification
     readStream contents asByteArray.
rsaCipher := RSA new privateKey: aPrivateKey.
rsaCipher useMD5.
^signature := rsaCipher sign: text.
```

The signature is returned as a ByteArray.

To verify a signature, again create an RSA instance, but set the public key, and the hash algorithm. Then send a verify:of: message to the RSA instance with the signature and the signed message as arguments. The message returns a Boolean: true if the signature is verified, false otherwise.

```
| text rsaCipher |
text := (Filename named: '..\readme.txt') asLogicalFileSpecification
     readStream contents asByteArray.
rsaCipher := RSA new publicKey: aPublicKey.
rsaCipher useMD5.
^rsaCipher verify: signature of: text.
```

Using a Custom Hash

The approach illustrated above employs either MD5 or SHA-1, as specified in the RSA instance by sending useMD5 or useSHA. Other hash algorithms are available, such as SHA256, and you can implement your own.

There are two ways to use an alternate hash algorithm to create the digest.

First, send a hashAlgorithm: message to the RSA instance with an instance of the algorithm. For example:

```
| text rsaCipher |
text := (Filename named: '..\readme.txt') asLogicalFileSpecification
      readStream contents asByteArray.
rsaCipher := RSA new privateKey: aPrivateKey.
rsaCipher hashAlgorithm: SHA256 new.
^signature := rsaCipher sign: text.
```

There is a restriction on this approach, that the hash class must implement a derEncodedDigestInfo method that returns a ByteArray with the Distinguished Encoding Rules (DER) information, as prescribed by RSA (PKCS#1/RFC#2437).

Note that SHA256 does not implement this method, and so the above example raises an exception.

An alternative is to digest the message separately, and then use that to create the signature. To do this, leave the hash algorithm unspecified, or set it to nil (rsaCipher hashAlgorithm: nil). If the hash algorithm is not specified, RSA assumes the message is already digested.

```
| text hash rsaCipher |
text := (Filename named: '..\readme.txt') asLogicalFileSpecification
      readStream contents asByteArray.
hash := SHA256 new hash: text.
rsaCipher := RSA new privateKey: aPrivateKey.
^signature := rsaCipher sign: hash.
```

Verifying the signature goes as before, except that the signature is verified against the hash:

```
| text hash rsaCipher |
text := (Filename named: '..\readme.txt') asLogicalFileSpecification
      readStream contents asByteArray.
hash := SHA256 new hash: text.
rsaCipher := RSA new publicKey: aPublicKey.
^rsaCipher verify: signature of: hash.
```

DSA

Digital Signature Algorithm (DSA) is a public-key algorithm only for creating digital signatures. It does not encrypt or decrypt a message. Instead, it combines a SHA-1 digest of the message with a random value and the private key, to produce two values, referred to as *r* and *s*, which are the signature.

To verify the signature, a SHA-1 digest of the message created, which is then combined with the public key and the s value. The result is a value, referred to as *v*. The signature is verified if v is equal to *r*; otherwise not.

The DSA class hides most of this complexity, providing an easy way to create and verify a DSA signature.

Generating DSA key pairs

Generating a DSA key pair follows the general pattern shown above ("Generating Keys"). Create an instance of DSAKeyGenerator with an appropriate key size, which must be a multiple of 64 between 512 and 1024. Then request the public and/or private key.

```
dsagen := DSAKeyGenerator keySize: 1024.
aPublicKey := dsagen publicKey.
aPrivateKey := dsagen privateKey.
```

The keys are cached in the DSAKeyGenerator instance. To use the same generator to produce another key pair, you must flush the cache first:

```
dsagen flush.
```

Only the keys are cleared, leaving the key size unchanged.

Digitally Signing with DSA

To digitally sign a message with DSA, all you need to do is create a DSA instance, set the private key, and send a sign: message with the message to be signed as argument:

```
| text dsa |
text := (Filename named: '..\readme.txt') asLogicalFileSpecification
    readStream contents asByteArray.
dsa := DSA new privateKey: aPrivateKey.
^signature := dsa sign: text
```

Because DSA uses only SHA-1 for the message digest, there is no need to specify the hash.

The signature returned is an instance of DSASignature, which holds two LargePositiveIntegers, *r* and *s*.

Verifying a DSA signature

Verifying the signature is similar: create a DSA instance with the public key, then send is a verify:of: message with the signature and signed message as arguments.

You will need to construct a DSASignature instance from the *r* and *s* values which you have received as the DSA signature. This is done by sending the class an r:s: instance creation method.

```
| text dsa signature |
text := (Filename named: '..\readme.txt') asLogicalFileSpecification
      readStream contents asByteArray.
"create the DSASignature from r and s"
signature := DSASignature r: r s: s.
" verify the signature "
dsa := DSA new publicKey: aPublicKey.
^dsa verify: signature of: text.
```

Diffie-Hellman key agreement

Diffie-Hellman is a different kind of public key algorithm, in that it neither encrypts nor signs a message. Instead, it allows remote parties to establish a shared secret value over an unprotected channel by exchanging public information. From that shared secret value, the two parties each create a symmetric session key to use for encrypting/decrypting a message.

The VisualWorks implementation is based on the shared secret generation algorithm as described in RFC 2631. In this variant of DH, the generator, public/private key pairs, and shared secret are derived using two large primes, *p* and *q*. These do not need to be secret, and often are precomputed to be appropriate sizes, as described below. The second prime, *q*, is added to the basic DH algorithm to increase security, as described in RFC 2631.

Basic DH Shared Secret Generation

Suppose Alice and Bob need to exchange a message in a secure fashion. Neither has a public key or a symmetric session key to exchange, so they are starting from the beginning. VisualWorks makes it simple to generate and share the necessary information and generate the shared secret and keys from which they can create a session key.

They need two large primes, referred to as *q* and *p*, and a generator, *g*, which is a large integer. The simple approach is for one party to generate the required values and for the other to accept them. To generate the values, one party, say Alice, creates an instance of the DH algorithm by:

```
aliceDH := DH new
```

which generates all three values. Alice extracts *p* and *g* and provides them to Bob:

```
p := aliceDH p.
q := aliceDH q.
g := aliceDH g.
```

Bob then uses the parameters to create his own instance of DH:

```
bobDH := DH p: p q: q g: g.
```

Note that it is important that the right parameters are assigned. In particular, *p* and *g* are the same size, but cannot be exchanged.

The secret value is established in two phases. First both parties generate their own private/public key pairs:

```
aPublicValue := aliceDH PublicValue.
bPublicValue := bobDH PublicValue.
```

The private key can usually stay hidden inside the algorithm instance.

Alice and Bob now exchange the public keys, which can be done over an unprotected communication channel, then continue with second phase in which they compute the shared secret value using the other party's public key:

```
aSharedSecret := aliceDH sharedSecretUsing: bPublicValue.
bSharedSecret := bobDH sharedSecretUsing: aPublicValue.
```

The shared secrets are both the same value, and it is computationally infeasible to produce them without knowing one of the private keys.

Note that the public value and the shared secret are the same size as the value *p*, which should be at least 512 bits, and the size of the private value is up to the size of *q*, which should be at least 160 bits. This example, using `DH new`, sets the sizes of *p* and *q* to these sizes by default. To set larger sizes, refer to "Controlling Parameter Generation" below.

Using the Shared Secret for Encryption

As said before, Diffie-Hellman is not itself used for encryption or signing. Instead, the secret value generated and shared by the communicating parties is used as *key material*; a symmetric session key has to be created from it.

Since the shared secret value is the same size as the *p* value, which should be a minimum of 512 bits, it is too large to use as the key for most symmetric ciphers. The parties must agree in advance on a method for creating a suitable key from the secret value. To maintain security, the algorithm is often very elaborate, and is left to your own imagination (referred to below simply as (CryptoClass genDHKey:).

