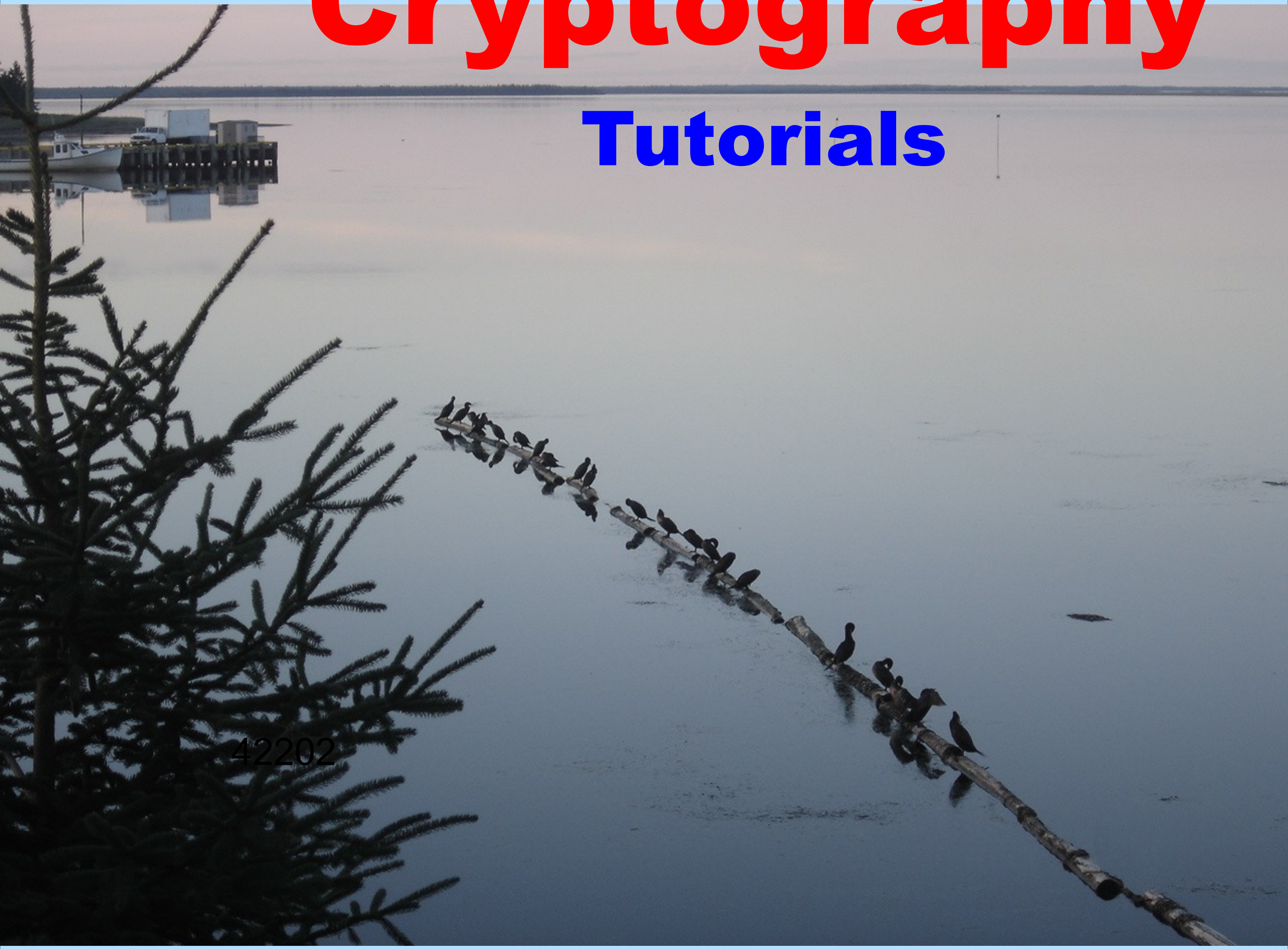
Herong's
Tutorial Examples

Cryptography

Tutorials



Herong Yang

www.herongyang.com/cryptography

Cryptography Tutorials - Herong's Tutorial Examples

Version 5.20, 2013

Dr. Herong Yang

HerongYang.com
Programming
Tutorial Notes

Table of Contents

Keywords: Cryptography, Encryption, Security, Tutorial, Example

Previous Version: http://www.herongyang.com/crypto/index2.html

About This Book

This section provides some detailed information about this book - Cryptography Tutorials - Herong's Tutorial Examples.

Website URL: http://www.herongyang.com/Cryptography

Title: Cryptography Tutorials - Herong's Tutorial Examples

Author: Dr. Herong Yang

Category: Cryptography/Tutorial

Version/Edition: Version 5.20, 2013

Number of pages: 309

Short description: This cryptography tutorial book is a collection of notes and sample codes written by the author while he was learning cryptography technologies himself. Topics include blowfish, certificate, cipher, decryption, DES, digest, encryption, keytool, MD5, OpenSSL, PEM, PKCS#8, PKCS#12, private key, public key, RSA, secret key, SHA1, SSL, X.509.

Long description: This cryptography tutorial book is a collection of notes and sample codes written by the author while he was learning cryptography technologies himself. Topics include blowfish, CA, certificate, certification path, cipher, CSR, decryption, DER, DES, digest, encryption, Java, JCE, JDK, keytool, MD5, message, OpenSSL, PEM, PKCS#8, PKCS#12, private key, public key, RSA, secret key, self-signed certificate, SHA1, SSL, X.509. Key sections are: Basic Concepts - Cipher - DES Algorithm - DES Algorithm - Illustrated with Java Programs - DES Algorithm - Java Implementation - JDK/JCE - Cipher for Encryption and Decryption - Cipher - Blowfish Algorithm - 8366 Hex Digits of PI - Message Digest - MD5 Algorithm Overview - MD5 Implementation in Java, PHP, PerlMD - Message Digest - SHA1 Algorithm Overview - SHA1 Implementation in Java, PHP, Perl - RSA public key encryption algorithm and Java implementation - DSA (Digital Signature Algorithm) and Java implementation - What is OpenSSL? - Installing OpenSSL on Windows - Generating RSA Key Pairs - Viewing Components of RSA Keys - Encrypting RSA Keys - What is a certificate? - Generating Self-Signed Certificates - Viewing Components of Certificates - Why Certificates Need to Be Signed by CAs? - Generating a Certificate Signing Request for Your Own Public Key - Viewing Components of Certificate Signing Request - Signing a Certificate Signing Request - keytool and Java keystore files - Certification Path Validation - Using Certificates in Web Browsers. - Certificate Format X.509, DER and PEM - Private Key Format PKCS#8 - Private Key and Certificate File Format PKCS#12

Keywords: Cryptography, Encryption, Tutorial, Example, Code, Free, Book.

Previous Version: http://www.herongyang.com/crypto/index2.html

Cost/Price: Free

Viewing statistics:

This book has been viewed a total of:

- 1,191,405 times as of December 2012.
- 936,329 times as of December 2011.
- 687,681 times as of December 2010.
- 417,742 times as of December 2009.
- 237,675 times as of December 2008.
- 190,670 times as of December 2007.

Revision history:

- Version 5.20, 2013. Added more notes on RSA and DSA algorithms.
- Version 5.00, 2009. Updated and formatted in hyPub format.
- Version 4.00, 2007. Added more notes OpenSSL and keytool.

Copyright:

-
- Material in this book may not be published, broadcasted, rewritten or redistributed in any form.
- The example codes is provided as-is, with no warranty of any kind.

Cryptography Terminology

This section provides descriptions on some commonly used cryptography terminologies

Blowfish: A Feistel network iterating a simple encryption function 16 times. The block size is 64 bits, and the key can be any length up to 448 bits. Although there is a complex initialization phase required before any encryption can take place, the actual encryption of data is very efficient on large microprocessors.

CBC (Cipher Block Chaining): An operation mode for block ciphers, where each plaintext block is XORed with the previous ciphertext block before encryption.

Certification Chain: Also called Certificate Path. An ordered list of certificates where the subject entity of one certificate is identical to the issuing entity of the next certificate.

Certificate Path: Also called Certification Chain. An ordered list of certificates where the subject entity of one certificate is identical to the issuing entity of the next certificate.

CFB (Cipher FeedBack): An operation mode for block ciphers, where each block of plaintext is XORed with the encrypted version of the previous ciphertext to generate the current ciphertext block.

DES (Data Encryption Standard): A 16-round Feistel cipher with block size of 64 bits. DES was developed by IBM in 1974 in response to a federal government public invitation for data encryption algorithms. In 977, DES was published as a federal standard, FIPS PUB 46.

ECB (Electronic CodeBook): An operation mode for block ciphers, where each plaintext block is encrypted independent from other blocks.

MD5 (Message Digest 5): A message-digest algorithm, which takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given pre-specified target message digest.

OFB (Output FeedBack): An operation mode for block ciphers, where each block of plaintext is XORed with the encrypted version of the previous ciphertext to generate the current ciphertext block.

OpenSSL: A cryptography toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) network protocols and related cryptography standards required by them.

PKCS5Padding: A padding schema for block ciphers, where the number of padded bytes equals to "8 - numberOfBytes(clearTextMessage) mod 8", and the value of each padded byte is an integer value of the number of padded bytes.

RSA (Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman): A public key algorithm invented in 1976 by three MIT mathematicians, Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman.

SHA1 (Secure Hash Algorithm 1): A message-digest algorithm, which takes an input message of any length < 2^64 bits and produces a 160-bit output as the message digest.

Cryptography Basic Concepts

This chapter describes some basic concepts of cryptography: what is cryptography, what is function and what is encryption.

What Is Cryptography?

This section describes what is cryptography - The study of techniques related to all aspects of data security.

Cryptography - The study of techniques related to all aspects of data security. The word "cryptography" is derived from the ancient Greek words "kryptos" (hidden) and "graphia" (writing).

Some aspects of data security:

- Confidentiality - Keeping data secret.
- Data Integrity - Ensuring data has not been altered.
- Entity Authentication - Identifying parties involved.
- Data Origin Authentication - Identifying the data origin.

Cryptanalysis - The study of techniques to defeat cryptographic techniques.

What Is Function?

This section describes what is function - Two sets of elements, X and Y, and a rule, f, which assigns to each element in X, to one element in Y.

Function - Two sets of elements, X and Y, and a rule, f, which assigns to each element in X, to one element in Y. A function is denoted as f: X -> Y.

Domain - The set of elements, X, in the above definition.

Codomain - The set of elements, Y, in the above definition.

If x is element in X, and y is the element in Y assigned to x by a function f, y is called the image of x, and denoted as y = f(x).

If y = f(x), x is called a preimage of y.

Image of Function - The set of all elements in Y which has at least one preimage in X.

1-1 Function - A function that satisfies that "if x <> y, then f(x) <> f(y)". No two elements in the domain will have the same image for a 1-1 function.

Bijection - A 1-1 function that satisfies that "image of f is Y".

Inverses Function - The function g of a bijection f that satisfies that "if y = f(x), then x = g(y)".

One-way Function - A bijection that satisfies that "the computation of its inverse function is very hard, and close to infeasible".

Trapdoor One-way Function - A one-way function that satisfies that "the computation of its inverse function becomes feasible, if additional information is given".

Permutation - A bijection that satisfies "f: X -> X".

Involution - A permutation that satisfies that "inverse function of f is f".

What Is Encryption?

This section describes what is encryption - A bijection function that uses a key, encryption key, to compute the image.

Encryption Function - A bijection function that uses a key, encryption key, to compute the image.

Clear Text Space - The domain of an encryption function.

Cipher Text Space - The codomain of an encryption function.

Decryption Function - The inversion function of an encryption function. Decryption function also uses a key, decryption key, to compute the image.

Encryption Scheme - An algorithm that defines a cleartext space, a ciphertext space, and a set of encryption keys. For each encryption key, the scheme also defines an encryption function, a decryption key, and a decryption function. An encryption scheme is also called a cipher.

Symmetric Key Encryption - An encryption scheme that "the decryption function uses the same key as its encryption function". Symmetric key encryption is also called one key, or secret key encryption.

Asymmetric Key Encryption - An encryption scheme that "the decryption function uses needs a different key than the key used in its encryption function, and it is almost impossible to compute one from another". Asymmetric key encryption is also called public key encryption.

Block Cipher - An encryption scheme that "cleartext is broken up into blocks of fixed length, and encrypted one block at a time".

Stream Cipher - An encryption scheme that "cleartext is encrypted as a continuous stream alphabets".

Simple Substitution Cipher - An encryption scheme that "Each alphabet in the cleartext is substituted by a single alphabet defined by the key".

Homophonic Substitution Cipher - An encryption scheme that "Each alphabet in the cleartext is substituted by an alphabetic string randomly selected from a set of strings defined by the key".

Transposition Cipher - A block cipher that "Alphabets in a block is permuted in an order defined by the key".

Product Cipher - An encryption scheme that "uses multiple ciphers in which the ciphertext of one cipher is used as the cleartext of the next cipher". Usually, substitution ciphers and transposition ciphers are used alternatively to construct a product cipher.

Introduction to DES Algorithm

This chapter provides tutorial examples and notes about DES (Data Encryption Standard) algorithm. Topics include description of block cipher, DES encryption algorithm, Round Keys Generation, DES decryption algorithm.

Conclusions:

- DES encryption is a 64-bit block cipher.
- 16 round keys are derived from a single 64-bit key.
- Decryption algorithm is identical to the encryption algorithm except for the order of the round keys.

What Is Block Cipher?

This section describes what is block cipher - An encryption scheme in which 'the clear text is broken up into blocks of fixed length, and encrypted one block at a time'.

Block Cipher - An encryption scheme in which "the clear text is broken up into blocks of fixed length, and encrypted one block at a time".

Usually, a block cipher encrypts a block of clear text into a block of cipher text of the same length. In this case, a block cipher can be viewed as a simple substitute cipher with character size equal to the block size.

ECB Operation Mode - Blocks of clear text are encrypted independently. ECB stands for Electronic Code Book. Main properties of this mode:

- Identical clear text blocks are encrypted to identical cipher text blocks.
- Re-ordering clear text blocks results in re-ordering cipher text blocks.
- An encryption error affects only the block where it occurs.

CBC Operation Mode - The previous cipher text block is XORed with the clear text block before applying the encryption mapping. Main properties of this mode:

- An encryption error affects only the block where is occurs and one next block.

Product Cipher - An encryption scheme that "uses multiple ciphers in which the cipher text of one cipher is used as the clear text of the next cipher". Usually, substitution ciphers and transposition ciphers are used alternatively to construct a product cipher.

Iterated Block Cipher - A block cipher that "iterates a fixed number of times of another block cipher, called round function, with a different key, called round key, for each iteration".

Feistel Cipher - An iterate block cipher that uses the following algorithm:

```
Input:
   T: 2t bits of clear text
   k1, k2, ..., kr: r round keys
   f: a block cipher with bock size of t

Output:
   C: 2t bits of cipher text

Algorithm:
   (L0, R0) = T, dividing T in two t-bit parts
   (L1, R1) = (R0, L0 ^ f(R0, k1))
   (L2, R2) = (R1, L1 ^ f(R1, k2))
   ......
   C = (Rr, Lr), swapping the two parts
```

^ is the XOR operation.

DES Cipher - A 16-round Feistel cipher with block size of 64 bits. DES stands for Data Encryption Standard.

DES (Data Encryption Standard) Cipher Algorithm

This section describes DES (Data Encryption Standard) algorithm - A 16-round Feistel cipher with block size of 64 bits.

DES Cipher - A 16-round Feistel cipher with block size of 64 bits. DES stands for Data Encryption Standard.

DES was developed by IBM in 1974 in response to a federal government public invitation for data encryption algorithms. In 977, DES was published as a federal standard, FIPS PUB 46.

DES algorithm:

```
Input:
   T: 64 bits of clear text
   k1, k2, ..., k16: 16 round keys
   IP: Initial permutation
   FP: Final permutation
   f(): Round function
```

```
Output:
   C: 64 bits of cipher text

Algorithm:
   T' = IP(T), applying initial permutation
   (L0, R0) = T', dividing T' into two 32-bit parts
   (L1, R1) = (R0, L0 ^ f(R0, k1))
   (L2, R2) = (R1, L1 ^ f(R1, k2))
   ......
   C' = (R16, L16), swapping the two parts
   C = FP(C'), applying final permutation
```

where ^ is the XOR operation.

The round function f(R,k) is defined as:

```
Input:
   R: 32-bit input data
   k: 48-bit round key
   E: Expansion permutation
   P: Round permutation
   s(): S boxes function

Output
   R' = f(R,k): 32-bit output data

Algorithm
   X = E(R), applying expansion permutation and returning 48-bit data
   X' = X ^ k, XOR with the round key
   X" = s(X'), applying S boxes function and returning 32-bit data
   R' = P(X"), applying the round permutation
```

The S boxes function s(X) is defined as:

```
Input:
   X: 48-bit input data
   S1, S2, ..., S8: 8 S boxes - 4 x 16 tables

Output:
   X' = s(X): 32-bit output data

Algorithm:
   (X1, X2, ..., X8) = X, dividing X into 8 6-bit parts
   X' = (S1(X1), S2(X2), ..., S8(X8))
      where Si(Xi) is the value at row r and column c of S box i with
         r = 2*b1 + b6
         c = 8*b2 + 4*b3 + 2*b3 + b4
         b1, b2, b3, b4, b5, b6 are the 6 bits of the Xi
```

DES cipher algorithm supporting tables:

Initial Permutation - IP:

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8

```
57    49   41    33    25   17     9    1
59    51   43    35    27   19    11    3
61    53   45    37    29   21    13    5
63    55   47    39    31   23    15    7
```

Final Permutation - FP:

```
40     8   48    16    56   24    64   32
39     7   47    15    55   23    63   31
38     6   46    14    54   22    62   30
37     5   45    13    53   21    61   29
36     4   44    12    52   20    60   28
35     3   43    11    51   19    59   27
34     2   42    10    50   18    58   26
33     1   41     9    49   17    57   25
```

Expansion permutation - E:

```
32     1    2     3     4    5
 4     5    6     7     8    9
 8     9   10    11    12   13
12    13   14    15    16   17
16    17   18    19    20   21
20    21   22    23    24   25
24    25   26    27    28   29
28    29   30    31    32    1
```

Round permutation - P:

```
16   7  20  21
29  12  28  17
 1  15  23  26
 5  18  31  10
 2   8  24  14
32  27   3   9
19  13  30   6
22  11   4  25
```

S boxes - S1, S2, ..., S8:

```
                        S1

14  4  13  1   2 15  11  8   3 10   6 12   5  9   0  7
 0 15   7  4  14  2  13  1  10  6  12 11   9  5   3  8
 4  1  14  8  13  6   2 11  15 12   9  7   3 10   5  0
15 12   8  2   4  9   1  7   5 11   3 14  10  0   6 13

                        S2

15  1   8 14   6 11   3  4   9  7   2 13  12  0   5 10
 3 13   4  7  15  2   8 14  12  0   1 10   6  9  11  5
 0 14   7 11  10  4  13  1   5  8  12  6   9  3   2 15
13  8  10  1   3 15   4  2  11  6   7 12   0  5  14  9

                        S3

10  0   9 14   6  3  15  5   1 13  12  7  11  4   2  8
```

```
13  7   0  9   3  4   6 10   2  8   5 14  12 11  15  1
13  6   4  9   8 15   3  0  11  1   2 12   5 10  14  7
 1 10  13  0   6  9   8  7   4 15  14  3  11  5   2 12

                        S4

 7 13  14  3   0  6   9 10   1  2   8  5  11 12   4 15
13  8  11  5   6 15   0  3   4  7   2 12   1 10  14  9
10  6   9  0  12 11   7 13  15  1   3 14   5  2   8  4
 3 15   0  6  10  1  13  8   9  4   5 11  12  7   2 14

                        S5

 2 12   4  1   7 10  11  6   8  5   3 15  13  0  14  9
14 11   2 12   4  7  13  1   5  0  15 10   3  9   8  6
 4  2   1 11  10 13   7  8  15  9  12  5   6  3   0 14
11  8  12  7   1 14   2 13   6 15   0  9  10  4   5  3

                        S6

12  1  10 15   9  2   6  8   0 13   3  4  14  7   5 11
10 15   4  2   7 12   9  5   6  1  13 14   0 11   3  8
 9 14  15  5   2  8  12  3   7  0   4 10   1 13  11  6
 4  3   2 12   9  5  15 10  11 14   1  7   6  0   8 13

                        S7

 4 11   2 14  15  0   8 13   3 12   9  7   5 10   6  1
13  0  11  7   4  9   1 10  14  3   5 12   2 15   8  6
 1  4  11 13  12  3   7 14  10 15   6  8   0  5   9  2
 6 11  13  8   1  4  10  7   9  5   0 15  14  2   3 12

                        S8

13  2   8  4   6 15  11  1  10  9   3 14   5  0  12  7
 1 15  13  8  10  3   7  4  12  5   6 11   0 14   9  2
 7 11   4  1   9 12  14  2   0  6  10 13  15  3   5  8
 2  1  14  7   4 10   8 13  15 12   9  0   3  5   6 11
```

DES Key Schedule (Round Keys Generation) Algorithm

This section describes DES (Data Encryption Standard) algorithm - A 16-round Feistel cipher with block size of 64 bits.

Key schedule algorithm:

```
Input:
    K: 64-bit key
    PC1: Permuted choice 1
    PC2: Permuted choice 2
    r1, r2, ..., r16: left shifts (rotations)

Output:
    k1, k2, ..., k16: 16 48-bit round keys

Algorithm:
    K' = PC1(K), applying permuted choice 1 and returning 56 bits
    (C0, D0) = K', dividing K' into two 28-bit parts
    (C1, D1) = (r1(C0), r1(D0)), shifting to the left
    k1 = PC2(C1,D1), applying permuted choice 2 and returning 48 bits
    (C2, D2) = (r2(C1), r2(D1)), shifting to the left
    k2 = PC2(C2,D2), applying permuted choice 2 and returning 48 bits
    ......
    k16 = PC2(C16,D16)
```

DES key schedule supporting tables:

Permuted Choice 1 - PC1:

```
57   49    41   33    25    17     9
 1   58    50   42    34    26    18
10    2    59   51    43    35    27
19   11     3   60    52    44    36
63   55    47   39    31    23    15
 7   62    54   46    38    30    22
14    6    61   53    45    37    29
21   13     5   28    20    12     4
```

Permuted Choice 2 - PC2:

```
14    17   11    24     1    5
 3    28   15     6    21   10
23    19   12     4    26    8
16     7   27    20    13    2
41    52   31    37    47   55
30    40   51    45    33   48
44    49   39    56    34   53
46    42   50    36    29   32
```

Left shifts (number of bits to rotate) - r1, r2, ..., r16:

```
 r1  r2  r3  r4  r5  r6  r7  r8  r9 r10 r11 r12 r13 r14 r15 r16
  1   1   2   2   2   2   2   2   1   2   2   2   2   2   2   1
```

DES Decryption Algorithm

This section describes DES decryption algorithm - identical to the encryption algorithm step by step in the same order, only with the subkeys applied in the reverse order.

The decryption algorithm of a block cipher should be identical to encryption algorithm step by step in reverse order. But for DES cipher, the encryption algorithm is so well designed, that the decryption algorithm is identical to the encryption algorithm step by step in the same order, only with the subkeys applied in the reverse order.

DES decryption algorithm:

```
Input:
   CC: 64 bits of cipher text
   k16, k15, ..., k1: 16 round keys
   IP: Initial permutation
   FP: Final permutation
   f(): Round function

Output:
   TT: 64 bits of clear text

Algorithm:
   CC' = IP(CC), applying initial permutation
   (LL0, RR0) = CC', dividing CC' into two 32-bit parts
   (LL1, RR1) = (RR0, LL0 ^ f(RR0, k16))
   (LL2, RR2) = (RR1, LL1 ^ f(RR1, k15))
   ......
   TT' = (RR16, LL16), swapping the two parts
   TT = FP(TT'), applying final permutation
```

Here is how to approve the decryption algorithm:

```
Let:
   T: 64 bits of clear text
   C: 64 bits of cipher text encrypted from T
   CC: 64 bits of cipher text
   TT: 64 bits of clear text decrypted from CC

If:
   CC = C

Then:
   TT = T

Prove:
   CC' = IP(CC)               First step of decryption
      = IP(C)                 Assumption of CC = C
      = IP(FP(C'))            Last step of encryption
      = C'                    IP is the inverse permutation of FP

   (LL0, RR0) = CC'           Initializing step in decryption
      = C'                    CC' = C'
      = (R16, L16)            Swapping step in encryption

   (LL1, RR1) = (RR0, LL0 ^ f(RR0, k16))
                              First round of decryption
      = (L16, R16 ^ f(L16, k16))
                              Previous result
      = (R15, (L15 ^ f(R15,k16)) ^ f(R15, k16))
                              (L16, R16) = (R15, L15 ^ f(R15, k16))
      = (R15, L15)           ^ reverse itself

   ......

   (LL16, RR16) = (R0, L0)

   TT' = (RR16, LL16)         Swapping in decryption
      = (L0, R0)              Previous result
      = T'                    Initializing step in encryption

   TT = FP(TT')               Last step in decryption
      = FP(T')                Previous result
      = FP(IP(T))             First step in encryption
      = T                     FP is the inverse permutation of IP
```

DES Algorithm - Illustrated with Java Programs

This chapter provides tutorial examples and notes about DES algorithm illustrated with Java programs. Topics include Java illustration program of DES key schedule algorithm and DES encryption algorithm.

DESSubkeysTest.java - DES Key Schedule Algorithm Illustration

This section provides a tutorial Java program, DESSubkeysTest.java, to illustrate how DES key schedule algorithm works.

As an illustration to the DES key schedule algorithm described in the previous chapter, I wrote the following Java program, DESKSubkeysTest.java:

```
/* DESSubkeysTest.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
class DESSubkeysTest {
   public static void main(String[] a) {
      try {
         byte[] theKey = getTestKey();
         byte[][] subKeys = getSubkeys(theKey);
         boolean ok = validateSubkeys(subKeys);
         System.out.println("DES subkeys test result: "+ok);
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
   static final int[] PC1 = {
      57, 49, 41, 33, 25, 17,  9,
       1, 58, 50, 42, 34, 26, 18,
      10,  2, 59, 51, 43, 35, 27,
      19, 11,  3, 60, 52, 44, 36,
      63, 55, 47, 39, 31, 23, 15,
       7, 62, 54, 46, 38, 30, 22,
      14,  6, 61, 53, 45, 37, 29,
      21, 13,  5, 28, 20, 12,  4
   };
   static final int[] PC2 = {
      14, 17, 11, 24,  1,  5,
       3, 28, 15,  6, 21, 10,
      23, 19, 12,  4, 26,  8,
      16,  7, 27, 20, 13,  2,
      41, 52, 31, 37, 47, 55,
      30, 40, 51, 45, 33, 48,
      44, 49, 39, 56, 34, 53,
      46, 42, 50, 36, 29, 32
   };
   static final int[] SHIFTS = {
      1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1
   };
   private static byte[][] getSubkeys(byte[] theKey)
      throws Exception {
      printBytes(theKey,"Input key");
      int activeKeySize = PC1.length;
      int numOfSubKeys = SHIFTS.length;
      byte[] activeKey = selectBits(theKey,PC1);
      printBytes(activeKey,"After permuted choice 1 - Active key");
      int halfKeySize = activeKeySize/2;
      byte[] c = selectBits(activeKey,0,halfKeySize);
      byte[] d = selectBits(activeKey,halfKeySize,halfKeySize);
      byte[][] subKeys = new byte[numOfSubKeys][];
      for (int k=0; k<numOfSubKeys; k++) {
         c = rotateLeft(c,halfKeySize,SHIFTS[k]);
         d = rotateLeft(d,halfKeySize,SHIFTS[k]);
         byte[] cd = concatenateBits(c,halfKeySize,d,halfKeySize);
         printBytes(cd,"Subkey #"+(k+1)+" after shifting");
         subKeys[k] = selectBits(cd,PC2);
         printBytes(subKeys[k],"Subkey #"+(k+1)
            +" after permuted choice 2");
      }
      return subKeys;
   }
   private static byte[] rotateLeft(byte[] in, int len, int step) {
      int numOfBytes = (len-1)/8 + 1;
      byte[] out = new byte[numOfBytes];
      for (int i=0; i<len; i++) {
         int val = getBit(in,(i+step)%len);
         setBit(out,i,val);
      }
      return out;
   }
   private static byte[] concatenateBits(byte[] a, int aLen, byte[] b,
      int bLen) {
      int numOfBytes = (aLen+bLen-1)/8 + 1;
      byte[] out = new byte[numOfBytes];
      int j = 0;
      for (int i=0; i<aLen; i++) {
         int val = getBit(a,i);
         setBit(out,j,val);
         j++;
      }
      for (int i=0; i<bLen; i++) {
         int val = getBit(b,i);
         setBit(out,j,val);
         j++;
      }
      return out;
   }
   private static byte[] selectBits(byte[] in, int pos, int len) {
      int numOfBytes = (len-1)/8 + 1;
      byte[] out = new byte[numOfBytes];
      for (int i=0; i<len; i++) {
         int val = getBit(in,pos+i);
         setBit(out,i,val);
      }
      return out;
   }
   private static byte[] selectBits(byte[] in, int[] map) {
      int numOfBytes = (map.length-1)/8 + 1;
      byte[] out = new byte[numOfBytes];
      for (int i=0; i<map.length; i++) {
         int val = getBit(in,map[i]-1);
         setBit(out,i,val);
//       System.out.println("i="+i+", pos="+(map[i]-1)+", val="+val);
      }
      return out;
   }
   private static int getBit(byte[] data, int pos) {
      int posByte = pos/8;
      int posBit = pos%8;
      byte valByte = data[posByte];
      int valInt = valByte>>(8-(posBit+1)) & 0x0001;
      return valInt;
   }
   private static void setBit(byte[] data, int pos, int val) {
      int posByte = pos/8;
      int posBit = pos%8;
      byte oldByte = data[posByte];
      oldByte = (byte) (((0xFF7F>>posBit) & oldByte) & 0x00FF);
      byte newByte = (byte) ((val<<(8-(posBit+1))) | oldByte);
      data[posByte] = newByte;
   }
   private static void printBytes(byte[] data, String name) {
      System.out.println("");
      System.out.println(name+":");
      for (int i=0; i<data.length; i++) {
         System.out.print(byteToBits(data[i])+" ");
      }
      System.out.println();
   }
   private static String byteToBits(byte b) {
      StringBuffer buf = new StringBuffer();
      for (int i=0; i<8; i++)
         buf.append((int)(b>>(8-(i+1)) & 0x0001));
      return buf.toString();
   }
   private static byte[] getTestKey() {
      String strKey = " 00010011 00110100 01010111 01111001"
                     +" 10011011 10111100 11011111 11110001";
      byte[] theKey = new byte[8];
      for (int i=0; i<8; i++) {
         String strByte = strKey.substring(9*i+1,9*i+1+8);
         theKey[i] = (byte) Integer.parseInt(strByte,2);
      }
      return theKey;
   }
   private static boolean validateSubkeys(byte[][] subKeys) {
      boolean ok = true;
      String[] strKeys = {
         " 00011011 00000010 11101111 11111100 01110000 01110010",//1
         " 01111001 10101110 11011001 11011011 11001001 11100101",//2
         " 01010101 11111100 10001010 01000010 11001111 10011001",//3
         " 01110010 10101101 11010110 11011011 00110101 00011101",//4
         " 01111100 11101100 00000111 11101011 01010011 10101000",//5
         " 01100011 10100101 00111110 01010000 01111011 00101111",//6
         " 11101100 10000100 10110111 11110110 00011000 10111100",//7
         " 11110111 10001010 00111010 11000001 00111011 11111011",//8
         " 11100000 11011011 11101011 11101101 11100111 10000001",//9
         " 10110001 11110011 01000111 10111010 01000110 01001111",//0
         " 00100001 01011111 11010011 11011110 11010011 10000110",//1
         " 01110101 01110001 11110101 10010100 01100111 11101001",//2
         " 10010111 11000101 11010001 11111010 10111010 01000001",//3
         " 01011111 01000011 10110111 11110010 11100111 00111010",//4
         " 10111111 10010001 10001101 00111101 00111111 00001010",//5
         " 11001011 00111101 10001011 00001110 00010111 11110101"};
      for (int k=0; k<16; k++) {
         for (int i=0; i<6; i++) {
            String strByte = strKeys[k].substring(9*i+1,9*i+1+8);
            byte keyByte = (byte) Integer.parseInt(strByte,2);
            if (keyByte!=subKeys[k][i]) ok = false;
         }
      }
      return ok;
   }
}
```

In this program, the input key is hard coded to the same value used by J. Orlin Grabbe in "The DES Algorithm Illustrated". The subkeys generated by the algorithm are validated with those mentioned by J. Orlin Grabbe.

See the next section for the execution output of this tutorial program.

DES Key Schedule Algorithm Illustration Program Output

This section provides the execution output of the tutorial Java program, DESSubkeysTest.java, to illustrate how DES key schedule algorithm works.

