Cincom

Cincom Smalltalk™ VisualWorks®

Security Guide

VisualWorks 8.3

P46-0143-08

SIMPLIFICATION THROUGH INNOVATION

Notice

Part Number: P46-0143-08

Software Release: 8.3

This document is subject to change without notice.

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

This document, the VisualWorks *Security Guide*, describes the 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.

Organization

This Guide begins with a general, and brief, introduction to software security and the problems it addresses.

The subsequent chapters introduce specific support for security technologies within VisualWorks as well as their use. The following chapters then describe the libraries provided by the security framework and how to use them when building your applications:

"Introduction to Security" gives 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.

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

"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.

"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.

"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.

"X.509 Certificates" describes X.509 certificates and how to use them in VisualWorks.

"Transport Layer Security" describes the implementation and use of the TLS/SSL protocol within VisualWorks.

"ASN.1" describes ASN.1 and the design of its VisualWorks implementation.

Conventions

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

Typographic Conventions

The following fonts are used to indicate special terms:

Examples	Description
template	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
`c:\windows`	Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).

Examples	Description
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 tools.
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 > **Open...**	Indicates the name of an item (**Open...**) 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
<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 personal-use 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 issues, send email to: helpna@cincom.com.

Before contacting Technical Support, please be prepared to provide the following information:

- The release number, which is displayed in the Welcome Workspace when you start VisualWorks.
- Any modifications (patch files, auxiliary code, or examples) distributed by Cincom that you have loaded into the image. Choose **Help > About VisualWorks** in the VisualWorks Launcher window. All installed patches can be found in the resulting dialog under **Patches** on the **System** tab.
- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, use **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.

- The hardware platform, operating system, and other system information you are using.

You can contact Technical Support as follows:

E-mail	Send questions about VisualWorks to: helpna@cincom.com.
Web	Visit: http://supportweb.cincom.com and choose the link to Support.
Telephone	Within North America, call Cincom Technical Support at +1-800-727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time. Outside North America, contact the local authorized reseller of Cincom products.

Personal-Use Licensees

VisualWorks Personal Use License (PUL) 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 an important public resource for VisualWorks developers:

- A mailing list for users of VisualWorks PUL, which serves a growing community of users. To subscribe or unsubscribe, send a message to: vwnc-request@cs.uiuc.edu with the SUBJECT of **subscribe** or **unsubscribe**. You can then address emails to: vwnc@cs.uiuc.edu.

The Smalltalk news group, comp.lang.smalltalk, carries on general discussions about different dialects of Smalltalk, including VisualWorks, and is a good source for advice.

Online Help

VisualWorks includes an online help system.

To display the online documentation browser, open the **Help** pull-down menu from the VisualWorks main window's menu bar (i.e., the Launcher window), and select one of the help options.

Additional Sources of Information

Cincom provides a variety of tutorials, specifically for VisualWorks.

The guide you are reading is but one manual in the VisualWorks documentation library.

Complete documentation is available in the `/doc` directory of your VisualWorks installation.

Books on Computer Security

The following is a short list of books that our 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*. Addison-Wesley (November 21, 2008).

Schneier, Bruce. *Applied Cryptography*. Addison-Wesley (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 a variety of specifications, available on the web. The following are a sample:

Secure Random Numbers	RFC 4086
Secure Hashes & MACS	
MD5	RFC 1321
SHA	FIPS 180-1

SHA-1, 256, 384, 512	FIPS 180-2
HMAC	RFC 2104, FIPS 198a
Encryption Algorithms	
DES	FIPS 46-3
Blowfish	The Blowfish Encryption Algorithm
AES	FIPS 197
Public-Key Algorithms	
Diffie-Hellman	RFC 2631
DSA	FIPS 186-2
RSA	PKCS #1, RFC 2437
Keys & Certificates	
X.509	Algorithms RFC 3279, Cerificates RFC 5820, Attribute Certificates RFC 3281
Password-Based Cyptography	PKCS #5, RFC 2898
Private-Key Information Syntax	PKCS #8
Personal Information Exchange Syntax	PKCS #12
Secure Protocols	
SSL & TLS	RFC 2246, RFC 4346, RFC 5346
HTTPS	RFC 2818

Chapter

1

Introduction to Security

Topics

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 are available in VisualWorks, and gives guidelines for employing them within your Smalltalk application.

Overview

Cryptography provides a way to convert intelligible data into an unintelligible form, 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 cryptography uses the same key for encrypting and decrypting the data. Accordingly, the key must be known to all persons who are authorized to decode and view the data. Symmetric-key cipher support is described in Symmetric-key Cryptography. Issues and techniques for keeping the key itself secure are outside the scope of this document.

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. These methods are described in Public Key Cryptography.

Beginning with VisualWorks release 7.9, cryptographic algorithms are no longer implemented in VisualWorks. Instead, VisualWorks accesses external libraries implemented on each OS platform. On Windows Vista and later, the Windows `bcrypt.dll` library is used. On other platforms, the OpenSSL `libcrypto.so` library is used. A uniform interface to these libraries is provided by using the Xtreams add-in (the `Xtreams-Crypto` parcel in `/contributed/Xtreams`).

Loading Security Components

Most encryption functions are provided by the Xstreams-Xtras parcel in `/contributed/Xtreams`. Access to those features, and additional non-cryptographic functions, are provided by parcels the parcels described below. Using the Parcel Manager, they are listed in the **Security** folder. The parcels are as follows.