```
| plaintext cipher key |
plaintext := (Filename named: '..\readme.txt') asLogicalFileSpecification
        readStream contents asByteArray.
cipher := AES newBP_ECB.
key := CryptoClass genDHKey: aSecret.
cipher setKey: key.
ciphertext := cipher encrypt: plaintext.
ciphertext asByteString inspect
```

For example, browse the implementation of the SSL 3.0 algorithm in nextChunkOfMaterialFor: in the SSLConnection class. It is generally recommended to have a security expert develop your algorithm.

Using pre-defined parameter values

Instead of beginning by generating the *p*, *q*, and *g* parameters using DH new, as described above, there are several reasons to use values that were previously generated. Alice might already have a DH public/private key pair that she wants to reuse. Or, she might generate the *p*, *q*, and *g* (and the keys) separately, setting specific size requirements (see "Controlling Parameter Generation" below) or using facilities other than those provided by VisualWorks.

In any case, the scenario is very much the same as described above. She still must provide Bob with the *p*, *q*, and *g* values so he can generate his own key pair. She might also have to do this, if she only has the parameters but not a key pair produced using them.

As shown above, Bob and Alice can create instances of DH using the instance creation method p:q:g: as follows:

```
aliceDH := DH new p: p q: q g: g.
bobDH := DH new p: p q: q g: g.
```

Not all Diffie-Hellman implementation use the *q* parameter, which was added to the original algorithm by the variant described in RFC 2631, to correct a weakness in the original. If the *q* parameter is not available, the DH instances can be created using the p:g: instance creation method instead:

```
aliceDH := DH new p: p g: g.
bobDH := DH new p: p g: g.
```

Then the key pairs can be generated and exchanged, and the shared secret produced, as described above.

Controlling Parameter Generation

Using DH new to generate the *p*, *q*, and *g* parameters creates them with default sizes of 512 bits for *p* and *g* (*g* is created as the same size as *p*), and 160 bits for *q*. These are recommended *minimal* sizes. As computers become faster, it becomes necessary for parameter sizes to grow to maintain a reasonable level of security. To generate parameters with specified sizes, use an instance of DHParameterGenerator.

Required parameters are bit-lengths of *q* and *p* referred to as *m* and *l*, respectively. So, for example to set the parameter sizes slightly higher than the defaults:

```
m := 172.
l := 520.
pg := DHParameterGenerator m: m l: l.
```

With the generator created, the *p* and *q* values are generated upon first access:

```
p := pg p.
q := pg q.
```

Use these values to create a new DH instance as shown above "Using pre-defined parameter values").

You can generate the values without accessing them by sending a generateP message to the generator:

```
pg generateP.
```

To generate a new set of parameters with the same generator, it has to be flushed using the message #flush. You can then generate new values:

```
pg flush.
pg generateP.
```

Similar to the key generators mentioned earlier, it can be created with a pre-existing instances of a random generator.

Events, specified by the #eventsTriggered method, allow you to monitor the progress of generation.

6

Secure Socket Layer

The SSL parcel provides implementation of Netscape's SSL protocol version 3.0.

Limitations

We now support most of the relevant SSL 3.0 cipher suites. We do not support the one cipher suite based on IDEA, the three cipher suites based on of Fortezza, nor the “exportable” variants of the strong cipher suites. This brings the list of supported cipher suites to the following:

```
SSL_RSA_WITH_NULL_MD5
SSL_RSA_WITH_NULL_SHA
SSL_RSA_WITH_DES_CBC_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA
SSL_RSA_WITH_RC4_128_MD5
SSL_RSA_WITH_RC4_128_SHA
SSL_DHE_DSS_WITH_DES_CBC_SHA
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
SSL_DHE_RSA_WITH_DES_CBC_SHA
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
SSL_DH_anon_WITH_RC4_128_MD5
SSL_DH_anon_WITH_DES_CBC_SHA
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
```

We do not support DH certificates, i.e., certificates with Diffie-Hellman parameters, however both RSA and DSA certificates are now fully supported.

With client authentication we ignore the list of acceptable certificate authority names provided by the server in the CertificateRequest message and pass whatever certificate compatible with specified certificate types is available. It is up to the server to accept or reject it.

Usage

The main components of the public API are SSLContext, SSLSession and SSLConnection.

SSLContext represents a server/client context maintaining a collection of fairly static preconfigured communication parameters and options, like supported cipher suites, various certificate registries, etc. It is also responsible for maintaining the collection of resumable sessions for SSL servers.

SSLContext is a root component used by an application to create a suitable environment for SSL communication. Once specific parameters and options are set up an instance of SSLContext builds instances of SSLConnection upon request.

SSLConnection implements an actual connection between two parties. It is responsible for encryption/decryption of SSL records (basic units of SSL communication) and maintenance of negotiated encryption parameters. It provides the bulk of the public API: connecting, accepting connection, closing connection, secure data streams. As such it is the component that is most exposed to an SSL application.

SSLSession is usually hidden from the application, except in case where an application wants to use several connections with the same party. It is much faster to “resume” an existing session for a new connection and use previously negotiated encryption parameters than renegotiating a new parameter suite for each new connection. So for the cases when an application wants to resume an existing session, it has to maintain the SSLSession and hand it to the handshake API when connecting a new connection.

The usual usage pattern goes something like this. First an instance of a context has to be created and initialized with supported cipher suites:

```
context := SSLContext newWithSecureCipherSuites.
```

Note that the collection of cipher suites can be specified explicitly using the SSLContext class method suites:.

Then an instance of a connection is built around an input-output binary stream. This is a stream provided by the lower level communication layer, usually a stream built from a TCP socket (Note that SSL is explicitly designed for connection-oriented protocols). So given a connected socket the stream can be built like this:

```
ioStream := (aSocketAccessor asExternalConnection
    withEncoding: #binary) readAppendStream.
```

The connection is then created by the context as follows:

```
connection := context connectionFor: ioStream.
```

Now we are ready to perform an SSL handshake. Each SSL connection connects a client with a server, playing two very different roles in the SSL handshake. To get a connection to handshake as a server send it message accept. To get a connection to handshake as a client use message connect.

```
connection connect.
```

Once the handshake finishes we are almost ready to transfer data securely. All we need is to get a secure data stream from the connection. The protocol is the same as the one used for ExternalConnection (the ones used for creating socket streams), accepting the appendStream, readStream, and readAppendStream messages, returning a stream using default encoding, or a specific encoding can be ordered with the withEncoding: message, as in the following example.

```
stream := (connection withEncoding: #binary) readAppendStream.
```

Note that SSL is not designed to transport application data on its own. Instead it is to be used for tunneling an application specific protocol used on top of it. It can be a completely custom proprietary protocol specifically designed for a given application or it can be protocol well known like HTTP for example. An important requirement on the application protocol is that it has to be self-delimiting, i.e., it has to be able to determine the start and end of it's messages without any hints from SSL.

An important thing to remember is that to force the data written into a stream to leave SSL buffers and be sent to the other party the stream has to be explicitly flushed (same as with a normal socket stream).

```
stream nextPutAll: #[1 2 3 4 5].
stream flush.
data := stream next: 5.
Transcript show: data printString.
```

To close a connection send message #close to the data stream.

```
stream close.
```

SSL specification requires connections to be properly closed with a close_notify alert sent to the other party. If there are circumstances in which it is not desirable to do that, message shutdown can be sent to the SSLConnection to skip the notification. Note that close is the preferred way though, because missing close_notification creates a vulnerability to a truncation attack.

SSL Exceptions

An essential requirement for security technologies is being able to detect and stay in control when things go wrong. Therefore it is important to know how are these situations captured by the SSL framework. Exceptions are an obvious implementation choice.

SSL exceptions are all subclassed from a generic SSLException class. They are divided into 2 categories, SSLErrors and SSLWarnings. SSLErrors are fatal exceptions, i.e., if an SSLError occurs the operation in progress cannot complete and has to be aborted, usually rendering the SSL connection itself unusable.

SSLWarnings are resumable exceptions, they are meant to warn the user that there was a problem and it is up to the user (or the application) to decide if the operation should be completed or not. If the warning is a problem serious enough given the circumstances, just return from the exception handler and the operation will be aborted, otherwise resume the exception to proceed with the operation. Keep in mind that most of the warnings report serious security issues though, so think twice before ignoring a warning.

Handshake and Certificates

The example as presented above would actually fail. The reason is that the context was not set up quite right. The default context settings are tuned to provide a secure connection and this cannot be achieved without authenticating the server party. (Servers usually do not care about a client's identity so clients are not required to authenticate, by default).

Authentication is performed using X.509 certificates. Without going into detail, it is enough to know that a certificate is a digitally signed data structure containing information identifying:

- the entity that the certificate was issued for, called the subject
- a public key for a specific public key algorithm (like RSA), usually referred to as the subject public key

- identification of the entity that issued the certificate, the issuer.

The signature of the certificate is generated by the issuer, and is there to prevent modification of the certificate contents. Since a signature can only be verified using the public key of the issuer, the issuer usually also has its own public key certificate.

A set of certificates related by the subject-issuer relationship forms a certificate chain. The last certificate in the chain usually belongs to a well known *certificate authority* (CA). If a party that needs to be authenticated (usually the server) provides a certificate chain to another party (the client) that knows the certificate of the CA of the chain, then the client can verify the authenticity of the whole chain. If the server further proves possession of the private key corresponding to the subject certificate of the chain, the client can be reasonably assured of the authenticity of the server (provided that the private keys of the chain certificates were not compromised). These are the basic ideas behind the certificate-based authentication used by SSL. (For details on X.509 certificates, refer to RFC 2459).

In order to perform server authentication the client needs to be set up with a collection of “trusted” certificate authority certificates. This collection is maintained in an instance of X509Registry.

It is absolutely essential that an X509Registry is initialized with true certificates of trusted CAs. The X.509 framework does nothing to ensure this; it is the sole responsibility of the user of the framework to obtain CA certificates from a trusted source and ensure that they are not compromised in the process. The level of precautions necessary to protect this process depends on the level of security required by the application. Specific procedures are outside of the scope of this document, and would likely need to be tailored for specific circumstances, especially for applications with higher security requirements. However, we can provide a few suggestions that might be acceptable for applications where the threat level is relatively low.

One potential source of CA certificates are web browsers. They also need a CA certificate registry and usually have a fairly extensive set of those bundled with them. Some browsers provide a way to export the bundled certificates. In Microsoft Internet Explorer version 6, for example, open **Internet Options**, select the **Content** tab, and click the **Certificates ...** button. Select the **Trusted Root Certification Authorities** tab for a listing of CA certificates. Select any of them and click the **Export** button. The export wizard offers several export formats; select **Base-64 encoded X.509** format, then specify a file name, such as `test.cer`. You can use the following code to read the certificate in and to add it to an X509Registry:

```
| certificate registry |
registry := Security.X509.X509Registry new.
certificate := Security.X509.Certificate fromFile: 'test.cer'.
registry addCertificate: certificate.
```

There are also other applications that need access to CA certificates and might be bundling them as well. On most Linux distributions you can often find CA certificates in a file named **`ca-bundle.crt`**. The OpenSSL library on RedHat systems usually maintains this file in directory **`/usr/share/ssl/certs`**. You can directly import the contents of this file as follows:

```
| certificates registry |
registry := Security.X509.X509Registry new.
certificates := Security.X509.CertificateFileReader
    readFromFile: 'ca-bundle.crt'.
registry addCertificates: certificates.
```

It is wise to review the contents of the certificate bundle and only import those certificates that your application actually needs. Adding a CA certificate to your registry is deceivingly simple and does not convey the degree of trust actually involved in that action. Be sure to understand what it is you are trusting a CA to do and ensure that it matches the security requirements of your application.

In general, the contents of the registry is maintained using messages addCertificate:, certificateFor:, and removeCertificate:. SSLContext needs to be linked to an instance of registry when created:

```
registry := X509Registry new addCertificate: anX509Certificate.
ctx := SSLContext newWithAllCipherSuitesUsing: registry.
```

Note that the context creation methods without the registry parameter use 'X509Registry default' for the registry.

A server has to have both its own certificate chain and the corresponding private key registered with its SSLContext. Therefore, these are registered with the context as a pair, an array with first item being a certificate chain and second item being the corresponding private key. The message to use to set up the context depends on the certificate type. The simplest case is an RSA certificate for encryption. This kind of certificate is registered using rsaCertificatePair:

```
chainKeyPair := Array with: anArrayOfX509Certificates
    with: anRSAPrivateKey.
ctx rsaCertificatePair: chainKeyPair.
```

The server presents its certificate chain to the client during SSL handshake. The client then validates the chain using its certificate registry. If the validation fails the client signals with a SSLBadCertificate warning. Other certificate related warnings are SSLCertificateExpired, SSLCertificateRevoked, SSLCertificateUnknown, and SSLUnsupportedCertificate.

A simpler variant of client certificate setup can be used if the client can obtain the server certificate via some other reliable means, such as by using a HTTPS-enabled web browser to obtain the certificate of an HTTPS server. In this case the party that is validating a certificate received from the other party can maintain a list of certificates that it considers valid and match any incoming certificate with the list. If there is a bit-equivalent certificate found in the list, then the certificate is valid. The list of “valid” certificates is maintained by the SSLContext. The list of valid certificates can be maintained using messages addValidCertificate:, validCertificateFor:, and removeValidCertificate:.

```
certificate := X509Certificate fromString: X509RSAPrivateKey
    example1TestCertificate.
context addValidCertificate: certificate
```

With the contexts set up, the handshake has a good chance of succeeding.

Certificate Subject Validation

SSL can do most of the validation by itself, but there is one check that only the application (or the user) can perform reliably: certificate subject validation. The purpose of this check is to determine that the certificate belongs to the party that the application is actually trying to reach. Without this check an impostor can present his own perfectly valid certificate and impersonate the other party.

This check is done by analyzing the subject's distinguished name embedded in the certificate to see that it is the name of the other party. This is the task of the subject validation block that should be provided by the application when the handshake is initiated. This block is passed as a parameter to connectSubject:, which is a variant of connect. For a server wishing to authenticate its clients the accept variant is acceptSubject:. The validation block takes one argument and it will be evaluated with a dictionary representing subject's distinguished name. The keys in the dictionary are so called attribute names and values are attribute values. For example web browsers are relying on a convention that the subject name attribute called “commonName” or also just “CN” contains a DNS name of the server site that the certificate was issued for. The block has to return a Boolean, indicating validity of the subject name. If the block returns false, an SSLBadCertificate warning will be signaled. Here is an example subject validation block:

```
connection connectSubject:
    [ :dnd | (dnd at: 'CN') = 'www.thesecureserver.com' ]
```

This states the requirement that the certificate presented by the server during the handshake has to have the specified value for the commonName attribute of the subject distinguished name.

Client Authentication

Some servers may need to authenticate its clients. In this case the server should use acceptSubject: instead of accept. For the client to pass the authentication it has to have a proper certificate and a corresponding private key set in its SSLContext. The server will validate the client certificate with its certificate registry and the subject validation block provided with the acceptSubject: message.

Note that SSL has provisions for a client to refuse to authenticate. In this case it is up to the server to decide if it allows the handshake to proceed without the client authentication. Obviously this kind of decision has to be made at the application level. Therefore, when the client refuses to authenticate, the server connection will signal an SSLNoCertificate warning. The application should resume the warning if it wants to proceed, or return the warning if it wants to break the handshake off.

```
[ connection acceptSubject: [ :dnd |
    ((dnd at: 'O') = 'TrustMe Inc') & ('*Purchasing*' match: (dnd at: 'OU')) ]
] on: SSLNoCertificate do: [ :ex | ex resume ]
```

Certificates for Signing

Certificates can have different purposes. SSL distinguishes between certificates that can be used for signing and those that can be used for encryption or key exchange. (A certificate should never be used for both, even though it may be technically possible). Certificates discussed previously would have to be encryption/key exchange certificates, for example encryption RSA certificates. However, SSL can accommodate signing certificates as well. We support both RSA signing certificates and DSA signing certificates.

To register such certificate with SSLContext use rsaSigningCertificatePair: or dsaCertificatePair: message.

Clients always authenticate with a signing certificate. Just register the certificate pair with the client's SSLContext.

A signing certificate has to be registered with the context as a signing certificate. If it is a certificate for RSA signing, the message rsaSigningCertificatePair: should be used. In case of DSS signing it should be a dssCertificatePair: message. (DSS certificates are only used for signing).

Apart from the signing certficate which will be used for authentication, a server also needs a pair of public/private encryption keys to be able to exchange a “secret” value with the client during the handshake. If such encryption key pair is not associated with server's certificate, a pair of “temporary” keys will have to be used. The type of keys obviously depends on the type of selected cipher suite.

As with the certificate pair, the key pair has to be registered with the SSLContext. In case of RSA cipher suite, message rsaKeyPair: should be used. With this setup the certificate will be used to authenticate the server. The private key of the certificate will be used to sign the temporary public key to prove to the client that the server really owns the private key for the certificate. The temporary public key will be sent to the client which will use the key to encrypt the “secret” value for the server. The server then uses the temporary private key to decrypt the secret value. An example server setup for this situation would look something like this:

```
certificate := X509Certificate fromString: X509RSAPrivateKey
        example1TestCertificate.
privateKey := (X509RSAPrivateKey fromString: X509RSAPrivateKey
        example1) getKey.
context signingCertificatePair: (Array with: (Array with: certificate)
        with: privateKey).
context rsaKeyPair: RSAEncryptionKey example1024BitKeyPair reverse
```

Because the type and specific properties of the certificate are all captured in the certificate itself, it is possible to use a much simpler API. You can send a certificate:key: message to SSLContext, which does the right thing.

Diffie-Hellman Key Exchange

Because of the nature of Diffie-Hellman key exchange, there are several options how to use temporary keys for DHE cipher suites (Refer to “Diffie-Hellman key agreement” on page 5-8 for the details on the meaning of the various parameters mentioned below):

- Both keys are precomputed and registered with the context using dhKeyPair: message. In this case the first element of the pair is a 3-

element array containing the group parameters p and g, and the public key y, in that order. The second element of the pair is a standalone private value x.

- Just the group parameters are registered with the context and a fresh pair of the private/public values x/y are generated for each handshake. In this case the parameters are registered using dhParameters: message as an (Array with: p with: g).
- If nothing is registered with the context an SSLNoDHParameters exception is signaled. However if the exception is resumed, SSL will generate whole DH parameter suite on the fly and proceed the handshake with those. Note however that generation of the parameter p is time consuming and the delay caused by that may not be acceptable. In that case we recommend pregenerating the p and g parameters and dhParameters: as described above.

Anonymous Handshake

By now you're probably wondering if you always have to deal with this messy certificate stuff. The answer is usually yes, unless you're in a situation when you don't need to authenticate the other party because you are certain that you know who the other party is. In this situation you can do a so-called anonymous handshake. Keep in mind that this type of handshake is strongly discouraged though, because dropping authentication means dropping prevention of man-in-the-middle attacks.

A client-side handshake is anonymous by default. Therefore, an anonymous handshake, in SSL terms, is the case when the server does not authenticate, either. To setup a context for anonymous handshake there's no need to do any of the certificate registration mentioned above. To complete the handshake the server still needs a pair of public/private encryption keys for protection of the shared “secret.” The key pair has to be registered with the server's SSLContext for the handshake to complete successfully. In SSL, anonymity of the handshake is dictated by the selected cipher suite. SSL 3.0 defines only a few of those, they have DH_anon in the name and are all based on Diffie-Hellman key exchange. There is no anonymous cipher suite defined for the RSA key exchange, and the SSLConnection will signal an SSLNoServerCertificate warning if the server doesn't have a suitable certificate. However, this warning is again resumable, so if the handshake proceeds, the server will skip the authentication and send a temporary RSA key instead, allowing an anonymous handshake even though the cipher suite is not an anonymous. Getting the server to perform an anonymous RSA handshake would look like following.