If you run the DES key schedule algorithm illustration program described in the previous section with JDK 1.4.1, you will get:

```

Input key:
00010011 00110100 01010111 01111001 10011011 10111100 11011111 111...

After permuted choice 1 - Active key:
11110000 11001100 10101010 11110101 01010110 01100111 10001111

Subkey #1 after shifting:
11100001 10011001 01010101 11111010 10101100 11001111 00011110

Subkey #1 after permuted choice 2:
00011011 00000010 11101111 11111100 01110000 01110010

Subkey #2 after shifting:
11000011 00110010 10101011 11110101 01011001 10011110 00111101

Subkey #2 after permuted choice 2:
01111001 10101110 11011001 11011011 11001001 11100101

Subkey #3 after shifting:
00001100 11001010 10101111 11110101 01100110 01111000 11110101

Subkey #3 after permuted choice 2:
01010101 11111100 10001010 01000010 11001111 10011001
```

```
Subkey #4 after shifting:
00110011 00101010 10111111 11000101 10011001 11100011 11010101

Subkey #4 after permuted choice 2:
01110010 10101101 11010110 11011011 00110101 00011101

Subkey #5 after shifting:
11001100 10101010 11111111 00000110 01100111 10001111 01010101

Subkey #5 after permuted choice 2:
01111100 11101100 00000111 11101011 01010011 10101000

Subkey #6 after shifting:
00110010 10101011 11111100 00111001 10011110 00111101 01010101

Subkey #6 after permuted choice 2:
01100011 10100101 00111110 01010000 01111011 00101111

Subkey #7 after shifting:
11001010 10101111 11110000 11000110 01111000 11110101 01010110

Subkey #7 after permuted choice 2:
11101100 10000100 10110111 11110110 00011000 10111100

Subkey #8 after shifting:
00101010 10111111 11000011 00111001 11100011 11010101 01011001

Subkey #8 after permuted choice 2:
11110111 10001010 00111010 11000001 00111011 11111011

Subkey #9 after shifting:
01010101 01111111 10000110 01100011 11000111 10101010 10110011

Subkey #9 after permuted choice 2:
11100000 11011011 11101011 11101101 11100111 10000001

Subkey #10 after shifting:
01010101 11111110 00011001 10011111 00011110 10101010 11001100

Subkey #10 after permuted choice 2:
10110001 11110011 01000111 10111010 01000110 01001111

Subkey #11 after shifting:
01010111 11111000 01100110 01011100 01111010 10101011 00110011

Subkey #11 after permuted choice 2:
00100001 01011111 11010011 11011110 11010011 10000110

Subkey #12 after shifting:
01011111 11100001 10011001 01010001 11101010 10101100 11001111

Subkey #12 after permuted choice 2:
01110101 01110001 11110101 10010100 01100111 11101001

Subkey #13 after shifting:
01111111 10000110 01100101 01010111 10101010 10110011 00111100

Subkey #13 after permuted choice 2:
10010111 11000101 11010001 11111010 10111010 01000001

Subkey #14 after shifting:
```

```
11111110 00011001 10010101 01011110 10101010 11001100 11110001

Subkey #14 after permuted choice 2:
01011111 01000011 10110111 11110010 11100111 00111010

Subkey #15 after shifting:
11111000 01100110 01010101 01111010 10101011 00110011 11000111

Subkey #15 after permuted choice 2:
10111111 10010001 10001101 00111101 00111111 00001010

Subkey #16 after shifting:
11110000 11001100 10101010 11110101 01010110 01100111 10001111

Subkey #16 after permuted choice 2:
11001011 00111101 10001011 00001110 00010111 11110101
DES subkeys test result: true
```

DESCipherTest.java - DES Cipher Algorithm Illustration

This section provides a tutorial Java program, DESCipherTest.java, to illustrate how DES cipher algorithm works.

As an illustration to the DES cipher algorithm described in the previous chapter, I wrote the following Java program, DESKCipher.java:

```
/* DESCipherTest.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
class DESCipherTest {
   public static void main(String[] a) {
      try {
         byte[][] subKeys = getTestSubkeys();
         byte[] theMsg = getTestMsg();
         byte[] theCph = cipherBlock(theMsg,subKeys);
         boolean ok = validateCipher(theCph);
         System.out.println("DES cipher test result: "+ok);
      } catch (Exception e) {
         e.printStackTrace();
         return;
      }
   }
   static final int[] IP = {
      58, 50, 42, 34, 26, 18, 10, 2,
      60, 52, 44, 36, 28, 20, 12, 4,
      62, 54, 46, 38, 30, 22, 14, 6,
      64, 56, 48, 40, 32, 24, 16, 8,
      57, 49, 41, 33, 25, 17,  9, 1,
      59, 51, 43, 35, 27, 19, 11, 3,
      61, 53, 45, 37, 29, 21, 13, 5,
      63, 55, 47, 39, 31, 23, 15, 7
   };
   static final int[] E = {
      32,  1,  2,  3,  4,  5,
       4,  5,  6,  7,  8,  9,
       8,  9, 10, 11, 12, 13,
      12, 13, 14, 15, 16, 17,
      16, 17, 18, 19, 20, 21,
      20, 21, 22, 23, 24, 25,
      24, 25, 26, 27, 28, 29,
      28, 29, 30, 31, 32,  1
   };
   static final int[] P = {
      16,  7, 20, 21,
      29, 12, 28, 17,
       1, 15, 23, 26,
       5, 18, 31, 10,
       2,  8, 24, 14,
      32, 27,  3,  9,
      19, 13, 30,  6,
      22, 11,  4, 25
   };
   static final int[] FP = {
      40, 8, 48, 16, 56, 24, 64, 32,
      39, 7, 47, 15, 55, 23, 63, 31,
      38, 6, 46, 14, 54, 22, 62, 30,
      37, 5, 45, 13, 53, 21, 61, 29,
      36, 4, 44, 12, 52, 20, 60, 28,
      35, 3, 43, 11, 51, 19, 59, 27,
      34, 2, 42, 10, 50, 18, 58, 26,
      33, 1, 41,  9, 49, 17, 57, 25
   };
   private static byte[] cipherBlock(byte[] theMsg, byte[][] subKeys)
      throws Exception {
      if (theMsg.length<8)
         throw new Exception("Message is less than 64 bits.");
      printBytes(theMsg,"Input message");
      theMsg = selectBits(theMsg,IP); // Initial Permutation
      printBytes(theMsg,"After initial permutation");
      int blockSize = IP.length;
      byte[] l = selectBits(theMsg,0,blockSize/2);
      byte[] r = selectBits(theMsg,blockSize/2,blockSize/2);
      int numOfSubKeys = subKeys.length;
      for (int k=0; k<numOfSubKeys; k++) {
               byte[] rBackup = r;
         r = selectBits(r,E); // Expansion
         printBytes(r,"R: After E expansion");
         r = doXORBytes(r,subKeys[k]); // XOR with the sub key
         printBytes(r,"R: After XOR with the subkey");
         r = substitution6x4(r); // Substitution
         printBytes(r,"R: After S boxes");
         r = selectBits(r,P); // Permutation
         printBytes(r,"R: After P permutation");
         r = doXORBytes(l,r); // XOR with the previous left half
         printBytes(r,"Right half at round #"+(k+1));
         l = rBackup; // Taking the previous right half
      }
      byte[] lr = concatenateBits(r,blockSize/2,l,blockSize/2);
      printBytes(lr,"After 16 rounds");
      lr = selectBits(lr,FP); // Inverse Permutation
      printBytes(lr,"After final permutation");
      return lr;
   }
   private static byte[] doXORBytes(byte[] a, byte[] b) {
      byte[] out = new byte[a.length];
      for (int i=0; i<a.length; i++) {
         out[i] = (byte) (a[i] ^ b[i]);
      }
      return out;
   }
   static final int[] S = {
14,  4, 13,  1,  2, 15, 11,  8,  3, 10,  6, 12,  5,  9,  0,  7, // S1
 0, 15,  7,  4, 14,  2, 13,  1, 10,  6, 12, 11,  9,  5,  3,  8,
 4,  1, 14,  8, 13,  6,  2, 11, 15, 12,  9,  7,  3, 10,  5,  0,
15, 12,  8,  2,  4,  9,  1,  7,  5, 11,  3, 14, 10,  0,  6, 13,
15,  1,  8, 14,  6, 11,  3,  4,  9,  7,  2, 13, 12,  0,  5, 10, // S2
 3, 13,  4,  7, 15,  2,  8, 14, 12,  0,  1, 10,  6,  9, 11,  5,
 0, 14,  7, 11, 10,  4, 13,  1,  5,  8, 12,  6,  9,  3,  2, 15,
13,  8, 10,  1,  3, 15,  4,  2, 11,  6,  7, 12,  0,  5, 14,  9,
10,  0,  9, 14,  6,  3, 15,  5,  1, 13, 12,  7, 11,  4,  2,  8, // S3
13,  7,  0,  9,  3,  4,  6, 10,  2,  8,  5, 14, 12, 11, 15,  1,
13,  6,  4,  9,  8, 15,  3,  0, 11,  1,  2, 12,  5, 10, 14,  7,
 1, 10, 13,  0,  6,  9,  8,  7,  4, 15, 14,  3, 11,  5,  2, 12,
 7, 13, 14,  3,  0,  6,  9, 10,  1,  2,  8,  5, 11, 12,  4, 15, // S4
13,  8, 11,  5,  6, 15,  0,  3,  4,  7,  2, 12,  1, 10, 14,  9,
10,  6,  9,  0, 12, 11,  7, 13, 15,  1,  3, 14,  5,  2,  8,  4,
 3, 15,  0,  6, 10,  1, 13,  8,  9,  4,  5, 11, 12,  7,  2, 14,
 2, 12,  4,  1,  7, 10, 11,  6,  8,  5,  3, 15, 13,  0, 14,  9, // S5
14, 11,  2, 12,  4,  7, 13,  1,  5,  0, 15, 10,  3,  9,  8,  6,
 4,  2,  1, 11, 10, 13,  7,  8, 15,  9, 12,  5,  6,  3,  0, 14,
11,  8, 12,  7,  1, 14,  2, 13,  6, 15,  0,  9, 10,  4,  5,  3,
12,  1, 10, 15,  9,  2,  6,  8,  0, 13,  3,  4, 14,  7,  5, 11, // S6
10, 15,  4,  2,  7, 12,  9,  5,  6,  1, 13, 14,  0, 11,  3,  8,
 9, 14, 15,  5,  2,  8, 12,  3,  7,  0,  4, 10,  1, 13, 11,  6,
 4,  3,  2, 12,  9,  5, 15, 10, 11, 14,  1,  7,  6,  0,  8, 13,
 4, 11,  2, 14, 15,  0,  8, 13,  3, 12,  9,  7,  5, 10,  6,  1, // S7
13,  0, 11,  7,  4,  9,  1, 10, 14,  3,  5, 12,  2, 15,  8,  6,
 1,  4, 11, 13, 12,  3,  7, 14, 10, 15,  6,  8,  0,  5,  9,  2,
 6, 11, 13,  8,  1,  4, 10,  7,  9,  5,  0, 15, 14,  2,  3, 12,
13,  2,  8,  4,  6, 15, 11,  1, 10,  9,  3, 14,  5,  0, 12,  7, // S8
 1, 15, 13,  8, 10,  3,  7,  4, 12,  5,  6, 11,  0, 14,  9,  2,
 7, 11,  4,  1,  9, 12, 14,  2,  0,  6, 10, 13, 15,  3,  5,  8,
 2,  1, 14,  7,  4, 10,  8, 13, 15, 12,  9,  0,  3,  5,  6, 11
   };
   private static byte[] substitution6x4(byte[] in) {
      in = splitBytes(in,6); // Splitting byte[] into 6-bit blocks
//    printBytes(in,"R: After splitting");
      byte[] out = new byte[in.length/2];
      int lhByte = 0;
      for (int b=0; b<in.length; b++) { // Should be sub-blocks
         byte valByte = in[b];
         int r = 2*(valByte>>7&0x0001)+(valByte>>2&0x0001); // 1 and 6
         int c = valByte>>3&0x000F; // Middle 4 bits
         int hByte = S[64*b+16*r+c]; // 4 bits (half byte) output
         if (b%2==0) lhByte = hByte; // Left half byte
         else out[b/2] = (byte) (16*lhByte + hByte);
      }
      return out;
   }
   private static byte[] splitBytes(byte[] in, int len) {
      int numOfBytes = (8*in.length-1)/len + 1;
      byte[] out = new byte[numOfBytes];
      for (int i=0; i<numOfBytes; i++) {
         for (int j=0; j<len; j++) {
            int val = getBit(in, len*i+j);
            setBit(out,8*i+j,val);
         }
      }
      return out;
   }
   private static byte[] concatenateBits(byte[] a, int aLen, byte[] b,
      int bLen) {
```

```
      int numOfBytes = (aLen+bLen-1)/8 + 1;
      byte[] out = new byte[numOfBytes];
      int j = 0;
      for (int i=0; i<aLen; i++) {
         int val = getBit(a,i);
         setBit(out,j,val);
         j++;
      }
      for (int i=0; i<bLen; i++) {
         int val = getBit(b,i);
         setBit(out,j,val);
         j++;
      }
      return out;
   }
   private static byte[] selectBits(byte[] in, int pos, int len) {
      int numOfBytes = (len-1)/8 + 1;
      byte[] out = new byte[numOfBytes];
      for (int i=0; i<len; i++) {
         int val = getBit(in,pos+i);
         setBit(out,i,val);
      }
      return out;
   }
   private static byte[] selectBits(byte[] in, int[] map) {
      int numOfBytes = (map.length-1)/8 + 1;
      byte[] out = new byte[numOfBytes];
      for (int i=0; i<map.length; i++) {
         int val = getBit(in,map[i]-1);
         setBit(out,i,val);
//       System.out.println("i="+i+", pos="+(map[i]-1)+", val="+val);
      }
      return out;
   }
   private static int getBit(byte[] data, int pos) {
      int posByte = pos/8;
      int posBit = pos%8;
      byte valByte = data[posByte];
      int valInt = valByte>>(8-(posBit+1)) & 0x0001;
      return valInt;
   }
   private static void setBit(byte[] data, int pos, int val) {
      int posByte = pos/8;
      int posBit = pos%8;
      byte oldByte = data[posByte];
      oldByte = (byte) (((0xFF7F>>posBit) & oldByte) & 0x00FF);
      byte newByte = (byte) ((val<<(8-(posBit+1))) | oldByte);
      data[posByte] = newByte;
   }
   private static void printBytes(byte[] data, String name) {
      System.out.println("");
      System.out.println(name+":");
      for (int i=0; i<data.length; i++) {
         System.out.print(byteToBits(data[i])+" ");
      }
      System.out.println();
   }
   private static String byteToBits(byte b) {
      StringBuffer buf = new StringBuffer();
      for (int i=0; i<8; i++)
         buf.append((int)(b>>(8-(i+1)) & 0x0001));
      return buf.toString();
   }
   private static byte[][] getTestSubkeys() {
      String[] strKeys = {
         " 00011011 00000010 11101111 11111100 01110000 01110010",//1
         " 01111001 10101110 11011001 11011011 11001001 11100101",//2
         " 01010101 11111100 10001010 01000010 11001111 10011001",//3
         " 01110010 10101101 11010110 11011011 00110101 00011101",//4
         " 01111100 11101100 00000111 11101011 01010011 10101000",//5
         " 01100011 10100101 00111110 01010000 01111011 00101111",//6
         " 11101100 10000100 10110111 11110110 00011000 10111100",//7
         " 11110111 10001010 00111010 11000001 00111011 11111011",//8
         " 11100000 11011011 11101011 11101101 11100111 10000001",//9
         " 10110001 11110011 01000111 10111010 01000110 01001111",//0
         " 00100001 01011111 11010011 11011110 11010011 10000110",//1
         " 01110101 01110001 11110101 10010100 01100111 11101001",//2
         " 10010111 11000101 11010001 11111010 10111010 01000001",//3
         " 01011111 01000011 10110111 11110010 11100111 00111010",//4
         " 10111111 10010001 10001101 00111101 00111111 00001010",//5
         " 11001011 00111101 10001011 00001110 00010111 11110101"};
      byte[][] subKeys = new byte[16][];
      for (int k=0; k<16; k++) {
         byte[] theKey = new byte[6];
         for (int i=0; i<6; i++) {
            String strByte = strKeys[k].substring(9*i+1,9*i+1+8);
            theKey[i] = (byte) Integer.parseInt(strByte,2);
         }
         subKeys[k] = theKey;
      }
      return subKeys;
   }
   private static byte[] getTestMsg() {
      String strMsg = " 00000001 00100011 01000101 01100111"
                     +" 10001001 10101011 11001101 11101111";
      byte[] theMsg = new byte[8];
      for (int i=0; i<8; i++) {
         String strByte = strMsg.substring(9*i+1,9*i+1+8);
         theMsg[i] = (byte) Integer.parseInt(strByte,2);
      }
      return theMsg;
   }
   private static boolean validateCipher(byte[] cipher ) {
      String strCipher = " 10000101 11101000 00010011 01010100"
                        +" 00001111 00001010 10110100 00000101";
      boolean ok = true;
      for (int i=0; i<8; i++) {
         String strByte = strCipher.substring(9*i+1,9*i+1+8);
         byte cipherByte = (byte) Integer.parseInt(strByte,2);
         if (cipherByte!=cipher[i]) ok = false;
      }
      return ok;
   }
}
```

In this program, the input clear text block and the subkeys are hard coded with the values used by J. Orlin Grabbe in "The DES Algorithm Illustrated". The cipher text block generated by the algorithm is validated with the result mentioned by J. Orlin Grabbe.

See the next section for the execution output of this tutorial program.

DES Cipher Algorithm Illustration Program Output

This section provides the execution output of the tutorial Java program, DESCipherTest.java, to illustrate how DES cipher algorithm works.

If you run the DES encryption algorithm illustration program described in the previous section with JDK 1.4.1, you will get:

```
Input message:
00000001 00100011 01000101 01100111 10001001 10101011 11001101 111...

After initial permutation:
11001100 00000000 11001100 11111111 11110000 10101010 11110000 101...

R: After E expansion:
01111010 00010101 01010101 01111010 00010101 01010101

R: After XOR with the subkey:
01100001 00010111 10111010 10000110 01100101 00100111

R: After S boxes:
01011100 10000010 10110101 10010111

R: After P permutation:
00100011 01001010 10101001 10111011

Right half at round #1:
11101111 01001010 01100101 01000100

R: After E expansion:
01110101 11101010 01010100 00110000 10101010 00001001

R: After XOR with the subkey:
00001100 01000100 10001101 11101011 01100011 11101100

R: After S boxes:
11111000 11010000 00111010 10101110

R: After P permutation:
00111100 10101011 10000111 10100011

Right half at round #2:
11001100 00000001 01110111 00001001

R: After E expansion:
11100101 10000000 00000010 10111010 11101000 01010011

R: After XOR with the subkey:
10110000 01111100 10001000 11111000 00100111 11001010

R: After S boxes:
00100111 00010000 11100001 01101111

R: After P permutation:
01001101 00010110 01101110 10110000

Right half at round #3:
```

```
10100010 01011100 00001011 11110100

R: After E expansion:
01010000 01000010 11111000 00000101 01111111 10101001

R: After XOR with the subkey:
00100010 11101111 00101110 11011110 01001010 10110100

R: After S boxes:
00100001 11101101 10011111 00111010

R: After P permutation:
10111011 00100011 01110111 01001100

Right half at round #4:
01110111 00100010 00000000 01000101

R: After E expansion:
10111010 11101001 00000100 00000000 00000010 00001010

R: After XOR with the subkey:
11000110 00000101 00000011 11101011 01010001 10100010

R: After S boxes:
01010000 11001000 00110001 11101011

R: After P permutation:
00101000 00010011 10101101 11000011

Right half at round #5:
10001010 01001111 10100110 00110111

R: After E expansion:
11000101 01000010 01011111 11010000 11000001 10101111

R: After XOR with the subkey:
10100110 11100111 01100001 10000000 10111010 10000000

R: After S boxes:
01000001 11110011 01001100 00111101

R: After P permutation:
10011110 01000101 11001101 00101100

Right half at round #6:
11101001 01100111 11001101 01101001

R: After E expansion:
11110101 00101011 00001111 11100101 10101011 01010011

R: After XOR with the subkey:
00011001 10101111 10111000 00010011 10110011 11101111

R: After S boxes:
00010000 01110101 01000000 10101101

R: After P permutation:
10001100 00000101 00011100 00100111

Right half at round #7:
00000110 01001010 10111010 00010000
```

```
R: After E expansion:
00000000 11000010 01010101 01011111 01000000 10100000

R: After XOR with the subkey:
11110111 01001000 01101111 10011110 01111011 01011011

R: After S boxes:
01101100 00011000 01111100 10101110

R: After P permutation:
00111100 00001110 10000110 11111001

Right half at round #8:
11010101 01101001 01001011 10010000

R: After E expansion:
01101010 10101011 01010010 10100101 01111100 10100001

R: After XOR with the subkey:
10001010 01110000 10111001 01001000 10011011 00100000

R: After S boxes:
00010001 00001100 01010111 01110111

R: After P permutation:
00100010 00110110 01111100 01101010

Right half at round #9:
00100100 01111100 11000110 01111010

R: After E expansion:
00010000 10000011 11111001 01100000 11000011 11110100

R: After XOR with the subkey:
10100001 01110000 10111110 11011010 10000101 10111011

R: After S boxes:
11011010 00000100 01010010 01110101

R: After P permutation:
01100010 10111100 10011100 00100010

Right half at round #10:
10110111 11010101 11010111 10110010

R: After E expansion:
01011010 11111110 10101011 11101010 11111101 10100101

R: After XOR with the subkey:
01111011 10100001 01111000 00110100 00101110 00100011

R: After S boxes:
01110011 00000101 11010001 00000001

R: After P permutation:
11100001 00000100 11111010 00000010

Right half at round #11:
11000101 01111000 00111100 01111000

R: After E expansion:
01100000 10101011 11110000 00011111 10000011 11110001
```

```
R: After XOR with the subkey:
00010101 11011010 00000101 10001011 11100100 00011000

R: After S boxes:
01111011 10001011 00100110 00110101

R: After P permutation:
11000010 01101000 11001111 11101010

Right half at round #12:
01110101 10111101 00011000 01011000

R: After E expansion:
00111010 10111101 11111010 10001111 00000010 11110000

R: After XOR with the subkey:
10101101 01111000 00101011 01110101 10111000 10110001

R: After S boxes:
10011010 11010001 10001011 01001111

R: After P permutation:
11011101 10111011 00101001 00100010

Right half at round #13:
00011000 11000011 00010101 01011010

R: After E expansion:
00001111 00010110 00000110 10001010 10101010 11110100

R: After XOR with the subkey:
01010000 01010101 10110001 01111000 01001101 11001110

R: After S boxes:
01100100 01111001 10011010 11110001

R: After P permutation:
10110111 00110001 10001110 01010101

Right half at round #14:
11000010 10001100 10010110 00001101

R: After E expansion:
11100000 01010100 01011001 01001010 11000000 01011011

R: After XOR with the subkey:
01011111 11000101 11010100 01110111 11111111 01010001

R: After S boxes:
10110010 11101000 10001101 00111100

R: After P permutation:
01011011 10000001 00100111 01101110

Right half at round #15:
01000011 01000010 00110010 00110100

R: After E expansion:
00100000 01101010 00000100 00011010 01000001 10101000

R: After XOR with the subkey:
```

```
11101011 01010111 10001111 00010100 01010110 01011101

R: After S boxes:
10100111 10000011 00100100 00101001

R: After P permutation:
11001000 11000000 01001111 10011000

Right half at round #16:
00001010 01001100 11011001 10010101

After 16 rounds:
00001010 01001100 11011001 10010101 01000011 01000010 00110010 00110100

After final permutation:
10000101 11101000 00010011 01010100 00001111 00001010 10110100 00000101
DES cipher test result: true
```

Now we got all elements of the implementation working, and validated, let's put them together to make a generic DES encryption program. See the next chapter.

DES Algorithm Java Implementation

This chapter provides tutorial examples and notes about DES algorithm implementation in Java language. Topics include an example Java implementation of DES encryption and decryption algorithm; test cases of single block cleartext and ciphertext.

Exercise: Improve CipherDES.java to handle multiple input blocks.

CipherDES.java - A Java Implementation of DES

This section provides a tutorial Java program, CipherDES.java - A Java Implementation of DES encryption and decryption algorithm.

Merging the illustration programs from the previous chapter together, I got the following simple Java implementation of the DES algorithm, CipherDES.java:

```
/* CipherDES.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
package herong;
import java.io.*;
public class CipherDES {
   public static void main(String[] a) {
      if (a.length<4) {
         System.out.println("Usage:");
         System.out.println("java DesCipher encrypt/decrypt keyFile"
            +" msgFile cphFile");
         return;
      }
      String mode = a[0];
      String keyFile = a[1];
      String msgFile = a[2];
      String cphFile = a[3];
      try {
         byte[] theKey = readBytes(keyFile);
         byte[][] subKeys = getSubkeys(theKey);
         byte[] theMsg = readBytes(msgFile);
         byte[] theCph = cipher(theMsg,subKeys,mode);
         writeBytes(theCph,cphFile);
         printBytes(theKey,"Key block");
         printBytes(theMsg,"Input block");
         printBytes(theCph,"Output block");
      } catch (Exception e) {
         e.printStackTrace();
         return;
      }
   }
   static final int[] IP = {
      58, 50, 42, 34, 26, 18, 10, 2,
      60, 52, 44, 36, 28, 20, 12, 4,
      62, 54, 46, 38, 30, 22, 14, 6,
      64, 56, 48, 40, 32, 24, 16, 8,
      57, 49, 41, 33, 25, 17,  9, 1,
      59, 51, 43, 35, 27, 19, 11, 3,
      61, 53, 45, 37, 29, 21, 13, 5,
      63, 55, 47, 39, 31, 23, 15, 7
   };
   static final int[] E = {
      32,  1,  2,  3,  4,  5,
       4,  5,  6,  7,  8,  9,
       8,  9, 10, 11, 12, 13,
      12, 13, 14, 15, 16, 17,
      16, 17, 18, 19, 20, 21,
      20, 21, 22, 23, 24, 25,
      24, 25, 26, 27, 28, 29,
      28, 29, 30, 31, 32,  1
   };
   static final int[] P = {
      16,  7, 20, 21,
      29, 12, 28, 17,
       1, 15, 23, 26,
       5, 18, 31, 10,
       2,  8, 24, 14,
      32, 27,  3,  9,
      19, 13, 30,  6,
      22, 11,  4, 25
   };
   static final int[] INVP = {
      40, 8, 48, 16, 56, 24, 64, 32,
      39, 7, 47, 15, 55, 23, 63, 31,
      38, 6, 46, 14, 54, 22, 62, 30,
      37, 5, 45, 13, 53, 21, 61, 29,
      36, 4, 44, 12, 52, 20, 60, 28,
      35, 3, 43, 11, 51, 19, 59, 27,
      34, 2, 42, 10, 50, 18, 58, 26,
      33, 1, 41,  9, 49, 17, 57, 25
   };
   public static byte[] cipher(byte[] theMsg, byte[][] subKeys,
      String mode) throws Exception {
      if (theMsg.length<8)
         throw new Exception("Message is less than 64 bits.");
      theMsg = selectBits(theMsg,IP); // Initial Permutation
      int blockSize = IP.length;
      byte[] l = selectBits(theMsg,0,blockSize/2);
      byte[] r = selectBits(theMsg,blockSize/2,blockSize/2);
      int numOfSubKeys = subKeys.length;
      for (int k=0; k<numOfSubKeys; k++) {
               byte[] rBackup = r;
         r = selectBits(r,E); // Expansion
         if (mode.equalsIgnoreCase("encrypt"))
            r = doXORBytes(r,subKeys[k]); // XOR with the sub key
         else
            r = doXORBytes(r,subKeys[numOfSubKeys-k-1]);
         r = substitution6x4(r); // Substitution
         r = selectBits(r,P); // Permutation
         r = doXORBytes(l,r); // XOR with the previous left half
         l = rBackup; // Taking the previous right half
      }
      byte[] lr = concatenateBits(r,blockSize/2,l,blockSize/2);
      lr = selectBits(lr,INVP); // Inverse Permutation
      return lr;
   }
   private static byte[] doXORBytes(byte[] a, byte[] b) {
      byte[] out = new byte[a.length];
      for (int i=0; i<a.length; i++) {
         out[i] = (byte) (a[i] ^ b[i]);
      }
      return out;
   }
   static final int[] S = {
14,  4, 13,  1,  2, 15, 11,  8,  3, 10,  6, 12,  5,  9,  0,  7, // S1
 0, 15,  7,  4, 14,  2, 13,  1, 10,  6, 12, 11,  9,  5,  3,  8,
 4,  1, 14,  8, 13,  6,  2, 11, 15, 12,  9,  7,  3, 10,  5,  0,
15, 12,  8,  2,  4,  9,  1,  7,  5, 11,  3, 14, 10,  0,  6, 13,
15,  1,  8, 14,  6, 11,  3,  4,  9,  7,  2, 13, 12,  0,  5, 10, // S2
 3, 13,  4,  7, 15,  2,  8, 14, 12,  0,  1, 10,  6,  9, 11,  5,
 0, 14,  7, 11, 10,  4, 13,  1,  5,  8, 12,  6,  9,  3,  2, 15,
13,  8, 10,  1,  3, 15,  4,  2, 11,  6,  7, 12,  0,  5, 14,  9,
10,  0,  9, 14,  6,  3, 15,  5,  1, 13, 12,  7, 11,  4,  2,  8, // S3
13,  7,  0,  9,  3,  4,  6, 10,  2,  8,  5, 14, 12, 11, 15,  1,
13,  6,  4,  9,  8, 15,  3,  0, 11,  1,  2, 12,  5, 10, 14,  7,
 1, 10, 13,  0,  6,  9,  8,  7,  4, 15, 14,  3, 11,  5,  2, 12,
 7, 13, 14,  3,  0,  6,  9, 10,  1,  2,  8,  5, 11, 12,  4, 15, // S4
13,  8, 11,  5,  6, 15,  0,  3,  4,  7,  2, 12,  1, 10, 14,  9,
10,  6,  9,  0, 12, 11,  7, 13, 15,  1,  3, 14,  5,  2,  8,  4,
 3, 15,  0,  6, 10,  1, 13,  8,  9,  4,  5, 11, 12,  7,  2, 14,
 2, 12,  4,  1,  7, 10, 11,  6,  8,  5,  3, 15, 13,  0, 14,  9, // S5
14, 11,  2, 12,  4,  7, 13,  1,  5,  0, 15, 10,  3,  9,  8,  6,
 4,  2,  1, 11, 10, 13,  7,  8, 15,  9, 12,  5,  6,  3,  0, 14,
11,  8, 12,  7,  1, 14,  2, 13,  6, 15,  0,  9, 10,  4,  5,  3,
12,  1, 10, 15,  9,  2,  6,  8,  0, 13,  3,  4, 14,  7,  5, 11, // S6
10, 15,  4,  2,  7, 12,  9,  5,  6,  1, 13, 14,  0, 11,  3,  8,
 9, 14, 15,  5,  2,  8, 12,  3,  7,  0,  4, 10,  1, 13, 11,  6,
 4,  3,  2, 12,  9,  5, 15, 10, 11, 14,  1,  7,  6,  0,  8, 13,
 4, 11,  2, 14, 15,  0,  8, 13,  3, 12,  9,  7,  5, 10,  6,  1, // S7
13,  0, 11,  7,  4,  9,  1, 10, 14,  3,  5, 12,  2, 15,  8,  6,
 1,  4, 11, 13, 12,  3,  7, 14, 10, 15,  6,  8,  0,  5,  9,  2,
 6, 11, 13,  8,  1,  4, 10,  7,  9,  5,  0, 15, 14,  2,  3, 12,
13,  2,  8,  4,  6, 15, 11,  1, 10,  9,  3, 14,  5,  0, 12,  7, // S8
 1, 15, 13,  8, 10,  3,  7,  4, 12,  5,  6, 11,  0, 14,  9,  2,
 7, 11,  4,  1,  9, 12, 14,  2,  0,  6, 10, 13, 15,  3,  5,  8,
 2,  1, 14,  7,  4, 10,  8, 13, 15, 12,  9,  0,  3,  5,  6, 11
   };
   private static byte[] substitution6x4(byte[] in) {
      in = splitBytes(in,6); // Splitting byte[] into 6-bit blocks
      byte[] out = new byte[in.length/2];
      int lhByte = 0;
      for (int b=0; b<in.length; b++) { // Should be sub-blocks
         byte valByte = in[b];
         int r = 2*(valByte>>7&0x0001)+(valByte>>2&0x0001); // 1 and 6
         int c = valByte>>3&0x000F; // Middle 4 bits
         int hByte = S[64*b+16*r+c]; // 4 bits (half byte) output
         if (b%2==0) lhByte = hByte; // Left half byte
         else out[b/2] = (byte) (16*lhByte + hByte);
      }
      return out;
   }
   private static byte[] splitBytes(byte[] in, int len) {
      int numOfBytes = (8*in.length-1)/len + 1;
      byte[] out = new byte[numOfBytes];
      for (int i=0; i<numOfBytes; i++) {
         for (int j=0; j<len; j++) {
            int val = getBit(in, len*i+j);
            setBit(out,8*i+j,val);
         }
      }
      return out;
   }
   static final int[] PC1 = {
      57, 49, 41, 33, 25, 17,  9,
       1, 58, 50, 42, 34, 26, 18,
      10,  2, 59, 51, 43, 35, 27,
      19, 11,  3, 60, 52, 44, 36,
      63, 55, 47, 39, 31, 23, 15,
       7, 62, 54, 46, 38, 30, 22,
      14,  6, 61, 53, 45, 37, 29,
      21, 13,  5, 28, 20, 12,  4
   };
   static final int[] PC2 = {
      14, 17, 11, 24,  1,  5,
       3, 28, 15,  6, 21, 10,
      23, 19, 12,  4, 26,  8,
      16,  7, 27, 20, 13,  2,
      41, 52, 31, 37, 47, 55,
      30, 40, 51, 45, 33, 48,
      44, 49, 39, 56, 34, 53,
      46, 42, 50, 36, 29, 32
   };
   static final int[] SHIFTS = {
      1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1
   };
   public static byte[][] getSubkeys(byte[] theKey)
      throws Exception {
      int activeKeySize = PC1.length;
      int numOfSubKeys = SHIFTS.length;
      byte[] activeKey = selectBits(theKey,PC1);
      int halfKeySize = activeKeySize/2;
      byte[] c = selectBits(activeKey,0,halfKeySize);
      byte[] d = selectBits(activeKey,halfKeySize,halfKeySize);
      byte[][] subKeys = new byte[numOfSubKeys][];
      for (int k=0; k<numOfSubKeys; k++) {
         c = rotateLeft(c,halfKeySize,SHIFTS[k]);
         d = rotateLeft(d,halfKeySize,SHIFTS[k]);
         byte[] cd = concatenateBits(c,halfKeySize,d,halfKeySize);
         subKeys[k] = selectBits(cd,PC2);
      }
      return subKeys;
   }
   private static byte[] rotateLeft(byte[] in, int len, int step) {
      int numOfBytes = (len-1)/8 + 1;
      byte[] out = new byte[numOfBytes];
      for (int i=0; i<len; i++) {
         int val = getBit(in,(i+step)%len);
         setBit(out,i,val);
      }
      return out;
   }
   private static byte[] concatenateBits(byte[] a, int aLen, byte[] b,
      int bLen) {
      int numOfBytes = (aLen+bLen-1)/8 + 1;
      byte[] out = new byte[numOfBytes];
      int j = 0;
      for (int i=0; i<aLen; i++) {
         int val = getBit(a,i);
         setBit(out,j,val);
         j++;
      }
      for (int i=0; i<bLen; i++) {
         int val = getBit(b,i);
         setBit(out,j,val);
         j++;
      }
      return out;
   }
   private static byte[] selectBits(byte[] in, int pos, int len) {
      int numOfBytes = (len-1)/8 + 1;
      byte[] out = new byte[numOfBytes];
      for (int i=0; i<len; i++) {
         int val = getBit(in,pos+i);
         setBit(out,i,val);
      }
      return out;
   }
   private static byte[] selectBits(byte[] in, int[] map) {
      int numOfBytes = (map.length-1)/8 + 1;
      byte[] out = new byte[numOfBytes];
      for (int i=0; i<map.length; i++) {
         int val = getBit(in,map[i]-1);
         setBit(out,i,val);
      }
      return out;
   }
   private static int getBit(byte[] data, int pos) {
      int posByte = pos/8;
      int posBit = pos%8;
      byte valByte = data[posByte];
      int valInt = valByte>>(8-(posBit+1)) & 0x0001;
      return valInt;
   }
   private static void setBit(byte[] data, int pos, int val) {
      int posByte = pos/8;
      int posBit = pos%8;
      byte oldByte = data[posByte];
      oldByte = (byte) (((0xFF7F>>posBit) & oldByte) & 0x00FF);
      byte newByte = (byte) ((val<<(8-(posBit+1))) | oldByte);
      data[posByte] = newByte;
   }
   private static byte[] readBytes(String in) throws Exception {
      FileInputStream fis = new FileInputStream(in);
      int numOfBytes = fis.available();
      byte[] buffer = new byte[numOfBytes];
      fis.read(buffer);
      fis.close();
      return buffer;
   }
   private static void writeBytes(byte[] data, String out)
      throws Exception {
      FileOutputStream fos = new FileOutputStream(out);
      fos.write(data);
      fos.close();
   }
   private static void printBytes(byte[] data, String name) {
      System.out.println("");
      System.out.println(name+":");
      for (int i=0; i<data.length; i++) {
         System.out.print(byteToBits(data[i])+" ");
      }
      System.out.println();
   }
   private static String byteToBits(byte b) {
      StringBuffer buf = new StringBuffer();
      for (int i=0; i<8; i++)
         buf.append((int)(b>>(8-(i+1)) & 0x0001));
      return buf.toString();
   }
}
```

If you run this program with JDK 1.4.1 with test key and clear text block mentioned in the previous chapter, you will get:

```
java -cp . CipherDES encrypt TestKey.des TestMsg.clr TestMsg.cph

Key block:
00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001

Input block:
00000001 00100011 01000101 01100111 10001001 10101011 11001101 11101111

Output block:
10000101 11101000 00010011 01010100 00001111 00001010 10110100 00000101

java -cp . CipherDES decrypt TestKey.des TestMsg.cph TestMsg.bck

Key block:
00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001

Input block:
10000101 11101000 00010011 01010100 00001111 00001010 10110100 00000101

Output block:
00000001 00100011 01000101 01100111 10001001 10101011 11001101 11101111
```

As you can see from the output, the program works correctly.