- PKCS5 - password based cryptography
- PKCS8 - secure private key storage/export/import
- X509 - public key certificates
- TLS - transport layer security (TLS/SSL)
- TLS-Classic - classic stream facade for TLS

A few other parcels are loaded by these as prerequisites.

Importing Security Components into a Name Space

Security classes are located in different name spaces:

- PKCS5 - Security.*
- PKCS8 - Security.*
- X509 - Security.X509.*
- TLS - Xtreams.*

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.

Chapter

2

Hashes and Message Digests

Topics

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.

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.

Working with Hashes

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

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`.

Chapter

3

Random Number Generators

Topics

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.

This chapter describes the use of class `DSSRandom`, a VisualWorks RNG, 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.

Overview

VisualWorks provides a several PRNGs, but not all of them are satisfactory for use in security applications (e.g., refer to the description of class Random in the the *Basic Libraries 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: 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.

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).

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.

Chapter

4

Symmetric-key Cryptography

Topics

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.

Encryption operations are provided by external libraries, with the API provided by Xtreams, in the Xtreams-Xtras parcel. Because of the dependence on enternal libraries, supported algorithms may vary by platform.

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 (select **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

The Cipher Read/Write Streams encrypt or decrypt bytes passing through the stream using a pre-configured symmetric cipher (DES, AES, Blowfish, ...). The algorithm, direction (encrypt or decrypt), the key and initialization vector is configured when the stream is created. The underlying stream must be binary and the cipher streams themselves consume/produce bytes as well.

There are several required parameters for creating cipher streams. Which algorithms and modes are available depends on the underlying implementation of Cipher. Commonly available algorithms are AES, DES, 3DES and RC4. Commonly available modes

are CBC, CFB, OFB, CTR. The mode has significant impact on the security properties of the encrypted data; if in doubt use CBC.

The second required parameter is the key, which is a `ByteArray` of appropriate size. Usually a given cipher supports a specific key size (e.g., a DES key is 8 bytes, a 3DES is 24 bytes), or it can support multiple sizes (e.g., AES-128 supports 16 bytes, AES-256 supports 32 bytes).

Finally, depending on cipher mode, an initialization vector ("IV") may be needed. It is again a `ByteArray` where the size corresponds to the block size of the cipher (e.g., DES is 8 bytes, 3DES is 8 bytes, AES is 16 bytes). Stream ciphers (e.g., RC4) do not need a mode or an IV.

The following table lists some of the commonly used algorithms and their properties

Algorithm	Key Size	Block Size
AES	16B	16B
AES	24B	16B
AES	32B	16B
DES	8B	8B
3DES	24B	8B
RC4	8-256B	n/a

Here is an example using a stream cipher.

```
| random key encrypted |
 random := [ (Random new next * 256) floor ] reading
  contentsSpecies: ByteArray; yourself.
 key := random read: 16.
 encrypted :=
  ((ByteArray new writing encrypting: 'RC4' key: key)
   encoding: #ascii)
   write: 'Message in a bottle!'; close; terminal.
 ((encrypted reading decrypting: 'RC4' key: key)
  encoding: #ascii) rest.
```

Block Ciphers

Block Cipher Implementations

In the case of block ciphers, the content must be processed in block-sized quantities, where the block size depends on the cipher. This does not directly impact the behavior of the API, but some amount of read-ahead from the source or write-behind to the destination is at times necessary. Similarly, write streams must be closed to make sure the last block has been written. Note that overall data size must be a multiple of the block size of the cipher. Consequently it may have to be padded to make it so. Since the encrypted bytes do not necessarily carry the information about the actual amount of bytes written, it might be necessary to communicate the original size separately.

```
| random key iv message padding encrypted |
 random := [ (Random new next * 256) floor ] reading
  contentsSpecies: ByteArray; yourself.
 key := random read: 16.
 iv := random read: 16.
 message := 'Message in a bottle!'.
 padding := ByteArray new: (iv size - (message size \\ iv size)).
 encrypted := ((ByteArray new writing
  encrypting: 'AES' mode: 'CBC' key: key iv: iv) encoding: #ascii)
   write: message; write: padding; close; terminal.
 ((encrypted reading decrypting: 'AES' mode: 'CBC' key: key iv: iv)
  encoding: #ascii) read: message size.
```

Password-based Encryption and Authentication

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 VisualWorks implementation 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 API

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
```

Chapter

5

Public Key Cryptography

Topics

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 key 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.

Encryption operations are provided by external libraries, with the API provided by Xtreams, in the `Xtreams-Xtras` parcel. Because of the dependence on enternal libraries, supported algorithms may vary by platform. The primary focus of this API is using keys regardless of their underlying implementation. A key can be used for different purposes depending on the algorithm. An attempt to use a key for wrong purpose will most likely result in implementation specific error.

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 holder 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.

To sign a sequence of bytes ("a message"), first a digest of the message has to be computed and then signed using the private key.

```
digest := (message reading hashing: 'SHA1') -= 0;
 close;
 digest.
signature := dsaPrivateKey sign: digest
```

A specific signing algorithm may require a specific type of digest. For example, RSA can use MD5 or SHA1, but DSA requires SHA1. Some algorithms support different ways to pad the input to the required size (generally determined by the size of the key). Unless there are no alternatives available for a given algorithm message, sign:hash:padding: has to be used, specifying which algorithm should be used:

```
digest := (message reading hashing: 'MD5') -= 0;
 close;
 digest.
signature := rsaPrivateKey sign: digest hash: 'MD5' padding: 'PKCS1'
```

The corresponding public key must be used to verify a signature. Again, there are two variants of the verification method: one with and one without explicit hash and padding arguments subject to the same conditions as the signing method. The result of the verification is a Boolean indicating if the signature checks out.