```
context := SSLContext new initializeSuites:
    (Array with: SSLCipherSuite SSL_RSA_WITH_DES_CBC_SHA).
context rsaKeyPair: (Array with: anRSAPublicKey with: anRSAPrivateKey).
connection := context connectionFor: ioStream.
[ connection accept
] on: SSLNoServerCertificate do: [ :ex | ex resume ]
```

The client will refuse to handshake a non-anonymous cipher suite if the server doesn't authenticate. That is unless the connect message is used (instead of connectSubject:). Omitting the subject validation block is considered an indication that the client is not interested to authenticate the server and thus allowing an anonymous handshake regardless of the chosen cipher suite.

Session Renegotiation

Occasionally, when an SSL connection is maintained for an extended time period, it might be desirable to renegotiate a new set of security parameters. SSL does have provisions to do that without rebuilding the connection from scratch. Renegotiation has to be initiated from the server side by sending message renegotiate to the SSLConnection. This invokes a full SSL handshake. The client side SSLConnection handles renegotiation on its own; it is completely transparent for the client application.

Session Resumption

The last topic that we will cover here is session resumption. Negotiation of security parameters is expensive. Therefore it is often desirable to reuse the previously negotiated security parameters for additional connections to the same party. The results of the negotiation are captured in an instance of SSLSession. To get a new connection to reuse previously negotiated parameters it has to be built around the session that captures the parameters using method connectionFor:using: instead of the previously mentioned connectionFor:, for example:

```
session := previousConnection session.
newConnection := context connectionFor: aNewIOStream using: session.
newConnection connect.
```

There's no need to use connectSubject: for session resumption, because the session caches the previously specified subject validation block. Ultimately it is up to the server to decide if it allows to resume a session. If resumption is refused the handshake will automatically fallback to full negotiation. A session has to be resumable to resume successfully (i.e. without doing the full negotiation). Session resumability can be tested with isResumable message. A server context does not allow session resumption by default, therefore the sessions it creates will be non-resumable. To get the server context to create resumable sessions it has to receive allowResumableSessions message. Any session can be explicitly made non-resumable using message beNonResumable.

7

ASN.1

Overview

SSL certificates are defined by the X.509 (see "Handshake and Certificates" on page 6-4) specification using ASN.1. Thus, X.509 certificate support requires ASN.1 support. This chapter briefly describes ASN.1 and the design of the VisualWorks ASN.1 implementation. It concludes in a short walkthrough describing the steps involved in setting up and using an ASN.1 decoder, and a list of known limitations.

Readers requiring more information about ASN.1 should consult either the several ISO/IEC specifications that define ASN.1 or Olivier Dubuisson's *ASN.1: Communication Between Heterogeneous Systems*. The latter is very highly recommended.

This section presumes a modest awareness of the issues and distinctions involved in marshalers and type systems, a good knowledge of Smalltalk coding conventions, and a willingness to browse the code discussed.

ASN.1

ASN.1 is an abstract syntax notation. It is used to specify the abstract syntax of the data and message types used to transfer data between communicating applications. ASN.1 is used in many of the RFCs that define Internet protocols. For example, the types used in data transfers by SNMP, LDAP, and Kerberos, and the certificates exchanged in SSL, are defined in ASN.1.

Any abstract data transfer syntax defined in ASN.1 may have a different concrete syntax in each communicating application. For example, a type defined in ASN.1 may be concretely manifested in one application as a C structure and in another as a Smalltalk class. This is expected. The ultimate purpose of an abstract syntax notation is to support clear specification of a transfer syntax: of the representation of its abstract data and message types on the wire between the communicating applications. That encoding must be independent of the concrete syntax used by any communication endpoint. To establish that independence, ASN.1 is accompanied by a set of encoding rules that specify how any abstract type defined in ASN.1 is to be linearized as a collection of bits. There are several such sets of encoding rules. BER (Basic Encoding Rules), DER (Distinguished Encoding Rules), and XER (XML Encoding Rules) are among the best known and most frequently used.

Thus, whenever ASN.1 is used to specify a transfer syntax, two things must be specified:

- the particular abstract data and message types used in the transfer
- the encoding rule set used to linearize those types.

The X.509 specification follows this requirement. It both defines the type `Certificate` and the types of a certificate's several components, and specifies that DER is to be used to encode and decode certificates.

In summary, ASN.1 is a language for defining types and messages, that, in conjunction with one of the ASN.1 encoding rule sets, will possess a well-defined bit-level representation in communication.

The ASN.1 specification, unlike that of CORBA, does not include language-specific mapping specifications, which detail the mapping of ASN.1 abstract types to the concrete, native types of particular computer language. Instead, that mapping is defined through a language-specific ASN.1 implementation, implemented by a language vendor. That fact underlines the importance of the following documentation.

ASN.1 Fundamental Types

ASN.1 defines several abstract, fundamental types. These fundamental types provide the basis upon which the users of ASN.1 may define the derived subtypes useful for their needs. These fundamental types include:

- a comparatively standard set of basic types, like `BOOLEAN`, `INTEGER`, `NULL`, `REAL`, and `ENUMERATED`
- a large set of (sometimes outmoded) character types, like `NumericString`, `PrintableString`, `IA5String`, and `VideoTexString`

- two constructed types containing elements of the same type, SEQUENCE OF and SET OF, usually mapped to concrete, language-specific collection types
- two constructed types containing elements of different types, SEQUENCE and SET, usually mapped to classes in languages like Smalltalk and to structures in languages like C
- additional notions common to most type systems, like CHOICE, ENUMERATED, and ANY
- other miscellaneous types, some specific to ASN.1, particularly OBJECT IDENTIFIER, but including types like UTCTime and GeneralizedTime

ASN.1 Modules

Like CORBA, ASN.1 also defines the notion of a MODULE, an object that contains a set of related type definitions. All ASN1. type declarations must occur within the scope of a module, and a module is the basic unit of ASN.1 compilation. ASN.1 modules may both import and export module elements.

ASN.1's Constructed Types

It is important to note that an ASN.1 SET or SET OF has no relation to the set of mathematics. The ASN.1 specification is strongly targeted toward the problems of data transfer. A SEQUENCE or a SEQUENCE OF guarantees the transmission order of its elements, while a SET or a SET OF does not. That is the only significant difference between a SEQUENCE and a SET, or a SEQUENCE OF and a SET OF. Thus, unlike a mathematical set, an ASN.1 SET or SET OF may have several elements of the same value and type. But, you cannot know which one of them you will get first when a SET or SET OF is transmitted. Recognizing the problematic nature of this notion, the ASN.1 community does not now recommend the use of SET or SET OF types in any circumstance where a SEQUENCE or SEQUENCE OF type may reasonably serve instead.

ASN.1's OID

An ASN.1 OBJECT IDENTIFIER is a node in a single public registration tree. This tree is used solely for the registration of persistent objects that require an identifier which is universally unambiguous and available world-wide. The graph is managed by the OSI. It may include, among its registered objects, any well defined piece of information, definition, or specification that requires a name to identify it during communication. For example, it includes attributes of distinguished names, and the objects managed under SNMP.

In a registration tree each node can be unambiguously identified by the path, from the tree's root, that is taken to reach it. Each node in the registration tree has an arbitrary number of ordered and numbered subnodes, as well as a unique, scoped name. Consequently, each node can be unambiguously identified either by a series of names, a series of integers, or both. For example, the following five path specifications refer to the same node in the OSI registration tree:

```
2.1.2.0
joint-iso-itu-t.asn1.ber-derived.canonical-encoding
joint-iso-itu-t.1.ber-derived.canonical-encoding
2.asn1.2.canonical-encoding
2.asn1.2.0
```

ASN.1 OIDs make up the bulk of the data traffic in applications like SNMP, where nearly all of the objects in data exchanges are registered objects. Hence, any implementation of ASN.1 must optimize the encoding and decoding of ASN.1 OIDs, and implement a mechanism for translating between the implementation's optimized representation and the various, alpha-numeric representations shown above.

Type Definition in ASN.1

ASN.1 implements several constructs for type definition and this section can do no more than address the basics. Consult Dubuisson's *ASN.1: Communication between Heterogeneous Systems* for more details.

A new ASN.1 type, called a subtype, must be defined within an ASN.1 module. Every module in ASN.1 must comply with the following minimal form:

```
ModuleName DEFINITIONS ::=
BEGIN
    assignments
END
```

Within the scope of the module, a new type is defined by assigning a type specification to a type name. The following is simple subtype definition:

```
Age := INTEGER
```

The new type named Age is defined as being a subtype of INTEGER.

Similarly, a constructed type is defined by naming and specifying the type of each constituent element, as in the following:

```
Person ::= SEQUENCE {
     first   IA5String
     lastl   IA5String
     age     AGE
}
```

Here, a new type named Person is defined as a SEQUENCE of three elements, named first, last, and age. The first two elements are IA5Strings. The last element is of type Age, defined previously as a subtype of INTEGER.

ASN.1 Constraints

To further specify types, ASN.1 also supports constructs for expressing type constraints and type constraint combinations. The latter include unions and intersections of constraints. A few illustrative examples will show how type ASN.1 constraints are expressed:

```
Age ::= INTEGER (0..120)
ExactIty32Bits ::= BIT STRING (SIZE (32))
NoSs ::= IA5String (FROM (ALL EXCEPT (“s” | “S”)))
```

In the first example, Age is constrained to the range of positive integers between 0 and 120 inclusive. In the second, ExactIty32Bits is constrained to be a bit string of size 32. In the last, NoSs is constrained to be an IA5String including all elements of the IA5 character set excluding ‘s’ and ‘S’.

This is only a sample of the several constraint constructs supported in ASN.1.

ASN.1 Type Tags

ASN.1 associates what it calls a UNIVERSAL type tag—a default, numeric type identifier—with each of its fundamental types. For example, the UNIVERSAL tag for a BOOLEAN is 1, and the UNIVERSAL tag for an INTEGER is 2. Any subtype of an ASN.1 type inherits the UNIVERSAL tag of its parent type.

UNIVERSAL tags are one of the several tag classes supported by ASN.1. Since the use of some tag classes has been discouraged by the ASN.1 community since 1994, we will not dwell on the semantics and import of all the various tag classes here. But one class of tags, the class of context-specific tags, is important, and requires explanation.

Type tags are critical in the encoding and decoding of ASN.1 data traffic. In an ASN.1 encoded byte stream, a numeric type tag is usually encoded prior to the value. Thus a decoder, passing over an encoded byte stream, is informed of the type of each value coming over the wire before it encounters the value itself. This significantly aids decoding and eliminates the need for backtracking.

However, in order to disambiguate the decoding of some types, ASN.1 also supports the use of context-specific tags within SEQUENCEs, SETS, CHOICEs, and other constructed types.

Consider the following SET, remembering that an ASN.1 SET does not guarantee the order of transmission of its elements:

```
Id ::= SET {
    ssn           NumericString
    employeeId    IA5String
    }
```

In this case, whether an encoder writes out the bytes for the ssn or the employeeId first, the receiving decoder will not be confused about which element it is decoding, because each value will be preceded by a distinguishing UNIVERSAL type tag (18 for the NumericString, 22 for the IA5String). This happy state of affairs breaks down if a SET contains multiple elements of the same type, or a SEQUENCE contains multiple elements of the same type, some of which are optional. To take the simplest case, consider the following invalid SET definition:

```
Id ::= SET {
    ssn           NumericString
    employeeId    NumericString
    }
```

In this case, a decoder, because SETs do not guarantee order of transmission, could not know, when it encounters the type tag for a NumericString, whether the following value was the ssn or the employeeId element of the set. This sort of ambiguity is not allowed in ASN.1, and is averted by overriding the use of UNIVERSAL tags within the scope of the SET declaration. The following example shows how the previous example can be corrected using a context-specific tag:

```
Id ::= SET {
    ssn           [0]    NumericString
    employeeId           NumericString
    }
```

The bracketed 0 in the code above instructs any ASN.1 encoder to replace the UNIVERSAL tag for ssn, an 18, indicating a NumericString, with a 0 followed by an 18. As a result, any ASN.1 decoder, with knowledge of how this SET subtype was defined, can unambiguously tell which element it is decoding, because the ssn value will be preceded by a 0 and a 18 while the employeeID value will be preceded by only an 18. However, if the marshaler does not have such type information, it will be unable to properly decode an ssn. The tag 0 is not associated with an ASN.1 fundamental type, and the decoder will fail.

If the context-specific tagging in the above is deemed too costly or verbose, the declaration can replace the EXPLICIT context-specific tag in the previous case with an IMPLICIT context-specific tag, as in the following:

```
Id ::= SET {
    ssn            [0]   IMPLICIT NumericString
    employeeId           NumericString
    }
```

In this case the UNIVERSAL tag 18 will be replaced, while encoding the ssn element, with only a 0 rather than with a 0 followed by 18 as it is when an EXPLICIT context-specific tag is used.

Considerations such as these, very particular to the lower levels of encoding and decoding, should arguably not be reflected at the level of an abstract syntax. But this is how ASN.1 works, and it is important to know that.

There are many other subtleties in ASN.1's tagging and type definition system, to which Dubuisson's book is the best available introduction.

ASN.1 Encoding Rules

ASN.1 supports over a half-dozen sets of encoding rules. Each set specifies the manner in which each of the types defined in the ASN.1 specification is to be represented in bits. Every user-defined subtype inherits the encoding rules that apply to its parent type.

The various sets of encoding rules supported by ASN.1 differ by virtue of the requirements they were designed to meet. BER (Basic Encoding Rules) was the first conceived, and is the most commonly used. It is far from compact. It does not guarantee a unique encoding for all values. But, it is relatively easy to decode. CER (Canonical Encoding Rules) and DER (Distinguished Encoding Rules) are both derived from BER, but guarantee that each value of each type will have one and only one proper encoding. Thus, a BER marshaler can always decode a CER or DER encoded byte stream, but not vice versa. PER (Packed Encoding Rules) strives for compactness above all else, and is extremely difficult to write marshalers for. In PER, the value of an INTEGER type that may have only two values, say, 213457634 and 213457635 will be encoded by exactly one bit, indicating whether it is the first or the second of the two values that is being transmitted.

The most commonly used ASN.1 encoding rules, BER and DER, are *triplet encodings*. Each value is transmitted as a triplet consisting of a type tag (T), a length (L), and a value (V). These are more specifically called *TLV encodings*.

Those interested in a more complete description of the various ASN.1 encoding rules should consult Dubuisson.

Packaging

The VisualWorks ASN.1 implementation is delivered in the **`net\`** subdirectory of the VisualWorks installation. The implementation consists of four parcels that should be loaded in the order shown below:

ASN1-Support
: This parcel defines several ideas basic to ASN.1, like the notions of a Module and an ObjectIdentifier. It also defines SMINode, a class used to implement trees of ASN.1 ObjectIdentifiers, and Encoding, the class used to optionally store the original encoding of an object.

ASN1-Constraints
: This parcel contains a VisualWorks implementation of the ASN.1 constraint system.

ASN1-Types
: This parcel contains definitions of the ASN.1 fundamental types. It also implements part of the machinery for retaining the original encoding of an object.

ASN1
: This parcel implements the read-write streams that encode or decode according to the two sets of encoding rules supported by the implementation, BER and DER. Extensions to the several ASN.1 fundamental types, the methods invoked in support of encoding and decoding, are also implemented here.

Design of the ASN.1 Implementation

The intent of this section is to expose the organization of the present VisualWorks ASN.1 implementation. After reading this section, you should be aware of the function of the major class hierarchies present in the implementation, and of the principles used to design the implementation's architecture. All of the critical class hierarchies in the ASN.1 implementation are well documented, but this section should provide you with an overview that integrates the comments in the code.

Outline

The current VisualWorks ASN.1 implementation is the successor of several previous ones. It improves upon them by more cleanly isolating the required components of a marshaler based on a foreign, abstract type system, like ASN.1. In the following we will describe each of the required components and point to the Smalltalk classes that implement it.

The list of the required components in an ASN.1 implementation falls out naturally from what is involved in marshaling using a foreign type system with multiple sets of encoding rules. When encoding—translating a Smalltalk object into bytes—the marshaler must discover or infer what ASN.1 type the Smalltalk object is mapped to, and then employ knowledge, both of that type and of the encoding rules in effect for that type, to produce the correct bytes. When decoding—translating bytes into Smalltalk objects—the marshaler must discover or infer the encoded ASN.1 type, and then use knowledge, both of that type and the decoding rules in effect for that type, to produce the correct Smalltalk object. So, the components needed in an ASN.1 implementation are:

- a full representation of the foreign, ASN.1 type system
- a bi-directional mapping between Smalltalk classes and ASN.1 types
- a representation of the several sets of encoding and decoding rules used to marshal ASN.1 types

Each of these involve other, ancillary components as described below

The ASN.1 Type System

The Types

A marshaler based on a foreign, abstract type system requires a representation of that type system in order to use, operate upon, and reason about it. This representation captures the high-level characteristics of each abstract type. It should be largely independent of what native Smalltalk class any ASN.1 abstract type is mapped to, and of the encoding rules used to marshal the abstract type. This representation captures, for example, the fact that an ASN.1 SEQUENCE possesses elements and may have extensions.

The abstract type system is implemented in the `ASN1.Entity` hierarchy. Though methods used in encoding and decoding are implemented in the abstract type hierarchy, they all dispatch directly to a marshaler and are effectively contentless.

The Type Extension Machinery

An abstract type system is useful only if it can be extended to create new subtypes. The subtype creation API should be one that an ASN.1 compiler could easily make use of. (An ASN.1 compiler reads an ASN.1 text file and produces Smalltalk objects that represent the types described.)

The type extension API is implemented in class ASN1.Module in the latter's definitions protocol. It thereby enforces fidelity to the idea that ASN.1 subtypes should be created only within modules. The API requires that a Module be defined before a subtype is created, and it uses methods named after the involved parent type in order to register newly defined subtypes, as in the following:

```
| module |
module := ASN1.Module new: #Temporary.
module INTEGER: #Age.
```

This code creates a subtype of INTEGER named Age in the module named Temporary. A supplementary API that does the same is shown below:

```
| module |
module := ASN1.Module new: #Temporary.
INTEGER named: #Age in: module.
```

All ASN.1 type classes understand the message named:in:.

Module reimplements doesNotUnderstand: in order to allow the lookup of already defined subtypes by name, as in the following:

```
| module |
module := ASN1.Module new: #Temporary.
INTEGER named: #Age in: module.
module Age: #AnotherAge
```

This creates a subtype of Age named AnotherAge. The subtype creation machinery is also implemented so that reference may be made to an as yet undefined subtype in the specification of a new subtype. This allows a user to ignore the order of the definition of types when defining several mutually implicated types. Support for references to as yet undefined types is implemented in ASN1.TypeReference.

All subtypes are created as instances of their parent type class.

Other methods used in type definition are implemented on the instance side in the ASN1.Entity hierarchy or in the ASN1.AbstractElement hierarchy. For example, the methods used to record or access the tagging mode—UNIVERSAL, EXPLICIT, or IMPLICIT—of the elements of an ASN.1 constructed type are implemented in ASN1.ChoiceElement.

The Constraint Specification Machinery

Since ASN.1 subtype definition usually involves the specification of constraints, the ASN.1 constraint system must also be represented in Smalltalk. Constraints, and constraint combinations, are implemented in the ASN1.Constraint hierarchy. ASN1.Type has a constraint instance variable, and any type instance can be asked whether an object satisfies its constraints using permits: anObject. The following example illustrates the specification of a simple range constraint:

```
| module |
module := ASN1.Module new: #Temporary.
( module INTEGER: #Age) constraint: (Constraint from: 0 to: 120).
```

Age now permits only integral values between 0 and 120 inclusive.

The APIs of the type definition and the constraint specification machinery are intended for use by an ASN.1 compiler, but can be and are used to define new subtypes directly.

The Mapping of Smalltalk Classes to the ASN.1 Type System

Marshaling in the presence of an abstract type system presupposes a mapping between

- the abstract types that regulate marshaling and
- the concrete types that are the input for encoding and the output of decoding.

In defining this mapping, we will inevitably discuss some topics that encroach upon the design and API of the marshalers, also discussed in the following major section.

One-To-One Base Type Mappings

Most ASN.1 base types are mapped, one-to-one, to the closest available Smalltalk native type. For example, ASN.1 type BOOLEAN is mapped to Boolean, and ASN.1 type INTEGER is mapped to Integer. For encoding, these mapping are accomplished by implementing the two methods encodeASN1With: aMarshaler and tagBER in the involved Smalltalk classes. Please examine the implementors of these methods. For decoding the mapping is effected in the low-level decoding rules used by marshalers. Examine, for example, the implementors of decodeBOOLEAN: and decodeINTEGER:.

Many-To-One Base Type Mappings, Encoding Policies, and Type Wrappers

One-to-one mappings are not always possible, even for simple types. Both ASN1.GeneralizedTime and ASN1.UTCTime are properly mapped, by default, to Timestamp. Many of the several ASN.1 string types are properly mapped, by default, to ByteArray. These are all many-to-one mappings, and many-to-one mappings create a problem. When an ASN.1 type involved in a many-to-one Smalltalk mapping is decoded into a native Smalltalk type, the associated ASN.1 type information is lost in the translation. Smalltalk cannot reliably re-encode the decoded object in the way that it was originally encoded. In most applications this is not a worry, but whenever it becomes one, it would be useful, when decoding, to retain the original ASN.1 encoding, or knowledge of the original ASN.1 type, or both. This is accomplished using the VisualWorks framework for encoding policies.

All marshalers have a default encoding policy. Encoding policies are implemented in the EncodingPolicy hierarchy and a marshaler's is set using the method encodingPolicy:. The two most important are RetrainEncodings and RetainAllEncodings. But, there are three in all:

- The first, RetainEncodings, the default encoding policy, will retain original encodings and type information, if and only if the ASN.1 type being decoded has its retainEncoding instance variable set to true. That value is set using the method retainEncoding: aBoolean, implemented in class ASN1.Type. By default, the value of retainEncoding is set to false.
- The second important policy, RetainAllEncodings, will always retain original encodings and type information in the marshaler's output, irrespective of the value of the retainEncoding instance variable in any ASN.1 subtype.
- The third policy, PrettyPrint, prints decoded entities onto a ByteString, and is useful in debugging decoding problems.

If RetainEncodings or RetainAllEncodings are used, the original encoding and ASN.1 abstract type of a base type are retained by wrapping the decoded value in an instance of ASN1.TypeWrapper. That class declares instance variables for both type and encoding. It stores encodings in an instance of class ASN1.Encoding.

All values within type wrappers are guaranteed to be re-encoded in the same manner in which they were originally encoded, within the degree of play allowed by the decoding rules in effect. This means that some difference might be observed under BER (because it does not strictly enforce a unique encoding for each ASN.1 value), but no difference will be seen under DER (which does enforce unique encoding).

Constructed Type Mappings, Structs, and User-Defined Mappings

In the absence of ASN.1 type information, marshalers map, the ASN.1 SEQUENCE, SEQUENCE OF, SET, and SET OF types to class OrderedCollection.

However, if ASN.1 type information is available, two of these types, SEQUENCE and SET—the only two ASN.1 types that it makes sense to map to something like a Smalltalk application class—are, by default, mapped to instances of class ASN1.Struct.

Class Struct is a variant of class Dictionary. It implements the machinery needed to allow its instances to act like the instances of an ordinary Smalltalk class. If a Struct had #ssn as one of its keys, it will appropriately respond to the accessors ssn: and ssn. A Struct also records the order in which its associations were added. The later is critical for re-encoding those Structs that were decoded from ASN.1 SEQUENCE subtypes, because a SEQUENCE must guarantee the order in which its elements are encoded or transmitted. If encodings are being retained, the encodings are stored in the Struct's encoding instance variable. The original ASN.1 type is stored in the Struct's name instance variable.

However, ASN.1 SEQUENCEs and SETs may also be mapped to user-created Smalltalk classes by setting the mapping instance variable of a SEQUENCE or SET subtype using mapping: aClass. User-defined Smalltalk classes mapped to ASN.1 SEQUENCE or SET types must declare instance variables for all of the ASN.1 types elements, with the same name as the element name, and with the usual accessors. The user-defined class must also respond to new. If the user-defined class will be expected to retain encodings, its declaration must also include an encoding instance variable.

Imported Type Mappings

Nearly any foreign type system will define types that are not paralleled in Smalltalk. For example, though an ASN.1 BOOLEAN should obviously be decoded as a Smalltalk Boolean, there is no obvious mapping for an ASN.1 OBJECT IDENTIFIER, an ASN.1 ENUMERATOR, or an ASN.1 BIT STRING, which has semantics involving the notion of unused bits, a notion not found in any native Smalltalk string class. In such cases, a new concrete class must be created in Smalltalk to represent these types. They are, for the cases mentioned, ASN1.ObjectIdentifier, ASN1.Enumeration, and ASN1.BitString respectively. They are all subclasses of either ASN1.Imported or ASN1.TypeWrapper.

Note, that such imported concrete types are distinct, and should be kept distinct, from the representation of their corresponding abstract types that they are mapped to. For example, class ASN1.OBJECT_IDENTIFIER represents an ASN.1 abstract type, while class ASN1.ObjectIdentifier represents that abstract type's manifestation as a Smalltalk-specific concrete type.

Note that ASN1.ObjectIdentifiers, unlike all other objects in the implementation, are designed to always retain or cache their encodings. This is an optimization required by ASN.1's use in SNMP, where ASN.1 OBJECT IDENTIFIERs usually make up more than 40% of the data traffic and frequently recur. Any failure to cache the encoding of these OIDs would be alarming in its cost.

SMINode

Instances of class ASN1.SMINode represent the nodes of the OSI object registration tree. The class also supports the task of converting the default representation of ASN1.ObjectIdentifiers as ByteArrays into the various alpha-numeric representations that an object identifier may go by. Class SMINode declares currently unused instance variables that will not become significant until the current ASN.1 implementation is integrated with the release's preview SNMP implementation.

The Encoding Rules

Marshalers perform the brute work of turning byte streams into Smalltalk objects and vice versa. They are represented in Smalltalk as subtypes of Stream and they implement the sets of low-level encoding rules that complement the ASN.1 specification.

The VisualWorks implementation of ASN.1 now supports only the most two frequently used sets of encoding rules, BER and DER. These are both triplet encodings in which each value is encoded as a triplet consisting of a type tag, a length in bytes, and a value. Thus, all of the supported ASN.1 marshaling streams are subclasses of TLVStream. The hierarchy is shown below:

```
TLVStream
    BERStreamBasic
        BERStreamDefinite
            DERStream
```

TLVStream is a generic superclass for ASN.1 triplet marshalers. Its subclasses accommodate the most common ways in which tags and lengths may be encoded. BERStreamBasic is intended to be the superclass for SNMP marshaling streams. It assumes that lengths will always be encoded in three bytes and that tags will always be encoded in one byte.

(Both of these conventions are standard practice in SNMP.) BERStreamDefinite assumes that lengths are encoded in the fewest possible number of bytes and that tags may be encoded in more than one byte. (The latter liberty may be required in marshaling subtype systems more complex than those used in SNMP.) BERStreamDefinite is thus a perfect superclass for DERStream. DERStream overrides, when necessary, those low-level marshaling rules, implemented in its superclasses, that allow a non-unique encoding for a given ASN.1 type and value pair. In general, the DER encoding rules eliminate the several encoding options permitted under BER, for example, the option of encoding the length 2 in one, two, or three bytes. Under DER the length 2 is always encoded in a single byte

All of the ASN.1 marshaling streams support a uniform set of entry points for encoding and decoding. The API consists of the following four methods:

unmarshalObjectType: anAsn1Type
: Decodes the current contents of the marshaling stream into an object, using the type information provided in anAsn1Type.

unmarshalObject
: Decodes the current contents of the marshaling stream into an object, in the absence of type information.

marshalObject: anObject withType: anAsn1Type
: Encodes an object onto the marshaling stream, using the type information provided in anAsn1Type.

marshalObject: anObject
: Encodes an object onto the marshaling stream, in the absence of type information.

There are two decoding methods and two encoding methods. In each case, one of the methods has knowledge of the ASN.1 type to be encoded or decoded, and the other does not. In other words, one is type-agnostic and the other is type-aware. Both methods are supported because, with an ASN.1 triplet encoding, it is often possible, to correctly marshal without detailed type information. This is a consequence of two facts.

- First, type-agnostic encoding is often possible because several Smalltalk classes are supplied with a default mapping to a basic ASN.1 type through the method encodeASN1With:. Thus, if you attempt to marshal an instance of one of these mapped classes, using marshalObject:, all will go well. However, marshaling will fail with an instance of an unmapped class, that is, with one that does not implement encodeASN1With:. All the Smalltalk classes that map naturally to an ASN.1 fundamental type implement encodeASN1With:.

- Second, type-agnostic decoding is possible because all ASN.1 fundamental types have a default, UNIVERSAL tag, and a default encoding under any set of encoding rules. Because all user-defined ASN.1 subtypes are derived from the basic types through the application of constraints, and because such derivation does not always entail a tag change, it is often possible to decode correctly on the basis of the tag alone, without any additional knowledge about the encoded type. However, this strategy breaks down whenever the user-defined type involves use of the context-specific tags discussed above. Such tags are commonly used in the definition of ASN.1 constructed types, like SEQUENCE or SET. Constructed types may have optional elements of the same simple type, and, if so, elements must be uniquely tagged within the scope of the SEQUENCE or SET to remove ambiguity.

Because type-agnostic decoding often works, it is worth supporting. It is often useful in the initial examination a DER or BER encoded data, allowing a developer to get an idea of what the data looks like before going to the trouble of defining ASN.1 type information, that would optimize decoding. If type-agnostic encoding fails, the involved ASN.1 subtypes must be defined in Smalltalk, to allow decoding at all.

Using the ASN.1 Implementation

At the moment, ASN.1 is primarily used in VisualWorks to decode incoming bytes, and the best example of its use is found in the code of the X509 parcel. The following section is a brief walk through, describing the initial steps a knowledgeable developer might take in working up an X.509 certificate decoder. Please load and examine the shipped X.509 code while reading this section.

Getting the Encoded Bytes

The first step in devising an ASN.1 decoder is writing the API that will deliver up the encoded data. A DER-encoded X.509 certificate as a good example of such data. Certificates are readily accessible on most systems. Please refer to your system’s documentation for instruction on how to obtain certificates if you run on an OS other than Windows XP. Most systems also have a certificate inspector that parses and displays a certificate, and you can use that tool to check the VisualWorks decoding.

To get a raw certificate on a Windows XP system, open the Windows Internet Explorer and select **Tools → Internet Options...** to open the Internet Options window. Then select the **Content** tab. Then left-click on the **Certificates...** button near the center of the window. On the resulting Certificates window, select **Trusted Root Certification Authorities**. Then select any one of the authorities listed and left-click on **Export...**. Follow through the various screens of the Certificate Export Wizard, and export the certificate to a file, ensuring that you select the format named 'DER encoded binary X.509'. Other operating systems have similar facilities.

Once you have a certificate file, and have created the corresponding Smalltalk Filename, the following code will answer the relevant ByteArray:

```
aFilename contentsOfEntireBinaryFile asByteArray
```

In a real application, you may have to write something more complex to deliver up your data, and there are several examples of such code in the X.509 parcel. Look at class Security.X509.Certificate's class-side instance creation protocol.

When you print the ByteArray answered by the code shown above to a Transcript, you will see an array like the following:

```
#[48 130 3 123 48 130 2 99 160 3 2 1 2 2 16 196 187 216 192 202 255 86
165 17 211 86 150 97 153 34 48 48 13 6 9 42 134 72 134 247 13 1 1 4 5 0
48 29 49 27...etc...]
```

As mentioned above, ASN.1 BER and DER encodings are both triplet encodings. Such encodings result in a very typical patterning of nested and serial triplets, and, once you become familiar with ASN.1 (learning, for example, that 48 is one of the type tags for an ASN.1 SEQUENCE), you will be able to perform a high-level parse on an ASN.1 ByteArray by eye. For example, the first few bytes of the preceding array may be resolved into the following triplets:

```
T: 48
L: 130 3 123 (a three-byte length)
V:
    T: 48
    L: 130 2 99 (a three-byte length)
    V:
        T: 160
        L: 3 (a one-byte length)
        V: 2 1 2
        T: 2
        L: 16 a one-byte length)
        V: 196 187 216 192 202 255 86 165 17 211 86 150 97 153 34 48
        T: 48
        L: 13 (a one-byte length)
        V: 6 9 42 134 72 134 247 13 1 1 4 5 0
        T: 48
        L: 29 (a one-byte length)
        V: 49 27...etc...
```

But, however impressive a parse by eye might be—and this might not be the right one—the aim is to get the machine to do it for you, in a way that is both more definitive and far more illuminating

Type-Agnostic Marshaling

To decode this ByteArray, you need to use a marshaling stream. Because type-agnostic decoding often works, it is worth trying it in this case, even though it is clear that this stream includes a number of SEQUENCE's, any of which may have used the EXPLICIT or IMPLICIT tagging modes, confounding the default association between type tags and value. If you wished to attempt a type-agnostic decoding of a sample certificate, you would try code like the following:

```
(ASN1.DERStream with: aFilename contentsOfEntireBinaryFile asByteArray)
    reset;
    unmarshalObject.
```

In this code, a DER marshaling stream is created with our DER-encoded ByteArray. Then the stream is reset and sent unmarshalObject, rather than unmarshalObjectType: anAsn1Type.

Sadly, this approach will not work in the case of certificates, because certificates do include SEQUENCEs that reassign type tags. If the code above is executed, a TagUnknown error will raised. This is a a nearly certain indication that the marshaling stream requires full type information in order to decode a certificate correctly.

Defining ASN.1 Types

To define the ASN.1 types needed to decode a certificate, you must refer to the X.509 specification, and in it find the ASN.1 code that defines a certificate and its components. Then you must write the corresponding Smalltalk code to define those types. This step would be unnecessary if the VisualWorks implementation included an ASN.1 compiler, but it does not, as yet.

There are two basic steps in defining the types:

- creating a module to hold type definitions
- defining and registering new types in the module

Module Creation

A module serves to organize related type definitions into a single, named entity.

A module for the X.509 certificate types may be created thus:

```
ASN1.Module new: ‘X509’.
```

In the X509 code, examine the shared variable ASN1Module in class X509Object, and the methods that support its initialization.

Type Definitions

Adding ASN.1 type definition is a matter of translating the ASN.1 text in the relevant specification into the Smalltalk used to define ASN.1 abstract types. Several examples of type definition are provided in the X509 code by the many implementors of the method initializeAsn1Types. We will discuss only two examples. All of these methods contain non-executable comments with the text of the source ASN.1 type definition in the X.509 specification. Thus, this set of methods provides you with a good range of examples for producing Smalltalk ASN.1 type definition code from ASN.1 source text. The first example, implemented in class Security.X509.Certificate, reads as follows:

```
initializeAsn1Types
"
    Certificate  ::=  SEQUENCE  {
      tbsCertificate       TBSCertificate,
      signatureAlgorithm   AlgorithmIdentifier,
      signatureValue       BIT STRING  }
"
    (ASN1Module SEQUENCE: self asn1TypeName)
      addElement: #tbsCertificate type: #TBSCertificate;
      addElement: #signatureAlgorithm type: #AlgorithmIdentifier;
      addElement: #signatureValue type: #BIT_STRING;
      mapping: self;
      retainEncoding: true
```

The ASN.1 text defines a Certificate as a SEQUENCE consisting of three elements. The corresponding Smalltalk code does the same, and also specifies that a Certificate is to retain its original encoding and map itself to the Smalltalk class in which this method is implemented, Security.X509.Certificate. As is required to support both this mapping and the retention of encodings, the definition of class Security.X509.Certificate declares the instance variables tbsCertificate, signatureAlgorithm, signatureValue, and encoding, with the usual accessing methods. If the line

```
mapping:self;
```

was removed from the method above, an ASN.1 Certificate would instead be decoded as a Struct, because it is a SEQUENCE, rather than a SEQUENCE OF, a SET OF, or an ASN.1 base type or imported type.

The second example is excerpted from the implementation of initializeAsn1Types in class Security.X509.TBSCertificate.