Java Implementation of DES - Test Cases

This section provides two test cases for the Java implementation of DES algorithm, CipherDES.java.

Once I got my DES implementation working, I created the following testing Java program to performed several test cases so that I can validate the results with other DES implementations:

```
/* CipherDesTest.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.io.*;
import herong.CipherDES;
class CipherDesTest {
   public static void main(String[] a) {
      if (a.length<1) {
         System.out.println("Usage:");
         System.out.println("java CipherDesTest 1");
         return;
      }
      String test = a[0];
      try {
         byte[] theKey = null;
         byte[] theMsg = null;
         byte[] theExp = null;
         if (test.equals("1")) {
            theKey = hexToBytes("133457799BBCDFF1");
            theMsg = hexToBytes("0123456789ABCDEF");
            theExp = hexToBytes("85E813540F0AB405");
         } else if (test.equals("2")) {
            theKey = hexToBytes("38627974656B6579"); // "8bytekey"
            theMsg = hexToBytes("6D6573736167652E"); // "message."
            theExp = hexToBytes("7CF45E129445D451");
         } else {
            System.out.println("Usage:");
            System.out.println("java CipherDesTest 1");
            return;
         }
         byte[][] subKeys = CipherDES.getSubkeys(theKey);
         byte[] theCph = CipherDES.cipher(theMsg,subKeys,"encrypt");
         System.out.println("Key     : "+bytesToHex(theKey));
         System.out.println("Message : "+bytesToHex(theMsg));
         System.out.println("Cipher  : "+bytesToHex(theCph));
         System.out.println("Expected: "+bytesToHex(theExp));
      } catch (Exception e) {
         e.printStackTrace();
         return;
      }
   }
   public static byte[] hexToBytes(String str) {
      if (str==null) {
         return null;
      } else if (str.length() < 2) {
         return null;
      } else {
         int len = str.length() / 2;
         byte[] buffer = new byte[len];
         for (int i=0; i<len; i++) {
             buffer[i] = (byte) Integer.parseInt(
                str.substring(i*2,i*2+2),16);
         }
         return buffer;
      }

   }
   public static String bytesToHex(byte[] data) {
      if (data==null) {
         return null;
      } else {
         int len = data.length;
         String str = "";
         for (int i=0; i<len; i++) {
            if ((data[i]&0xFF)<16) str = str + "0"
               + java.lang.Integer.toHexString(data[i]&0xFF);
            else str = str
               + java.lang.Integer.toHexString(data[i]&0xFF);
         }
         return str.toUpperCase();
      }
   }
}
```

Note that this program, CipherDesTest.java, can be easily modified to add other testing cases. To run CipherDesTest.java, you need compile CipherDES.java into a package called "herong", and run them together.

Case 1 - Taken from The DES Algorithm Illustrated:

```
java -cp . CipherDesTest 1

Key     : 133457799BBCDFF1
Message : 0123456789ABCDEF
Cipher  : 85E813540F0AB405
Expected: 85E813540F0AB405
```

Case 2 - Modified from DES Source Code:

```
java -cp . CipherDesTest 2

Key     : 38627974656B6579
Message : 6D6573736167652E
Cipher  : 7CF45E129445D451
Expected: 7CF45E129445D451
```

DES Algorithm - Java Implementation in JDK JCE

This chapter provides tutorial examples and notes about DES algorithm implementation in the JDK JCE package. Topics include JCE classes related to DES; testing tutorial programs for DES encryption and decryption; PKCS5Padding schema to pad cleartext as 8-byte blocks.

Conclusion

- Sun JDK JCE implementation of DES algorithm works as expected.
- PKCS5Padding is a simple but clever way pad cleartext message be multiples of 64-bit blocks.
- Sun implementation of PKCS5Padding works well.

DES Java Implementation in JDK by Sun

This section describes how DES algorithm is implemented in JDK. DES algorithm is implemented as part of the JCE package which is integrated in the JDK since version 1.4.

DES algorithm has been implemented in Java by Sun as part of the JDK JCE (Java Cryptography Extension) package. JCE was previously an optional package (extension) to the Java 2 SDK, Standard Edition, versions 1.2.x and 1.3.x. JCE has now been integrated into the Java 2 SDK, version 1.4.x, and new versions.

Sun's implementation of DES includes the following features:

- Offers both DES and Triple DES algorithms.
- Offers Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Propagating Cipher Block Chaining (PCBC) modes.
- Offers PKCS5 padding options

The following JCE classes and interfaces are involved to support DES encryption and decryption:

- javax.crypto.spec.DESKeySpec - This class specifies a DES encryption and decryption key material.
- javax.crypto.spec.DESedeKeySpec - This class specifies a DES-EDE ("triple-DES") encryption and decryption key material.
- javax.crypto.SecretKeyFactory - This class represents a factory for secret keys. SecretKeyFactory allows you to build an opaque secret key object from a given key specification (key material), like a DESKeySpec object. It also allows you to retrieve the underlying key material of an opaque secret key object.
- javax.crypto.Cipher - This class provides the functionality of a cryptographic cipher for encryption and decryption. In order to create a Cipher object, the application calls the Cipher's getInstance() method, and passes the name of the requested transformation of a specific cryptographic algorithm (e.g. DES) to it.

Steps of Using DES Algorithm in JDK JCE

This section describes how to use DES algorithm provided in the JDK JCE package. Steps include building a secret key object from key material; creating and initializing a cipher object.

To use DES algorithm in the JDK JCE package, we need to use several JCE classes and follow the steps:

1. Creating a DESKeySpec object from a 8-byte key material by calling the javax.crypto.spec.DESKeySpec constructor.

2. Creating a SecretKeyFactory object with DES algorithm name, mode name, and padding name by calling the javax.crypto.SecretKeyFactory.getInstance() static method.

3. Converting the DESKeySpec object to a SecretKey object through the SecretKeyFactory object by calling the javax.crypto.SecretKeyFactory.generateSecret() method.

4. Creating a Cipher object with DES algorithm name, mode name, and padding name by calling the javax.crypto.Cipher.getInstance() static method.

5. Initializing the Cipher object with the SecretKey object by calling the javax.crypto.Cipher.init() method.

6. Ciphering through input data as byte arrays by calling the javax.crypto.Cipher.update() method.

7. Finalizing the ciphering process by calling the javax.crypto.Cipher.doFinal() method.

Testing DES Algorithm in JDK JCE

This section provides a tutorial example to test DES algorithm implemented in the JDK JCE package.

To test out Sun's DES implementation, I created the following testing Java program to perform several test cases to understand those classes and methods.

```
/**
 * JceSunDesTest.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.security.spec.*;
import javax.crypto.*;
import javax.crypto.spec.*;
class JceSunDesTest {
   public static void main(String[] a) {
      if (a.length<1) {
         System.out.println("Usage:");
         System.out.println("java JceSunDesTest 1/2");
         return;
      }
      String test = a[0];
      try {
         byte[] theKey = null;
         byte[] theMsg = null;
         byte[] theExp = null;
         if (test.equals("1")) {
            theKey = hexToBytes("133457799BBCDFF1");
            theMsg = hexToBytes("0123456789ABCDEF");
            theExp = hexToBytes("85E813540F0AB405");
         } else if (test.equals("2")) {
            theKey = hexToBytes("38627974656B6579"); // "8bytekey"
            theMsg = hexToBytes("6D6573736167652E"); // "message."
            theExp = hexToBytes("7CF45E129445D451");
         } else {
            System.out.println("Usage:");
            System.out.println("java JceSunDesTest 1/2");
            return;
         }
         KeySpec ks = new DESKeySpec(theKey);
         SecretKeyFactory kf
            = SecretKeyFactory.getInstance("DES");
         SecretKey ky = kf.generateSecret(ks);
         Cipher cf = Cipher.getInstance("DES/ECB/NoPadding");
         cf.init(Cipher.ENCRYPT_MODE,ky);
         byte[] theCph = cf.doFinal(theMsg);
         System.out.println("Key     : "+bytesToHex(theKey));
         System.out.println("Message : "+bytesToHex(theMsg));
         System.out.println("Cipher  : "+bytesToHex(theCph));
         System.out.println("Expected: "+bytesToHex(theExp));
      } catch (Exception e) {
         e.printStackTrace();
         return;
      }
   }
   public static byte[] hexToBytes(String str) {
      if (str==null) {
         return null;
      } else if (str.length() < 2) {
         return null;
      } else {
         int len = str.length() / 2;
         byte[] buffer = new byte[len];
         for (int i=0; i<len; i++) {
             buffer[i] = (byte) Integer.parseInt(
                str.substring(i*2,i*2+2),16);
         }
         return buffer;
      }

   }
   public static String bytesToHex(byte[] data) {
      if (data==null) {
         return null;
      } else {
         int len = data.length;
         String str = "";
         for (int i=0; i<len; i++) {
            if ((data[i]&0xFF)<16) str = str + "0"
               + java.lang.Integer.toHexString(data[i]&0xFF);
            else str = str
               + java.lang.Integer.toHexString(data[i]&0xFF);
         }
         return str.toUpperCase();
      }
   }
}
```

Note that this program, CipherDesTest.java, can be easily modified to add other testing cases. I am using the ECB mode and NoPadding option to make it compatible with original DES standard.

Case 1 - Taken from The DES Algorithm Illustrated:

```
java JceSunDesTest 1

Key     : 133457799BBCDFF1
Message : 0123456789ABCDEF
Cipher  : 85E813540F0AB405
Expected: 85E813540F0AB405
```

Case 2 - Modified from DES Source Code:

```
java JceSunDesTest 2

Key     : 38627974656B6579
Message : 6D6573736167652E
Cipher  : 7CF45E129445D451
Expected: 7CF45E129445D451
```

This result is identical to my Java implementation program described in the previous chapter!

What Is PKCS5Padding?

This section describes what is PKCS5Padding - a schema to pad cleartext to be multiples of

8-byte blocks.

As you can see from previous tutorials, DES algorithm requires that the input data to be 8-byte blocks. If you want to encrypt a text message that is not multiples of 8-byte blocks, the text message must be padded with additional bytes to make the text message to be multiples of 8-byte blocks.

PKCS5Padding is a padding scheme described in: RSA Laboratories, "PKCS #5: Password-Based Encryption Standard," version 1.5, November 1993.

PKCS5Padding schema is actually very simple. It follows the following rules:

- The number of bytes to be padded equals to "8 - numberOfBytes(clearText) mod 8". So 1 to 8 bytes will be padded to the clear text data depending on the length of the clear text data.
- All padded bytes have the same value - the number of bytes padded.

PKCS5Padding schema can also be explained with the diagram below, if M is the original clear text and PM is the padded clear text:

```
If numberOfBytes(clearText) mod 8 == 7, PM = M + 0x01
If numberOfBytes(clearText) mod 8 == 6, PM = M + 0x0202
If numberOfBytes(clearText) mod 8 == 5, PM = M + 0x030303
...
If numberOfBytes(clearText) mod 8 == 0, PM = M + 0x0808080808080808
```

JceSunDesPaddingTest.java - JCE DES Padding Test

This section provides a tutorial example to test the PKCS5Padding schema used by the DES implementation in the JDK JCE package.

Let's modify the above testing program to see how PKCS5Padding schema works.

```
/**
 * JceSunDesPaddingTest.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.security.spec.*;
import javax.crypto.*;
import javax.crypto.spec.*;
class JceSunDesPaddingTest {
   public static void main(String[] a) {
      if (a.length<2) {
         System.out.println("Usage:");
         System.out.println("java JceSunDesPaddingTest 1/2/3"
            + " algorithm");
         return;
      }
      String test = a[0];
      String algorithm = a[1];
      try {
         byte[] theKey = null;
         byte[] theMsg = null;
         byte[] theExp = null;
         if (test.equals("1")) {
            theKey = hexToBytes("133457799BBCDFF1");
            theMsg = hexToBytes("0123456789ABCDEF");
            theExp = hexToBytes("85E813540F0AB405");
         } else if (test.equals("2")) {
            theKey = hexToBytes("38627974656B6579"); // "8bytekey"
            theMsg = hexToBytes("6D6573736167652E"); // "message."
            theExp = hexToBytes("7CF45E129445D451");
         } else if (test.equals("3")) {
            theKey = hexToBytes("133457799BBCDFF1"); // = test 1
            theMsg = hexToBytes("0808080808080808"); // PKCS5Padding
            theExp = hexToBytes("FDF2E174492922F8");
         } else {
            System.out.println("Wrong option. For help enter:");
            System.out.println("java JceSunDesPaddingTest");
            return;
         }
         KeySpec ks = new DESKeySpec(theKey);
         SecretKeyFactory kf
            = SecretKeyFactory.getInstance("DES");
         SecretKey ky = kf.generateSecret(ks);
         Cipher cf = Cipher.getInstance(algorithm);
         cf.init(Cipher.ENCRYPT_MODE,ky);
         byte[] theCph = cf.doFinal(theMsg);
         System.out.println("Key     : "+bytesToHex(theKey));
         System.out.println("Message : "+bytesToHex(theMsg));
         System.out.println("Cipher  : "+bytesToHex(theCph));
         System.out.println("Expected: "+bytesToHex(theExp));
      } catch (Exception e) {
         e.printStackTrace();
         return;
      }
   }
   public static byte[] hexToBytes(String str) {
      if (str==null) {
         return null;
      } else if (str.length() < 2) {
         return null;
      } else {
         int len = str.length() / 2;
         byte[] buffer = new byte[len];
         for (int i=0; i<len; i++) {
             buffer[i] = (byte) Integer.parseInt(
                str.substring(i*2,i*2+2),16);
         }
         return buffer;
      }

   }
   public static String bytesToHex(byte[] data) {
      if (data==null) {
         return null;
      } else {
         int len = data.length;
         String str = "";
         for (int i=0; i<len; i++) {
            if ((data[i]&0xFF)<16) str = str + "0"
               + java.lang.Integer.toHexString(data[i]&0xFF);
            else str = str
               + java.lang.Integer.toHexString(data[i]&0xFF);
         }
         return str.toUpperCase();
      }
   }
}
```

Let's run this program with different options:

```
java JceSunDesPaddingTest 1 DES/ECB/NoPadding

Key     : 133457799BBCDFF1
Message : 0123456789ABCDEF
Cipher  : 85E813540F0AB405
Expected: 85E813540F0AB405

java JceSunDesPaddingTest 1 DES/ECB/PKCS5Padding

Key     : 133457799BBCDFF1
Message : 0123456789ABCDEF
Cipher  : 85E813540F0AB405FDF2E174492922F8
Expected: 85E813540F0AB405

java JceSunDesPaddingTest 1 DES

Key     : 133457799BBCDFF1
Message : 0123456789ABCDEF
Cipher  : 85E813540F0AB405FDF2E174492922F8
Expected: 85E813540F0AB405

java JceSunDesPaddingTest 3 DES/ECB/PKCS5Padding

Key     : 133457799BBCDFF1
Message : 0808080808080808
Cipher  : FDF2E174492922F8FDF2E174492922F8
Expected: FDF2E174492922F8
```

The results tells us that:

- If the input block is 8-byte long, PKCS5Padding adds another a block of 8-byte.
- The Cipher.getInstance("DES") method call is identical to Cipher.getInstance("DES/ECB/PKCS5Padding").
- If the input block is 8-byte long, PKCS5Padding adds another a block of 8-byte with value of 0x08 for each padded byte.

DES Encryption Operation Modes

This chapter provides tutorial examples and notes about DES encryption operation modes. Topics include how different blocks of plaintext can be coupled together to improve the strength of the DES encryption algorithm; ECB (Electronic CodeBook), CBC (Cipher Block Chaining), CFB (Cipher FeedBack), and OFB (Output FeedBack) Operation Modes.

Conclusion

- DES operation modes are ways to couple previous plaintext blocks with the current plaintext block to improve the strength of the encryption.
- Initial vector is used to parameterize the operation modes.
- Sun JCE implementation of DES operation modes work as expected.

DES Encryption Operation Mode Introduction

This section describes what are DES encryption operation modes and notations used to describe how each operation mode works.

DES encryption algorithm defines how a single 64-bit plaintext block can be encrypted. It does not define how a real plaintext message with an arbitrary number of bytes should be padded and arranged into 64-bit input blocks for the encryption process. It does not define how one input block should be coupled with other blocks from the same original plaintext message to improve the encryption strength.

(FIPS) Federal Information Processing Standards Publication 81 published in 1980 provided the following block encryption operation modes to address how blocks of the same plaintext message should be coupled:

- ECB - Electronic Code Book operation mode.
- CBC - Cipher Block Chaining operation mode.
- CFB - Cipher Feedback operation mode
- OFB - Output Feedback operation mode

See http://www.itl.nist.gov/fipspubs/fip81.htm for details.

In order to describe these operation modes, we need to define the following notations:

P = P[1], P[2], P[3], ..., P[i], ... - Representing the original plaintext message, P, being arranged into multiple 64-bit plaintext blocks. P[i] represents plaintext block number i.

Ek(P[i]) - Representing the DES encryption algorithm applied on a single 64-bit plaintext block, P[i], with a predefined key, k.

C = C[1], C[2], C[3], ..., C[i], ... - Representing the final ciphertext message, C, being regrouped from multiple 64-bit ciphertext blocks. C[i] represents ciphertext block number i.

IV - Called "Initial Vector", representing a predefined 64-bit initial value.

With these notations, we are ready to describe different operation modes that can be applied the DES encryption algorithm.

What is ECB (Electronic CodeBook) Operation Mode?

This section describes what is ECB (Electronic CodeBook) Operation Mode - each plaintext block is encrypted independently without any input from other blocks.

ECB (Electronic CodeBook) is the simplest operation mode comparing to other operation modes. It can be described by the formula and the diagram below with notations defined earlier:

```
C[i] = Ek(P[i])

 P[1]--|       P[2]--|       P[3]--|
       |             |             |
     Ek()          Ek()          Ek()
       |             |             |
     C[1]          C[2]          C[3]
```

As you can see from the formula and the diagram, in ECB mode, each ciphertext block is obtained by applying the DES encryption process to the current plaintext block directly. So the current ciphertext block has not dependency on any previous plaintext blocks.

The disadvantage of ECB mode is that identical plaintext blocks are encrypted to identical ciphertext blocks; thus, it does not hide data patterns well. In some senses it doesn't provide message confidentiality at all, and it is not recommended for cryptographic protocols.

wikipedia has a striking example of the degree to which ECB can reveal patterns in the plaintext. The example uses a bitmap file of an image as the plaintext message. After applying DES encryption in ECB mode, the ciphertext message can be viewed as a new bitmap image file. The new image does reveal major patterns of the original image very clearly.

What is CBC (Cipher Block Chaining) Operation Mode?

This section describes what is CBC (Cipher Block Chaining) Operation Mode - each plaintext block is XORed with the ciphertext of the previous block before encryption.

CBC (Cipher Block Chaining) operation mode can be described with notations defined earlier as the following formula and diagram:

```
C[i] = Ek(P[i] XOR C[i-1])
C[1] = Ek(P[1] XOR IV)

      IV
       |       ------|       ------|
       |     /       |     /       |
P[1]--XOR   / P[2]--XOR   / P[3]--XOR
       |   /         |   /         |
     Ek() /        Ek() /        Ek()
       | /           | /           |
     C[1]          C[2]          C[3]
```

As you can see from the formula and the diagram, in CBC mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted to generate the current ciphertext block. In this way, each ciphertext block is depending on all plaintext blocks up to that point. Note that for the first block, the Initial Vector (IV) is used as the previous ciphertext block.

What is CFB (Cipher FeedBack) Operation Mode?

This section describes what is CFB (Cipher FeedBack) Operation Mode - each plaintext block is XORed with the encrypted version of the ciphertext of the previous block to be the ciphertext block.

CFB (Cipher FeedBack) operation mode can be described with notations defined earlier as the following formula and diagram:

```
C[i] = P[i] XOR Ek(C[i-1])
C[1] = P[1] XOR Ek(IV)

      IV
       |        -----|        -----|
     Ek()     /    Ek()     /    Ek()
       |     /       |     /       |
       |    /        |    /        |
P[1]--XOR  /  P[2]--XOR  /  P[3]--XOR
       |  /          |  /          |
       | /           | /           |
     C[1]          C[2]          C[3]
```

As you can see from the formula and the diagram, in CBC mode, each block of plaintext is XORed with the encrypted version of the previous ciphertext to generate the current ciphertext block. In this way, each ciphertext block is depending on all plaintext blocks up to that point. Note that for the first block, the Initial Vector (IV) is used as the previous ciphertext block.

What is OFB (Output FeedBack) Operation Mode?

This section describes what is OFB (Output FeedBack) Operation Mode - each plaintext block is XORed with the current output block to be the ciphertext block. The current output block is the encrypted version of the previous output block.

OFB (Output FeedBack) operation mode can be described with notations defined earlier as the following formula and diagram:

```
C[i] = P[i] XOR O[i]
O[i] = Ek(O[i-1])
O[1] = E(IV)

      IV
       |        -----|        -----|
     Ek()     /    Ek()     /    Ek()
       |--O[1]       |--O[2]       |--O[3]
       |             |             |
P[1]--XOR     P[2]--XOR     P[3]--XOR
       |             |             |
     C[1]          C[2]          C[3]
```

As you can see from the formula and the diagram, in OFB mode, each block of plaintext is XORed with the current output block to generate the current ciphertext block. The current output block is obtained by applying the encryption process on the previous output block. Note that for the first block, the Initial Vector (IV) is used as the previous output block.

DES Operation Modes in JCE

This section describes what DES operation modes are implemented in the JDK JCE package, and how to use them.

Sun has implemented all 4 operation modes described above in their JDK JCE (Java Cryptography Extension) package. To use DES operation modes properly, you need to:

1. Specify the operation mode name as part of the algorithm name when calling Cipher.getInstance(algorithm) to create a cipher object like:

```
   Cipher cObj1 = Cipher.getInstance("DES/ECB/NoPadding");
   Cipher cObj2 = Cipher.getInstance("DES/CBC/NoPadding");
   Cipher cObj3 = Cipher.getInstance("DES/CFB/NoPadding");
   Cipher cObj4 = Cipher.getInstance("DES/OFB/NoPadding");
```

2. Initialize the cipher object with the key and the IV (Initial Vector) by using the IvParameterSpec class like:

```
   AlgorithmParameterSpec apsObj = new IvParameterSpec(theIV);
   cObj.init(Cipher.ENCRYPT_MODE, keyObj, apsObj);
```

JCE DES Operation Mode Testing Program

This section provides a tutorial example program to test DES operation modes implemented in the JDK JCE package.

To test out JCE DES operation mode implementation, I wrote the following testing program:

```
/**
 * JceSunDesOperationModeTest.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.security.spec.*;
import javax.crypto.*;
import javax.crypto.spec.*;
class JceSunDesOperationModeTest {
   public static void main(String[] a) {
      if (a.length<1) {
         System.out.println("Usage:");
         System.out.println(
            "java JceSunDesOperationModeTest 1/2/3/4");
         return;
      }
      String test = a[0];
      try {
         byte[] theKey = null;
         byte[] theIVp = null;
         byte[] theMsg = null;
         byte[] theExp = null;
         String algorithm = null;
         if (test.equals("1")) {
            algorithm = "DES/ECB/NoPadding";
            theKey = hexToBytes("0123456789ABCDEF");
            theMsg = hexToBytes(
               "4E6F772069732074686520074696D6520666F7220616C6C20");
            // "Now is the time for all "
            theExp = hexToBytes(
               "3FA40E8A984D43156A271787AB8883F9893D51EC4B563B53");
         } else if (test.equals("2")) {
            algorithm = "DES/CBC/NoPadding";
            theKey = hexToBytes("0123456789ABCDEF");
            theIVp = hexToBytes("1234567890ABCDEF");
            theMsg = hexToBytes(
               "4E6F772069732074686520074696D6520666F7220616C6C20");
            // "Now is the time for all "
            theExp = hexToBytes(
               "E5C7CDDE872BF27C43E934008C389C0F683788499A7C05F6");
         } else if (test.equals("3")) {
            algorithm = "DES/CFB/NoPadding";
            theKey = hexToBytes("0123456789ABCDEF");
            theIVp = hexToBytes("1234567890ABCDEF");
            theMsg = hexToBytes(
               "4E6F77206973207468652074696D6520666F7220616C6C20");
            // "Now is the time for all "
            theExp = hexToBytes(
               "F3096249C7F46E51A69E839B1A92F78403467133898EA622");
         } else if (test.equals("4")) {
            algorithm = "DES/OFB/NoPadding";
            theKey = hexToBytes("0123456789ABCDEF");
            theIVp = hexToBytes("1234567890ABCDEF");
            theMsg = hexToBytes(
               "4E6F77206973207468652074696D6520666F7220616C6C20");
            // "Now is the time for all "
            theExp = hexToBytes(
               "F3096249C7F46E5135F24A242EEB3D3F3D6D5BE3255AF8C3");
         } else {
            System.out.println("Wrong option. For help enter:");
            System.out.println("java JceSunDesOperationModeTest");
            return;
         }
         KeySpec ks = new DESKeySpec(theKey);
         SecretKeyFactory kf
            = SecretKeyFactory.getInstance("DES");
         SecretKey ky = kf.generateSecret(ks);
         Cipher cf = Cipher.getInstance(algorithm);
         if (theIVp == null) {
            cf.init(Cipher.ENCRYPT_MODE, ky);
         } else {
            AlgorithmParameterSpec aps = new IvParameterSpec(theIVp);
            cf.init(Cipher.ENCRYPT_MODE, ky, aps);
         }
         byte[] theCph = cf.doFinal(theMsg);
         System.out.println("Key     : "+bytesToHex(theKey));
         if (theIVp != null) {
            System.out.println("IV      : "+bytesToHex(theIVp));
         }
         System.out.println("Message : "+bytesToHex(theMsg));
         System.out.println("Cipher  : "+bytesToHex(theCph));
         System.out.println("Expected: "+bytesToHex(theExp));
      } catch (Exception e) {
         e.printStackTrace();
         return;
      }
   }
   public static byte[] hexToBytes(String str) {
      if (str==null) {
         return null;
      } else if (str.length() < 2) {
         return null;
      } else {
         int len = str.length() / 2;
         byte[] buffer = new byte[len];
         for (int i=0; i<len; i++) {
             buffer[i] = (byte) Integer.parseInt(
                str.substring(i*2,i*2+2),16);
         }
         return buffer;
      }
   }
   public static String bytesToHex(byte[] data) {
      if (data==null) {
         return null;
      } else {
         int len = data.length;
         String str = "";
         for (int i=0; i<len; i++) {
            if ((data[i]&0xFF)<16) str = str + "0"
               + java.lang.Integer.toHexString(data[i]&0xFF);
            else str = str
               + java.lang.Integer.toHexString(data[i]&0xFF);
         }
         return str.toUpperCase();
      }
   }
}
```

This program provides 4 tests: one for each operation mode. All tests share the same plaintext message, "Now is the time for all ". For CBC, CFB and OFB modes, the same IV is used, 0x1234567890ABCDEF.

JCE DES Operation Mode Testing Program Result

This section provides testing results of a tutorial example program to test DES operation modes implemented in the JDK JCE package.

I used my testing program, JceSunDesOperationModeTest.java, to test the cases listed in the http://www.itl.nist.gov/fipspubs/fip81.htm:

```
java JceSunDesOperationModeTest 1 -- with ECB
Key     : 0123456789ABCDEF
Message : 4E6F772069732074686520746696D6520666F7220616C6C20
Cipher  : 3FA40E8A984D48156A271787AB8883F9893D51EC4B563B53
Expected: 3FA40E8A984D43156A271787AB8883F9893D51EC4B563B53

java JceSunDesOperationModeTest 2 -- with CBC
Key     : 0123456789ABCDEF
IV      : 1234567890ABCDEF
Message : 4E6F772069732074686520746696D6520666F7220616C6C20
Cipher  : E5C7CDDE872BF27C43E934008C389C0F683788499A7C05F6
Expected: E5C7CDDE872BF27C43E934008C389C0F683788499A7C05F6

java JceSunDesOperationModeTest 3 -- with CFB
Key     : 0123456789ABCDEF
IV      : 1234567890ABCDEF
Message : 4E6F772069732074686520746696D6520666F7220616C6C20
Cipher  : F3096249C7F46E51A69E839B1A92F78403467133898EA622
Expected: F3096249C7F46E51A69E839B1A92F78403467133898EA622

java JceSunDesOperationModeTest 4 -- with OFB
Key     : 0123456789ABCDEF
IV      : 1234567890ABCDEF
Message : 4E6F772069732074686520746696D6520666F7220616C6C20
Cipher  : F3096249C7F46E5135F24A242EEB3D3F3D6D5BE3255AF8C3
Expected: F3096249C7F46E5135F24A242EEB3D3F3D6D5BE3255AF8C3
```

Outputs of the first 3 test cases match well with the expected values documented in http://www.itl.nist.gov/fipspubs/fip81.htm. The output of the last test case can not be compared, since it is not document in FIPS 81.

DES in Stream Cipher Modes

This chapter provides tutorial examples and notes about DES stream cipher modes. 1-bit, 1-byte and 8-byte stream ciphers, Topics include modifying DES CFB and OFB block ciphers to stream ciphers; using DES stream ciphers provided in the JDK JCE package.

Conclusion

- DES stream cipher modes are ways to operate DES encryption algorithm in block sizes less than 64 bits.
- Sun JCE implementation of DES stream cipher modes work as expected.

Introducting DES Stream Cipher Modes

This section describes what are DES encryption stream cipher modes and how CFB and OFB block operation modes can be modified as stream modes.

As we know from previous tutorials, DES algorithm is a block cipher algorithm. But it can be operated in different ways to become stream ciphers.

(FIPS 81) Federal Information Processing Standards Publication 81 published in 1980 defined the following 4 operation modes:

- ECB - Electronic Code Book operation mode.
- CBC - Cipher Block Chaining operation mode.
- CFB - Cipher Feedback operation mode
- OFB - Output Feedback operation mode

See http://www.itl.nist.gov/fipspubs/fip81.htm for details.

In FIPS 81, the last two operation modes, CFB and OFB, are defined in ways so that they can be used as stream ciphers.

In order to describe these operation modes, we need to define the following notations:

P = P[1], P[2], P[3], ..., P[i], ... - Representing the original plaintext message, P, being arranged into multiple 64-bit plaintext blocks. P[i] represents plaintext block number i.

E(P[i]) - Representing the DES encryption algorithm applied on a single 64-bit plaintext block, P[i], with a predefined key, k.

C = C[1], C[2], C[3], ..., C[i], ... - Representing the final ciphertext message, C, being regrouped from multiple 64-bit ciphertext blocks. C[i] represents ciphertext block number i.

IV - Called "Initial Vector", representing a predefined 64-bit initial value.

CFB (Cipher FeedBack) as a Stream Cipher

This section describes how DES CFB (Cipher FeedBack) operation mode can be modified as a 1-bit stream cipher or a 1-byte stream cipher.

CFB (Cipher FeedBack) operation mode as a block cipher can be described with notations defined earlier as the following formula and diagram:

```
C[i] = P[i] XOR E(C[i-1])
C[1] = P[1] XOR E(IV)

      IV
       |        -----|        -----|
      E()     /     E()     /     E()
       |     /       |     /       |
       |    /        |    /        |
P[1]--XOR  /  P[2]--XOR  /  P[3]--XOR
       |  /          |  /          |
       | /           | /           |
     C[1]          C[2]          C[3]
```

As you can see from the formula and the diagram, in CBC mode, each block of plaintext is XORed with the encrypted version of the previous ciphertext to generate the current ciphertext block. In this way, each ciphertext block is depending on all plaintext blocks up to that point. Note that for the first block, the Initial Vector (IV) is used as the previous ciphertext block.

In order to run the CFB operation mode as a stream cipher, FIPS 81 defines CFB variations where plaintext blocks can have any size less than 64 bits. To describe CFB variations, we need the following additional notations:

k - Representing the size plaintext blocks. k can have a value between 1 and 64.

Fk() - Representing a filter function that take the first k bits of a 64-bit block.

I = I[1], I[2], I[3], ..., I[i], ... - Representing the input block, I, used as input to the DES encryption process.

Sk() - Representing a shifting function that shifts k bits out of the input block from the left side. The missing k bits are taken from the ciphertext block.