```
rsaPublicKey
 verify: signature
 of: digest
 hash: 'MD5'
 padding: 'PKCS1'
```

The sign: methods return a ByteArray, but the encoding of the signature in the ByteArray again depends on the algorithm and its underlying implementation. The encoding does not matter if the signature will be used with same implementation. However, if the signature will be processed by different implementation, or needs to be serialized for storage or transport, the bytes may need to be decoded into the constituent elements (usually large positive integers). Private keys support an API to turn the signature bytes into its elements, and public keys provide API to turn signature elements back into bytes.

```
digest := (message reading hashing: 'SHA1') -= 0;
 close;
 digest.
signature := dsaPrivateKey sign: digest.
elements := dsaPrivateKey signatureElementsFrom: signature.
bytes := dsaPublicKey signatureByteFrom: elements.
 "bytes = signature"
```

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."

With encryption capable algorithms (e.g. RSA) the public key can be used to encrypt a sequence of bytes which only the holder of the

corresponding private key can decrypt back. Generally the sequence of bytes cannot exceed the size of the keys. As with signatures, the bytes may have to be padded to key size and the padding method has to be specified.

```
encrypted := rsaPublicKey encrypt: message passing: 'PKCS1'.
decrypted := rsaPrivateKey decrypt: encrypted padding: 'PKCS1'.
"decrypted = message"
```

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.

Key agreement algorithms allow two parties to establish a shared secret over an unprotected channel. The two parties have to be in possession of compatible keys, where compatible means not just the same algorithm but also same (algorithm dependent) parameters. The two parties then exchange their public keys over the channel (they don't need to be encrypted, they are public) and then each use their private key and the other party's public key to arrive at the same, shared secret value.

```
bobPublic := PublicKey DH: bobPublicKeyElements.
secret := alicePrivate derive: bobPublic.
```

PKCS8 Encoding for Private Keys

PKCS #8: Private-Key Information Syntax Standard describes a binary encoding for transfer and storage of private-keys. Private-key information includes a private key for some public-key algorithm and a set of attributes.

The standard also describes encoding for encryption of private keys to protect their confidentiality. It proposes few specific methods of

encryption based on password-based encryption algorithms from PKCS #5. Given the highly sensitive nature of private keys, there is rarely a good reason to store or transport them unencrypted.

The standard does not cover the actual format for specific kinds of private keys. Following the openssl man page, VisualWorks follows the definitions specified in: PKCS #11 .

VisualWorks supports two types of keys, represented by classes:

- RSAPrivateKey
- DSAPrivateKey

Encryption algorithm support is limited by the capabilities of the PKCS#5 implementation, which currently allows us to support:

- DES for encryption and MD5 for key derivation (OID pbeWithMD5AndDES-CBC)
- DES for encryption and SHA1 for key derivation (OID pbeWithSHA1AndDES- CBC)

These are among the most widely supported algorithms. We intend to add more secure versions in the near future. The default algorithm used for encryption is DES with SHA1.

Reading and Writing Keys

To encode a key on a stream, send a writeOn: message, for example

```
key := (RSAKeyGenerator bitSize: 512) privateKey.
stream := ByteArray new readWriteStream.key writeOn: stream.
```

Note that the stream must be binary and positionable. If you are writing the key to a file, use an ExternalReadWriteStream.

To read the key back from the stream send a readKeyFrom: message, for example:

```
stream reset.
PKCS8 readKeyFrom: stream.
```

This will return whichever key type is next in the stream.

PKCS8 encoded keys are sometimes distributed in additional ASCII "armoring," called PEM format. This is basically the PKCS#8 bytes encoded in Base-64 encoding with some identification header and footer lines attached. For convenience we also provide equivalent reading methods for this PEM format, readPEMFrom: and readKeyPEMFrom:. So, you could read a key from a **.pem** file like this:

```
PKCS8 readKeyPEMFrom: 'key.pem' asFilename readStream.
```

Encrypting Keys

Private keys are highly sensitive information, therefore it is generally preferrable to keep the key encrypted while it is not in use.

The API to store a key encrypted is similar to the one for reading and writing, just requiring an additional password: keyword and parameter. The password parameter must be a ByteArray. It is up to the application to take care of the character encoding issues.

Here is a write example:

```
stream reset.
password := 'very secret password' asByteArrayEncoding: #utf_16.
key writeOn: stream password: password.
```

To read the key back:

```
stream reset.
PKCS8 readKeyFrom: stream password: password.
```

The default algorithm used for key encryption is PKCS #5 password-based encryption using DES and SHA1, which is used to derive the encryption key from the provided password (identified by the PKCS #5 specification as pbeWithSHA1AndDES-CBC). It is possible to use other algorithms as well. See the documentation of the PKCS5 package for more details about these algorithms. Here we will describe only how to choose and tune an algorithm for key encryption.

Consider the following example:

```
stream reset.
info := EncryptedPrivateKeyInfo PBE_MD5_DES.
info algorithmIdentifier parameters count: 2048.
info key: key password: password.
info writeOn: stream.
```

You work with the EnryptedPrivateKeyInfo objects to tune encryption parameters. Choose the desired encryption algorithm by using the corresponding instance creation method, in this case PBE_MD5_DES (as opposed to the default which uses SHA1 instead of MD5). With the EncryptedPrivateKeyInfo instance in hand, you can adjust specific parameters. In this example we increase the "iteration count" to 2048. Once the algorithm is set up, you can store the key by sending the key:password: message to the EnryptedPrivateKeyInfo instance.

Note that the key is encrypted immediately, so any parameters must be tuned before this step. Finally, write the EnryptedPrivateKeyInfo instance out on the stream.

Reading this is much easier because all the information about the encryption algorithm and specific parameters is stored along with the encrypted key.