```
"
    TBSCertificate  ::=  SEQUENCE  {
        version [0]  EXPLICIT Version DEFAULT v1,
        serialNumber CertificateSerialNumber,
        signature AlgorithmIdentifier,
        issuer Name,
        validity Validity,
        subject Name,
        subjectPublicKeyInfo SubjectPublicKeyInfo,
        issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                    -- If present, version MUST be v2 or v3
        subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
                    -- If present, version MUST be v2 or v3
        extensions      [3]  EXPLICIT Extensions OPTIONAL
                    -- If present, version MUST be v3       }
"
    ( ASN1Module SEQUENCE: self asn1TypeName )
        addElement: #version type: #Version tag: 0 tagging: #explicit
            default: 0;
        addElement: #serialNumber type: #CertificateSerialNumber;
        addElement: #signature type: #AlgorithmIdentifier;
        addElement: #issuer type: #Name;
        addElement: #validity type: #Validity;
        addElement: #subject type: #Name;
        addElement: #subjectPublicKeyInfo type: #SubjectPublicKeyInfo;
        addOptionalElement: #issuerUniqueID type: #UniqueIdentifier tag: 1
  tagging: #implicit;
        addOptionalElement: #subjectUniqueID type: #UniqueIdentifier tag: 2
  tagging: #implicit;
        addOptionalElement: #extensions type: #Extensions tag: 3 tagging:
  #explicit;
        mapping: self;
        retainEncoding: true.
```

This is a more complex case, but is still straightforward. Note the protocol used to add optional elements, to set default values, to identify the context-specific tags, and to mark them as either IMPLICIT or EXPLICIT tags. The only tricky bit, in this case, is that knowledge of another part of the specification was used to set the default value of version to 0, the value used to identify ‘v1’.

Type-In Hand Marshaling

Once you have defined the ASN.1 subtypes relevant to the ByteString you wish to decode—irrespective of whether you have mapped ASN.1 SEQUENCEs or SETs to Smalltalk, user-defined classes by using mapping: aClass—you can attempt type-in-hand rather than type-agnostic decoding. Code like the following will do for Certificate:

```
(ASN1.DERStream with: aFilename contentsOfEntireBinaryFile asByteArray)
    reset;
    unmarshalObjectType: (aModule find: #Certificate)
```

Code like this will be found in the class-side instance creation methods of Security.X509.Certificate.

Debugging Tips and Error Types

In general, stepping through the code from a marshaler's entry points—unmarshalObjectType:, unmarshalObject, marshalObject:withType:, or marshalObject: —is your best approach to solving a decoding or encoding problem.

The various subclasses of ASN1.Asn1Error have been designed to identify the most usual ways in which decoding, encoding, or ASN.1 type definition may go astray. It may pay to cruise through the ASN1.Asn1Error class hierarchy, browsing all the methods in which those classes are referenced, to get a grip on what those ways are.

Known Limitations

The ASN.1 implementation has several known limitations, though none of them are known to significantly restrict the implementation's use in support of either X.509 or SNMP. The limitations are:

- Modules do not yet support importing and exporting or global tagging.
- Constraint extensibility, and some ASN.1 constraint types are not supported. These unsupported constraint types are:
    - regular expression constraints
    - the WITH COMPONENTS constrait for constructed types,
    - the OCTET STRING constraints ENCODED BY and CONTAINING ENCODED BY
    - user defined CONSTRAINED BY constraints

- Some ASN.1 fundamental types and type definition constructs are not supported, for example, `EXTERNAL`, `EMBEDDED PDV`, `RELATIVE OID`, and `REAL`.
- Exception markers are not supported.
- Tags encoded in multiple bytes may cause decoding problems.
- Some valid, but infrequently used, BER encodings, like indefinite length, are not supported.

Index

Symbols

<Operate> button ix
<Select> button ix
<Window> button ix

A

AES 4-4
ARC4 4-11
ASN.1 7-1
asymmetric-key, see publick-key cryptography

B

BER 7-2, 7-7, 7-13, 7-14
block cipher 4-1
block ciphers 4-4
block padding 4-7
BlockCipher class 4-4
BlockCipher instance creation 4-5
blockSize method 4-9
Blowfish 4-5
BP 4-7
buttons
    mouse ix

C

CBC 4-5
CER 7-7
certificate authority 6-5
certificates
    handshake and 6-4
    signing 6-8
CFB 4-6
cipher block chaining 4-5
cipher feedback 4-6
cipher modes 4-5
ciphertext stealing 4-7
constructed types 7-3
context-specific tags 7-16
conventions
    typographic viii
CORBA 7-2, 7-3
CS 4-7

D

decrypt: method 4-4
decryptInPlace: method 4-4
decryptInPlace:from:to: method 4-4, 4-9
decryption
    block cipher 4-7
    RSA 5-3
    stream cipher 4-11
    symmetric-key 4-3
DER 7-2, 7-7, 7-13, 7-14
derEncodedDigestInfo method 5-6
DES 4-5
Diffie-Hellman 5-8
digital envelope 5-4
digital signature 5-1, 5-2
    DSA 5-7
    RSA 5-5
DSA 5-2, 5-7
DSSRandom class 3-1
Dubuisson 7-1, 7-4, 7-7, 7-8

E

ECB 4-5
electronic code book 4-5
encoding rules 7-7
EncodingPolicy 7-12
encrypt: method 4-4
encryptInPlace: method 4-4
encryptInPlace:from:to: method 4-4, 4-9
encryption
    block cipher 4-9
    public-key 5-1
    RSA 5-3
    stream cipher 4-11
    symmetric-key 4-3
encyrption
    block cipher 4-7

F

flush method 5-3
fonts viii

G

generate key pair 5-3
getting help x

H

hashAlgorithm: method 5-6

I
imported types 7-13
K
Kerberos 7-1
key: method 4-3
KeyGenerator class 5-2
keySize: method 5-2
L
LDAP 7-1
M
MODULE 7-3
module 7-4
mouse buttons ix
  <Operate> button ix
  <Select> button ix
  <Window> button ix
N
name space 1-3
Name Space, importing Net Clients into 1-3
Net Clients
  importing into a Name Space 1-3
notational conventions viii
O
OBJECT IDENTIFIER 7-3
OFB 4-6
output feedback 4-6
P
parcels 1-2
PER 7-7
privateKey method 5-3
public/private key pair 5-2, 5-8
public-key cryptography 1-2, 5-1
publicKey method 5-3
R
RC4 4-11
resetDefault message 3-7
resetDefaultFrom: message 3-7
RSA 5-2
S
Secure Socket Layer. *See* SSL.
SEQUENCE 7-3
SEQUENCE OF 7-3
session, SSL
  renegotiation 6-11
  resumption 6-11
SET 7-3
SET OF 7-3
shared secret 5-8
sign: method 5-5, 5-7
SMINode 7-14
SNMP 7-1, 7-3, 7-14, 7-15
special symbols viii
SSL
  anonymous handshake 6-10
  certificate 6-4
  certificate chain 6-5
  client authentication 6-8
  exceptions 6-4
  handshake and certificates 6-4
  introduction 6-1
  limitations in implementation 6-1
  session
    renegotiation 6-11
    resumption 6-11
  signing certificates 6-8
  using 6-2
SSLConnection class 6-2
SSLContext class 6-2, 6-6
SSLError class 6-4
SSLException class 6-4
SSLSession class 6-2
SSLWarning class 6-4
stream cipher 4-1, 4-11
Struct 7-13
symbols used in documentation viii
symmetric-key cryptography 1-2, 4-1
T
technical support x
TLV encoding 7-8
triple-key encryption 4-8
triplet encoding 7-8, 7-14
type mapping 7-11
type tags 7-5
TypeWrapper 7-12
typographic conventions viii
U
useMD5 method 5-5
useSHA method 5-5
V
verify:of: method 5-5, 5-8
X
X509Registry 6-5
XER 7-2