With the above notations, a k-bit CFB operation mode variation can be described as:

```
C[i] = P[i] XOR Fk(E(I[i]))
I[i] = Sk(I[i-1], C[i-1])
I[1] = IV

      IV
       |
     I[1]------Sk()--I[2]------Sk()--I[3]
       |       /       |       /       |
      E()     /       E()     /       E()
       |     /         |     /         |
     Fk()   /        Fk()   /        Fk()
       |   /           |   /           |
P[1]--XOR /     P[2]--XOR /     P[3]--XOR
       | /             | /             |
     C[1]            C[2]            C[3]
```

Note that there are some special cases of CFB variations:

- The 64-bit CFB variation is the original CFB operation mode.
- The 8-bit CFB variation is a stream cipher that takes 1 byte (8 bits) from the plaintext message at a time.
- The 1-bit CFB variation is a stream cipher that takes 1 bit from the plaintext message at a time.

OFB (Output FeedBack) as a Stream Cipher

This section describes how DES OFB (Output FeedBack) operation mode can be modified as a 1-bit stream cipher or a 1-byte stream cipher.

OFB (Output FeedBack) operation mode can be described with notations defined earlier as the following formula and diagram:

```
C[i] = P[i] XOR O[i]
O[i] = E(O[i-1])
O[1] = E(IV)

      IV
       |        -----|        -----|
      E()     /     E()     /     E()
       |--O[1]       |--O[2]       |--O[3]
       |             |             |
P[1]--XOR     P[2]--XOR     P[3]--XOR
```

```
       |             |             |
     C[1]          C[2]          C[3]
```

As you can see from the formula and the diagram, in OFB mode, each block of plaintext is XORed with the current output block to generate the current ciphertext block. The current output block is obtained by applying the encryption process on the previous output block. Note that for the first block, the Initial Vector (IV) is used as the previous output block.

In order to run the OFB operation mode as a stream cipher, FIPS 81 defines OFB variations where plaintext blocks can have any size less than 64 bits. To describe OFB variations, we need the following additional notations:

k - Representing the size plaintext blocks. k can have a value between 1 and 64.

Fk() - Representing a filter function that take the first k bits of a 64-bit block.

I = I[1], I[2], I[3], ..., I[i], ... - Representing the input block, I, used as input to the DES encryption process.

Sk() - Representing a shifting function that shifts k bits out of the input block from the left side. The missing k bits are taken from the output block.

With the above notations, a k-bit OFB operation mode variation can be described as:

```
C[i] = P[i] XOR Fk(O[i])
O[i] = E(I[i])
I[i] = Sk(I[i-1], O[i-1])
I[1] = IV

      IV
       |
     I[1]------Sk()--I[2]------Sk()--I[3]
       |       /       |       /       |
      E()     /       E()     /       E()
       |--O[1]         |--O[2]         |--O[3]
     Fk()            Fk()            Fk()
       |               |               |
P[1]--XOR       P[2]--XOR       P[3]--XOR
       |               |               |
     C[1]            C[2]            C[3]
```

Note that there are some special cases of OFB variations:

- The 64-bit OFB variation is the original OFB operation mode.
- The 8-bit OFB variation is a stream cipher that takes 1 byte (8 bits) from the plaintext message at a time.
- The 1-bit OFB variation is a stream cipher that takes 1 bit from the plaintext message at a time.

CFB and OFB Stream Ciphers Implemented in JCE

This section describes how DES CFB and OFB (Output FeedBack) stream ciphers are implemented in the JDK JCE package.

Sun has implemented stream cipher modes for both CFB and OFB modes, but with restrictions that the feedback sizes must be multiples of 8 bits. To use CFB or OFB in a stream cipher mode, you need to specify the feedback size in bits right after the mode name when calling Cipher.getInstance(algorithm) to create a cipher object like:

```
   Cipher cObj1 = Cipher.getInstance("DES/CFB8/NoPadding");
   Cipher cObj2 = Cipher.getInstance("DES/CFB16/NoPadding");
   Cipher cObj3 = Cipher.getInstance("DES/CFB24/NoPadding");
   ...
   Cipher cObja = Cipher.getInstance("DES/OFB8/NoPadding");
   Cipher cObjb = Cipher.getInstance("DES/OFB16/NoPadding");
   Cipher cObjc = Cipher.getInstance("DES/OFB24/NoPadding");
   ...
```

JCE DES Stream Ciphers Testing Program

This section provides a tutorial example on how to use DES stream ciphers provided in the JDK JCE package.

To test out JCE DES stream cipher mode implementation, I wrote the following testing program:

```
/**
 * JceSunDesStreamCipherTest.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.security.spec.*;
import javax.crypto.*;
import javax.crypto.spec.*;
class JceSunDesStreamCipherTest {
   public static void main(String[] a) {
      if (a.length<1) {
         System.out.println("Usage:");
         System.out.println(
            "java JceSunDesStreamCipherTest 1/2/3/4");
         return;
      }
      String test = a[0];
      try {
         byte[] theKey = null;
         byte[] theIVp = null;
         byte[] theMsg = null;
         byte[] theExp = null;
         String algorithm = null;
         if (test.equals("1")) {
            algorithm = "DES/CFB8/NoPadding";
            theKey = hexToBytes("0123456789ABCDEF");
            theIVp = hexToBytes("1234567890ABCDEF");
            theMsg = hexToBytes("4E6F772069732074686520");
            // "Now is the"
            theExp = hexToBytes("F31FDA07011462EE187F");
         } else if (test.equals("2")) {
            algorithm = "DES/CFB64/NoPadding";
            theKey = hexToBytes("0123456789ABCDEF");
            theIVp = hexToBytes("1234567890ABCDEF");
            theMsg = hexToBytes(
               "4E6F77206973207468652074696D6520666F7220616C6C20");
            // "Now is the time for all "
            theExp = hexToBytes(
               "F3096249C7F46E51A69E839B1A92F78403467133898EA622");
         } else if (test.equals("3")) {
            algorithm = "DES/OFB8/NoPadding";
            theKey = hexToBytes("0123456789ABCDEF");
            theIVp = hexToBytes("1234567890ABCDEF");
            theMsg = hexToBytes("4E6F7720697320746865");
            // "Now is the"
            theExp = hexToBytes("F34A2850C9C64985D684");
         } else if (test.equals("4")) {
            algorithm = "DES/OFB64/NoPadding";
            theKey = hexToBytes("0123456789ABCDEF");
            theIVp = hexToBytes("1234567890ABCDEF");
            theMsg = hexToBytes(
               "4E6F77206973207468652074696D6520666F7220616C6C20");
            // "Now is the time for all "
            theExp = hexToBytes(
               "F3096249C7F46E5135F24A242EEB3D3F3D6D5BE3255AF8C3");
         } else {
            System.out.println("Wrong option. For help enter:");
            System.out.println("java JceSunDesStreamCipherTest");
            return;
         }
         KeySpec ks = new DESKeySpec(theKey);
         SecretKeyFactory kf
            = SecretKeyFactory.getInstance("DES");
         SecretKey ky = kf.generateSecret(ks);
         Cipher cf = Cipher.getInstance(algorithm);
         if (theIVp == null) {
            cf.init(Cipher.ENCRYPT_MODE, ky);
         } else {
            AlgorithmParameterSpec aps = new IvParameterSpec(theIVp);
            cf.init(Cipher.ENCRYPT_MODE, ky, aps);
         }
         byte[] theCph = cf.doFinal(theMsg);
         System.out.println("Key     : "+bytesToHex(theKey));
         if (theIVp != null) {
            System.out.println("IV      : "+bytesToHex(theIVp));
         }
         System.out.println("Message : "+bytesToHex(theMsg));
         System.out.println("Cipher  : "+bytesToHex(theCph));
         System.out.println("Expected: "+bytesToHex(theExp));
      } catch (Exception e) {
         e.printStackTrace();
         return;
      }
   }
   public static byte[] hexToBytes(String str) {
      if (str==null) {
         return null;
      } else if (str.length() < 2) {
         return null;
      } else {
         int len = str.length() / 2;
         byte[] buffer = new byte[len];
         for (int i=0; i<len; i++) {
             buffer[i] = (byte) Integer.parseInt(
                str.substring(i*2,i*2+2),16);
         }
         return buffer;
      }
   }
   public static String bytesToHex(byte[] data) {
      if (data==null) {
         return null;
      } else {
         int len = data.length;
         String str = "";
         for (int i=0; i<len; i++) {
            if ((data[i]&0xFF)<16) str = str + "0"
               + java.lang.Integer.toHexString(data[i]&0xFF);
            else str = str
               + java.lang.Integer.toHexString(data[i]&0xFF);
         }
         return str.toUpperCase();
      }
   }
}
```

This program provides 4 tests: CFB8, CFB64, OFB8 and OFB64 with testing plaintext messages, keys and initial vectors used in FIPS81.

JCE DES Stream Ciphers Testing Program Result

This section provides results of the tutorial example on how to use DES stream ciphers provided in the JDK JCE package.

I used my testing program, JceSunDesStreamCipherTest.java, to test the cases listed in the http://www.itl.nist.gov/fipspubs/fip81.htm:

```
java JceSunDesStreamCipherTest 1 -- with 8-bit CFB
Key     : 0123456789ABCDEF
IV      : 1234567890ABCDEF
Message : 4E6F7720697320746865
Cipher  : F31FDA07011462EE187F
Expected: F31FDA07011462EE187F

java JceSunDesStreamCipherTest 2 -- with 64-bit CFB
Key     : 0123456789ABCDEF
IV      : 1234567890ABCDEF
Message : 4E6F77206973207468652074696D6520666F7220616C6C20
Cipher  : F3096249C7F46E51A69E839B1A92F78403467133898EA622
Expected: F3096249C7F46E51A69E839B1A92F78403467133898EA622

java JceSunDesStreamCipherTest 3 -- with 8-bit OFB
Key     : 0123456789ABCDEF
IV      : 1234567890ABCDEF
Message : 4E6F7720697320746865
Cipher  : F34A2850C9C64985D684
Expected: F34A2850C9C64985D684
```

```
java JceSunDesStreamCipherTest 4 -- with 64-bit CFB
Key     : 0123456789ABCDEF
IV      : 1234567890ABCDEF
Message : 4E6F77206973207468652074696D6520666F7220616C6C20
Cipher  : F3096249C7F46E5135F24A242EEB3D3F3D6D5BE3255AF8C3
Expected: F3096249C7F46E5135F24A242EEB3D3F3D6D5BE3255AF8C3
```

Outputs of all 4 test cases match well with the expected values documented in http://www.itl.nist.gov/fipspubs/fip81.htm.

PHP Implementation of DES - mcrypt

This chapter provides tutorial examples and notes about PHP implementation of DES. Topics include introduction of mcrypt library; mcrypt encryption functions; DES encryption and decryption test program and test result.

Conclusion

- mcrypt extension offers DES 64-bit ECB, 64-bit CBC, 8-bit CFB and 8-bit OFB operation modes.
- mcrypt pads plaintext messages with 0x00 bytes.
- If you are interested in PHP source code on DES implementation, see Paul Tero's DES page at http://www.tero.co.uk/des/.

mcrypt Library for PHP

This section describes the mcrypt library - encryption extension for PHP. mcrypt supports DES and many other encryption algorithms.

"mcrypt" is the suggested encryption extension for PHP. It supports a wide variety of block algorithms such as DES, TripleDES, Blowfish (default), 3-WAY, SAFER-SK64, SAFER-SK128, TWOFISH, TEA, RC2 and GOST in CBC, OFB, CFB and ECB cipher modes. See http://sourceforge.net/projects/mcrypt for details.

The best way to download and install "mcrypt" for your Windows system is to download the pre-compiled "mcrypt" version by the steps below:

1. Go to http://ftp.emini.dk/pub/php/win32/mcrypt/.

2. Download the binary file, libmcrypt.dll, 19-Jan-2004 02:27, 163k.

3. Save libmcrypt.dll to your PHP directory, like \php\.

4. Open PHP initialization file, \php\php.ini.

5. Uncomment line "extension=php_mcrypt.dll".

You are ready to use "mcrypt" encryption functions.

mcrypt Encryption Functions

This section describes mcrypt encryption functions and 4 DES ciphers (operation modes) - 8-byte ECB, 8-byte CBC, 1-byte CFB, and 1-byte OFB ciphers.

"mcrypt" encryption extension offers the following main functions to perform an encryption operation:

- mcrypt_module_open() - Creates a resource handle for the specified encryption algorithm and operation mode.
- mcrypt_generic_init() - Initializes an encryption handle with a key and an initial vector.
- mcrypt_generic() - Encrypts a plaintext message with an initialized encryption handle.
- mcrypt_generic_deinit() - De-initializes an encryption handle.
- mcrypt_module_close() - Closes an encryption resource handle.

To specify the DES encryption algorithm for the mcrypt_module_open() function, you need to use the predefined constant: MCRYPT_DES.

To specify a DES operation mode for the mcrypt_module_open() function, you also need to use one of 4 predefined constants representing 4 4 DES operation modes supported in mcrypt:

- MCRYPT_MODE_ECB - The 64-bit ECB (Electronic CodeBook) operation mode.
- MCRYPT_MODE_CBC - The 64-bit CBC (Cipher Block Chaining) operation mode.
- MCRYPT_MODE_CFB - The 8-bit CFB (Cipher FeedBack) operation mode
- MCRYPT_MODE_OFB - The 8-bit OFB (Output FeedBack) operation mode

For example, if you want to create cipher resource handle to use DES 1-byte CFB operation mode, the function call should be: mcrypt_module_open(MCRYPT_DES, ", MCRYPT_MODE_CFB, ").

mcrypt DES Encryption Testing Program

This section provides a tutorial example on how to use the DES algorithm in a specific operation mode as block or stream cipher.

To test out the DES operation modes and stream cipher modes implemented in the "mcrypt" extension, I wrote the following test PHP script:

```
<?php # des_mcrypt_operation_mode_test.php
# Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
#
   $key = pack("H*", "0123456789ABCDEF");
   $iv = pack("H*", "1234567890ABCDEF");
   # "Now is the time for all "
   $plain_text = pack("H*",
      "4E6F77206973207468652074696D6520666F7220616C6C20");

   $mode = MCRYPT_MODE_ECB;
   $expected = pack("H*",
      "3FA40E8A984D43156A271787AB8883F9893D51EC4B563B53");
   print("\n\nTest 1 - DES ECB 64-bit:");
   des_test($mode);

   $mode = MCRYPT_MODE_CBC;
   $expected = pack("H*",
      "E5C7CDDE872BF27C43E934008C389C0F683788499A7C05F6");
   print("\n\nTest 2 - DES CBC 64-bit:");
   des_test($mode);

   $mode = MCRYPT_MODE_CFB; # 8-bit
   $expected = pack("H*",
      "F31FDA07011462EE187F43D80A7CD9B5B0D290DA6E5B9A87");
   print("\n\nTest 3 - DES CFB 8-bit:");
   des_test($mode);

   $mode = MCRYPT_MODE_OFB; # 8-bit
   $expected = pack("H*",
      "F34A2850C9C64985D684AD96D772E2F243EA499ABEE8AE95");
   print("\n\nTest 4 - DES OFB 8-bit:");
   des_test($mode);

function des_test($mode) {
   global $key, $iv, $plain_text, $expected;
   $cipher = mcrypt_module_open(MCRYPT_DES, '', $mode, '');
   mcrypt_generic_init($cipher, $key, $iv);
   $cipher_text = mcrypt_generic($cipher, $plain_text);
   mcrypt_generic_deinit($cipher);
   mcrypt_module_close($cipher);
   print("\nThe key        : ".bin2Hex($key));
   print("\nThe IV         : ".bin2Hex($iv));
   print("\nThe plaintext : ".bin2Hex($plain_text));
   print("\nThe ciphertext: ".bin2Hex($cipher_text));
   print("\nThe expected  : ".bin2Hex($expected));
}
?>
```

If you run this script, you should get:

```

Test 1 - DES ECB 64-bit:
The key       : 0123456789abcdef
The IV        : 1234567890abcdef
The plaintext : 4e6f77206973207468652074696d6520666f7220616c6c20
The ciphertext: 3fa40e8a984d48156a271787ab8883f9893d51ec4b563b53
The expected  : 3fa40e8a984d43156a271787ab8883f9893d51ec4b563b53

Test 2 - DES CBC 64-bit:
The key       : 0123456789abcdef
The IV        : 1234567890abcdef
The plaintext : 4e6f77206973207468652074696d6520666f7220616c6c20
The ciphertext: e5c7cdde872bf27c43e934008c389c0f683788499a7c05f6
The expected  : e5c7cdde872bf27c43e934008c389c0f683788499a7c05f6

Test 3 - DES CFB 8-bit:
The key       : 0123456789abcdef
The IV        : 1234567890abcdef
The plaintext : 4e6f77206973207468652074696d6520666f7220616c6c20
The ciphertext: f31fda07011462ee187f43d80a7cd9b5b0d290da6e5b9a87
The expected  : f31fda07011462ee187f43d80a7cd9b5b0d290da6e5b9a87

Test 4 - DES OFB 8-bit:
The key       : 0123456789abcdef
The IV        : 1234567890abcdef
The plaintext : 4e6f77206973207468652074696d6520666f7220616c6c20
The ciphertext: f34a2850c9c64985d684ad96d772e2f243ea499abee8ae95
The expected  : f34a2850c9c64985d684ad96d772e2f243ea499abee8ae95
```

Output looks very good. All 4 test cases match the expected ciphertext message documented in http://www.itl.nist.gov/fipspubs/fip81.htm.

Block Padding in mcrypt

This section describes the padding schema used in PHP mcrypt library for MCRYPT_MODE_ECB and MCRYPT_MODE_CBC modes. mcrypt simply pads 0x00 to the end of the plaintext.

If you use mcrypt DES to encrypt a stream of bytes in MCRYPT_MODE_CFB or MCRYPT_MODE_OFB mode, you don't need to worry about padding the byte stream. Because MCRYPT_MODE_CFB and MCRYPT_MODE_OFB are 1-byte ciphers.

But if you use mcrypt DES to encrypt a stream of bytes in MCRYPT_MODE_ECB or MCRYPT_MODE_CBC mode, you have to worry about padding the byte stream. Because MCRYPT_MODE_ECB and MCRYPT_MODE_CBC are 8-byte ciphers.

"mycypt" offers a very simple padding method which simply adds 0x00 to the end of the plaintext message to make it multiples of 8 bytes (64 bits) for MCRYPT_MODE_ECB and MCRYPT_MODE_CBC modes.

This will create problems if you encrypt a message with PHP mcrypt in MCRYPT_MODE_ECB mode and send it to someone else who tries to decrypt it with a JDK JCE program. Because the JDK JCE package uses the DES/ECB/PKCS5Padding mode by default. The last block will fail to decrypt because the padding schema is different.

So if you want exchange DES encrypted messages with a Java programmer, you need to handle the PKCS5Padding padding yourself as part of your PHP script. See the PKCS5Padding schema section for detailed information.

Blowfish - 8-Byte Block Cipher

This chapter provides tutorial examples and notes about Blowfish block cipher. Topics include Blowfish encryption and decryption algorithm; Blowfish sub-key generation schema, Blowfish algorithm implementation in Java.

Conclusion

- Blowfish is a 64-bit block cipher.
- 18 sub-keys are derived from a single initial key.
- Decryption algorithm is identical to the encryption algorithm except for the order of the round keys.
- Markus Hahn provided a very efficient implementation, BlowfishJ.

Exercise: Now the algorithm is here, and the required initialization data is here, you can try to develop your own implementation in Java.

What Is Block Cipher?

This section describes what is block cipher - An encryption scheme in which 'the clear text is broken up into blocks of fixed length, and encrypted one block at a time'.

Block Cipher - An encryption scheme that "the clear text is broken up into blocks of fixed length, and encrypted one block at a time".

Usually, a block cipher encrypts a block of clear text into a block of cipher text of the same length. In this case, a block cipher can be viewed as a simple substitute cipher with character size equal to the block size.

ECB Operation Mode - Blocks of clear text are encrypted independently. ECB stands for Electronic Code Book. Main properties of this mode:

- Identical clear text blocks are encrypted to identical cipher text blocks.
- Re-ordering clear text blocks results in re-ordering cipher text blocks.
- An encryption error affects only the block where it occurs.

CBC Operation Mode - The previous cipher text block is XORed with the clear text block before applying the encryption mapping. Main properties of this mode:

- An encryption error affects only the block where is occurs and one next block.

Product Cipher - An encryption scheme that "uses multiple ciphers in which the cipher text of one cipher is used as the clear text of the next cipher". Usually, substitution ciphers and transposition ciphers are used alternatively to construct a product cipher.

Iterated Block Cipher - A block cipher that "iterates a fixed number of times of another block cipher, called round function, with a different key, called round key, for each iteration".

Feistel Cipher - An iterate block cipher that uses the following algorithm:

```
Input:
   T: 2t bits of clear text
   k1, k2, ..., kr: r round keys
   f: a block cipher with bock size of t

Output:
   C: 2t bits of cipher text

Algorithm:
   (L0, R0) = T, dividing T in two t-bit parts
   (L1, R1) = (R0, L0 XOR f(R0, k1))
   (L2, R2) = (R1, L1 XOR f(R1, k2))
   ......
   C = (Rr, Lr), swapping the two parts
```

Blowfish Cipher Algorithm

This section describes the Blowfish cipher algorithm - A 16-round Feistel cipher with block size of 64 bits developed by Bruce Schneier in 1993.

Blowfish - A 16-round Feistel cipher with block size of 64 bits.

Blowfish was developed by Bruce Schneier in 1993. Here is the algorithm presented in Bruce's paper:

```
Input:
   T: 64 bits of clear text
   P1, P2, ..., P18: 18 sub-keys
   F(): Round function

Output:
   C: 64 bits of cipher text

Algorithm:
   (xL, xR) = T, dividing T into two 32-bit parts
   Loop on i from = 1 to 16
      xL = xL XOR Pi
      xR = F(xL) XOR xR
      Swap xL and xR
   End of loop
   Swap xL and xR
   xR = xR XOR P17
   xL = xL XOR P18
   C = (xL, xR)
```

The round function F(A) is defined as:

```
Input:
   A: 32-bit input data
   S1[], S2[], S3[], S4[]: 4 S-box lookup tables, 256 long each

Output
   B = f(A): 32-bit output data

Algorithm
   (a, b, c, d) = A, dividing L into four 8-bit parts
   B = ( (S1[a] + S2[b] mod 2**32) XOR S3[c] ) + S[d] mod 2**32
```

Key Schedule (Sub-Keys Generation) Algorithm

This section describes the Blowfish Key Schedule (Sub-Keys Generation) algorithm. It uses the binary representation of the fractional portion of constant Pi - 3.1415927... as initial values.

Blowfish key schedule algorithm:

```
Input:
   K: The key - 32 bits or more
   PI: The binary representation of the fractional portion of "pi"
      = 3.1415927... - 3.0
      = 2/16 + 4*/16**2 + 3/16**3 + 15/16**4 + ...
      = 0x243f6a8885a308d313198a2e03707344...

Output:
   P1, P2, ..., P18: 18 32-bit sub-keys
   S1[], S2[], S3[], S4[]: 4 S-boxes, 32-bit 256-element arrays

Algorithm:
   (P1, P2, ..., P18, S1[], S2[], S3[], S4[]) = PI
   K' = (K, K, K, ...), Repeat the key to 18*32 bits long
   (P1, P2, ..., P18) = (P1, P2, ..., P18) XOR K'
   T = (0x000000, 0x000000), Setting initial clear text
   T = Blowfish(T), Applying Blowfish algorithm
   (P1, P2) = T, Updating first two sub-keys
   T = Blowfish(T), Applying Blowfish again
   (P3, P4) = T
   ......
   T = Blowfish(T)
   (P17, P18) = T
   T = Blowfish(T)
   (S1[0], S1[1]) = T
   T = Blowfish(T)
   (S1[2], S1[3]) = T
   ......
   T = Blowfish(T)
   (S1[254], S1[255]) = T
   T = Blowfish(T)
   (S2[0], S2[1]) = T
   ......
   T = Blowfish(T)
   (S4[254], S4[255]) = T
```

Note that:

- To initialize 18 sub-keys and 4 S tables, we need 18*32 + 4*256*32 = 576 + 32768 = 33344 binary digits of PI, or 33344/4 = 8366 hex digits of PI. See the last section of this chapter for the complete list of 8366 hex digits.
- To finalize 18 sub-keys and 4 S tables, we need to apply Blowfish algorithm (18+4*256)/2 = 1042/2 = 521 times.

BlowfishJ - Java Implementation by Markus Hahn

This section describes BlowfishJ - Java implementation of Blowfish by Markus Hahn.

One of the most popular Java implementation of Blowfish cipher is BlowfishJ, developed by Markus Hahn, http://come.to/hahn. Markus used a very efficient form of Blowfish algorithm:

```
Input:
   T: 64 bits of clear text
   P1, P2, ..., P18: 18 sub-keys
   F(): Round function

Output:
   C: 64 bits of cipher text

Algorithm:
   (L, R) = T, dividing T into two 32-bit parts
   L = L XOR P1
   R = R XOR F(L) XOR P2
   L = L XOR F(R) XOR P3
   R = R XOR F(L) XOR P4
   L = L XOR F(R) XOR P5
   R = R XOR F(L) XOR P6
   L = L XOR F(R) XOR P7
   R = R XOR F(L) XOR P8
   L = L XOR F(R) XOR P9
   R = R XOR F(L) XOR P10
   L = L XOR F(R) XOR P11
   R = R XOR F(L) XOR P12
   L = L XOR F(R) XOR P13
   R = R XOR F(L) XOR P14
   L = L XOR F(R) XOR P15
   R = R XOR F(L) XOR P16
   L = L XOR F(R) XOR P17
   R = R XOR P18
   C = (R, L)
```

Blowfish Decryption Algorithm

This section describes the Blowfish decryption algorithm, which is identical to the encryption algorithm step by step in the same order, only with the sub-keys applied in the reverse order.

The decryption algorithm of a block cipher should be identical to encryption algorithm step by step in reverse order. But for Blowfish cipher, the encryption algorithm is so well designed, that the decryption algorithm is identical to the encryption algorithm step by step in the same order, only with the sub-keys applied in the reverse order.

To help us to approve the decryption algorithm, we have to write the encryption algorithm and the decryption algorithm with temporary variables.

Encryption algorithm with temporary variables:

```
Input:
   T: 64 bits of clear text
   P1, P2, ..., P18: 18 sub-keys
   F(): Round function

Output:
   C: 64 bits of cipher text

Algorithm:
   (L0, R0) = T, dividing T into two 32-bit parts
   L1 = L0 XOR P1
   R2 = R0 XOR F(L1) XOR P2
   L3 = L1 XOR F(R2) XOR P3
   R4 = R2 XOR F(L3) XOR P4
   L5 = L3 XOR F(R4) XOR P5
   R6 = R4 XOR F(L5) XOR P6
   L7 = L5 XOR F(R6) XOR P7
   R8 = R6 XOR F(L7) XOR P8
   L9 = L7 XOR F(R8) XOR P9
   R10 = R8 XOR F(L9) XOR P10
   L11 = L9 XOR F(R10) XOR P11
   R12 = R10 XOR F(L11) XOR P12
   L13 = L11 XOR F(R12) XOR P13
   R14 = R12 XOR F(L13) XOR P14
   L15 = L13 XOR F(R14) XOR P15
   R16 = R14 XOR F(L15) XOR P16
   L17 = L15 XOR F(R16) XOR P17
   R18 = R16 XOR P18
   C = (R18, L17)
```

Decryption algorithm with temporary variables:

```
Input:
   CC: 64 bits of cipher text
   P1, P2, ..., P18: 18 sub-keys
   F(): Round function
```

```
Output:
   TT: 64 bits of clear text

Algorithm:
   (LL0, RR0) = CC, dividing CC into two 32-bit parts
   LL1 = LL0 XOR P18
   RR2 = RR0 XOR F(LL1) XOR P17
   LL3 = LL1 XOR F(RR2) XOR P16
   RR4 = RR2 XOR F(LL3) XOR P15
   LL5 = LL3 XOR F(RR4) XOR P14
   RR6 = RR4 XOR F(LL5) XOR P13
   LL7 = LL5 XOR F(RR6) XOR P12
   RR8 = RR6 XOR F(LL7) XOR P11
   LL9 = LL7 XOR F(RR8) XOR P10
   RR10 = RR8 XOR F(LL9) XOR P9
   LL11 = LL9 XOR F(RR10) XOR P8
   RR12 = RR10 XOR F(LL11) XOR P7
   LL13 = LL11 XOR F(RR12) XOR P6
   RR14 = RR12 XOR F(LL13) XOR P5
   LL15 = LL13 XOR F(RR14) XOR P4
   RR16 = RR14 XOR F(LL15) XOR P3
   LL17 = LL15 XOR F(RR16) XOR P2
   RR18 = R16 XOR P1
   TT = (RR18, LL17)
```

Here is how to approve the decryption algorithm:

```
Let:
   T: 64 bits of clear text
   C: 64 bits of cipher text encrypted from T
   CC: 64 bits of cipher text
   TT: 64 bits of clear text decrypted from CC

If:
   CC = C

Then:
   TT = T

Prove:
   (LL0, RR0) = CC            Initializing step in decryption
      = C                     Assumption of CC = C
      = (R18, L17)            Finalizing step in encryption

   LL1 = LL0 XOR P18          Applying P18 in decryption
      = R18 XOR P18           Previous result
      = R16 XOR P18 XOR P18   Applying P18 in encryption
      = R16

   RR2 = RR0 XOR F(LL1) XOR P17
                              Applying P17 in decryption
      = L17 XOR F(R16) XOR P17
                              Previous result
      = L15 XOR F(R16) XOR P17 XOR F(R16) XOR P17
                              Applying P17 in encryption
      = L15

   ......

   LL17 = LL15 XOR F(RR16) XOR P2
                              Applying P2 in decryption
      = R2 XOR F(L1) XOR P2
                              Previous result
      = R0 XOR F(L1) XOR P2 XOR F(L1) XOR P2
                              Applying P2 in encryption
      = R0

   RR18 = RR16 XOR P1         Applying P1 in decryption
      = L1 XOR P1             Previous result
      = L0 XOR P1 XOR P1      Applying P1 in encryption
      = L0

   TT = (RR18, LL17)          Finalizing step in decryption
      = (L0, R0)              Initializing step in encryption
      = T
```

First 8366 Hex Digits of PI

This section provides first 8366 hex digits of constant PI, 3.1415927..., needed for Blowfish encryption algorithm.