```
stream reset.
PKCS8 readKeyFrom: stream password: password.
```

Chapter

6

X.509 Certificates

Topics

Overview of X.509 Certificate Usage

A public-key certificate provides a secure way to deliver public keys over the internet. X.509 is a specification for what has become the standard public-key certificate mechanism.

An X.509 certificate binds a “name” of a subject to a public key. The certificate itself is issued and digitally signed by a “certificate authority” (CA), vouching that the embedded public key does, in fact, belong to the named subject. The key can then be used to encrypt a message for the subject in confidence that only the subject, using the corresponding private key, will be able to read the message.

Someone who knows the issuer’s public key can use the key to validate the signature on the certificate. If it passes validation, the user can be assured that the certificate was not tampered with, and also that it was signed by the issuer, because only the corresponding private key could have been used to generate the signature, and issuer should be the only one who knows it. As long as the issuer can be trusted to only issue certificates to entities being identified by those certificates, then the certificate can be used as a trusted source of the subject’s public key.

This makes certificates helpful in solving key management problems. For example, it allows two formerly unaquainted parties to authenticate each other over an insecure communication channel, such as the internet.

Here is a simplified example of how this works. Suppose Alice wants to talk to Bob. Alice knows and trusts a certificate issuer, CA, and CA has issued a certificate to Bob. Given this, Bob can prove his identity to Alice over an unsecured channel as follows:

1. Bob sends his certificate to Alice.
2. Alice verifies authenticity of the certificate, using the public key of CA which she knows and trusts.
3. Alice verifies that the subject name of the certificate identifies Bob.
4. Alice generates a unique, unpredictable sequence of bytes, a challenge.

5. Alice uses the public key from Bob's certificate to encrypt the challenge and sends it to Bob.
6. Bob decrypts the challenge using his private key and sends it back to Alice.
7. Alice compares Bob's response to the original challenge, if it matches she can be reasonably confident that she is talking to Bob and that the certificate contains his true public key.

While this process is fairly simple in theory, there are many complicating details involved that have to be just right. For example:

- How to generate a unique and unpredictable challenge.
- Bob and CA must be able to keep their private keys private.
- The probability of a key compromise grows with time, requiring that keys be replaced by new keys periodically, and that certificates restricted the time of validity of the key they carry.
- The CA must prevent issuing fraudulent certificates, making sure that only a certificate for Bob has Bob's true public key, and that Bob has the corresponding private key.

Addressing many of these issues constitute the core business of certificate authorities.

CA public keys are distributed in the form of self-signed certificates, for which the issuer and the subject is the same entity. Because the signature on the certificate does not provide any security for these, self-signed certificates must be delivered in another way. Certificates for established CAs are often pre-bundled with software, such as web browsers and operating systems. Certificates distributed in this way might be adequately secure for low-risk applications, such as web browsing. In general, however, higher security requirements call for more stringent deployment processes.

Large CAs rarely issue end-user certificates using their top level keys, because these keys need to be guarded under extremely strict security provisions. Instead they issue intermediate keys and certificates, indicating in the certificate that the associated keys are allowed to sign other certificates. The issuer of a second level certificate would be the subject of the top level certificate. The second level certificate and keys are then used to issue customer

certificates. The sequence of certificates linked by matching issuer-subject names is called a certificate chain or path. An application needs to know only the top level certificate key in order to be able to validate an entire chain.

Further explanation of certification processes is beyond scope for this document. For more information, there are a variety of resources, such as:

- The X.509 standard itself, at www.ietf.org/dyn/wg/charter/pkix-charter.html
- The corresponding RFC, currently 5280
- *Planning for PKI: Best Practices Guide for Deploying Public Key Infrastructure*, by Russ Housley and Tim Polk.

Processing X.509 Certificates

X.509 certificates are stored and transmitted in a standard binary format, and are often base-64 encoded to simplify their handling. The format is generally transparent from the perspective of VisualWorks X.509 support.

The validation process is mostly authomatic, handled by the X.509 framework, but requires a registry of trusted certificates. However, the X.509 framework cannot automatically validate the self-signed root certificates in the registry. Accordingly, it is the 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.

Importing Certificates

Whether you receive a root certificate from a CA directly, or export them from another registry, you will generally import the certificate into VIsualWorks from a file. The method you use to read and import the file varies depending on the file format.

PEM format files are base64 encoded, begin with “--- BEGIN CERTIFICATE ---” and end with “--- END CERTIFICATE ---”. Files

in this format can be read in by sending a fromFile: message to the Certificate class, with the file name as argument. For example,

```
Security.X509.Certificate fromFile: 'sample.pem'
```

The file name extension varies based on platform and source.

Files not in PEM format are generally raw byte binary files. You can import these from a ReadStream. For example:

```
stream := ('sample.cer' asFilename withEncoding: #binary) readStream.
 [ Certificate readFrom: file ] ensure: [ file close ]
```

Additional instance creation methods are available for other formats. Browse the Certificate class for these.

Once the certificate is read into VisualWorks and represented as a Certificate, it can be added to the registry.

Setting up an X509Registry

VisualWorks validates certificates against a set of trusted, self-signed certificates that are held in an instance of X509Registry. Any certificate chain for which the root certificate is in the registry can be validated.

VisualWorks provides a default registry instance, accessed by

```
X509Registry default
```

The registry is initially empty. This registry is used, for example, by VisualWork NetClients HTTPS support. An application may populate and use the default registry, or create and use its own.

It is absolutely essential for validation that the X509Registry is initialized with true certificates of trusted CA authorities. For applications where the threat level is relatively low, web browsers and some operating systems provide a source of CA certificates. They usually have a fairly extensive set of certificates, and provide a way to export the certificates to files which can then be read into VisualWorks. For applications with higher security requirements, self-signed certificates should be obtained from the CA using a secure alternate method.

Given a certificate exported into a file named `test.pem`, you can read the certificate and add it to an X509Registry as follows:

```
| certificate registry |
 registry := Security.X509.X509Registry new.
 certificate := Security.X509.Certificate fromFile: 'test.pem'.
 registry addTrusted: certificate.
```

If you receive a trusted certificate from another source, you can add it to the register in the same way.

Operating systems also frequently maintain a registry. On most Linux distributions, for example, there are CA certificates in the file `ca#bundle.crt`. 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'.
 certificates do: [:certificate | registry addTrusted: certificate].
```

It is wise to review the contents of the certificate bundle and only import those certificates that your application actually needs.

Certificate Validation

The primary access point for validating a certificate is the validateCertficateChain: message, which is sent to the active X509Registry. The method expects a certificate chain, which is a sequence of certificates ordered in ascending order of the subject-issuer relationship, subject before its issuer.

This method performs all the necessary checks, and raises an exception if any check fails. The method checks, for example, for every adjacent pair in the chain:

- whether any certificates have been revoked (checked against the revoked certificates in the X509Registry)
- whether all certificates are currently valid (within the notBefore/notAfter period)

- whether there are any critical certificate extensions that the framework cannot process (if so, validation must fail)
- whether the subject of one certificate match the issuer of the other
- whether the public key of the issuer validates the signature of the subject

Exceptions are all resumable, giving the user a chance to disregard these security warnings in specific cases.

The method also allows collecting a list of all problems of the entire chain so they can be presented all at once.

If the method completes without exceptions, the chain passes validation.

Subject Verification

Verifying the final subject cannot be done automatically, so must be performed manually. It is the user's responsibility to make sure that the subject of the certificate identifies the intended subject.

Suppose Alice wants to encrypt something for Bob, receives a certificate and the certificate checks out as valid. She still needs to make sure that the certificate identifies Bob as its subject. Otherwise, someone might send his own perfectly valid certificate, pretending to be Bob, thus subverting the authentication. For this check, the user needs to analyze the subject name and be satisfied that it is legitimate, based on its contents.

The subject is represented by a `Name` instance. A `Name` is a collection of `AttributeValueAssertion` instances, which are associations in which the keys represent well known aspects of an X.509 name. The various attributes have well established acronyms, for example:

- C=Country
- L=Locality
- O=Organization
- OU=Organization Unit
- CN=Common Name

The Name object typically contains several of these, as can be seen by sending a printOpenSSLString message to a certificate. For example,

```
aCertificate subject printOpenSSLString
```

might return

```
'C=ZA, S=Western Cape, L=Cape Town, O=Thawte Consulting cc, OU=Certification
 Services Division, CN=Thawte Server CA, E=server-certs@thawte.com'
```

The attributes can be used to verify the subject during connection, using either the acronyms or accessor messages.

Certificate Structure

You can get a quick overview of the contents and structure of a certificate by inspecting it and navigating to its "openssl" attribute. A sample certificate is provided for this purpose. Inspect:

```
Security.X509.Certificate fromBase64: (Security.X509.Certificate thawte)
```

and select the **openssl** attribute.

A Certificate object has several components. Normal Smalltalk objects represent components where possible, such as Time objects for validity dates. Most components are accessible using accessor messages. For example, the publicKey message returns the embedded key, which then can be used to encrypt a message:

```
aCertificate :=  Security.X509.Certificate sampleCertificate.
message := 'Message in a bottle!' asByteArrayEncoding: #utf8.
encrypted := (Security.RSA new publicKey: aCertificate publicKey)
 encrypt: message
```

(In a case like this, you want to make sure to use the public key only for purposes consistent with the certificate's intention. A root certificate's public key, for example, is only to be used for signing certificates, not for encryption.)

Some components remain complicated X.509 objects, such as the issuer and subject names which are represented as X509.Name objects.

At the highest level, a certificate consists of two elements: the TBSCertificate (to-be-signed) and the signature. The TBSCertificate contains all of the information in the certificate except the signature.

Within the TBSCertificate are fields representing all of the required and optional X.509 fields. Each is represented by appropriate accessor methods. A few of the standard fields are:

- issuer and subject - represented by Name instances
- validity - “not before” and “not after” Time instances
- version - the X.509 version employed by the certificate
- serial number - provided by the CA
- signature algorithm identifier - algorithm used to sing the certificate
- subject public key information - a SubjectPublicKeyInfo instance holding the public key and the algorithm
- extensions - X.509 version 2 and 3 extensions used by the certificate

Browse the Certificate and TBSCertificate classes for accessor methods.