Here are the first 8366 hex digits of the fractional portion of constant "pi", 3.1415927..., needed for Blowfish encryption algorithm. I extracted then from the source code provided in BlowfishJ by Markus Hahn.

```
243F6A8885A308D313198A2E03707344A4093822299F31D0082EFA98EC4E6C89
452821E638D01377BE5466CF34E90C6CC0AC29B7C97C50DD3F84D5B5B5470917
9216D5D98979FB1BD1310BA698DFB5AC2FFD72DBD01ADFB7B8E1AFED6A267E96
BA7C9045F12C7F9924A19947B3916CF70801F2E2858EFC16636920D871574E69
A458FEA3F4933D7E0D95748F728EB658718BCD5882154AEE7B54A41DC25A59B5
9C30D5392AF26013C5D1B023286085F0CA417918B8DB38EF8E79DCB0603A180E
6C9E0E8BB01E8A3ED71577C1BD314B2778AF2FDA55605C60E65525F3AA55AB94
5748986263E8144055CA396A2AAB10B6B4CC5C341141E8CEA15486AF7C72E993
B3EE1411636FBC2A2BA9C55D741831F6CE5C3E169B87931EAFD6BA336C24CF5C
7A325381289586773B8F48986B4BB9AFC4BFE81B6628219361D809CCFB21A991
487CAC605DEC8032EF845D5DE98575B1DC262302EB651B8823893E81D396ACC5
0F6D6FF383F442392E0B4482A484200469C8F04A9E1F9B5E21C66842F6E96C9A
670C9C61ABD388F06A51A0D2D8542F68960FA728AB5133A36EEF0B6C137A3BE4
BA3BF0507EFB2A98A1F1651D39AF017666CA593E82430E888CEE8619456F9FB4
7D84A5C33B8B5EBEE06F75D885C12073401A449F56C16AA64ED3AA62363F7706
1BFEDF72429B023D37D0D724D00A1248DB0FEAD349F1C09B075372C980991B7B
25D479D8F6E8DEF7E3FE501AB6794C3B976CE0BD04C006BAC1A94FB6409F60C4
5E5C9EC2196A246368FB6FAF3E6C53B51339B2EB3B52EC6F6DFC511F9B30952C
CC814544AF5EBD09BEE3D004DE334AFD660F2807192E4BB3C0CBA85745C8740F
D20B5F39B9D3FBDB5579C0BD1A60320AD6A100C6402C7279679F25FEFB1FA3CC
8EA5E9F8DB3222F83C7516DFFD616B152F501EC8AD0552AB323DB5FAFD238760
53317B483E00DF829E5C57BBCA6F8CA01A87562EDF1769DBD542A8F6287EFFC3
AC6732C68C4F5573695B27B0BBCA58C8E1FFA35DB8F011A010FA3D98FD2183B8
4AFCB56C2DD1D35B9A53E479B6F84565D28E49BC4BFB9790E1DDF2DAA4CB7E33
62FB1341CEE4C6E8EF20CADA36774C01D07E9EFE2BF11FB495DBDA4DAE909198
EAAD8E716B93D5A0D08ED1D0AFC725E08E3C5B2F8E7594B78FF6E2FBF2122B64
8888B812900DF01C4FAD5EA0688FC31CD1CFF191B3A8C1AD2F2F2218BE0E1777
EA752DFE8B021FA1E5A0CC0FB56F74E818ACF3D6CE89E299B4A84FE0FD13E0B7
7CC43B81D2ADA8D9165FA2668095770593CC7314211A1477E6AD206577B5FA86
C75442F5FB9D35CFEBCDAF0C7B3E89A0D6411BD3AE1E7E4900250E2D2071B35E
226800BB57B8E0AF2464369BF009B91E5563911D59DFA6AA78C14389D95A537F
207D5BA202E5B9C5832603766295CFA911C819684E734A41B3472DCA7B14A94A
1B5100529A532915D60F573FBC9BC6E42B60A47681E6740008BA6FB5571BE91F
```

```
F296EC6B2A0DD915B6636521E7B9F9B6FF34052EC585566453B02D5DA99F8FA1
08BA47996E85076A4B7A70E9B5B32944DB75092EC4192623AD6EA6B049A7DF7D
9CEE60B88FEDB266ECAA8C71699A17FF5664526CC2B19EE1193602A575094C29
A0591340E4183A3E3F54989A5B429D656B8FE4D699F73FD6A1D29C07EFE830F5
4D2D38E6F0255DC14CDD20868470EB266382E9C6021ECC5E09686B3F3EBAEFC9
3C9718146B6A70A1687F358452A0E286B79C5305AA5007373E07841C7FDEAE5C
8E7D44EC5716F2B8B03ADA37F0500C0DF01C1F040200B3FFAE0CF51A3CB574B2
25837A58DC0921BDD19113F97CA92FF69432477322F547013AE5E58137C2DADC
C8B576349AF3DDA7A94461460FD0030EECC8C73EA4751E41E238CD993BEA0E2F
3280BBA1183EB3314E548B384F6DB9086F420D03F60A04BF2CB8129024977C79
5679B072BCAF89AFDE9A771FD9930810B38BAE12DCCF3F2E5512721F2E6B7124
501ADDE69F84CD877A5847187408DA17BC9F9ABCE94B7D8CEC7AEC3ADB851DFA
63094366C464C3D2EF1C18473215D908DD433B3724C2BA1612A14D432A65C451
50940002133AE4DD71DFF89E10314E5581AC77D65F11199B043556F1D7A3C76B
3C11183B5924A509F28FE6ED97F1FBFA9EBABF2C1E153C6E86E34570EAE96FB1
860E5E0A5A3E2AB3771FE71C4E3D06FA2965DCB999E71D0F803E89D65266C825
2E4CC9789C10B36AC6150EBA94E2EA78A5FC3C531E0A2DF4F2F74EA7361D2B3D
1939260F19C279605223A708F71312B6EBADFE6EEAC31F66E3BC4595A67BC883
B17F37D1018CFF28C332DDEFBE6C5AA56558218568AB9802EECEA50FDB2F953B
2AEF7DAD5B6E2F841521B62829076170ECDD4775619F151013CCA830EB61BD96
0334FE1EAA0363CFB5735C904C70A239D59E9E0BCBAADE14EECC86BC60622CA7
9CAB5CABB2F3846E648B1EAF19BDF0CAA02369B9655ABB5040685A323C2AB4B3
319EE9D5C021B8F79B540B19875FA09995F7997E623D7DA8F837889A97E32D77
11ED935F166812810E358829C7E61FD696DEDFA17858BA9957F584A51B227263
9B83C3FF1AC24696CDB30AEB532E30548FD948E46DBC312858EBF2EF34C6FFEA
FE28ED61EE7C3C735D4A14D9E864B7E342105D14203E13E045EEE2B6A3AAABEA
DB6C4F15FACB4FD0C742F442EF6ABBB5654F3B1D41CD2105D81E799E86854DC7
E44B476A3D816250CF62A1F25B8D2646FC8883A0C1C7B6A37F1524C369CB7492
47848A0B5692B285095BBF00AD19489D1462B17423820E0058428D2A0C55F5EA
1DADF43E233F70613372F0928D937E41D65FECF16C223BDB7CDE3759CBEE7460
4085F2A7CE77326EA607808419F8509EE8EFD85561D99735A969A7AAC50C06C2
5A04ABFC800BCADC9E447A2EC3453484FDD567050E1E9EC9DB73DBD3105588CD
675FDA79E3674340C5C43465713E38D83D28F89EF16DFF20153E21E78FB03D4A
E6E39F2BDB83ADF7E93D5A68948140F7F64C261C94692934411520F77602D4F7
BCF46B2ED4A20068D40824713320F46A43B7D4B7500061AF1E39F62E97244546
14214F74BF8B88404D95FC1D96B591AF70F4DDD366A02F45BFBC09EC03BD9785
7FAC6DD031CB850496EB27B355FD3941DA2547E6ABCA0A9A28507825530429F4
0A2C86DAE9B66DFB68DC1462D7486900680EC0A427A18DEE4F3FFEA2E887AD8C
B58CE0067AF4D6B6AACE1E7CD3375FECCE78A399406B2A4220FE9E35D9F385B9
EE39D7AB3B124E8B1DC9FAF74B6D185626A36631EAE397B23A6EFA74DD5B4332
6841E7F7CA7820FBFB0AF54ED8FEB397454056ACBA48952755533A3A20838D87
FE6BA9B7D096954B55A867BCA1159A58CCA9296399E1DB33A62A4A563F3125F9
5EF47E1C9029317CFDF8E80204272F7080BB155C05282CE395C11548E4C66D22
48C1133FC70F86DC07F9C9EE41041F0F404779A45D886E17325F51EBD59BC0D1
F2BCC18F41113564257B7834602A9C60DFF8E8A31F636C1B0E12B4C202E1329E
AF664FD1CAD181156B2395E0333E92E13B240B62EEBEB92285B2A20EE6BA0D99
DE720C8C2DA2F728D012784595B794FD647D0862E7CCF5F05449A36F877D48FA
C39DFD27F33E8D1E0A476341992EFF743A6F6EABF4F8FD37A812DC60A1EBDDF8
991BE14CDB6E6B0DC67B55106D672C372765D43BDCD0E804F1290DC7CC00FFA3
B5390F92690FED0B667B9FFBCEDB7D9CA091CF0BD9155EA3BB132F88515BAD24
7B9479BF763BD6EB37392EB3CC1159798026E297F42E312D6842ADA7C66A2B3B
12754CCC782EF11C6A124237B79251E706A1BBE64BFB63501A6B101811CAEDFA
3D25BDD8E2E1C3C9444216590A121386D90CEC6ED5ABEA2A64AF674EDA86A85F
BEBFE98864E4C3FE9DBC8057F0F7C08660787BF86003604DD1FD8346F6381FB0
7745AE04D736FCCC83426B33F01EAB71B08041873C005E5F77A057BEBDE8AE24
55464299BF582E614E58F48FF2DDFDA2F474EF388789BDC25366F9C3C8B38E74
B475F25546FCD9B97AEB26618B1DDF84846A0E79915F95E2466E598E20B45770
8CD55591C902DE4CB90BACE1BB8205D011A862487574A99EB77F19B6E0A9DC09
662D09A1C4324633E85A1F0209F0BE8C4A99A0251D6EFE101AB93D1D0BA5A4DF
A186F20F2868F169DCB7DA83573906FEA1E2CE9B4FCD7F5250115E01A70683FA
A002B5C40DE6D0279AF88C27773F8641C3604C0661A806B5F0177A28C0F586E0
006058AA30DC7D6211E69ED72338EA6353C2DD94C2C21634BBCBEE5690BCB6DE
```

```
EBFC7DA1CE591D766F05E4094B7C018839720A3D7C927C2486E3725F724D9DB9
1AC15BB4D39EB8FCED54557808FCA5B5D83D7CD34DAD0FC41E50EF5EB161E6F8
A28514D96C51133C6FD5C7E756E14EC4362ABFCEDDC6C837D79A323492638212
670EFA8E406000E03A39CE37D3FAF5CFABC277375AC52D1B5CB0679E4FA33742
D382274099BC9BBED5118E9DBF0F7315D62D1C7EC700C47BB78C1B6B21A19045
B26EB1BE6A366EB45748AB2FBC946E79C6A376D26549C2C8530FF8EE468DDE7D
D5730A1D4CD04DC62939BBDBA9BA4650AC9526E8BE5EE304A1FAD5F06A2D519A
63EF8CE29A86EE22C089C2B843242EF6A51E03AA9CF2D0A483C061BA9BE96A4D
8FE51550BA645BD62826A2F9A73A3AE14BA99586EF5562E9C72FEFD3F752F7DA
3F046F6977FA0A5980E4A91587B086019B09E6AD3B3EE593E990FD5A9E34D797
2CF0B7D9022B8B5196D5AC3A017DA67DD1CF3ED67C7D2D281F9F25CFADF2B89B
5AD6B4725A88F54CE029AC71E019A5E647B0ACFDED93FA9BE8D3C48D283B57CC
F8D5662979132E28785F0191ED756055F7960E44E3D35E8C15056DD488F46DBA
03A161250564F0BDC3EB9E153C9057A297271AECA93A072A1B3F6D9B1E6321F5
F59C66FB26DCF3197533D928B155FDF5035634828ABA3CBB28517711C20AD9F8
ABCC5167CCAD925F4DE817513830DC8E379D58629320F991EA7A90C2FB3E7BCE
5121CE64774FBE32A8B6E37EC3293D4648DE53696413E680A2AE0810DD6DB224
69852DFD09072166B39A460A6445C0DD586CDECF1C20C8AE5BBEF7DD1B588D40
CCD2017F6BB4E3BBDDA26A7E3A59FF453E350A44BCB4CDD572EACEA8FA6484BB
8D6612AEBF3C6F47D29BE463542F5D9EAEC2771BF64E6370740E0D8DE75B1357
F8721671AF537D5D4040CB084EB4E2CC34D2466A0115AF84E1B0042895983A1D
06B89FB4CE6EA0486F3F3B823520AB82011A1D4B277227F8611560B1E7933FDC
BB3A792B344525BDA08839E151CE794B2F32C9B7A01FBAC9E01CC87EBCC7D1F6
CF0111C3A1E8AAC71A908749D44FBD9AD0DADECBD50ADA380339C32AC6913667
8DF9317CE0B12B4FF79E59B743F5BB3AF2D519FF27D9459CBF97222C15E6FC2A
0F91FC719B941525FAE59361CEB69CEBC2A8645912BAA8D1B6C1075EE3056A0C
10D25065CB03A442E0EC6E0E1698DB3B4C98A0BE3278E9649F1F9532E0D392DF
D3A0342B8971F21E1B0A74414BA3348CC5BE7120C37632D8DF359F8D9B992F2E
E60B6F470FE3F11DE54CDA541EDAD891CE6279CFCD3E7E6F1618B166FD2C1D05
848FD2C5F6FB2299F523F357A632762393A8353156CCCD02ACF081625A75EBB5
6E16369788D273CCDE96629281B949D04C50901B71C65614E6C6C7BD327A140A
45E1D006C3F27B9AC9AA53FD62A80F00BB25BFE235BDD2F671126905B2040222
B6CBCF7CCD769C2B53113EC01640E3D338ABBD602547ADF0BA38209CF746CE76
77AFA1C52075606085CBFE4E8AE88DD87AAAF9B04CF9AA7E1948C25C02FB8A8C
01C36AE4D6EBE1F990D4F869A65CDEA03F09252DC208E69FB74E6132CE77E25B
578FDFE33AC372E6
```

Secret Key Generation and Management

This chapter provides tutorial notes and example codes on secret keys. Topics include secret keys for Blowfish, DES, or HmacMD5 encryption algorithms; key generator class; test program to generate secret keys and save them in files.

Conclusions:

- Secret keys are used for symmetric encryption algorithms like Blowfish, DES, or HmacMD5.
- The javax.crypto.KeyGenerator can be used to generate secret keys.
- Secret keys are usually stored in files in raw format.

Sample programs listed in this chapter have been tested with JDK 1.3 to 1.6.

javax.crypto.SecretKey - The Secret Key Interface

This section provides a quick introduction of secret key and symmetric encryption algorithm. The secret key interface, javax.crypto.SecretKey, is also described.

What is a **Secret Key**? A secrete key is the key used in a symmetric encryption algorithm, where the same key is used both the encryption process and the decryption process.

Known symmetric encryption algorithms:

- Blowfish - The block cipher designed by Bruce Schneier.
- DES - The Digital Encryption Standard as described in FIPS PUB 46-2.
- DESede - Triple DES Encryption (DES-EDE).

The secrect key concept is supported in JDK through the JCE (Java Cryptography Extension) package. The first thing I want to learn in JCE is the javax.crypto.SecretKey interface.

javax.crypto.SecretKey is an interface providing a grouping point for various secret keys. It extents java.security.Key, and inherits 3 methods:

getAlgorithm() - Returns the algorithm name used to generate the key.

getEncoded() - Returns the key as a byte array in its primary encoding format.

getFormat() - Returns the name of the primary encoding format of this key.

Last update: 2013.

javax.crypto.KeyGenerator - Generating Secret Keys

This section provides a quick introduction of the secret key generation class, javax.crypto.KeyGenerator.

To generate secret keys, I need to learn the javax.crypto.KeyGenerator class.

javax.crypto.KeyGenerator is an abstract class providing a link to implementation classes of secret key generration algorithms provided by various security package providers. Major methods in the KeyGenerator class:

getInstance() - Returns a KeyGenerator object of the specified algorithm from the implementation of the specified provider. If provider is not specified, the default implementation is used. This is a static method.

init() - Initializes this key generator with the specified key size.

generateKey() - Generates a key and returns it as a SecretKey object.

getAlgorithm() - Returns the algorithm name of this generator.

getProvider() - Returns the provider as a Provider object of this generator.

Steps of generating a secret key of a given algorithm are:

```
      KeyGenerator kg = KeyGenerator.getInstance(algorithm);
      kg.init(keySize);
      SecretKey ky = kg.generateKey();
```

See next sections for a sample program on how to use the KeyGenerator class.

Last update: 2013.

Converting Secret Keys to and from Byte Arrays

This section provides a quick introduction of the SecretKeySpec class and the KeySpec interface. They can be used to convert secret keys into byte arrays to store them in external files.

Converting a secret key to a byte array is supported by the SecretKey interface with the getEncoded() method. All secret key implementation classes use the RAW encoding format.

Converting a secret key from a byte array is not so easy. Two options are available:

- Using the generic SecretKeySpec class to convert a byte array to a SecretKey object directly.
- Using algorithm specific KeySpec implementations to convert a byte array to a KeySpec object first, then using SecretKeyFactory to convert a KeySpec object to a SecretKey object.

1. javax.crypto.spec.SecretKeySpec is a class represents a secret key in a generic fashion. It offers the following constructor and methods:

SecretKeySpec() - Convert a secret key from the specified byte array according to the specified algorithm and constructs a SecretKeySpec object based on the secret key.

getAlgorithm() - Returns the algorithm name used to generate the key.

getEncoded() - Returns the key as a byte array in its primary encoding format.

getFormat() - Returns the name of the primary encoding format of this key.

2. java.security.spec.KeySpec is an interface providing a grouping point for algorithm specific key specification implementations. There are two implementations for secret key algorithms:

javax.crypto.spec.DESKeySpec is a KeySpec implementation for DES algorithm. It offers DESKeySpec() to construct a KeySpec object from the specific byte array.

javax.crypto.spec.DESedeKeySpec is a KeySpec implementation for DESede algorithm. It offers DESedeKeySpec() to construct a KeySpec object from the specific byte array.

javax.crypto.spec.SecretKeyFactory is a class as a conversion tool between SecretKey objects and algorithm specific KeySpec objects. It offers the following methods:

getInstance() - Returns a SecretKeyFactory object of the specified algorithm and the specified security package provider. If not specified, the default security pacage provider will be used.

generateSecret() - Generates a SecretKey object from the specified KeySpec object, and returns it.

getKeySpec() - Converts a SecretKey object to a KeySpec object, and returns it.

getAlgorithm() - Returns the algorithm name of this object.

getProvider() - Returns the provider as a provider object of this object.

See the next section on how to use the SecretKeySpec class and the KeySpec interface in a sample program.

Last update: 2013.

JceSecretKeyTest.java - Secret Key Test Program

This section provides a quick tutorial example on how to write a sample program to generate a secret key for Blowfish, DES, or HmacMD5 encryption, save the secret key to a file, and read it back.

The following sample program shows you how to generate a secret key, write it a file, and read it back.

```
/**
 * JceSecretKeyTest.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.io.*;
import java.security.spec.*;
import javax.crypto.*;
import javax.crypto.spec.*;
class JceSecretKeyTest {
   public static void main(String[] a) {
      if (a.length<3) {
         System.out.println("Usage:");
         System.out.println("java JceSecretKeyTest keySize output"
            +" algorithm");
         return;
      }
      int keySize = Integer.parseInt(a[0]);
      String output = a[1];
      String algorithm = a[2]; // Blowfish, DES, DESede, HmacMD5
      try {
         writeKey(keySize,output,algorithm);
         readKey(output,algorithm);
      } catch (Exception e) {
         System.out.println("Exception: "+e);
         return;
      }
   }
   private static void writeKey(int keySize, String output,
         String algorithm) throws Exception {
      KeyGenerator kg = KeyGenerator.getInstance(algorithm);
      kg.init(keySize);
      System.out.println();
      System.out.println("KeyGenerator Object Info: ");
      System.out.println("Algorithm = "+kg.getAlgorithm());
      System.out.println("Provider = "+kg.getProvider());
      System.out.println("Key Size = "+keySize);
      System.out.println("toString = "+kg.toString());

      SecretKey ky = kg.generateKey();
      String fl = output+".key";
      FileOutputStream fos = new FileOutputStream(fl);
      byte[] kb = ky.getEncoded();
      fos.write(kb);
      fos.close();
      System.out.println();
      System.out.println("SecretKey Object Info: ");
      System.out.println("Algorithm = "+ky.getAlgorithm());
      System.out.println("Saved File = "+fl);
      System.out.println("Size = "+kb.length);
      System.out.println("Format = "+ky.getFormat());
      System.out.println("toString = "+ky.toString());
   }
   private static void readKey(String input, String algorithm)
      throws Exception {
      String fl = input+".key";
      FileInputStream fis = new FileInputStream(fl);
      int kl = fis.available();
      byte[] kb = new byte[kl];
      fis.read(kb);
      fis.close();
      KeySpec ks = null;
      SecretKey ky = null;
      SecretKeyFactory kf = null;
      if (algorithm.equalsIgnoreCase("DES")) {
               ks = new DESKeySpec(kb);
         kf = SecretKeyFactory.getInstance("DES");
         ky = kf.generateSecret(ks);
      } else if (algorithm.equalsIgnoreCase("DESede")) {
               ks = new DESedeKeySpec(kb);
         kf = SecretKeyFactory.getInstance("DESede");
         ky = kf.generateSecret(ks);
      } else {
         ks = new SecretKeySpec(kb,algorithm);
         ky = new SecretKeySpec(kb,algorithm);
      }

      System.out.println();
      System.out.println("KeySpec Object Info: ");
      System.out.println("Saved File = "+fl);
      System.out.println("Length = "+kb.length);
      System.out.println("toString = "+ks.toString());

      System.out.println();
      System.out.println("SecretKey Object Info: ");
      System.out.println("Algorithm = "+ky.getAlgorithm());
      System.out.println("toString = "+ky.toString());
   }
}
```

Here is the result of my first test. It is done with JDK 1.6.

```
java -cp . JceSecretKeyTest 56 key1 Blowfish

KeyGenerator Object Info:
Algorithm = Blowfish
Provider = SunJCE version 1.6
Key Size = 56
toString = javax.crypto.KeyGenerator@b09e89

SecretKey Object Info:
Algorithm = Blowfish
Saved File = key1.key
Size = 7
Format = RAW
toString = javax.crypto.spec.SecretKeySpec@2685016e

KeySpec Object Info:
Saved File = key1.key
Length = 7
```

```
toString = javax.crypto.spec.SecretKeySpec@2685016e

SecretKey Object Info:
Algorithm = Blowfish
toString = javax.crypto.spec.SecretKeySpec@2685016e
```

The program seems to be working:

- Since I am not specifying the provider name, the implementation of the Blowfish algorithm provided in the default security package was selected. Of course, Sun is the provider of the default security package.
- The Blowfish key is only 7 bytes when "encoded" in RAW format.
- When importing the blowfish key back from the 7 raw bytes, SecretKeyScep class is used instead of SecretKeyFactory class.

In the second test, I wants to try DES algorithm:

```
java -cp . JceSecretKeyTest 56 key2 DES

KeyGenerator Object Info:
Algorithm = DES
Provider = SunJCE version 1.6
Key Size = 56
toString = javax.crypto.KeyGenerator@b09e89

SecretKey Object Info:
Algorithm = DES
Saved File = key2.key
Size = 8
Format = RAW
toString = com.sun.crypto.provider.DESKey@fffe7965

KeySpec Object Info:
Saved File = key2.key
Length = 8
toString = javax.crypto.spec.DESKeySpec@a401c2

SecretKey Object Info:
Algorithm = DES
toString = com.sun.crypto.provider.DESKey@fffe7965
```

Of course, you can continue testing with DESede and HmacMD5.

Last update: 2013.

Cipher - Secret Key Encryption and Decryption

This chapter provides tutorial notes and example codes on the cipher process. Topics include the cipher process - data encryption and decryption; the cipher class, javax.crypto.Cipher; the cipher sample program and test results with Blowfish and DES encryption algorithms.

Conclusions:

- Secret keys are used for symmetric encryption algorithms like Blowfish, DES, or HmacMD5.
- The javax.crypto.Cipher class can be used to perform encryption or decryption on input data.

Sample programs listed in this chapter have been tested with JDK 1.3 to 1.6.

javax.crypto.Cipher - The Secret Key Encryption Class

This section provides a quick introduction of the cipher class, javax.crypto.Cipher, to encrypt input data with a secret key.

In the previous chapter, I learned how to use classes and interfaces in the JCE (Java Cryptography Extension) package to generate and manage secret keys. Now I am ready to learn how to use secret keys to encrypt or cipher any input data using the javax.crypto.Cipher class.

javax.crypto.Cipher is a class that provides functionality of a cryptographic cipher for encryption and decryption. It has the following major methods.

getInstance() - Returns a Cipher object of the specified transformation (algorithm plus options) from the implementation of the specified provider. If provider is not specified, the default implementation is used. This is a static method.

init() - Initializes this cipher for the specified operation mode with the specified key or public key certificate. Two important operation modes are Cipher.ENCRYPT_MODE and Cipher.DECRYPT_MODE.

update() - Feeds additional input data to this cipher and generates partial output data.

doFinal() - Feeds the last part of the input data to this cipher and generates the last part of the output data.

getBlockSize() - Returns the block size of this cipher.

getAlgorithm() - Returns the algorithm name of this cipher.

getProvider() - Returns the provider as a Provider object of this cipher.

See the next section how to use the javax.crypto.Cipher class in a sample program.

Last update: 2013.

JceSecretCipher.java - Secret Key Encryption Sample Program

This section provides a tutorial example on how to write a secret key encryption and description program using the javax.crypto.Cipher class.

The following sample program shows you how to do encryption and decryption with a secret key using the javax.crypto.Cipher class:

```
/**
 * JceSecretCipher.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.io.*;
import java.security.spec.*;
import javax.crypto.*;
import javax.crypto.spec.*;
class JceSecretCipher {
   public static void main(String[] a) {
      if (a.length<5) {
         System.out.println("Usage:");
         System.out.println("java JceSecretCipher algorithm mode"
            +" keyFile input output");
         return;
      }
      String algorithm = a[0];
      String mode = a[1];
      String keyFile = a[2];
      String input = a[3];
      String output = a[4];
      try {
         SecretKey ky = readKey(keyFile,algorithm);
         secretCipher(algorithm, mode, ky, input, output);
      } catch (Exception e) {
         System.out.println("Exception: "+e);
         return;
      }
   }
   private static SecretKey readKey(String input, String algorithm)
      throws Exception {
      String fl = input;
      FileInputStream fis = new FileInputStream(fl);
      int kl = fis.available();
      byte[] kb = new byte[kl];
      fis.read(kb);
      fis.close();
      KeySpec ks = null;
      SecretKey ky = null;
      SecretKeyFactory kf = null;
      if (algorithm.equalsIgnoreCase("DES")) {
               ks = new DESKeySpec(kb);
         kf = SecretKeyFactory.getInstance("DES");
         ky = kf.generateSecret(ks);
      } else if (algorithm.equalsIgnoreCase("DESede")) {
               ks = new DESedeKeySpec(kb);
         kf = SecretKeyFactory.getInstance("DESede");
         ky = kf.generateSecret(ks);
      } else {
         ks = new SecretKeySpec(kb,algorithm);
         ky = new SecretKeySpec(kb,algorithm);
      }
      System.out.println();
      System.out.println("KeySpec Object Info: ");
      System.out.println("Saved File = "+fl);
      System.out.println("Length = "+kb.length);
      System.out.println("toString = "+ks.toString());
      System.out.println();
      System.out.println("SecretKey Object Info: ");
      System.out.println("Algorithm = "+ky.getAlgorithm());
      System.out.println("toString = "+ky.toString());
      return ky;
   }
   private static void secretCipher(String algorithm, String mode,
      SecretKey ky, String input, String output) throws Exception {
      Cipher cf = Cipher.getInstance(algorithm);
      if (mode.equalsIgnoreCase("encrypt"))
         cf.init(Cipher.ENCRYPT_MODE,ky);
      else if (mode.equalsIgnoreCase("decrypt"))
         cf.init(Cipher.DECRYPT_MODE,ky);
      else
         throw new Exception("Invalid mode: "+mode);
      System.out.println();
      System.out.println("Cipher Object Info: ");
      System.out.println("Block Size = "+cf.getBlockSize());
      System.out.println("Algorithm = "+cf.getAlgorithm());
      System.out.println("Provider = "+cf.getProvider());
      System.out.println("toString = "+cf.toString());

      FileInputStream fis = new FileInputStream(input);
      FileOutputStream fos = new FileOutputStream(output);
      int bufSize = 1024;
      byte[] buf = new byte[bufSize];
      int n = fis.read(buf,0,bufSize);
      int fisSize = 0;
      int fosSize = 0;
      while (n!=-1) {
         fisSize += n;
         byte[] out = cf.update(buf,0,n);
         fosSize += out.length;
         fos.write(out);
         n = fis.read(buf,0,bufSize);
      }
      byte[] out = cf.doFinal();
      fosSize += out.length;
      fos.write(out);
      fis.close();
      fos.close();
      System.out.println();
      System.out.println("Cipher Process Info: ");
      System.out.println("Input Size = "+fisSize);
      System.out.println("Output Size = "+fosSize);
   }
}
```

Note that:

- This sample program requires a secret key file as described in the previous section.
- If want to, you could also create a secret key by yourself as any byte array. But the size of the key should meet the requirement of the encryption algorithm. For example, DES requires that the key to be 8 bytes long.

See the next section for test result of this sample program.

Last update: 2013.

Blowfish Secret Key Encryption Tests

This section provides the test result of Blowfish secret key encryption and decryption using the javax.crypto.Cipher class.

Here is the result of my first test of JceSecretCipher.java with the Blowfish algorithm. It is done with JDK 1.4.1. Here is the result of the encryption step:

```
java -cp . JceSecretCipher Blowfish encrypt bfish.key
   JceSecretCipher.java jce.cph

KeySpec Object Info:
Saved File = bfish.key
Length = 7
toString = javax.crypto.spec.SecretKeySpec@2685016e

SecretKey Object Info:
Algorithm = Blowfish
toString = javax.crypto.spec.SecretKeySpec@2685016e

Cipher Object Info:
Block Size = 8
Algorithm = Blowfish
Provider = SunJCE version 1.6
toString = javax.crypto.Cipher@b179c3

Cipher Process Info:
Input Size = 3684
Output Size = 3688
```

Here is the result of the decryption step:

```
java -cp . JceSecretCipher Blowfish decrypt bfish.key jce.cph jce.clr

KeySpec Object Info:
Saved File = bfish.key
Length = 7
toString = javax.crypto.spec.SecretKeySpec@2685016e
```

```
SecretKey Object Info:
Algorithm = Blowfish
toString = javax.crypto.spec.SecretKeySpec@2685016e

Cipher Object Info:
Block Size = 8
Algorithm = Blowfish
Provider = SunJCE version 1.6
toString = javax.crypto.Cipher@b179c3

Cipher Process Info:
Input Size = 3688
Output Size = 3684
```

Now checking the decryption output with the original text:

```
C:\herong>comp JceSecretCipher.java jce.clr
Comparing JceSecretCipher.java and jce.clr...
Files compare OK
```

Note that:

- bfish.key is the key file generated by KeyGenerator with the Blowfish algorithm.
- The last command confirms that the encryption process and the decryption process work correctly.
- The block size of Blowfish encryption algorithm is 8 bytes.

You can do more tests with DES and HmacMD5 keys.

Last update: 2013.

Introduction of RSA Algorithm

This chapter provides tutorial notes and example codes on RSA public key encryption algorithm. Topics include illustration of public key algorithm; proof of RSA encryption algorithm; security of public key; efficient way of calculating exponentiation and modulus; generating large prime numbers.

Conclusions:

- RSA (Rivest, Shamir and Adleman) uses public key and private key to encrypt and decrypt messages.
- RSA public keys are generated using 2 large prime numbers.
- RSA public key security is based the difficult level of factoring prime numbers. Given P and q to calculate n = p*q is easy. But given n to find p and q is very difficult.
- RSA encryption operation requires calculation of "C = M**e mod n", which can be done by loop of "C = C*C * M**e[i]".
- Most RSA tools are using public keys generated from large probable prime numbers, because generating large prime numbers is very expensive.

What Is Public Key Encryption?

This section describes public key encryption, also called asymmetric encryption, which uses a pair of keys, a private key and a public key to encrypt and decrypt messages.

What Is Public Key Encryption? Public key encryption is also called asymmetric encryption, which uses a pair of keys, a private key and a public key. Text encrypted by one key can be decrypted the other key.

Public key encryption can be used to secure a two-way communication.

For example, Bob wants to communicate with Susan in a secure way. He will generate a private key and a public key. He will encrypt the message with the private key, and send the encrypted message and the public key to Susan. Susan will then decrypt the message with the public key.

If Susan wants to send a response, she will encrypt the response with the public key and send the encrypted response back to Bob. Bob will then decrypt the response with the private key.

This communication process can be illustrated with the following diagram:

```
Bob          Bob              Internet       Susan          Susan

           | Encryption  |                 | Decryption |
Original   | with        |    Encrypted    | with       |    Original
message  ->| private key | -> message   -> | public key | -> message

           | Decryption  |                 | Encryption |
Original   | with        |    Encrypted    | with       |    Original
response <-| private key | <- response  <- | public key | <- response
```

The idea of public key encryption was invented in 1975 by two Stanford mathematicians, Whitfield Diffie and Martin Hellman. Diffie and Hellman failed to give any real algorithm and demonstrated only that public key encryption was possible in theory.

But one year later, in 1976, three MIT mathematicians, Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman invented a real public key algorithm. Their algorithm, called RSA algorithm, is based on factoring theory. See next section for more details.

Last update: 2013.

RSA Public Key Encryption Algorithm

This section describes the RSA public key encryption algorithm. Generating public and private keys used in RSA encryption requires two large prime numbers.

RSA public key encryption algorithm was invented in 1976 by three MIT mathematicians, Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. The name of the algorithm "RSA" represents the initials of their surnames.

The first part of the RSA algorithm is the public key and private key generation, which can be described as:

- Choose two distinct prime numbers p and q. For security purposes, the integers p and q should be chosen at random, and should be of similar bit-length of 1024 bits or higher

- Compute n = p*q. n is used as the modulus for both public and private keys. Its length, usually expressed in bits, is the key length.
- Compute m = (p-1)*(q-1). m is actually the Euler's totient function value of n.
- Choose an integer e such that 1 < e < n and greatest common divisor gcd(e, m) = 1; i.e., e and m are coprime numbers.
- Compute d such that d*e mod m = 1. d is also called the modular multiplicative inverse of e with modulo m.
- Package the public key as {n,e}.
- Package the private key as {n,d}.

The second part of the RSA algorithm is the message encryption and decryption, which can be described as:

To encrypt a message, the sender can follow these steps:

- Divide the original message into blocks so that each block can be converted to a number, M < n.
- Compute the encrypted block with the public key as C = M**e mod n.
- Deliver encrypted blocks as the encrypted message to the owner of the private key.

To decrypt a message, the owner of the private key can follow these steps:

- Divide the encrypted message back into blocks of the same block size used in the encryption process.
- Decrypt the block with the private key as M = C**d mod n.
- Put decrypted block together to get the original message.

Last update: 2013.

Illustration of RSA Algorithm: p,q=5,7

This section provides a tutorial example to illustrate how RSA public key encryption algorithm works with 2 small prime numbers 5 and 7.

To demonstrate the RSA public key encryption algorithm, let's start it with 2 smaller prime numbers 5 and 7.

Generation the public key and private key with prime numbers of 5 and 7 can be illustrated as:

```
Given p as 5
Given q as 7
Compute n = p*q: n = 5*7 = 35
Compute m = (p-1)*(q-1): m = 4*6 = 24
Select e, such that e and m are coprime numbers: e = 5
Compute d, such that d*e mod m = 1: d = 29
The public key {n,e} is = {35,5}
The private key {n,d} is = {35,29}
```

With the public key of {35,5}, encryption of a cleartext M represented as number 23 can be illustrated as:

```
Given public key {n,e} as {35,5}
Given cleartext M represented in number as 23
Divide B into blocks: 1 block is enough
Compute encrypted block C = M**e mod n:
   C = 23**5 mod 35 = 6436343 mod 35 = 18
The ciphertext C represented in number is 18
```

With the private key of {35,29}, decryption of the ciphertext C represented as number 18 can be illustrated as:

```
Given private key {n,e} as {35,29}
Given ciphertext C represented in number as 18
Divide C into blocks: 1 block is enough
Compute encrypted block M = C**d mod n:
   M = 18**29 mod 35
     = 18*18**28 mod 35
     = 18*(18**4)**7 mod 35
     = 18*(104976)**7 mode 35
     = 18*(104976 mod 35)**7 mod 35
     = 18*(11)**7 mod 35
     = 18*19487171 mod 35
     = 350769078 mod 35
     = 23
The cleartext M represented in number is 23
```

Cool. RSA public key encryption algorithm works. We are getting the original cleartext 23 back using the decryption algorithm and private key!

Notice that I had to compute "18**29 mod 35" in multiple steps, because 18**29 is too big to be computed directly.

Last update: 2013.

Illustration of RSA Algorithm: p,q=7,19

This section provides a tutorial example to illustrate how RSA public key encryption algorithm works with 2 small prime numbers 7 and 19.

As the second example to illustrate how RSA public key encryption algorithm works, let's take prime numbers of 7 and 19 as presented by Paul Johnston at http://pajhome.org.uk/crypt/rsa/rsa.html.

Generation the public key and private key with prime numbers of 5 and 7 can be illustrated as:

```
Given p as 7
Given q as 19
Compute n = p*q: n = 7*19 = 133
Compute m = (p-1)*(q-1): m = 6*18 = 108
Select e, such that e and m are coprime numbers: e = 5
Compute d, such that d*e mod m = 1: d = 65
The public key {n,e} is = {133,5}
The private key {n,d} is = {133,65}
```

With the public key of {133,5}, encryption of a cleartext M represented as number 6 can be illustrated as:

```
Given public key {n,e} as {133,5}
Given cleartext M represented in number as 6
Divide M into blocks: 1 block is enough
Compute encrypted block C = M**e mod n:
   C = 6**5 mod 133 = 7776 mod 133 = 62
The ciphertext c represented in number is 62
```

With the private key of {133,65}, decryption of the ciphertext represented as number 62 can be illustrated as:

```
Given private key {n,e} as {133,65}
Given ciphertext C represented in number as 62
Divide C into blocks: 1 block is enough
Compute encrypted block M = C**d mod n:
   M = 62**65 mod 133
     = 62*62**64 mod 133
     = 62*(62**4)**16 mod 133
     = 62*(14776336)**16 mode 133
     = 62*(14776336 mod 133)**16 mod 133
     = 62*(36)**16 mod 133
     = 62*(36**4 mod 133)**4 mod 133
     = 62*(92)**4 mod 133
     = 4441636352 mod 133
     = 6
The cleartext M represented in number is 6
```

Very nice. We are getting the original cleartext 6 back with no problem with the private key.

Last update: 2013.

Proof of RSA Public Key Encryption

This section describes steps to prove RSA Public Key Encryption algorithm. Fermat's little theorem is the key part of the proof.

To proof the RSA public key encryption algorithm, we need to proof the following:

```
Given that:
   p and q are 2 distinct prime numbers
   n = p*q
   m = (p-1)*(q-1)
   e satisfies 1 > e > n and e and m are coprime numbers
   d satisfies d*e mod m = 1
   M satisfies 0 => M > n
   C = M**e mod n
the following is true:
   M == C**d mod n
```

One way to prove the above is to use steps presented by Avi Kak at https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture12.pdf

(a) Prove that "M**(e*d) mod p == M mod p" is true:

```
M**(e*d) mod p
   = M**(k1*m+1) mod p                # because "d*e mod m = 1"
   = (M**(k1*m))*M mop p              # factoring 1 M out
   = (M**(k1*(q-1)*(p-1)))*M mod p    # because "m = (p-1)*(q-1)"
   = (M**(k2*(p-1)))*M mod p          # set "k2 = k1*(q-1)"
   = ((M**(p-1) mod p)**k2)*M mod p   # rearranging "mod p"

If M and p are coprime numbers:
M**(e*d) mod p
   = (1**k2)*M mod p                  # Fermat's little theorem:
                                      #    M**(p-1) mod p = 1
                                      # when M and p are coprimes
   = M mod p                          # done

Else
   M == k*p must be true              # because p is a prime number
   M mod p == 0
   M**(e*d) mod p == 0
   M**(e*d) mod p == M mod p          # done
```

(b) Prove that "M**(e*d) mod q == M mod q" is true in the same process as above.

(c) Prove that "M == C**d mod n" is true:

```
M**(e*d) mod p == M mod p    (1) # see proof (a)
M**(e*d) - M == k3*p         (2) # because of modulus operation

M**(e*d) mod q == M mod q    (3) # see proof (b)
M**(e*d) - M == k4*q         (4) # because of modulus operation

M**(e*d) - M == k5*p*q       (5) # because of (2) and (4)
M**(e*d) - M == k5*n         (6) # because n = p*q
M mod n == M**(e*d) mod n    (7) # because of modulus operation
M == M**(e*d) mod n          (8) # because 0=< M < n
M == (M**e mod n)**d mod n   (9) # moving "mod n"

M == C**d mod n             (10) # because C = M**e mod n
```

See you can see, the key part of the proof process is the "Fermat's little theorem", which says that if p is a prime number, then for any integer a, the number "a**p # a" is an integer multiple of p.

See http://en.wikipedia.org/wiki/Fermat%27s_little_theorem for more details.

Last update: 2013.

How Secure Is RSA Algorithm?

This section discusses the security of RSA public key encryption algorithm. RSA private key is not 100% secure. But if the private key uses larger value of n = p*q, it will take a very long time to crack the private key.

The security of RSA public key encryption algorithm is mainly based on the integer factorization problem, which can be described as:

```
Given integer n as the product of 2 distinct prime numbers p and q,
find p and q.
```

If the above problem could be solved, the RSA encryption is not secure at all. This is because the public key {n,e} is known to the public. Any one can use the public key {n,e} to figure out the private key {n,d} using these steps:

- Compute p and q by factorizing n.
- Compute m = (p-1)*(q-1).
- Compute d such that d*e mod m = 1 to obtain the private key {n,d}

If n is small, the integer factorization problem is easy to solve by testing all possible prime numbers in the range of (1, n).

For example, given 35 as n, we can list all prime numbers in the range of (1, 35): 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, and 31, and try all combinations of them to find 5 and 7 are factors of 35.

As the value of n gets larger, the integer factorization problem gets harder to solve. But it is still solvable with the use of computers. For example, the RSA-100 number with 100 decimal digits, or 330 bits, has been factored by Arjen K. Lenstra in 1991:

```
RSA-100 = 15226050279225333605356183781326374297180681149613
          80688657908494580122963258952897654000350692006139

    p*q = 37975227936943673922808872755445627854565536638199
        * 40094690950920881030683735292761468389214899724061
```

If you are using the above RSA-100 number as n, your private key is not private any more.

As of today, the highest value of n that has been factored is RSA-678 number with 232 decimal digits, or 768 bits, factored by Thorsten Kleinjung et al. in 2009:

```
RSA-768 = 12301866845301177551304949583849627207728535695953
```

```
          34792197322452151726400507263657518745202199786469
          38995647494277406384592519255732630345373154826850
          79170261221429134616704292143116022212404792747377
          94080665351419597459856902143413

    p*q = 33478071698956898786044169848212690817704794983713
          76856891243138898288379387800228761471165253174308
          7737814467999489
        × 36746043666799590428244633799627952632279158164343
          08764267603228381573966651127923337341714339681027
          0092798736308917
```

As our computers are getting more powerful, factoring n of 1024 bits will soon become reality. This is why experts are recommending us:

- Stop using RSA keys with n of 1024 bits now.
- Use RSA keys with n of 2048 bits to keep your data safe up to year 2030.
- Use RSA keys with n of 3072 bits to keep your data safe beyond year 2031.

Last update: 2013.

How to Calculate "M**e mod n"

This section discusses the difficulties of calculating 'M**e mod n'. The intermediate result of 'M**e' is too big for most programming languages.

If you are interested to apply the RSA encryption yourself manually, we need to learn how to calculate "M**e mod n" and "C**d mod n", which looks simple, but difficult to carry out.

First let's see how difficult is to calculate "C**d mod n" directly even with smaller numbers like "62**65 mod 133" as we saw in the previous example.

Here is a sample Perl script to calculate "62**65 mod 133":

```
# PowerModTest.pl
# Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
#

  print("\n");
  print("Wrong answer:\n");
  $c = 62**65 % 133;
  print("62**65 % 133 = ".$c."\n");

  print("\n");
  print("Correct answer:\n");
  $c = 62*(((62**4%133)**4%133)**4%133) % 133;
  print("62*(((62**4%133)**4%133)**4%133) % 133 = ".$c."\n");

  exit(0);
```

If you run it, you will get:

```
Wrong answer:
62**65 % 133 = 21

Correct answer:
62*(((62**4%133)**4%133)**4%133) % 133 = 6
```

So, why we are getting the wrong answer, if you use the expression "62**65 % 133" that matches the formula in encryption algorithm directly? It could be integer overflow on the intermediate result. I am not sure.

You can try this in PHP script too:

```
<?php # PowerModTest.php
# Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
#
  print("\n");
  print("Wrong answer:\n");
  $c = pow(62,65) % 133;
  print("pow(62,65) % 133 = ".$c."\n");

  print("\n");
  print("Correct answer:\n");
  $c = 62*(pow(pow(pow(62,4)%133,4)%133,4)%133) % 133;
  print("62*(pow(pow(pow(62,4)%133,4)%133,4)%133) % 133 = ".$c."\n");
?>
```

Here is the PHP script output. The direct expression also gives a wrong answer.

```
Wrong answer:
pow(62,65) % 133 = 0

Correct answer:
62*(pow(pow(pow(62,4)%133,4)%133,4)%133) % 133 = 6
```

Conclusion, we can not carry out "M**e mod n" as two operations directly, because the intermediate result of "M**e" can be too big to be processed in exponentiation operations in most programming languages.

We need to find a different way to calculate "M**e mod n" correctly and efficiently.

Last update: 2013.

Efficient RSA Encryption and Decryption Operations

This section describes an efficient way of carrying out RSA encryption and decryption operations provided by authors of RSA algorithm.

One efficient way to carry out the RSA encryption operation of "C = M**e mod n" is to use the following algorithm provided by authors of RSA as:

```
Step 1. Represent e in binary format and
```

```
   store it binary digits in array e[0], e[1], ..., e[k]
Step 2. Set the variable C to 1
Step 3. For each i from 0 to k, repeat steps 3a and 3b
Step 3a. C is reset to C*C mod n
Step 3b. If e[i] = 1, C is reset to C*M mod n
```

The RSA decryption operation of "M = C**d mod n" can also be carried out using the same algorithm:

```
Step 1. Represent d in binary format and
   store it binary digits in array d[0], d[1], ..., d[k]
Step 2. Set the variable M to 1
Step 3. For each i from 0 to k, repeat steps 3a and 3b
Step 3a. M is reset to M**2 mod n
Step 3b. If d[i] = 1, M is reset to M*C mod n
```

Let's use an example presented in the previous tutorial to validate the algorithm:

```
Given C = 62, d = 65, and n = 133
Calculate M = C**d mod n = 62**65 mod 133

Step 1. Represent d, 65, in binary format in array d[]
   d[] = {1,0,0,0,0,0,1}, k = 6

Step 2. Set the variable M to 1
   M = 1

Step 3. For each i from 0 to 6, repeat steps 3a and 3b
   i = 0, d[0] = 1
   M = 1*1 mod 133 = 1       (1)
   M = 1*62 mod 133 = 62     (2)

   i = 1, d[1] = 0
   M = 62**2 mod 133 = 120   (3)

   i = 1, d[2] = 0
   M = 120**2 mod 133 = 36   (4)

   i = 1, d[3] = 0
   M = 36**2 mod 133 = 99    (5)

   i = 1, d[4] = 0
   M = 99**2 mod 133 = 92    (6)

   i = 1, d[5] = 0
   M = 92**2 mod 133 = 85    (7)

   i = 1, d[6] = 1
   M = 85**2 mod 133 = 43    (8)
   M = 43*62 mod 133 = 6     (9)
```

Looks good. It matches the result presented in the previous tutorial.

If we start with the last calculation (9) and combine backward other calculations, we can see why this algorithm works:

```
M = 43*62 mod 133                              # start with (9)
```

```
  = 85**2 *62 mod 133                          # combine (8)
  = (92**2)**2 *62 mod 133                     # combine (7)
  = ((99**2)**2)**2 *62 mod 133                # combine (6)
  = (((36**2)**2)**2)**2 *62 mod 133           # combine (5)
  = ((((120**2)**2)**2)**2)**2 *62 mod 133     # combine (4)
  = (((((62**2)**2)**2)**2)**2)**2 *62 mod 133 # combine (3), (2), (1)
  = 62**64 *62 mod 133                         # consolidate exp.
  = 62**65 mod 133
```

Last update: 2013.

Proof of RSA Encryption Operation Algorithm

This section discusses how to prove the RSA encryption operation algorithm.

We can also prove the encryption operation algorithm presented in the previous section in the following way:

```
Given integers M, e, and n
Express e as a binary expression:
   e = e[0]*2**k + e[1]*2**(k-1) + ... + e[k-1]*2**1 + e[k]*2**0

Given C = M**e mod n

Replace e with the binary expression:
   C = M**(e[0]*2**k + e[1]*2**(k-1) + ... + e[k-1]*2**1
       + M**e[k]*2**0) mod n

Split and simply the e[k] term:
   C = M**(e[0]*2**k + e[1]*2**(k-1) + ... ) * M**e[k] mod n

Factor remaining terms:
   C = (M**(e[0]*2**(k-1) + e[1]*2**(k-2) + ... + e[k-1]*2**0))**2
       * M**e[k] mod n

Split and simply the e[k-1] term:
   C = (M**(e[0]*2**(k-1) + e[1]*2**(k-2) + ...) * M**e[k-1])**2
       * M**e[k] mod n

Repeat last two steps for e[k-2], ... e[0] terms:
   C = ((...((M**e[0])**2 * M**e[1])**2 * ...)**2 * M**e[k-1])**2
       * M**e[k] mod n

Move "mod n" into each team:
   C = ((...((M**e[0] mod n)**2 * M**e[1] mod n)**2 * ...)**2
       * M**e[k-1] mod n)**2
       * M**e[k] mod n
```

As you can see from the last expression, the "(...)**2 * M**e[i] mod n" is a repeating pattern. This leads us to the following algorithm to calculate "C = M**e mod n":

```
Given integers M, e, and n

Calculcate "C = M**e mod n":
Step 1. Represent e in binary format and
   store it binary digits in array e[0], e[1], ..., e[k]
```

```
Step 2. Set the variable C to 1
Step 3. For each i from 0 to k, repeat step 3a
Step 3a. C is reset to "C**2 * M**e[i] mod n"
```

This algorithm is identical to the algorithm presented in the previous section, because "* M**e[i]" is optional if e[i]=0.

Last update: 2013.

Finding Large Prime Numbers

This section describes different ways to generate large prime numbers to be used to generate public key and private key. Today most RSA tools are using probable prime numbers.

Now we have an efficient algorithm for the RSA encryption and decryption operation, the next thing to look at is on how to generate public and private keys.

The first step of the public and private key generation process is to get 2 large prime numbers. This can be done in different ways:

1. Selecting existing prime numbers from prime number databases. If you need 2 prime numbers to generate a new pair public key and private key, you can select 2 prime numbers from an existing prime number database like http://www.bigprimes.net/, which currently have about 1.4 billion prime numbers.

The largest prime number currently stored in www.bigprimes.net is 32416190071, which has 11 decimal digits. If select 2 11-digit prime numbers, p and q, the product of p and q, n, will be have 22 decimal digits, which is about 50-bit long. This tells us that if we use prime numbers from http://www.bigprimes.net/, we can only generate 50-bit RSA keys.

This is not long enough to meet today's security standard. Remember, security experts recommend to use 2048-bit public and private keys if you want to keep your data safe upto year 2030.

2. Generating prime numbers using prime number generating algorithms. If you need 2 prime numbers to generate a new pair public key and private key, you can generate 2 prime numbers using a prime number generating algorithm like "Sieve of Eratosthenes" which iteratively marking composite (i.e. not prime) by calculating multiples of each known prime and keeping numbers that are not marked as prime numbers at the end.

The "Sieve of Eratosthenes" algorithm is very efficient algorithm to generate prime numbers. But it is still takes too long to generate large prime numbers.

3. Generating probable prime numbers using probable prime number generating algorithms. If you need 2 prime numbers to generate a new pair public key and private key and don't want to wait for a long time to find them, you can generate 2 probable prime numbers using a probable prime number generating algorithm like the probablePrime() method provided in

java.math.BigInteger class.

Of course, public key and private key generated from probable prime numbers may fail to work in encryption and description. But the likelihood of failure is very small.

Today most RSA public key and private key tools are using probable prime numbers.

Last update: 2013.

RSA Implementation using java.math.BigInteger Class

This chapter provides tutorial notes and example codes on RSA implementation using Java BigInteger class. Topics include introduction of the java.math.BigInteger class; generating large probable prime numbers; generating RSA public key and private key; validating RSA keys; determining cleartext and ciphertext block sizes; padding last block of cleartext; implementing RSA encryption and decryption operations.

Conclusions:

- java.math.BigInteger class is designed to offer you enough methods to build a full solution for RSA public key encryption.

- The probablePrime() method can be used to generate very large positive probable prime numbers. The probability of generated numbers are prime numbers is 100-100/(2**100) percent.
- The gcd() method can be used to calculate the greatest common divisor for coprime number generation.
- The modInverse() method can be used to calculate the private key from the public key.
- The modPow() method can be used to carry out the RSA encryption and decryption operations.
- Cleartext block size can be set to "min(floor((RsaKeySize-1)/8), 256)".
- The block of cleartext can be padded using the same idea as the PKCS5Padding schema.
- Ciphertext block size can be set to "1+floor((RsaKeySize-1)/8)".
- A simple full implementation of RSA public key algorithm is presented using the java.math.BigInteger class.
- The implementation passed tests with RSA keys upto 3072 bits.

java.Math.BigInteger Class

This section describes the standard Java class, java.Math.BigInteger, which offers a number of useful methods to help generating large probable prime numbers, generating public key and private key, encryption and decryption messages.

In order to help implementing RSA public key encryption algorithm, Java offers a standard class called java.Math.BigInteger that provides:

- BigInteger(int bitLength, int certainty, Random rnd) - Constructs a randomly generated positive big integer that is probably prime with the specified certainty and bit length. This constructor can be used to generate large probable prime numbers "p" and "q" to be used to generate RSA public key and private key.
- static BigInteger probablePrime(int bitLength, Random rnd) - Returns a randomly generated positive big integer that is probably prime with the default certainty and specified bit length. This method can be used to generate large probable prime numbers "p" and 'q" to be used to generate RSA public key and private key.
- public boolean isProbablePrime(int certainty) - Returns true if this bit integer is probably prime, false if it's definitely composite. This method can be used to validate large probable prime numbers "p" and "q" to be used to generate RSA public key and private key.
- public BigInteger gcd(BigInteger val) - Returns a bit integer whose value is the greatest common divisor of this big integer and the specified big integer. This method can be used to select "e" component in the public key such that "gcd(e,(p-1)*(q-a)) = 1".

- public BigInteger modInverse(BigInteger m) - Returns a big integer x such that x*(this big integer) mod m = 1. This method can be used to compute "d" component in the private key such that "d*e mod (p-1)*(q-a) = 1".

- public BigInteger modPow(BigInteger exponent, BigInteger m) - Returns a big integer whose value is (this big integer)**exponent mod m. This method can be used to compute the ciphertext (encryption) "C" using "M**e mod p*q", where "M" is the original message. It also can be used to compute the original message (decryption) "M" using "C**d mod p*q".

Last update: 2013.

Generating Prime Number with BigInteger Class

This section provides a tutorial example on how to generate probable prime numbers using the java.math.BigInteger class in Java.

The first thing I want to try with the java.Math.BigInteger Class is to generate probable prime numbers. I am interested how fast large prime numbers can be generated and how good they are. Here is my first Java program using the java.Math.BigInteger Class:

```
/* PrimeGenerator.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.math.BigInteger;
import java.util.Random;
class PrimeGenerator {
   public static void main(String[] a) {
      if (a.length<2) {
         System.out.println("Usage:");
         System.out.println("java PrimeGenerator length certainty");
         return;
      }
      int length = Integer.parseInt(a[0]);
      int certainty = Integer.parseInt(a[1]);
      Random rnd = new Random();

      long t1 = System.currentTimeMillis();
      BigInteger p = new BigInteger(length,certainty,rnd);
      long t2 = System.currentTimeMillis();
      boolean ok = p.isProbablePrime(certainty);
      long t3 = System.currentTimeMillis();

      BigInteger two = new BigInteger("2");
      System.out.println("Probable prime: "+p);
      System.out.println("Validation: "+ok);
      System.out.println("Bit length: "+length);
      System.out.println("Certainty: "+certainty);
      System.out.println("Probability (%): "
         +(100.0-100.0/(two.pow(certainty)).doubleValue()));
      System.out.println("Generation time (milliseconds): "+(t2-t1));
      System.out.println("Validation time (milliseconds): "+(t3-t2));
   }
}
```

Compile and run it on my Windows 7 computer with JDK 1.6 starting with a small bit length and certainty to validate my program:

```
C:\herong>javac PrimeGenerator.java

C:\herong>java PrimeGenerator 4 4
Probable prime: 13
Validation: true
Bit length: 4
Certainty: 4
Probability (%): 93.75
Generation time (milliseconds): 1
Validation time (milliseconds): 39
```

Reviewing the output, we can see that prime number generated by the program, 13, does meet specified criteria:

- 13 expressed in binary format is 1101, which is 4-bit long as specified.
- 13 is a true prime number. Its probability is 100%, higher than than the specified probability of 93.75%, because probability = 100 - 100/(2**certainty) = 100 - 100/16 = 93.75.
- The generation time is short, only 1 millisecond.
- But the validation time is much longer, took 39 milliseconds.

I think my program works correctly. What do you think?

Last update: 2013.

Performance of Prime Number Generation

This section describes the performance of prime number generation using the java.math.BigInteger class. Larger prime numbers and higher certainties requires longer generation time.

Since my program is working, I want to use it to do more tests to see the impact on the generation time when I increase the bit length of the prime number.

In the first set of tests, I fixed the certainty to 16, which gives a probability of 99.99847412109375%. Testing results of different bit lengths are summarized in the table below:

```
                                Gen.   Val.
Bits Cert.  Probability         time   time  Prime
  16   16  99.99847412109375       1     32  64063
  32   16  99.99847412109375       2     40  4229630129
  64   16  99.99847412109375       3     34  18000105070399368923
 128   16  99.99847412109375      19     36  323465175613040279975...
 256   16  99.99847412109375      17     38  917924135915163081716...
 512   16  99.99847412109375     121     62  966939251609260746850...
1024   16  99.99847412109375     343    126  160235677041735401079...
```

```
2048   16  99.99847412109375    6746    743  3223557885769038951 86...
3072   16  99.99847412109375   24545   2362  5536132193091148463 16...
```

As you can see, generating a prime number with 1024 or less bits is pretty fast, taking less than 1 second. But it took 6.7 seconds to generate a 2048-bit prime number, and 24.5 seconds to generate a 3072-bit prime number.

In the second set of tests, I fixed the bit length to 3072. Testing results of different certainty levels are summarized in the table below:

```
                                Gen.   Val.
Bits Cert.  Probability         time   time  Prime
3072   16  99.99847412109375   34824   2344  565702937028652677653...
3072   32  99.99999997671694   57465   2334  481349137981705251267...
3072   64  100.0                8763   2341  353663618372094331380...
3072  128  100.0               63645   2413  430259008059707049953...
3072  256  100.0               34017   2352  440736874159753618303...
3072  512  100.0                7039   2357  486492819926034358966...
3072 1024  100.0              120450   2337  567519578195425341872...
```

It is interesting to see that probability rounds up to 100% when the certainty is 64 or higher.

Also the prime number generation time is not truly proportional to the certainty. For example, it took much less time for certainty=512 than certainty=16. This could be caused by the randomness of the algorithm.

Last update: 2013.

RSA Encryption Implementation using BigInteger Class

This section describes the outline of a simple RSA public key encryption implementation using the java.math.BigInteger class.

After learning methods provided in the java.math.BigInteger class, I think I am ready to implement RSA public key encryption algorithm in Java now. Here is the outline of my implementation logic:

1. Public key generation:

- Call probablePrime() method to generate a prime number, p.
- Call probablePrime() method to generate another prime number, q.
- Compute the modulus, n, as p*q.
- Compute the Euler's totient value, m, as (p-1)*(q-1).
- Find a public key, e, until m.gcd(e)==1.
- Find a private key, d, using e.modInverse(m).

2. Message encryption with public key:

- Divide original message into a sequence of blocks.
- Encrypt each block, M, using C = M.modPow(e,n).

3. Ciphertext decryption with private key:

- Divide ciphertext into a sequence of blocks.
- Decrypt each block, C, using M = C.modPow(d,n).

See next tutorials for Java implementation programs.

Last update: 2013.

RsaKeyGenerator.java for RSA Key Generation

This section describes the initial draft of a RSA public key and private key generation implementation using the java.math.BigInteger class.

The first step of RSA public key encryption implementation is to write a program to generate a pair of public key and private key. Here is my initial draft using the java.math.BigInteger class:

```
/* RsaKeyGenerator.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.math.BigInteger;
import java.util.Random;
class RsaKeyGenerator {
   public static void main(String[] a) {
      if (a.length<1) {
         System.out.println("Usage:");
         System.out.println("java RsaKeyGenerator size");
         return;
      }
      int size = Integer.parseInt(a[0]);

      Random rnd = new Random();
      BigInteger p = BigInteger.probablePrime(size/2,rnd);
      BigInteger q = p.nextProbablePrime();
      BigInteger n = p.multiply(q);
      BigInteger m = (p.subtract(BigInteger.ONE)).multiply(
         q.subtract(BigInteger.ONE));
      BigInteger e = getCoprime(m);
      BigInteger d = e.modInverse(m);

      System.out.println("p: "+p);
      System.out.println("q: "+q);
      System.out.println("m: "+m);
      System.out.println("Modulus: "+n);
      System.out.println("Key size: "+n.bitLength());
      System.out.println("Public key: "+e);
      System.out.println("Private key: "+d);
   }
   public static BigInteger getCoprime(BigInteger m) {
      Random rnd = new Random();
      int length = m.bitLength()-1;
      BigInteger e = BigInteger.probablePrime(length,rnd);
      while (! (m.gcd(e)).equals(BigInteger.ONE) ) {
               e = BigInteger.probablePrime(length,rnd);
      }
      return e;
   }
}
```

Some notes on RsaKeyGenerator.java:

- probablePrime() method uses the default certainty value of 100. This ensures that the probability of the resulting value being a prime number is 100-100/(2**100) percent.
- When generating the first prime number "p", I specified the bit length to be "size/2", This ensures that the bit length of the modulus "n", which the RSA key size, to be the requested value "size". The bit length of "n" is about 2 times of the bit length of "p".
- When calling probablePrime(), it requires a random number generator. I used a java.util,Random constructor to created a random number generator. Another option is to use a java.security.SecureRandom contrustor.
- To generate the second prime number "q", I used the p.nextProbablePrime() method, which returns the next prime number bigger than "p". This ensures that "q" is different than "p".
- To find a public key "e", I used a "while" loop to test different possible "e" until (m.gcd(e)).equals(BigInteger.ONE).

See the next tutorial for testing result of this program.

Last update: 2013.

RSA Keys Generated by RsaKeyGenerator.java

This section provides some RSA keys generated by my BigInteger implementation, RsaKeyGenerator.java, of RSA public key and private key generation algorithm.

Are you ready to see how my initial implementation of RSA public key and private key generation behaves? Here is the first test run using JDK 1.6:

```
C:\herong>javac RsaKeyGenerator.java

C:\herong>java RsaKeyGenerator 4
Exception in thread "main" java.lang.ArithmeticException: bitLength < 2
        at java.math.BigInteger.probablePrime(BigInteger.java:539)
        at RsaKeyGenerator.getCoprime(RsaKeyGenerator.java:35)
        at RsaKeyGenerator.main(RsaKeyGenerator.java:21)
```

Too bad. The program failed in the "getCoprime()" call to find a public key, "e". This is because the requested key size is too low, which does not give enough room to find a public key, "e".

Rerun the program with a higher key size. The problem should go away.

```
C:\herong>java RsaKeyGenerator 5
p: 3
q: 5
m: 8
Modulus: 15
Key size: 4
Public key: 5
Private key: 5
```

The program completed! But the result is not ideal. The private key is the same as the public key "5". This makes my private key not secure. The program could be enhanced to ensure the private key is different than the public key.

Rerun the program with a higher key size. Hope it will return a good pair of public key and private key.

```
C:\herong>java RsaKeyGenerator 7
p: 7
q: 11
m: 60
Modulus: 77
Key size: 7
Public key: 17
Private key: 53
```

Cool. The program generated a good pair of public key, {77,17} and {77,53}.

Rerun the program with different key sizes:

```
C:\herong>java RsaKeyGenerator 8
p: 11
q: 13
m: 120
Modulus: 143
Key size: 8
Public key: 53
Private key: 77

C:\herong>java RsaKeyGenerator 16
p: 151
q: 157
m: 23400
Modulus: 23707
Key size: 15
Public key: 10151
Private key: 4751

C:\herong>java RsaKeyGenerator 32
p: 34613
q: 34631
m: 1198613560
```

```
Modulus: 1198682803
Key size: 31
Public key: 756303523
Private key: 705860827

C:\herong>java RsaKeyGenerator 64
p: 2966449091
q: 2966449099
m: 8799820227293420820
Modulus: 8799820233226319009
Key size: 63
Public key: 3447872045926560979
Private key: 5997668047507921159

C:\herong>java RsaKeyGenerator 128
p: 17902136406704537069
q: 17902136406704537077
m: 320486487924256034368552058949822333168
Modulus: 320486487924256034404356331763231407313
Key size: 128
Public key: 138184930940463531660820083778072069237
Private key: 173448309040289888328993883042709949325

C:\herong>java RsaKeyGenerator 256
p: 248658744261550238073459677814507557459
q: 248658744261550238073459677814507557527
m: 618311710977310434529034534762836648859088873902738200302650613...
Modulus: 618311710977310434529034534762836648864062048787969205064...
Key size: 256
Public key: 394190853336940694532345943348534965939075733405768734...
Private key: 214295683817019610140890985852801296823028963507284 70...
```

Output looks good. But we need test them with the encryption and decryption process.

Last update: 2013.

RsaKeyValidator.java for RSA Key Validation

This section provides a tutorial example on how to validate RSA keys by encrypting and decrypting some random sample messages.

Before writing the RSA encryption and decryption programs, I want to validate RSA keys generated from by RsaKeyGenerator.java program.

Here is my RSA key validation program:

```
/* RsaKeyValidator.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.math.BigInteger;
import java.util.Random;
import java.io.*;
class RsaKeyValidator {
   private BigInteger n, e, d;
   public static void main(String[] a) {
      if (a.length<1) {
         System.out.println("Usage:");
         System.out.println("java RsaKeyValidator input");
         return;
      }
      String input = a[0];

      RsaKeyValidator validator = new RsaKeyValidator(input);
      validator.test();
      validator.test();
      validator.test();
   }

// Reading in RSA public key and private key
   RsaKeyValidator(String input) {
      try {
         BufferedReader in = new BufferedReader(new FileReader(input));
         String line = in.readLine();
         while (line!=null) {
            if (line.indexOf("Modulus: ")>=0) {
               n = new BigInteger(line.substring(9));
            }
            if (line.indexOf("Public key: ")>=0) {
               e = new BigInteger(line.substring(12));
            }
            if (line.indexOf("Private key: ")>=0) {
               d = new BigInteger(line.substring(13));
            }
            line = in.readLine();
         }
      } catch (Exception ex) {
         ex.printStackTrace();
      }
      System.out.println("--- Reading public key and private key ---");
      System.out.println("Modulus: "+n);
      System.out.println("Key size: "+n.bitLength());
      System.out.println("Public key: "+e);
      System.out.println("Private key: "+d);
   }

// Testing encryption and description
   public void test() {
      Random rnd = new Random();
      int size = rnd.nextInt(n.bitLength()-1);
      BigInteger text = new BigInteger(size,rnd);
      BigInteger cipher = text.modPow(e,n);
      BigInteger decrypted = cipher.modPow(d,n);
      boolean isPassed = text.equals(decrypted);
      System.out.println("--- RSA encryption test ---");
      System.out.println("Is test passed: "+isPassed);
      System.out.println("Original text: "+text);
      System.out.println("Cipher text: "+cipher);
      System.out.println("Decrypted text: "+decrypted);
   }
}
```

Some notes on RsaKeyValidator.java:

- Reading RSA key back from a file is easy, looping through each line, then parsing the modulus, the public key and the private key back.

- The test() method is designed to generate a random positive integer less that the modulus, n. Then random integer is encrypted and decrypted back to see if it changed or not.

Testing result is presented in the next tutorial.

Last update: 2013.

64-bit RSA Key Validated by RsaKeyValidator.java

This section provides a tutorial example on how a 64-bit RSA key is validated by RsaKeyValidator.java with 3 rounds of encryption and decryption tests.

To try my RsaKeyValidator.java program, I am going to use the 64-bit RSA key generated from the previous tutorial using RsaKeyGenerator.java:

```
C:\herong>java RsaKeyGenerator 64 > rsa_64.key

C:\herong>type rsa_64.key
p: 3372213049
q: 3372213067
m: 11371820901801285168
Modulus: 11371820908545711283
Key size: 64
Public key: 8383132602661586723
Private key: 2377205257342368779

C:\herong>java RsaKeyValidator rsa_64.key
--- Reading public key and private key ---
Modulus: 11371820908545711283
Key size: 64
Public key: 8383132602661586723
Private key: 2377205257342368779

--- RSA encryption test ---
Is test passed: true
Original text: 2314539574532842
Cipher text: 6717539904595114684
Decrypted text: 2314539574532842

--- RSA encryption test ---
Is test passed: true
Original text: 474153
Cipher text: 372517405590993925
Decrypted text: 474153

--- RSA encryption test ---
Is test passed: true
Original text: 490393355042176
Cipher text: 9836060768354999626
Decrypted text: 490393355042176
```

I think my 64-bit RSA key is good. It passed 3 all tests. What do you think?

Last update: 2013.

Converting Byte Sequences to Positive Integers

This section describes java.math.BigInteger methods that can be used to convert byte sequences into positive integers and convert back to byte sequences for RSA encryption and descryption operations.

From the previous tutorial, we learned that RSA keys generated from RsaKeyGenerator.java is working. Now we need to think about how to encrypt a byte sequences. We know this can be done by converting the byte sequence to positive integers, then applying the RSA encryption operation on the converted integers.

So we need to learn how to convert a byte sequence into a positive integer for the encryption purpose.

Similarly, we also need to learn how to convert a positive integer into a byte sequence after applying the RSA decryption operation to recover the original byte sequence.

For converting byte sequences into integers, the java.math.BigInteger class has offered 3 methods based the Java documentation:

- public BigInteger(byte[] val) - Translates a byte array containing the two's-complement binary representation of a BigInteger into a BigInteger. The input array is assumed to be in big-endian byte-order: the most significant byte is in the zeroth element.
- public BigInteger(int signum, byte[] magnitude) - Translates the sign-magnitude representation of a BigInteger into a BigInteger. The sign is represented as an integer signum value: -1 for negative, 0 for zero, or 1 for positive. The magnitude is a byte array in big-endian byte-order: the most significant byte is in the zeroth element.
- public byte[] toByteArray() - Returns a byte array containing the two's-complement representation of this BigInteger. The byte array will be in big-endian byte-order: the most significant byte is in the zeroth element. The array will contain the minimum number of bytes required to represent this BigInteger, including at least one sign bit, which is (ceil((this.bitLength() + 1)/8)).

Obviously, the best choice of converting a byte sequence into a positive integer is to use the BigInteger(int signum, byte[] magnitude) constructor as: "new BigInteger(1,byteArray)". This avoids getting negative numbers when the first bit of the byte sequence is a negative sign.

For converting a positive integer back to a byte sequence, we have to use toByteArray(). There is no other choices. But you may get an extra byte in the resulting byte sequence because of the extra sign bit added in the result. For example, the follow Java code will show that "barSignConverted" has 3 bytes, not 2 types.

```
byte[] barSign = {(byte)0xfe,(byte)0x31}; // 2 bytes of a Unicode text
BigInteger value = new BigInteger(1,barSign);  // 65073
byte[] barSignConverted = value.toByteArray();
```

```
System.out.println("Length: "+barSignConverted.length);
```

However, we can safely drop the first byte, if an extra byte is added for the sign bit, because it will contain the sign bit only, and we don't need the sign anyway.

Last update: 2013.

Cleartext Block Size for RSA Encryption

This section discusses what is the most efficient block size when dividing a Cleartext message into blocks for RSA public encryption. The suggested block size is 'floor((x-1)/8)', where 'x' is the RSA key size, or the key modulus bit length.

In the previous section, we learned how to convert a byte sequence into a positive integer, so that it can be encrypted by the RSA public key.