Creating a Certificate

For certain (usually well contained) deployments, it might be more suitable to maintain a private certification infrastructure than to rely on third party certificate authorities. This entails all of the associated maintenance and operation overhead of a CA, and requires a thorough understanding of the related standards and processes. The VisualWorks X509 framework provides the low level technical tools to do this. At the lowest level, all you need is the capability to create new certificates.

Create a Self-signed Certificate

To create a self-signed CA certificate, we need a CA name and a pair of keys, the public key to put into the certificate and the private key to sign it with.

```
caName := Security.X509.Name new
 country: 'US';
 locality: 'Cincinnati';
```

```
 organization: 'Cincom Systems';
 organizationUnit: 'Cincom Smalltalk';
 commonName: 'CST Test CA';
 yourself.
caName printOpenSSLString
```

results with:

```
'C=US, L=Cincinnati, O=Cincom Systems, OU=Cincom Smalltalk, CN=CST Test CA'
```

Given a public key (`rsaPublicKey` in the following example), a `Certificate` requires few more attributes as well:

```
ca := Security.X509.Certificate new
 serialNumber: 1000;
 issuer: caName;
 subject: caName;
 notBefore: Date today;
 notAfter: (Date today addDays: 7);
 publicKey: rsaPublicKey;
 forCertificateSigning;
 yourself.
```

The `forCertificateSigning` message is required for any CA certificate. It allows the certificate validation algorithm to distinguish CA certificates from end-user certificates. Otherwise, a malicious end-user could start issuing other certificates signed by the key from his or her own certificate which would seem like valid certificates from the CA hierarchy.

The last missing bit on the new certificate is the signature. A certificate is signed using the signing (private) key.

```
ca signUsing: rsaPrivateKey
```

We now have a complete certificate expressed as a Smalltalk object. To convert it to a standard binary format it has to be written into a binary stream (currently the stream has to be an internal stream):

```
caBytes := ByteArray new writeStream.
ca writeOn: caBytes.
caBytes contents.
```

Issue a Certificate

To issue a certificate, create a new certificate for a given subject with subject's public key and sign it using your CA private key:

```
subjectName := Security.X509.Name new CN: 'Test Subject'; yourself.
subjectKey := dsaPublicKey.
```

Create the new certificate as previously. Note, however, that it is important to express the usage of the associated key properly. For example, if the key from this certificates is to be used for signing data, rather than for signing other certificates, send a `forSigning` message. For example:

```
subject := Security.509.Certificate new
 serialNumber: 2000;
 issuer: caName;
 subject: subjectName;
 notBefore: Date today;
 notAfter: (Date today addDays: 7);
 publicKey: subjectKey;
 forSigning;
 yourself.

subject signUsing: (Security.RSA new
 useSHA;
 privateKey: rsaPrivateKey;
 yourself).
```

The usage options are supported by similar methods:

forCertificateSigning

Mark the certificate as suitable for signing other certificates.

forCRLSigning

Mark the certificate as suitable for signing CRLs.

forEncryption

Mark this certificate as suitable for encryption of data.

forKeyExchange

Mark this certificate as suitable for key exchange or key agreement depending on the key algorithm.

forSigning

Mark this certificate as suitable for signing anything except certificates and CRLs.

Chapter

7

Transport Layer Security

Topics

The TLS parcel provides an implementation of the Netscape SSL protocol version 3.0 and TLS 1.0, 1.1 and 1.2.

VisualWorks supports most of the relevant TLS/SSL cipher suites. The list of currently supported cipher suites can be browsed in `TLSCipherSuite` class.

Encryption services are provided by external libraries, which are accessed using Xtreams libraries.

General API

The main components of the public API are TLSContext, TLSCertificateStore, TLSSession and TLSConnection.

TLSContext

TLSContext represents a server/client context maintaining a collection of fairly static preconfigured communication parameters and options. The context specifies primarily the supported cipher suites and certificate registries, but can specify other parameters as well. It is also responsible for maintaining the collection of resumable sessions for TLS servers.

TLSContext is used by an application to create a suitable environment for TLS communication. The instance creation methods are:

newWithDefaults

Create a TLS context for all supported cipher suites, using the default X509Registry for certificates.

newWithSuites: criteria

Create a TLS context for all supported cipher suites, using aCertificateRegistry for certificates.

Once an instance of TLSContext is created and configured, requests to build instances of TLSConnection can be sent to it, using these messages:

newConnection: aSocket

Create a new TLSConnection reading and writing on aSocket.

newConnectionReading: input writing: output

Create a new TLSConnection reading and writing on separate input and output streams.

TLSCertificateStore

TLSCertificateStore manages trusted certificates for certificate authorities and known peer certificates. It also manages the certificates and keys that are used to authenticate to peers. The class encapsulates all operations with keys and certificates, such as

certificate validation, signing and verification, as well as encryption and decryption using public/private keys.

By default, TLSCertificateStore uses the X509Registry for storage of certificates (refer to Processing X.509 Certificates for more information about this registry).

TLSSession

TLSSession 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 (e.g., connectionFor:using:) when connecting a new connection.

TLSConnection

TLSConnection implements an actual connection between two parties, and maintains a pair of communication streams between the parties. It is the component that is most exposed to an TLS application.