But what happens if the converted positive integer is too large? We know that the RSA public key encryption algorithm will not work on integers that are greater than the modulus of the RSA key.

To resolve this problem, we have to divide the cleartext byte sequence into blocks, and convert each block into a positive number smaller than the modulus of the RSA so it can be encrypted correctly. The next question is then, what block size should we use?

One option is use 1 byte as the block size. This will ensure that the converted integer is in the range of 0 and 255. Any RSA key with a modulus value greater than 255 will work on 1-byte blocks. In other words, any RSA key with a key size of 9 bites or higher will work 1-byte blocks.

However, if we have a RSA key with a much higher key size, like 2048 bites, using 1-byte blocks is not that efficient. This is because larger blocks can be used with larger keys to reduce the number of blocks to reduce the total encryption time and ciphertext data size.

Let's use a RSA key with 2048 bits with a cleartext byte sequence of 200 bytes to see the impact of using different block size.

1. If we set the block size to 1 byte, we will have:

- 200 blocks after dividing the cleartext byte sequence.
- 200 8-bit integers to be encrypted.
- 200 2048-bit integers resulted from the encryption process.

2. If we set the block size to 200 bytes, we will have:

- 1 block after dividing the cleartext byte sequence.
- 1 1600-bit integer to be encrypted.

- 1 2048-bit integer resulted from the encryption process.

Obviously, using 200-bytes as the block size is much more efficient than 1-byte block size. This means the higher block size, the more efficient.

So we need to find the highest block size allowed for a given RSA key with x bits key size. This can be calculated as: "floor((x-1)/8)" in bytes. The resulting block size is at least 1 bit less than the RSA key bit length, which ensures that the integer value in each block is smaller than the key modulus.

Note that, we cannot use a block size that is the same bit length as the RSA key, because the result block may represent an integer greater than the key modulus. For example, block value 0xF0000000 is greater than a key modulus 0xEC134ECB. But their bit length is the same, 32 bits, 4 bytes.

Last update: 2013.

Cleartext Message Padding and Revised Block Size

This section discusses how to pad the cleartext message so that it can divided into blocks of the same size. The padding idea is borrowed from the PKCS5Padding schema.

Once the block size is determined, we need to find a schema to pad the cleartext message if it is not multiples of the block size.

Instead of developing a new padding schema, we can borrow the idea from the PKCS5Padding schema, which is defined as:

- The number of bytes to be padded equals to "blockSize - numberOfBytes(cleartext) mod blockSize".
- All padded bytes have the same value - the number of bytes padded.

But this padding schema has a limitation: The maximum number of bytes we can pad is 256, because that is the maximum value we can present in a single byte.

So we have to modify our block size logic and padding schema to make it work:

- Cleartext message block size "blockSize" should be calculated as "min(floor((RsaKeySize-1)/8), 256)".
- Number of padding bytes "numberOfPaddingBytes" should equal to "blockSize - numberOfBytes(cleartext) mod blockSize".
- All padding bytes should have the same value as "numberOfPaddingBytes mod blockSize".

Example 1:

```
Given the RSA key size: x = 16
Given the number of bytes in the cleartext message: y = 12
The block size should be: min(floor((16-1)/8),256) = 1
The number of padding bytes should be: 1 - 12 mod 1 = 1
The padding byte value should be: 1 mod 1 = 0
The padding byte sequence should be: 0x00
```

Example 2:

```
Given the RSA key size: x = 64
Given the number of bytes in the cleartext message: y = 12
The block size should be: min(floor((64-1)/8),256) = 7
The number of padding bytes should be: 7 - 12 mod 7 = 2
The padding byte value should be: 2 mod 7 = 0x02
The padding byte sequence should be: 0x0202
```

Example 3:

```
Given the RSA key size: x = 64
Given the number of bytes in the cleartext message: y = 21
The block size should be: min(floor((64-1)/8),256) = 7
The number of padding bytes should be: 7 - 21 mod 7 = 7
The padding byte value should be: 7 mod 7 = 0x00
The padding byte sequence should be: 0x00000000000000
```

Last update: 2013.

Ciphertext Block Size for RSA Encryption

This section discusses what is the most efficient block size when packaging encrypted integers resulted from the RSA encryption operation on ciphertext blocks. The suggested block size is '1+floor((x-1)/8)', where 'x' is the RSA key size, or the key modulus bit length.

The previous section, we figured out the best block size (can be called as cleartext block size) and padding schema for dividing the cleartext message into blocks for RSA encryption.

Now let's look that how we can package individual encrypted integers generated from cleartext message blocks into a ciphertext byte sequence.

We have 2 general options:

1. Convert encrypted integers into byte blocks with the minimum number of bytes for each integer. Then package byte blocks sequentially with block markers to separate them.

2. Convert encrypted integers into byte blocks with a equal number of bytes for each integer. Then package byte blocks sequentially with no block markers to separate them.

The option 2 seems to be better, because it avoids the trouble of designing the special block marker and informing the receiver of the encrypted message what the marker is.

If we go with option 2, we need to figure out what is the best block size for encrypted integer blocks. The only requirement is that the block must long enough to hold the highest possible encrypted integer, which is the RSA key modulus.

Base on this requirement, the best block size (can be called as ciphertext block size) for encrypted integer blocks is "ceiling(RsaKeySize/8)" in bytes, which can also be expressed as "1+floor((RsaKeySize-1)/8)" in bytes.

If we compare the ciphertext block size "1+floor((RsaKeySize-1)/8)" with the cleartext block size "min(floor((RsaKeySize-1)/8),256)", the ciphertext block size is always 1 byte larger, if the RSA key size is 2056 or less.

Last update: 2013.

RsaKeyEncryption.java for RSA Encryption Operation

This section provides a tutorial example on how to implement RSA encryption operation using the java.math.BigInteger class. The important part of the implementation is to determine the cleartext block size, ciphertext block size, and the padding of the last block.

Finally, we are ready to implement the RSA public key encryption operation using the java.math.BigInteger class. Here is my initial draft:

```
/* RsaKeyEncryption.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.math.BigInteger;
import java.io.*;
class RsaKeyEncryption {
   private BigInteger n, e;
   public static void main(String[] a) {
      if (a.length<3) {
         System.out.println("Usage:");
         System.out.println("java RsaKeyEncryption key input output");
         return;
      }
      String keyFile = a[0];
      String input = a[1];
      String output = a[2];

      RsaKeyEncryption encryptor = new RsaKeyEncryption(keyFile);
      encryptor.encrypt(input,output);
   }

// Reading in RSA public key
   RsaKeyEncryption(String input) {
      try {
         BufferedReader in = new BufferedReader(new FileReader(input));
         String line = in.readLine();
         while (line!=null) {
            if (line.indexOf("Modulus: ")>=0) {
               n = new BigInteger(line.substring(9));
            }
            if (line.indexOf("Public key: ")>=0) {
               e = new BigInteger(line.substring(12));
            }
            line = in.readLine();
         }
      } catch (Exception ex) {
         ex.printStackTrace();
      }
      System.out.println("--- Reading public key ---");
      System.out.println("Modulus: "+n);
      System.out.println("Key size: "+n.bitLength());
      System.out.println("Public key: "+e);
   }

// Encrypting original message
   public void encrypt(String intput, String output) {
      int keySize = n.bitLength();                       // In bits
      int clearTextSize = Math.min((keySize-1)/8,256);   // In bytes
      int cipherTextSize = 1 + (keySize-1)/8;            // In bytes
      System.out.println("Cleartext block size: "+clearTextSize);
      System.out.println("Ciphertext block size: "+cipherTextSize);
      try {
         FileInputStream fis = new FileInputStream(intput);
         FileOutputStream fos = new FileOutputStream(output);
         byte[] clearTextBlock = new byte[clearTextSize];
         byte[] cipherTextBlock = new byte[cipherTextSize];
         long blocks = 0;
         int dataSize = fis.read(clearTextBlock);
         boolean isPadded = false;

//       Reading input message
         while (dataSize>0) {
            blocks++;
            if (dataSize<clearTextSize) {
               padBytesBlock(clearTextBlock,dataSize);
               isPadded = true;
            }

            BigInteger clearText = new BigInteger(1,clearTextBlock);
            BigInteger cipherText = clearText.modPow(e,n);
            byte[] cipherTextData = cipherText.toByteArray();
            putBytesBlock(cipherTextBlock,cipherTextData);
            fos.write(cipherTextBlock);

            dataSize = fis.read(clearTextBlock);
         }

//       Adding a full padding block, if needed
         if (!isPadded) {
            blocks++;
            padBytesBlock(clearTextBlock,0);
            BigInteger clearText = new BigInteger(1,clearTextBlock);
            BigInteger cipherText = clearText.modPow(e,n);
            byte[] cipherTextData = cipherText.toByteArray();
            putBytesBlock(cipherTextBlock,cipherTextData);
            fos.write(cipherTextBlock);
         }

         fis.close();
         fos.close();
         System.out.println("Encryption block count: "+blocks);
      } catch (Exception ex) {
         ex.printStackTrace();
      }
   }

// Putting bytes data into a block
   public static void putBytesBlock(byte[] block, byte[] data) {
      int bSize = block.length;
      int dSize = data.length;
      int i = 0;
      while (i<dSize && i<bSize) {
          block[bSize-i-1] = data[dSize-i-1];
          i++;
      }
      while (i<bSize) {
          block[bSize-i-1] = (byte)0x00;
          i++;
      }
   }

// Padding input message block
   public static void padBytesBlock(byte[] block, int dataSize) {
      int bSize = block.length;
      int padSize = bSize - dataSize;
      int padValue = padSize%bSize;
      for (int i=0; i<padSize; i++) {
          block[bSize-i-1] = (byte) padValue;
      }
   }
}
```

Some notes on RsaKeyEncryption.java:

- For RSA encryption operation, I only need to read in the public key.
- The cleartext block size is calculated with "Math.min((keySize-1)/8,256)" as we decided in the previous section.
- The ciphertext block size is calculated with "1 + (keySize-1)/8" as we decided in the previous section.
- I created a method "padBytesBlock(byte[] block, int dataSize)" to pad the last block of the cleartext message.
- I used "new BigInteger(1,clearTextBlock)" to convert a byte array to a positive BigInteger by explicitly specifying the sign value.
- I used "clearText.modPow(e,n)" to compute the encrypted integer.
- I used "cipherText.toByteArray()" to dump the encrypted integer into a byte array with an extra sign bit included.
- I created a method "putBytesBlock(byte[] block, byte[] data)" to package the encrypted integer represented in a byte array into a fixed length byte block.
- The first loop in "putBytesBlock()" is to copy all magnitude bits from the integer data array into the block starting from the end of the array. The sign bit (should always be "0") is not

copied, if the integer is large and its number of magnitude bits is the same as the block size in bits. Bit the sign bit is copied, if the integer is smaller.

- The second loop in "putBytesBlock()" is to pad those extra bytes at the beginning with "0x00", if the integer is small.

Testing result is presented in the next tutorial.

Last update: 2013.

RsaKeyDecryption.java for RSA Decryption Operation

This section provides a tutorial example on how to implement RSA decryption operation using the java.math.BigInteger class. The important part of the implementation is to use the same block sizes and padding schema as the encryption operation.

Before testing RsaKeyEncryption.java, let's finish the implementation of RSA decryption operation first. Here is my initial draft:

```
/* RsaKeyDecryption.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.math.BigInteger;
import java.io.*;
class RsaKeyDecryption {
   private BigInteger n, d;
   public static void main(String[] a) {
      if (a.length<3) {
         System.out.println("Usage:");
         System.out.println("java RsaKeyDecryption key input output");
         return;
      }
      String keyFile = a[0];
      String input = a[1];
      String output = a[2];

      RsaKeyDecryption encryptor = new RsaKeyDecryption(keyFile);
      encryptor.decrypt(input,output);
   }

// Reading in RSA private key
   RsaKeyDecryption(String input) {
      try {
         BufferedReader in = new BufferedReader(new FileReader(input));
         String line = in.readLine();
         while (line!=null) {
            if (line.indexOf("Modulus: ")>=0) {
               n = new BigInteger(line.substring(9));
            }
            if (line.indexOf("Private key: ")>=0) {
               d = new BigInteger(line.substring(13));
            }
            line = in.readLine();
         }
      } catch (Exception ex) {
         ex.printStackTrace();
      }
      System.out.println("--- Reading private key ---");
      System.out.println("Modulus: "+n);
      System.out.println("Key size: "+n.bitLength());
      System.out.println("Private key: "+d);
   }

// Decrypting cipher text
   public void decrypt(String intput, String output) {
      int keySize = n.bitLength();                       // In bits
      int clearTextSize = Math.min((keySize-1)/8,256);   // In bytes
      int cipherTextSize = 1 + (keySize-1)/8;            // In bytes
      System.out.println("Cleartext block size: "+clearTextSize);
      System.out.println("Ciphertext block size: "+cipherTextSize);
      try {
         FileInputStream fis = new FileInputStream(intput);
         FileOutputStream fos = new FileOutputStream(output);
         byte[] clearTextBlock = new byte[clearTextSize];
         byte[] cipherTextBlock = new byte[cipherTextSize];
         long blocks = 0;
         int dataSize = 0;
         while (fis.read(cipherTextBlock)>0) {
            blocks++;
            BigInteger cipherText = new BigInteger(1,cipherTextBlock);
            BigInteger clearText = cipherText.modPow(d,n);
            byte[] clearTextData = clearText.toByteArray();
            putBytesBlock(clearTextBlock,clearTextData);

            dataSize = clearTextSize;
            if (fis.available()==0) {
               dataSize = getDataSize(clearTextBlock);
            }
            if (dataSize>0) {
               fos.write(clearTextBlock,0,dataSize);
            }
         }
         fis.close();
         fos.close();
         System.out.println("Decryption block count: "+blocks);
      } catch (Exception ex) {
         ex.printStackTrace();
      }
   }

// Putting bytes data into a block
   public static void putBytesBlock(byte[] block, byte[] data) {
      int bSize = block.length;
      int dSize = data.length;
      int i = 0;
      while (i<dSize && i<bSize) {
          block[bSize-i-1] = data[dSize-i-1];
          i++;
      }
      while (i<bSize) {
          block[bSize-i-1] = (byte)0x00;
          i++;
      }
   }

// Getting data size from a padded block
   public static int getDataSize(byte[] block) {
      int bSize = block.length;
      int padValue = block[bSize-1];
      return (bSize-padValue)%bSize;
   }
}
```

Some notes on RsaKeyDecryption.java:

- For RSA decryption operation, I only need to read in the private key.
- In order to recover the original cleartext, we have to use the same block sizes that were used in the encryption operation.
- I used "new BigInteger(1,cipherTextBlock)" to convert a byte array to a positive BigInteger by explicitly specifying the sign value.
- I used "cipherText.modPow(d,n)" to compute the decrypted integer.
- I used "clearTextData.toByteArray()" to dump the decrypted integer into a byte array with an extra sign bit included.
- The same method "putBytesBlock(byte[] block, byte[] data)" is used to package the decrypted integer value represented in a byte array into a fixed length byte block.
- I created a method "getDataSize(byte[] block)" to calculate the data size in the last block by excluding padded bytes.

Testing result is presented in the next tutorial.

Last update: 2013.

Testing RsaKeyEncryption.java with a 16-bit Key

This section provides a tutorial example on testing my RSA encryption and decryption implementation using the java.math.BigInteger class with a 16-bit key.

For the first test, I created a simple text file called, hello.text:

```
Hello world!
```

Then I generated a 16-bit RSA key:

```
C:\herong>java RsaKeyGeneration 16 > rsa_16.key

C:\herong>type rsa_16.key
p: 211
q: 223
m: 46620
Modulus: 47053
Key size: 16
Public key: 17837
Private key: 22373
```

Result of encryption and decryption is listed below:

```
C:\herong>java RsaKeyEncryption rsa_16.key
   hello.text hello.encrypted

--- Reading public key ---
Modulus: 47053
Key size: 16
Public key: 17837
Cleartext block size: 1
Ciphertext block size: 2
Encryption block count: 13

C:\herong>java RsaKeyDecryption rsa_16.key
   hello.encrypted hello.decrypted

--- Reading private key ---
Modulus: 47053
Key size: 16
Private key: 22373
Cleartext block size: 1
Ciphertext block size: 2
Decryption block count: 13

C:\herong>type hello.decrypted
Hello world!
```

My RSA encryption and decryption implementation seems to be working. To confirm this, I tested it again with a binary file:

```
C:\herong>copy \windows\System32\xcopy.exe my_xcopy.exe
        1 file(s) copied.

C:\herong>my_xcopy.exe
Invalid number of parameters
0 File(s) copied

C:\herong>java RsaKeyEncryption rsa_16.key
   my_xcopy.exe my_xcopy_exe.encrypted

--- Reading public key ---
Modulus: 47053
Key size: 16
Public key: 17837
Cleartext block size: 1
Ciphertext block size: 2
Encryption block count: 36865

C:\herong>java RsaKeyDecryption rsa_16.key
   my_xcopy_exe.encrypted my_xcopy_decrypted.exe

--- Reading private key ---
Modulus: 47053
Key size: 16
Private key: 22373
Cleartext block size: 1
Ciphertext block size: 2
Decryption block count: 36865
```

```
C:\herong>my_xcopy_decrypted.exe
Invalid number of parameters
0 File(s) copied
```

Looks very good. My RSA encryption and decryption implementation is working with my RSA 16-bit key.

Last update: 2013.

Testing RsaKeyEncryption.java with a 64-bit Key

This section provides a tutorial example on testing my RSA encryption and decryption implementation using the java.math.BigInteger class with a 64-bit key.

Now I want to test my RSA encryption and decryption implementation on a 64-bit key:

```
C:\herong>java RsaKeyGeneration 64 > rsa_64.key

C:\herong>type rsa_64.key
p: 3372213049
q: 3372213067
m: 11371820901801285168
Modulus: 11371820908545711283
Key size: 64
Public key: 8383132602661586723
Private key: 2377205257342368779
```

The first test uses the same hello.text file:

```
C:\herong>java RsaKeyEncryption rsa_64.key
   hello.text hello.encrypted

--- Reading public key ---
Modulus: 11371820908545711283
Key size: 64
Public key: 8383132602661586723
Cleartext block size: 7
Ciphertext block size: 8
Encryption block count: 2

C:\herong>java RsaKeyDecryption rsa_64.key
   hello.encrypted hello.decrypted

--- Reading private key ---
Modulus: 11371820908545711283
Key size: 64
Private key: 2377205257342368779
Cleartext block size: 7
Ciphertext block size: 8
Decryption block count: 2

C:\herong>type hello.decrypted
Hello world!
```

My RSA encryption and decryption implementation is still working with my RSA 64-bit key. Comparing with the output of the 16-bit key test, the block size is increased and the number of blocks is decreased.

Now repeating the test on the same binary file:

```
C:\herong>java RsaKeyEncryption rsa_64.key
   my_xcopy.exe my_xcopy_exe.encrypted

--- Reading public key ---
Modulus: 11371820908545711283
Key size: 64
Public key: 8383132602661586723
Cleartext block size: 7
Ciphertext block size: 8
Encryption block count: 5267

C:\herong>java RsaKeyDecryption rsa_64.key
   my_xcopy_exe.encrypted my_xcopy_decrypted.exe

--- Reading private key ---
Modulus: 11371820908545711283
Key size: 64
Private key: 2377205257342368779
Cleartext block size: 7
Ciphertext block size: 8
Decryption block count: 5267

C:\herong>comp my_xcopy.exe my_xcopy_decrypted.exe
Comparing my_xcopy.exe and my_xcopy_decrypted.exe...
Files compare OK
```

Looks very good. My RSA encryption and decryption implementation is still working with my RSA 64-bit key.

Last update: 2013.

Testing RsaKeyEncryption.java with a 3072-bit Key

This section provides a tutorial example on testing my RSA encryption and decryption implementation using the java.math.BigInteger class with a 3072-bit key.

In the last test, I want to try my RSA encryption and decryption implementation on a large key, 3072-bit key:

```
C:\herong>java RsaKeyGeneration 3072 > rsa_3072.key

C:\herong>type rsa_3072.key
p: 128081139829885235542800955228499173923808960750433007737693766...
q: 128081139829885235542800955228499173923808960750433007737693766...
m: 164047783801226141039199633638385807841290101177468164207580823...
Modulus: 1640477838012261410391996336383858078412901011774681642O7...
Key size: 3071
Public key: 115818800736950873270224324475859346802845181289027278...
```

```
Private key: 2614812967765412146458218676402228314819986311563511l...
```

The first test uses the same hello.text file:

```
C:\herong>java RsaKeyEncryption rsa_3072.key
   hello.text hello.encrypted

--- Reading public key ---
Modulus: 16404778380122614103919963363838580784129010117746816420 7...
Key size: 3071
Public key: 11581880073695087327022432447585934680284518128902727 8...
Cleartext block size: 256
Ciphertext block size: 384
Encryption block count: 1

C:\herong>java RsaKeyDecryption rsa_3072.key
   hello.encrypted hello.decrypted

--- Reading private key ---
Modulus: 16404778380122614103919963363838580784129010117746816420 7...
Key size: 3071
Private key: 2614812967765412146458218676402228314819986311563511 1...
Cleartext block size: 256
Ciphertext block size: 384
Decryption block count: 1

C:\herong>type hello.decrypted
Hello world!
```

My RSA encryption and decryption implementation has no problem with large RSA keys. Now repeating the test on the same binary file:

```
C:\herong>java RsaKeyEncryption rsa_64.key
   my_xcopy.exe my_xcopy_exe.encrypted

C:\herong>java RsaKeyEncryption rsa_3072.key
   my_xcopy.exe my_xcopy_exe.encrypted

--- Reading public key ---
Modulus: 164047783801226141039199633638385807841290101177468164207...
Key size: 3071
Public key: 115818800736950873270224324475859346802845181289027278...
Cleartext block size: 256
Ciphertext block size: 384
Encryption block count: 145

C:\herong>java RsaKeyDecryption rsa_3072.key
   my_xcopy_exe.encrypted my_xcopy_decrypted.exe

--- Reading private key ---
Modulus: 164047783801226141039199633638385807841290101177468164207...
Key size: 3071
Private key: 261481296776541214645821867640222831481998631156351111...
Cleartext block size: 256
Ciphertext block size: 384
Decryption block count: 145

C:\herong>comp my_xcopy.exe my_xcopy_decrypted.exe
Comparing my_xcopy.exe and my_xcopy_decrypted.exe...
```

```
Files compare OK
```

No problem. My RSA encryption and decryption implementation works with large RSA keys too. But the encryption took about 2 minutes to finish on my computer.

Last update: 2013.

Introduction of DSA (Digital Signature Algorithm)

This chapter provides tutorial notes and example codes on DSA (Digital Signature Algorithm). Topics include introduction of DSA; illustration of DSA key generation, message signing and signature verification; proof of Digital Signature Algorithm.

Conclusions:

- DSA (Digital Signature Algorithm) uses public key and private key to generate and verify digital signatures.
- DSA public private keys are based 2 large prime numbers, p and q, where (p-1) mod q = 0
- DSA can not be used to encrypt messages.

What Is a Digital Signature?

This section describes what is a digital signature and what is the process of generating and verifying digital signature from a message.

What Is a Digital Signature? A digital signature is an electronic analogue of a written signature to provide assurance that the claimed signatory signed the information. In addition, a digital signature may be used to detect whether or not the information was modified after it was signed (i.e., to detect the integrity of the signed data).

The digital signature process can be divided into 2 parts:

1. Signature Generation:

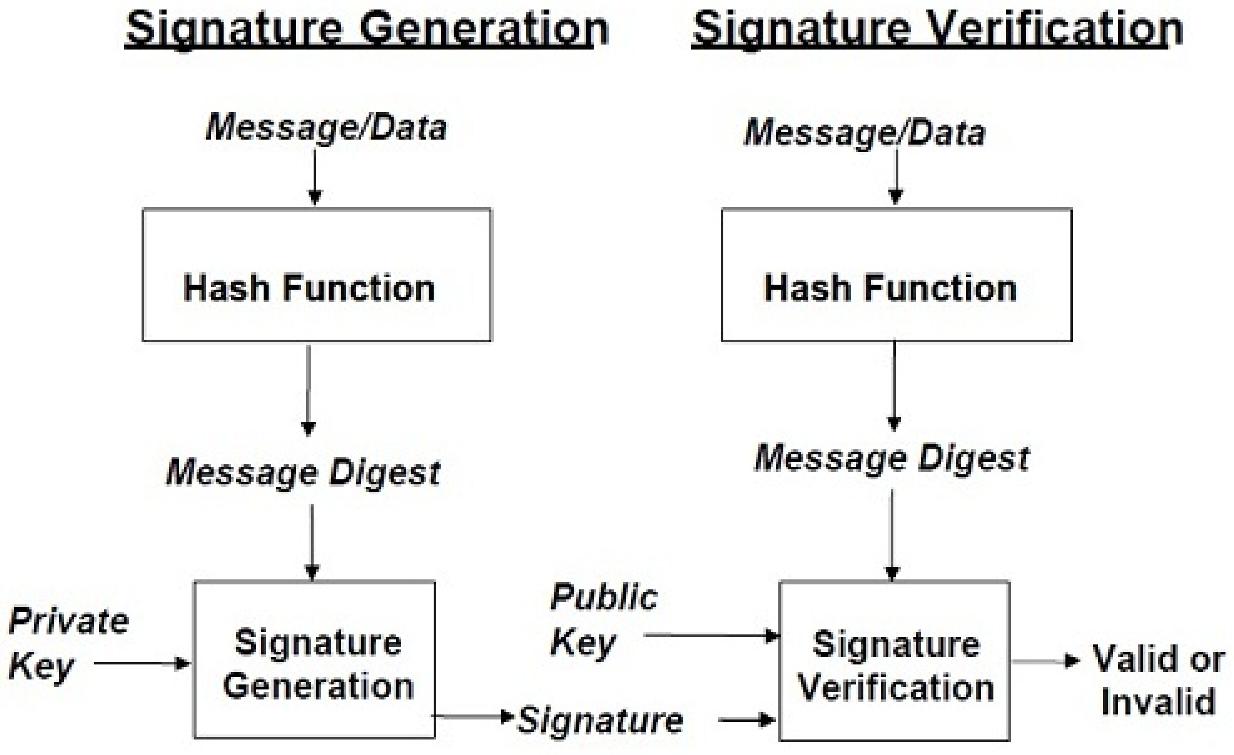
- Generating a pair of public key and provide key by the sender of the message.
- Generating the message digest from the message using a hash function.
- Generating the digital signature from the message digest with the private key.

- Sending the message, the digital signature, and the public key to receiver.

2. Signature Verification:

- Generating the message digest from the message using the same hash function.
- Verifying the digital signature with message digest using the public key.

Here is a diagram showing the digital signature process:



There are 2 popular algorithms that are used to generate and verify digital signature using public keys:

- RSA (Rivest, Shamir and Adleman) algorithm developed by Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman in 1976.
- DSA (Digital Signature Algorithm) algorithm developed by US government in 1991.

Last update: 2013.

What Is DSA (Digital Signature Algorithm)?

This section describes the DSA (Digital Signature Algorithm) algorithm, which consists of 2 parts: generation of a pair of public key and private key; generation and verification of digital signature.

What Is DSA (Digital Signature Algorithm)? DSA is a United States Federal Government standard for digital signatures. It was proposed by the National Institute of Standards and Technology (NIST) in August 1991 for use in their Digital Signature Standard (DSS), specified in FIPS 186 in 1993.

The first part of the DSA algorithm is the public key and private key generation, which can be described as:

- Choose a prime number q, which is called the prime divisor.
- Choose another primer number p, such that p-1 mod q = 0. p is called the prime modulus.
- Choose an integer g, such that 1 < g < p, g**q mod p = 1 and g = h**((p–1)/q) mod p. q is also called g's multiplicative order modulo p.
- Choose an integer, such that 0 < x < q.
- Compute y as g**x mod p.
- Package the public key as {p,q,g,y}.
- Package the private key as {p,q,g,x}.

The second part of the DSA algorithm is the signature generation and signature verification, which can be described as:

To generate a message signature, the sender can follow these steps:

- Generate the message digest h, using a hash algorithm like SHA1.
- Generate a random number k, such that 0 < k < q.
- Compute r as (g**k mod p) mod q. If r = 0, select a different k.
- Compute i, such that k*i mod q = 1. i is called the modular multiplicative inverse of k modulo q.
- Compute s = i*(h+r*x) mod q. If s = 0, select a different k.
- Package the digital signature as {r,s}.

To verify a message signature, the receiver of the message and the digital signature can follow these steps:

- Generate the message digest h, using the same hash algorithm.
- Compute w, such that s*w mod q = 1. w is called the modular multiplicative inverse of s modulo q.

- Compute u1 = h*w mod q.
- Compute u2 = r*w mod q.
- Compute v = (((g**u1)*(y**u2)) mod p) mod q.
- If v == r, the digital signature is valid.

Last update: 2013.

Illustration of DSA Algorithm: p,q=7,3

This section provides a tutorial example to illustrate how DSA digital signature algorithm works with small prime modulus p=7 and prime divisor q=3.

To demonstrate the DSA digital signature algorithm, let's try it with a smaller prime divisor q=3 and prime modulus p=7.

The process of generating the public key and private key can be illustrated as:

```
q = 3      # selected prime divisor
p = 7      # computed prime modulus: (p-1) mod q = 0
g = 4      # computed: 1 < g < p, g**q mod p = 1
           #           and g = h**((p–1)/q) mod p
           #           4**3 mod 7 = 1: 64 mod 7 = 1
x = 5      # selected: 0 < x < q
y = 2      # computed: y = g**x mod p = 4**5 mod 7
{7,3,4,2}  # the public key: {p,q,g,y}
{7,3,4,5}  # the private key: {p,q,g,x}
```

With the private key {p,q,g,x}={7,3,4,5}, the process of generating a digital signature out a message hash value of h=3 can be illustrated as:

```
h = 3      # the hash value as the message digest
k = 2      # selected: 0 < k < q
r = 2      # computed: r = (g**k mod p) mod q = (4**2 mod 7) mod 3
i = 5      # computed: k*i mod q = 1: 2*i mod 3 = 1
s = 2      # computed: s = i*(h+r*x) mod q = 5*(3+2*5) mod 3
{2,2}      # the digital signature
```

The process of verifying the digital signature {r,s}={2,2} with the public key {p,q,g,y}={7,3,4,2} can be illustrated as:

```
h = 3      # the hash value as the message digest
w = 5      # computed: s*w mod q = 1: 2*w mod 3 = 1
u1 = 0     # computed: u1 = h*w mod q = 3*5 mod 3 = 0
u2 = 1     # computed: u2 = r*w mod q = 2*5 mod 3 = 1
v = 2      # computed: v = (((g**u1)*(y**u2)) mod p) mod q
           #             = (((4**0)*(2**1)) mod 7) mod 3 = 2
v == r     # verification passed
```

Cool. DSA digital signature algorithm works. The verification value matches the expected value.

Last update: 2013.

Illustration of DSA Algorithm: p,q=23,11

This section provides a tutorial example to illustrate how DSA digital signature algorithm works with small prime modulus p=23 and prime divisor q=11.

As the second example to illustrate how DSA digital signature algorithm works, let's try it with another prime divisor q=11 and prime modulus p=23.

The process of generating the public key and private key can be illustrated as:

```
q = 11       # selected prime divisor
p = 23       # computed prime modulus: (p-1) mod q = 0
g = 4        # computed: 1 < g < p, g**q mod p = 1:
             #           and g = h**((p–1)/q) mod p
             #           4**11 mod 23 = 1: 4194304 mod 23 = 1
x = 7        # selected: 0 < x < q
y = 8        # computed: y = g**x mod p = 4**7 mod 23
{23,11,4,8}  # the public key: {p,q,g,y}
{23,11,4,7}  # the private key: {p,q,g,x}
```

With the private key {p,q,g,x}={23,11,4,7}, the process of generating a digital signature out a message hash value of h=3 can be illustrated as:

```
h = 3      # the hash value as the message digest
k = 5      # selected: 0 < k < q
r = 1      # computed: r = (g**k mod p) mod q = (4**5 mod 23) mod 11
i = 9      # computed: k*i mod q = 1: 5*i mod 11 = 1
s = 2      # computed: s = i*(h+r*x) mod q = 9*(3+1*7) mod 11
{1,2}      # the digital signature: {r,s}
```

The process of verifying the digital signature {r,s}={1,2} with the public key {p,q,g,y}={23,11,4,8} can be illustrated as:

```
h = 3      # the hash value as the message digest
w = 6      # computed: s*w mod q = 1: 2*w mod 11 = 1
u1 = 7     # computed: u1 = h*w mod q = 3*6 mod 11 = 7
u2 = 6     # computed: u2 = r*w mod q = 1*6 mod 11 = 6
v = 1      # computed: v = (((g**u1)*(y**u2)) mod p) mod q
           #             = (((4**7)*(8**6)) mod 23) mod 11 = 2
           #             = 16384*262144 mod 23 mod 11 = 1
v == r     # verification passed
```

Very nice. The verification value matches the expected value with no problem.

Last update: 2013.

Illustration of DSA Algorithm with Different k and h

This section provides a tutorial example to illustrate how DSA digital signature algorithm works with same DSA key paramters but different k and h values.

In the third example, I want to use the same DSA key parameters, the same hash value, and a different k value of 7.

The same DSA key parameters are used:

```
{23,11,4,8}  # the public key: {p,q,g,y}
{23,11,4,7}  # the private key: {p,q,g,x}
```

The process of generating a digital signature with the same private key {p,q,g,x}={23,11,4,7} and a different k value of 7 can be illustrated as:

```
h = 3      # the hash value as the message digest
k = 7      # selected: 0 < k < q
r = 8      # computed: r = (g**k mod p) mod q = (4**7 mod 23) mod 11
i = 8      # computed: k*i mod q = 1: 7*i mod 11 = 1
s = 10     # computed: s = i*(h+r*x) mod q = 8*(3+8*7) mod 11
{8,10}     # the digital signature: {r,s}
```

The process of verifying the digital signature {r,s}={8,10} with the same public key {p,q,g,y}={23,11,4,8} can be illustrated as:

```
h = 3      # the hash value as the message digest
w = 10     # computed: s*w mod q = 1: 10*w mod 11 = 1
u1 = 8     # computed: u1 = h*w mod q = 3*10 mod 11 = 8
u2 = 3     # computed: u2 = r*w mod q = 8*10 mod 11 = 3
v = 8      # computed: v = (((g**u1)*(y**u2)) mod p) mod q
           #             = (((4**8)*(8**3)) mod 23) mod 11 = 8
v == r     # verification passed
```

There is no problem. The digital signature is still good.