SSLConnection is responsible for encryption/decryption of SSL records (basic units of SSL communication) and maintenance of negotiated encryption parameters. It also provides the bulk of the public API for connecting, accepting connection, closing connection, and secure data streams. A few essential instance methods are the following:

accept

Handshake as a server without validating the client. This is the usual method, because the servers only rarely validate their clients. If client validation is required, use accept: instead.

connect: aSubject

Perform TLS client handshake. If subject is a block, perform a full handshake and use the block to verify that the server certificate matches the desired target. If subject is a Session, attempt a short/fast handshake with the cached session

parameters, but if the server rejects the session, fail over to full handshake using the verification block cached by the session. If the subject is nil, perform an insecure, anonymous handshake.

close

Close the secure channel.

A Usage Pattern

The usual usage pattern for establishing an TLS connection goes something like the following.

Note that the handshake entails authenticating against a `TLSCertificateStore`, which must be set up prior to this. By default, `TLSCertificateStore` refers to an `X509Registry`. Refer to the discussion of X.509 for details and instructions on setting up the registry.

First an instance of a context has to be created and initialized with supported cipher suites. There are messages to support standard common suite selections or to specify a collection:

```
context := TLSContext newWithDefaults.
```

To specify a collection of cipher suites, create the instance using the class method `newWithSuites:`.

The connection is then created from the context as follows:

```
connection := context newConnection: aSocketAccessor.
```

The `newConnection:` message creates read and write streams on the same `SocketAccessor`.

Now we are ready to perform a the handshake to establish the connection. Each TLS connection connects a client with a server, playing two very different roles in the handshake.

To connect as a client, send the connection a `connect:` message. In this case, the client does want to validate the subject, to prevent man-

in-the-middle attacks. The block argument performs the check, for example to verify by common name.

```
connection connect: [:cert | cert subject
 commonName = 'www.google.com'].
```

To connect as a server, send the connection an `accept` message. No validation of the client subject is usually required.

```
connection accept.
```

At this point you can ask the connection for a stream for reading or writing. With TLS-Classic loaded, you can get standard VisualWorks stream behavior by sending:

```
connection readAppendStream
```

which returns a binary read/append stream. To get a character-oriented read/write stream, you can send:

```
(connection withEncoding: #utf8) readAppendStream
```

emulating the pattern of `ExternalConnection`.

Without TLS-Classic loaded, use the Xtreams `reading` and `writing` messages to get individual streams.

It is important to explicitly flush the TLS buffers, to ensure that data is written onto the steam, as with any socket stream.

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

Both the streams and the connection should be explicitly closed, otherwise they can leak resources and secrets.

```
writeStream close.
connection close.
```

Different components of the TLS setup (connection, session, context, certificate store, session cache) have different lifecycle and can hold

external resources as well. It is important that they are also released when no longer needed (send message release). A session that is not resumable will be released automatically when the corresponding connection is closed. However, resumable sessions are managed by the session cache, so release the cache when no longer needed. Similarly a certificate store holds private keys, which need to be disposed properly, so releasing the store is equally important.

Note that TLS 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 TLS.

Handshake and Certificates

The default TLSCertificateStore settings are designed to provide a secure connection, and this cannot be achieved without authenticating the server party. (Servers do not usually care about client's identity, so clients are not required to authenticate by default.)

Authentication is performed using X.509 certificates. In order to authenticate a server, the client needs to have a collection of "trusted" CA certificates. This collection is maintained in an instance of X509Registry (refer to Processing X.509 Certificates on page 50 for instructions).

A server must register both its own certificate chain and the corresponding private key with its TLSCertificateStore. Therefore, these are registered with the context as a pair. To register the pair, send a certificate:key: message to the TLSCertificateStore. The first argument is either the server's certificate, or a chain of certificates as a sequenced collection (e.g., an Array). The chain must start with the server's own certificate, followed by its issuer's certificate. The

second argument is the private key corresponding to the public key in the server's certificate:

```
connection certificates certificate: chain key: correspondingPrivateKey
```

The server presents its certificate chain to the client during TLS handshake. The client then validates the chain using its certificate registry. If the validation fails the client signals TLSBadCertificate warning. Other certificate related warnings are TLSCertificateExpired, TLSCertificateRevoked, TLSCertificateUnknown and TLSUnsupportedCertificate.

Known Certificates

An alternative client-side setup can be useful where a client only connects to a single, or a few, specific servers. In this setup the client is pre-configured with specific server certificates. The certificates must be obtained via some other, secure and reliable means (for example, pre-bundled with the application). In this case the client validates the certificate received from the server by testing that the server certificate is bit-equivalent to a certificate in its list of "valid" certificates.

The list of valid certificates is maintained by the TLSCertificateStore. To add a certificate, declare it known by sending a known: message to the store:

```
connection certificates known: aCertificateOfAWellKnownServer.
```

Subject Validation

Even though TLS can do most of the validation by itself, it remains to verify that the certificate belongs to the party that the application is actually trying to reach. Without verifying the subject of the certificate, an imposter can present his own perfectly valid certificate.

This check is done by analyzing the subject's distinguished name embedded in the certificate to verify that it is the name of the other party. This task is performed by the subject validation block that is provided with the connect: message sent by the application when

the handshake is initiated. (For a server wishing to authenticate its clients, it accepts using accept:.)

The validation block takes one argument, an X509.Certificate. The block must return a Boolean, indicating if the certificate identifies the intended party. If the block returns false, an TLSBadCertificate warning will be signaled. Here is an example subject validation block:

```
connection connect:
 [:certificate | certificate subject country = 'USA' and:
  [certificate subject organizationUnit = 'Cincom']]
```

A fairly widespread convention on the web is embedding the URL of the server as the common-name attribute of the server certificate. This is how web browsers automate the subject validation when accessing HTTPS URLs. If the subject CN matches the URL the validation succeeds, otherwise it fails.

Certificates for Signing

Public keys can be used for different purposes depending on the underlying algorithm. X.509 certificates restrict the usage of their embedded public key to specific purpose. Accordingly, some certificates are suitable for TLS authentication (certificates for data signing and encryption) and some are not (certificate-signing certificates).

Certificates that are suitable for TLS can be separated into two basic categories: those that can be used for signing data and those that can be used for encryption or key exchange. Depending on the purpose of the certificate the TLS handshake works differently, and provide different advantages. For example a handshake using an encrypting RSA certificate will be more efficient, because it can facilitate both authentication and key exchange using a single private key operation which may be significant when trying to manage load on a busy server. On the other hand a signing DSA certificate with ephemeral Diffie-Hellman key exchange (discussed below) provides "forward secrecy" (later compromise of private key of the certificate does not compromise confidentiality of the TLS session), which is desirable for some applications.

If required to authenticate, clients always authenticate with a signing certificate.

The certificate is marked as for signing during its creation, by sending a `forSigning` message. For example:

```
subject := Security.509.Certificate new
 serialNumber: 2000;
 issuer: caName;
 subject: subjectName;
 notBefore: Date today;
 notAfter: (Date today addDays: 7);
 publicKey: subjectKeys publicKey;
 forSigning;
 yourself.

subject signUsing: (Security.RSA new
 useSHA;
 privateKey: caKeys privateKey;
 yourself).
```

The certificate and the private key corresponding to the public key in the certificate must be registered with the client's `TLSCertificateStoret`, by sending a `certificate:key:` message to the context. With this setup the certificate will be used to authenticate the server.

Diffie-Hellman Key Exchange

Signing certificates are usually used with Diffie-Hellman key exchange (cipher suites with identifier starting with DHE_). This algorithm requires parameters *p* and *g*. The parameters can be reused for a number of sessions or they can be regenerated from time to time. To facilitate either mode of use, these parameters are configured via a block captured by `TLSContext>>dheParameters`. The parameter block is invoked with another block as its argument. The parameter block must invoke its argument with the actual values of *p* and *g* (in that order). See `defaultDHEParametersValue` for an example.

Anonymous Handshake

In most cases it is necessary to validate a connection using certificates. There are, however a few situations when you do not need to authenticate the other party, because you are certain of that party's identity. In this situation you can do an "anonymous handshake."

A client-side handshake is anonymous by default. Therefore, an anonymous handshake, in TLS terms, is the case when the server does not authenticate, either.

Keep in mind that this type of handshake is strongly discouraged in general, because dropping authentication means dropping prevention of man-in-the-middle attacks.

In TLS, anonymity of the handshake is dictated by the selected cipher suite. The protocol defines only a few of those. All have DH_anon in the name and are based on Diffie-Hellman key exchange. Being anonymous, the certificate store is not needed in this case; however, the key exchange still requires the Diffie-Hellman parameters discussed earlier.

Session Renegotiation

Occasionally, when an TLS connection is maintained for an extended time period, it might be desirable to renegotiate a new set of security parameters. TLS does have provisions to do that without rebuilding the connection from scratch. Renegotiation can be initiated from either client or server side at any time by sending message renegotiate to the TLSConnection. The other side handles the renegotiation completely transparently. Normally a renegotiation will attempt to resume the same session (only refresh the connection keys), but if the session is not resumable it will fallback to full handshake automatically.

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 TLSSession. To get a new connection to reuse previously negotiated parameters it has to be built around the session that captures the parameters. The session can be passed in as the argument of the connect: message instead of a validation block. The session captures the validation block it was first negotiated with so if the resumption falls back to a full handshake, it will use the captured block.

```
session := previousConnection session.
newConnection := context newConnection: aSocketAccessor.
newConnection connect: session.
```

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. The ability to resume sessions is indicated by presence or absence of TLSSessionCache in the TLSContext. Any session can be explicitly made non-resumable by releasing it using message release. As was noted earlier the session cache is responsible for maintaining the set of active, resumable sessions. It is important to release the cache when no longer needed. Aside from releasing any associated external resources it also triggers destruction of any secrets associated with its sessions.

TLS Exceptions

An essential requirement for security technologies is being able to detect and stay in control when errors occur. Exception handling provides the means to do this.

TLS exceptions provided by subclasses of TLSError and TLSWarning. Some TLS warnings and errors represent protocol alerts, which are sent to the other side through the socket connection. These exceptions (as any other protocol messages) are also triggered as Announcements by the connection when they are sent or received. This is in addition to being signaled as exceptions as well.

TLSErrors are fatal exceptions; if an TLSError occurs, the operation in progress cannot complete and must be aborted, usually closing the connection as well. Fatal errors also render the associated session non-resumable.

TLSWarnings are resumable exceptions, 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 TLS warnings report serious security issues. To maintain security, ensure that it is safe to resume before doing so.

Chapter

8

ASN.1

Topics

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. SSL certificates are defined by the X.509 (see Handshake and Certificates) specification using 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.

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.

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
 last  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 `IA5String`s. 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 following comments are provided to expose the organization of the present VisualWorks ASN.1 implementation. The comments cover 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 in code comments; these comments provide an overview that integrates the comments in the code.

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 86165 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 `SEQUENCE`s or `SET`s 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

S

T

X