Now let's try on a different hash value of h=9 and generate the digital signature with same private key {p,q,g,x}={23,11,4,7}.

```
h = 9      # the hash value as the message digest
k = 7      # selected: 0 < k < q
r = 8      # computed: r = (g**k mod p) mod q = (4**7 mod 23) mod 11
i = 8      # computed: k*i mod q = 1: 7*i mod 11 = 1
s = 3      # computed: s = i*(h+r*x) mod q = 8*(9+8*7) mod 11
{8,3}      # the digital signature: {r,s}
```

Here is the verification process with the same public key {p,q,g,y}={23,11,4,8}:

```
h = 9      # the hash value as the message digest
w = 4      # computed: s*w mod q = 1: 3*w mod 11 = 1
u1 = 3     # computed: u1 = h*w mod q = 9*4 mod 11 = 3
u2 = 10    # computed: u2 = r*w mod q = 8*4 mod 11 = 10
v = 8      # computed: v = (((g**u1)*(y**u2)) mod p) mod q
           #             = (((4**3)*(8**10)) mod 23) mod 11 = 8
v == r     # verification passed
```

Looks very good.

Last update: 2013.

Proof of DSA Digital Signature Algorithm

This section describes steps to prove DSA digital signature algorithm. Fermat's little theorem is the key part of the proof.

To proof the DSA digital signature algorithm, we need to proof the following:

```
Given:
   p is a prime number                      (1)
   q is a prime number                      (2)
   (p-1) mod q = 0                          (3)
   1 < g < p                                (4)
   g = h**((p–1)/q) mod p                 (5.1)
   g**q mod p = 1                         (5.2)
   0 < x < q                                (6)
   y = g**x mod p                           (7)
   h                                        (8)
   0 < k < q                                (9)
   r = (g**k mod p) mod q                  (10)
   r > 0                                   (11)
   k*i mod q = 1                           (12)
   s = i*(h+r*x) mod q                     (13)
   s > 0                                   (14)
   s*w mod q = 1                           (15)
   u1 = h*w mod q                          (16)
   u2 = r*w mod q                          (17)
   v = (((g**u1)*(y**u2)) mod p) mod q     (18)

prove:
   v = r
```

One way to prove this is to use steps presented by William Stallings at http://mercury.webster.edu/aleshunas/COSC%205130/K-DSA.pdf

```
Fermat's little theorem:
   h**(p-1) mod p = 1                                        # as (21)

Rewrite expression:
   g**(f*q) mod p
      = (h**((p–1)/q) mod p)**(f*q) mod p                    # by (5.1)
      = h**(f*q*(p-1)/q) mod p
      = h**(f*(p-1)) mod p
      = (h**(p-1) mod p)**f mod p
      = 1**f mod p                                           # by (21)
      = 1                                                    # as (22)

Rewrite expression:
   y**u2 mod p
      = y**(r*w mod q) mod p                                 # by (17)
      = (g**x mod p)**(r*w mod q) mod p                      # by (7)
      = g**(x*(r*w mod q)) mod p
      = g**(x*(r*w-f1*q) mod p
      = g**(x*r*w-x*f1*q) mod p
      = g**((x*r*w mod q)+f2*q-x*f1*q) mod p
      = (g**(x*r*w mod q)*(g**(f*q)) mod p
      = (g**(x*r*w mod q)*1) mod p                          # by (22)
      = g**(x*r*w mod q) mod p                              # as (24)


Rewrite expression:
   v = (((g**u1)*(y**u2)) mod p) mod q                      # by (18)
      = (((g**(h*w mod q))*(y**u2)) mod p) mod q            # by (16)
      = (((g**(h*w mod q))*(g**(x*r*w mod q))) mod p) mod q # by (24)
      = ((g**((h*w mod q)+(x*r*w mod q))) mod p) mod q
      = ((g**(((h*w+x*r*w) mod q)+f*q)) mod p) mod q
      = (((g**((h*w+x*r*w) mod q))*(g**(f*q))) mod p) mod q
      = (((g**((h*w+x*r*w) mod q))*1)) mod p) mod q         # by (22)
      = ((g**((h*w+x*r*w) mod q)) mod p) mod q
      = ((g**(((h+x*r)*w) mod q)) mod p) mod q              # as (25)

Rewrite expression:
   k*s mod q
      = k*(i*(h+r*x) mod q) mod q                           # by (13)
      = (k*i mod q)*((h+r*x) mod q) mod q
      = 1*((h+r*x) mod q) mod q                             # by (12)
      = (h+r*x) mod q                                       # as (26)

Apply (26) to (25):
   v = ((g**((k*s*w) mod q)) mod p) mod q
      = (g**((k mod q)*(s*w mod q) mod q) mod p) mod q
      = (g**((k mod q)*(1) mod q) mod p) mod q              # by (15)
      = (g**(k mod q) mod p) mod q
      = (g**(k + f*q) mod p) mod q
      = (g**k mod p)*(g**(f*q) mod p) mod q
      = (g**k mod p)*1 mod q                                # by (22)
      = (g**k mod p) mod q
      = r                                                   # by (10)

Done.
```

See you can see, the key part of the proof process is the "Fermat's little theorem", which says that if p is a prime number, then for any integer a, the number "a**p # a" is an integer multiple of p. See http://en.wikipedia.org/wiki/Fermat%27s_little_theorem for more details.

It is interesting to see that both RSA and DSA are based on "Fermat's little theorem".

Last update: 2013.

Java Default Implementation of DSA

This chapter provides tutorial notes and example codes on the Java default implementation of DSA (Digital Signature Algorithm). Generating DSA key pair in Java; Examples of public key and private key pairs; Checking DSA key parameter; Generating DSA digital signature in Java using SHA1withDSA; Verifying DSA digital signature in Java.

Conclusions:

- Java provides a default implementation of DSA (Digital Signature Algorithm) in the java.security.* class package.
- The java.security.KeyPairGenerator class can be used to generate DSA public key and private key pair with algorithm name set to "DSA".
- DSA private key can can be save to a file using the PKCS#8 encoding format.
- DSA public key can can be save to a file using the X.509 encoding format.
- The java.security.Signature class can be used to generate and verify DSA digital signatures with algorithm name set to "SHA1withDSA".
- Java "SHA1withDSA" digital signatures are 46 bytes long.

Sample programs listed in this chapter have been tested with JDK 1.6.

DsaKeyGenerator.java - Generating DSA Key Pair

This section provides a tutorial example on how to using the Java default implementation of

DSA (Digital Signature Algorithm) to generate DSA key pair, public key and private key.

Implementing the DSA digital signature algorithm from scratch is not easy. What we can do is to use the default implementation of DSA in Java SE 6.0.

The first Java program I wrote is to generate a DSA key pair, public key and private key:

```
/**
 * DsaKeyGenerator.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.io.*;
import java.math.*;
import java.security.*;
import java.security.interfaces.*;
class DsaKeyGenerator {
   public static void main(String[] a) {
      if (a.length<2) {
         System.out.println("Usage:");
         System.out.println("java DsaKeyGenerator keySize output");
         return;
      }
      int keySize = Integer.parseInt(a[0]);
      String output = a[1];
      String algorithm = "DSA";
      try {
         genKeyPair(keySize,output,algorithm);
      } catch (Exception e) {
         System.out.println("Exception: "+e);
         return;
      }
   }
   private static void genKeyPair(int keySize, String output,
         String algorithm) throws Exception {
      KeyPairGenerator kg = KeyPairGenerator.getInstance(algorithm);
      kg.initialize(keySize);
      System.out.println();
      System.out.println("KeyPairGenerator Object Info: ");
      System.out.println("Algorithm = "+kg.getAlgorithm());
      System.out.println("Provider = "+kg.getProvider());
      System.out.println("Key Size = "+keySize);
      System.out.println("toString = "+kg.toString());
      KeyPair pair = kg.generateKeyPair();
      PrivateKey priKey = pair.getPrivate();
      PublicKey pubKey = pair.getPublic();
      String fl = output+".pri";
      FileOutputStream out = new FileOutputStream(fl);
      byte[] ky = priKey.getEncoded();
      out.write(ky);
      out.close();
      System.out.println();
      System.out.println("Private Key Info: ");
      System.out.println("Algorithm = "+priKey.getAlgorithm());
      System.out.println("Saved File = "+fl);
      System.out.println("Size = "+ky.length);
      System.out.println("Format = "+priKey.getFormat());
      System.out.println("toString = "+priKey.toString());
      fl = output+".pub";
      out = new FileOutputStream(fl);
      ky = pubKey.getEncoded();
      out.write(ky);
      out.close();
      System.out.println();
      System.out.println("Public Key Info: ");
      System.out.println("Algorithm = "+pubKey.getAlgorithm());
      System.out.println("Saved File = "+fl);
      System.out.println("Size = "+ky.length);
      System.out.println("Format = "+pubKey.getFormat());
      System.out.println("toString = "+pubKey.toString());
   }
}
```

Some notes on DsaKeyGenerator.java:

- The java.security.KeyPairGenerator.getInstance(algorithm) method is used to create a key pair generator object for the specified algorithm, "DSA".
- The java.security.KeyPairGenerator.initialize() method is used to initialize the key pair generator with the specified key size.
- The java.security.KeyPairGenerator.generateKeyPair() method is used to generate a new key pair.
- The java.security.Key.getEncoded() method is used to convert the key to a byte array using the default encoding.
- The java.io.FileOutputStream.write() method is used to dump the byte array to a file.

Testing result is presented in the next tutorial.

Last update: 2013.

DSA 512-bit and 1024-bit Key Pair Examples

This section provides 2 DSA key pair examples, 512-bit and 1024-bit, using DsaKeyGenerator.java running JDK 1.6.

Let's compile and run DsaKeyGenerator.java using JDK 1.6:

```
C:\herong>javac DsaKeyGenerator.java

C:\herong>java DsaKeyGenerator 16 test
Exception: java.security.InvalidParameterException: Modulus size must
range from 512 to 1024 and be a multiple of 64
```

Ok. We are not allowed to generate small DSA keys. Let's try the minimum DSA key size, 512 bits:

```
C:\herong>java DsaKeyGenerator 512 dsa_512

KeyPairGenerator Object Info:
```

```
Algorithm = DSA
Provider = SUN version 1.6
Key Size = 512
toString = sun.security.provider.DSAKeyPairGenerator@19b49e6

Private Key Info:
Algorithm = DSA
Saved File = dsa_512.pri
Format = PKCS#8
toString = Sun DSA Private Key
parameters:
p:
fca682ce 8e12caba 26efccf7 110e526d b078b05e decbcd1e b4a208f3 ae1617ae
01f35b91 a47e6df6 3413c5e1 2ed0899b cd132acd 50d99151 bdc43ee7 37592e17
q:
962eddcc 369cba8e bb260ee6 b6a126d9 346e38c5
g:
678471b2 7a9cf44e e91a49c5 147db1a9 aaf244f0 5a434d64 86931d2d 14271b9e
35030b71 fd73da17 9069b32e 2935630e 1c206235 4d0da20a 6c416e50 be794ca4

x:
8ad78f1c 6fd8512f a6f82053 5672a761 96fec0b2

Public Key Info:
Algorithm = DSA
Saved File = dsa_512.pub
Format = X.509
toString = Sun DSA Public Key
Parameters:
p:
fca682ce 8e12caba 26efccf7 110e526d b078b05e decbcd1e b4a208f3 ae1617ae
01f35b91 a47e6df6 3413c5e1 2ed0899b cd132acd 50d99151 bdc43ee7 37592e17
q:
962eddcc 369cba8e bb260ee6 b6a126d9 346e38c5
g:
678471b2 7a9cf44e e91a49c5 147db1a9 aaf244f0 5a434d64 86931d2d 14271b9e
35030b71 fd73da17 9069b32e 2935630e 1c206235 4d0da20a 6c416e50 be794ca4

y:
cde640d1 ebcaf6e2 caa83da1 ed475f5e 3f4fdfa5 802d4d5a f8bd07a6 d278418a
4b29f245 3ccff8ba 8daace97 e12d36a5 d50edd2d c93101da 213de2ef 189f8be4
```

Cool. Here is our first real DSA key pair. The toString() method actually dumps all DSA key parameters, p, q, g, x, y, in hexadecimal format.

Also note that the default encoding format is PKCS#8 for the private key, and X.509 for the public key.

The next test generates a 1024-bit DSA key pair:

```
C:\herong>java DsaKeyGenerator 1024 dsa_1024

KeyPairGenerator Object Info:
Algorithm = DSA
Provider = SUN version 1.6
Key Size = 1024
toString = sun.security.provider.DSAKeyPairGenerator@19b49e6

Private Key Info:
```

```
Algorithm = DSA
Saved File = dsa_1024.pri
Format = PKCS#8
toString = Sun DSA Private Key
parameters:
p:
fd7f5381 1d751229 52df4a9c 2eece4e7 f611b752 3cef4400 c31e3f80 b6512669
455d4022 51fb593d 8d58fabf c5f5ba30 f6cb9b55 6cd7813b 801d346f f26660b7
6b9950a5 a49f9fe8 047b1022 c24fbba9 d7feb7c6 1bf83b57 e7c6a8a6 150f04fb
83f6d3c5 1ec30235 54135a16 9132f675 f3ae2b61 d72aeff2 2203199d d14801c7
q:
9760508f 15230bcc b292b982 a2eb840b f0581cf5
g:
f7e1a085 d69b3dde cbbcab5c 36b857b9 7994afbb fa3aea82 f9574c0b 3d078267
5159578e bad4594f e6710710 8180b449 167123e8 4c281613 b7cf0932 8cc8a6e1
3c167a8b 547c8d28 e0a3ae1e 2bb3a675 916ea37f 0bfa2135 62f1fb62 7a01243b
cca4f1be a8519089 a883dfe1 5ae59f06 928b665e 807b5525 64014c3b fecf492a

x:
87a06897 5ef251b4 50510dee 08734119 5ca68c16

Public Key Info:
Algorithm = DSA
Saved File = dsa_1024.pub
Format = X.509
toString = Sun DSA Public Key
Parameters:
p:
fd7f5381 1d751229 52df4a9c 2eece4e7 f611b752 3cef4400 c31e3f80 b6512669
455d4022 51fb593d 8d58fabf c5f5ba30 f6cb9b55 6cd7813b 801d346f f26660b7
6b9950a5 a49f9fe8 047b1022 c24fbba9 d7feb7c6 1bf83b57 e7c6a8a6 150f04fb
83f6d3c5 1ec30235 54135a16 9132f675 f3ae2b61 d72aeff2 2203199d d14801c7
q:
9760508f 15230bcc b292b982 a2eb840b f0581cf5
g:
f7e1a085 d69b3dde cbbcab5c 36b857b9 7994afbb fa3aea82 f9574c0b 3d078267
5159578e bad4594f e6710710 8180b449 167123e8 4c281613 b7cf0932 8cc8a6e1
3c167a8b 547c8d28 e0a3ae1e 2bb3a675 916ea37f 0bfa2135 62f1fb62 7a01243b
cca4f1be a8519089 a883dfe1 5ae59f06 928b665e 807b5525 64014c3b fecf492a

y:
a28a43b9 5d736b5a 5afeb5a0 7d2c8965 ebf352a3 e29ba7e3 6511120c cca2b760
51cdfb87 fd9ee758 e5b11598 6663186f 468327bf 5ac500f1 89cb706f 6216abbc
4bb7258f 92150606 5db33698 3c31267c e78c9427 fab8dad0 c64b54f1 eff60ec6
01dd1abc 25d95693 803794d9 6733d565 69931f07 c772a513 2383ac6e abdafbc4
```

Last update: 2013.

DsaKeyChecker.java - Reading and Checking DSA Keys

This section provides a tutorial example on how to read DSA public key and private key back from the key files, assuming they are using PKCS#8 and X.509 encoding formats.

In my DsaKeyGenerator.java program, DSA public key and private key are saved in 2 separate files. If you want to learn how read them back into Java programs, you can read my Java program, DsaKeyChecker.java:

```
/**
 * DsaKeyChecker.java
 * Copyright (c) 2013 by Dr. Herong Yang
 */
import java.io.*;
import java.math.*;
import java.security.*;
import java.security.interfaces.*;
import java.security.spec.*;
class DsaKeyChecker {
   public static void main(String[] a) {
      if (a.length<1) {
         System.out.println("Usage:");
         System.out.println("java DsaKeyChecker input");
         return;
      }
      String input = a[0];
      String algorithm = "DSA";
      try {
         KeyPair pair = readKeyPair(input,algorithm);
         checkDsaKeyPair(pair);
      } catch (Exception e) {
         System.out.println("Exception: "+e);
         return;
      }
   }
   private static KeyPair readKeyPair(String input,
         String algorithm) throws Exception {
      KeyFactory keyFactory = KeyFactory.getInstance(algorithm);

      String priKeyFile = input+".pri";
      FileInputStream priKeyStream = new FileInputStream(priKeyFile);
      int priKeyLength = priKeyStream.available();
      byte[] priKeyBytes = new byte[priKeyLength];
      priKeyStream.read(priKeyBytes);
      priKeyStream.close();
      PKCS8EncodedKeySpec priKeySpec
         = new PKCS8EncodedKeySpec(priKeyBytes);
      PrivateKey priKey = keyFactory.generatePrivate(priKeySpec);

      String pubKeyFile = input+".pub";
      FileInputStream pubKeyStream = new FileInputStream(pubKeyFile);
      int pubKeyLength = pubKeyStream.available();
      byte[] pubKeyBytes = new byte[pubKeyLength];
      pubKeyStream.read(pubKeyBytes);
      pubKeyStream.close();
      X509EncodedKeySpec pubKeySpec
         = new X509EncodedKeySpec(pubKeyBytes);
      PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);

      return new KeyPair(pubKey, priKey);
   }
   private static void checkDsaKeyPair(KeyPair pair) {
      DSAPublicKey pubKey = (DSAPublicKey) pair.getPublic();
      DSAPrivateKey priKey = (DSAPrivateKey) pair.getPrivate();
      DSAParams params = priKey.getParams();
      BigInteger p = params.getP();
      BigInteger q = params.getQ();
      BigInteger g = params.getG();
      BigInteger x = priKey.getX();
      BigInteger y = pubKey.getY();
      System.out.println();
      System.out.println("DSA Key Parameters: ");
      System.out.println("p = "+p);
      System.out.println("q = "+q);
      System.out.println("g = "+g);
      System.out.println("x = "+x);
      System.out.println("y = "+y);

      System.out.println();
      System.out.println("DSA Key Verification: ");
      System.out.println("What's key size? "+p.bitLength());
      System.out.println("Is p a prime? "+p.isProbablePrime(200));
      System.out.println("Is q a prime? "+q.isProbablePrime(200));
      System.out.println("Is p-1 mod q == 0? "
         +p.subtract(BigInteger.ONE).mod(q));
      System.out.println("Is g**q mod p == 1? "+g.modPow(q,p));
      System.out.println("Is q > x? "+(q.compareTo(x)==1));
      System.out.println("Is g**x mod p == y? "+g.modPow(x,p).equals(y));
   }
}
```

Some notes on DsaKeyChecker.java:

- The java.security.KeyFactory.getInstance(algorithm) method is used to initiate a key factory object for the specified algorithm, "DSA".
- The java.security.spec.PKCS8EncodedKeySpec constructor is used to load the private key back from the byte array, because it is encoded in PKCS#8 format.
- The java.security.KeyFactory.keyFactory.generatePrivate(priKeySpec) method is used to convert the private key back from private key spec.
- The java.security.spec.X509EncodedKeySpec constructor is used to load the public key back from the byte array, because it is encoded in X.509 format.
- The java.security.KeyFactory.generatePublic(pubKeySpec) method is used to convert the public key back from public key spec.
- The checkDsaKeyPair() method is used to retrieve DSA key parameters from the DSA key pair and check their properties.

Testing result is presented in the next tutorial.

Last update: 2013.

Example of DSA Key Parameters and Properties

This section provides 2 DsaKeyChecker.java output examples to show DSA key parameters and properties.

Let's compile and run DsaKeyChecker.java using JDK 1.6 on my first DSA key pair, dsa_512.pub and dsa_512.pri, generated in the previous tutorial:

```
C:\herong>javac DsaKeyChecker.java

C:\herong>java DsaKeyChecker dsa_512

DSA Key Parameters:
p = 13232376895198612407547930718267435757728527029623408872245156039
7577130290363687191464521860412042373505217852403370487520714627982773
003935646236777459223
q = 857393771208094202104259627990318636601332086981
g = 54216440574364751416096484883257051280474283943804743768346673007
66108262613900542681289080713724597310673074119355136085795982097390 6
708903671851411897 96
x = 792647853324835944125296675259316105451780620466
y = 10783827985936883407800478884376885258012329124816552994400318669
41712227984308664513720074342723253116776610426060680530302231490625 4
4035938031595830343 40

DSA Key Verification:
What's key size? 512
Is p a prime? true
Is q a prime? true
Is p-1 mod q == 0? 0
Is g**q mod p == 1? 1
Is q > x? true
Is g**x mod p == y? true
```

The output looks good. The key size is 512 bits, measured by the number of bits of the prime modulus "p".

Below the output on my second DSA key pair, dsa_1024.pub and dsa_1024.pri:

```
C:\herong>java DsaKeyChecker dsa_1024

DSA Key Parameters:
p = 17801190547854226652823756245015999014523215636912067427327445031
44428657887370207706126952521234630795671567847784664499706507709207 2
78570500096683881440341297452211718185060472311500393010799593580673 9
53487170663198022620197149665241350609459137075949565146728556906067 9
41358375427073717274295513433206952 39
q = 864205495604807476120572616017955259175325408501
g = 17406820753240209518581198012352343653860449079456135097849583104
05999534884558231478515974089409507253077970949157594923683005742524 3
87610370844734671801488761181030830437549851909834726015504946913294 8
80833954923138500003616464826446084923040787218189599990564960977693 6
80177492737089620066891879567442107 30
x = 774290984479563168206130828532207106685994961942
y = 11413953692062257086993806233172330674938775529337393031977771373
12974694691091424011302322172177773213681844413974439315769846504493 3
01344275875756827386236711535481600955480809120630409696336526664982 9
96691708547428329737507308545970320128723518000534012439700593480613 3
15262434484712051661304973108924241 32

DSA Key Verification:
What's key size? 1024
Is p a prime? true
Is q a prime? true
Is p-1 mod q == 0? 0
Is g**q mod p == 1? 1
Is q > x? true
Is g**x mod p == y? true
```

Last update: 2013.

java.security.Signature - The Data Signing Class

This section describes the data signing and verification Java class, java.security.Signature, and its major methods like update(), sign() and verify().

The java.security.Signature class is an abstract class providing a link to implementation classes of digital signature algorithms provided by various security package providers. Major methods in the KeyPairGenerator class:

getInstance() - Returns a Signature object of the specified algorithm from the implementation of the specified provider. If provider is not specified, the default implementation is used. This is a static method.

initSign() - Initializes the current Signature object with the specified private key to be ready to take input data for generating a new signature.

initVerify() - Initializes the current Signature object with the specified public key to be ready to take input data for verifying an existing signature.

update() - Adds more data to the current Signature object for signature generation or signature verification.

sign() - Generates a new signature for the input data received so far in the current Signature object, and returns the signature as a byte array. It also removes the input data.

verify() - Verifies the input data received so far in the current Signature object against the specified signature, and returns true or false. It also removes the input data.

getAlgorithm() - Returns the algorithm name of the current Signature object.

getProvider() - Returns the provider as a Provider object of the current Signature object.

See the next section on how to write a sample program.

Last update: 2013.

DsaSignatureGenerator.java - Generating DSA Digital Signature

This section provides tutorial example on how to generate a digital signature for a message file with a DSA private key using the SHA1withDSA algorithm.

Here my DSA signature generation program, DsaSignatureGenerator.java, using the java.security.Signature class:

```
/**
 * DsaSignatureGenerator.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.io.*;
import java.security.*;
import java.security.spec.*;
class DsaSignatureGenerator {
   public static void main(String[] a) {
      if (a.length<3) {
         System.out.println("Usage:");
         System.out.println("java DsaSignatureGenerator keyFile"
            + " msgFile sigFile");
         return;
      }
      String keyFile = a[0];
      String msgFile = a[1];
      String sigFile = a[2];
      String keyAlgo = "DSA";
      String sigAlgo = "SHA1withDSA";

      try {
         PrivateKey priKey = readPrivateKey(keyFile,keyAlgo);
         sign(msgFile,sigFile,sigAlgo,priKey);
      } catch (Exception e) {
         System.out.println("Exception: "+e);
         return;
      }
   }
   private static PrivateKey readPrivateKey(String input,
         String algorithm) throws Exception {
      KeyFactory keyFactory = KeyFactory.getInstance(algorithm);
      FileInputStream priKeyStream = new FileInputStream(input);
      int priKeyLength = priKeyStream.available();
      byte[] priKeyBytes = new byte[priKeyLength];
      priKeyStream.read(priKeyBytes);
      priKeyStream.close();
      PKCS8EncodedKeySpec priKeySpec
         = new PKCS8EncodedKeySpec(priKeyBytes);
      PrivateKey priKey = keyFactory.generatePrivate(priKeySpec);
      System.out.println();
      System.out.println("Private Key Info: ");
      System.out.println("Algorithm = "+priKey.getAlgorithm());
      return priKey;
   }
   private static byte[] sign(String input, String output,
      String algorithm, PrivateKey priKey) throws Exception {
      Signature sg = Signature.getInstance(algorithm);
      sg.initSign(priKey);
      System.out.println();
      System.out.println("Signature Object Info: ");
      System.out.println("Algorithm = "+sg.getAlgorithm());
      System.out.println("Provider = "+sg.getProvider());
      FileInputStream in = new FileInputStream(input);
      int bufSize = 1024;
      byte[] buffer = new byte[bufSize];
      int n = in.read(buffer,0,bufSize);
      int count = 0;
      while (n!=-1) {
         count += n;
         sg.update(buffer,0,n);
         n = in.read(buffer,0,bufSize);
      }
      in.close();
      FileOutputStream out = new FileOutputStream(output);
      byte[] sign = sg.sign();
      out.write(sign);
      out.close();
      System.out.println();
      System.out.println("Sign Processing Info: ");
      System.out.println("Number of input bytes = "+count);
      System.out.println("Number of output bytes = "+sign.length);
      return sign;
   }
}
```

Some notes on DsaSignatureGenerator.java:

- The private key is converted from the private key file encoded in PKCS#8 format. The public key is not needed for generating the signature file.
- The java.security.Signature.getInstance(algorithm) method is used to create the signature object with the specified algorithm: "SHA1withDSA".
- The initSign(privateKey) method is used at the beginning to initialize the signature object with the private key.
- The update() method is used repeatedly to hash the message file in chunks.
- The sign() method is used at the end to generate the signature as a byte array from the final hash value.

Last update: 2013.

DsaSignatureGenerator.java Test Results

This section provides test results from DsaSignatureGenerator.java on a binary file with different DSA private keys. DSA digital signature size is 46 bytes when generated with the Java SHA1withDSA algorithm.

Want to see some test output from DsaSignatureGenerator.java? Here is the first test on the DsaSignatureGenerator.class binary file with the DSA 512-bit private key generated in the previous tutorial:

```
C:\herong>java DsaSignatureGenerator
   dsa_512.pri DsaSignatureGenerator.class dsa_512.sig

Private Key Info:
Algorithm = DSA

Signature Object Info:
```

```
Algorithm = SHA1withDSA
Provider = SUN version 1.6

Sign Processing Info:
Number of input bytes = 2859
Number of output bytes = 46

C:\herong>java HexWriter dsa_512.sig dsa_512.hex
Number of input bytes: 46

C:\herong>type dsa_512.hex
302C02141F5BD5A1B756874480D223B7
1CB18A85D8BEE09C02147C9B3CAD3317
A757F8C9CFC51A7925437BA17D0D
```

My DsaSignatureGenerator.java program seems to be working. A 46-byte digital signature is generated from the DsaSignatureGenerator.class file with a 512-bit DSA private key using the SHA-1 hashing algorithm.

What happens if I run the program again with the input message and the same private key? Is the digital signature going to be different?

```
C:\herong>java DsaSignatureGenerator
   dsa_512.pri DsaSignatureGenerator.class dsa_512x.sig

Private Key Info:
Algorithm = DSA

Signature Object Info:
Algorithm = SHA1withDSA
Provider = SUN version 1.6

Sign Processing Info:
Number of input bytes = 2859
Number of output bytes = 46

C:\herong>java HexWriter dsa_512x.sig dsa_512x.hex
Number of input bytes: 46

C:\herong>type dsa_512x.hex
302C021459E171244C1A1C23EFE33B72
4464DF8971C436A4021473F09A1EB7DA
D1224C6ACE291DDE1B965C735E4E
```

The output shows that a different digital signature is generated. This proves that a random number, "k", is used in the DSA signature generation process. A different new digital signature will be generated each time the same input message and same private key is used.

Here is the second test with the 1024-bit private key on the same binary file, DsaSignatureGenerator.class:

```
C:\herong>java DsaSignatureGenerator
   dsa_1024.pri DsaSignatureGenerator.class dsa_1024.sig

Private Key Info:
Algorithm = DSA
```

```
Signature Object Info:
Algorithm = SHA1withDSA
Provider = SUN version 1.6

Sign Processing Info:
Number of input bytes = 2859
Number of output bytes = 46

C:\herong>java HexWriter dsa_1024.sig dsa_1024.hex
Number of input bytes: 46

C:\herong>type dsa_1024.hex
302C02147182608511EA79A038806F0A
6206D7D086DCD27B02145496D01B967A
73B9E60626B0523502A4C12697AE
```

Of course, we are getting a different digital signature. But it has the same length: 46 bytes.

Last update: 2013.

DsaSignatureVerifier.java - Verifying DSA Digital Signature

This section provides tutorial example on how to verifying a digital signature for a message file with the DSA public key using the SHA1withDSA algorithm.

When a DSA digital signature is generated, it should be delivered to the receiver together with the original document, and the DSA public key for her/him to verify.

Now let's pretend we are the document receiver and want to verify the digital signature to ensure that no one has modified the document or the signature. I wrote another Java program to do this using the java.security.Signature class:

```
/**
 * DsaSignatureVerifier.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.io.*;
import java.security.*;
import java.security.spec.*;
class DsaSignatureVerifier {
   public static void main(String[] a) {
      if (a.length<3) {
         System.out.println("Usage:");
         System.out.println("java DsaSignatureGenerator keyFile"
            + " msgFile sigFile");
         return;
      }
      String keyFile = a[0];
      String msgFile = a[1];
      String sigFile = a[2];
      String keyAlgo = "DSA";
      String sigAlgo = "SHA1withDSA";

      try {
         PublicKey pubKey = readPublicKey(keyFile,keyAlgo);
         byte[] sign = readSignature(sigFile);
         verify(msgFile,sigAlgo,sign,pubKey);
      } catch (Exception e) {
         System.out.println("Exception: "+e);
         return;
      }
   }
   private static PublicKey readPublicKey(String input,
         String algorithm) throws Exception {
      FileInputStream pubKeyStream = new FileInputStream(input);
      int pubKeyLength = pubKeyStream.available();
      byte[] pubKeyBytes = new byte[pubKeyLength];
      pubKeyStream.read(pubKeyBytes);
      pubKeyStream.close();
      X509EncodedKeySpec pubKeySpec
         = new X509EncodedKeySpec(pubKeyBytes);
      KeyFactory keyFactory = KeyFactory.getInstance(algorithm);
      PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);
      System.out.println();
      System.out.println("Public Key Info: ");
      System.out.println("Algorithm = "+pubKey.getAlgorithm());
      return pubKey;
   }
   private static byte[] readSignature(String input)
         throws Exception {
      FileInputStream signStream = new FileInputStream(input);
      int signLength = signStream.available();
      byte[] signBytes = new byte[signLength];
      signStream.read(signBytes);
      signStream.close();
      return  signBytes;
   }
   private static boolean verify(String input, String algorithm,
         byte[] sign, PublicKey pubKey) throws Exception {
      Signature sg = Signature.getInstance(algorithm);
      sg.initVerify(pubKey);
      System.out.println();
      System.out.println("Signature Object Info: ");
      System.out.println("Algorithm = "+sg.getAlgorithm());
      System.out.println("Provider = "+sg.getProvider());
      FileInputStream in = new FileInputStream(input);
      int bufSize = 1024;
      byte[] buffer = new byte[bufSize];
      int n = in.read(buffer,0,bufSize);
      int count = 0;
      while (n!=-1) {
         count += n;
         sg.update(buffer,0,n);
         n = in.read(buffer,0,bufSize);
      }
      in.close();
      boolean ok = sg.verify(sign);
      System.out.println("Verify Processing Info: ");
      System.out.println("Number of input bytes = "+count);
      System.out.println("Verification result = "+ok);
      return ok;
   }
}
```

Some notes on the sample program, DsaSignatureVerifier.java:

- The public key is converted from the public key file encoded in X.509 format. The receiver has no access to the private key.
- The java.security.Signature.getInstance(algorithm) method is used to create the signature object with the specified algorithm: "SHA1withDSA".
- The initSign(publicKey) method is used at the beginning to initialize the signature object with the public key.
- The update() method is used repeatedly to hash the message file in chunks.
- The verify(signature) method is used at the end to verify the signature against the final hash value.

Last update: 2013.

DsaSignatureVerifier.java Test Results

This section provides test results from DsaSignatureVerifier.java on DSA digital signatures generated on a binary file. The verification will pass if the original document and the digital signature have not been modified.

Now let's see the verification result of the first DSA digital signature generated previously.

```
C:\herong>java DsaSignatureVerifier
   dsa_512.pub DsaSignatureGenerator.class dsa_512.sig

Public Key Info:
Algorithm = DSA

Signature Object Info:
Algorithm = SHA1withDSA
Provider = SUN version 1.6
Verify Processing Info:
Number of input bytes = 2859
Verification result = true
```

Cool. The signature verification passed. So nobody changed DsaSignatureGenerator.class or dsa_512.sig.

What will happen if DsaSignatureGenerator.class is changed? Let's create a modified version by adding characters to the end of the file:

```
C:\herong>copy DsaSignatureGenerator.class+con
   DsaSignatureGenerator.modified

DsaSignatureGenerator.class
con
A^Z
        1 file(s) copied.
```

The DsaSignatureGenerator.java will tell us that the signature failed to pass the verification:

```
C:\herong>java DsaSignatureVerifier dsa_512.pub
   DsaSignatureGenerator.modified dsa_512.sig

Public Key Info:
Algorithm = DSA

Signature Object Info:
Algorithm = SHA1withDSA
Provider = SUN version 1.6
Verify Processing Info:
Number of input bytes = 529
Verification result = false
```

We can also verify the digital signature generated with the 1024-bit private key:

```
C:\herong>java DsaSignatureVerifier
   dsa_1024.pub DsaSignatureGenerator.class dsa_1024.sig

Public Key Info:
Algorithm = DSA

Signature Object Info:
Algorithm = SHA1withDSA
Provider = SUN version 1.6
Verify Processing Info:
Number of input bytes = 2859
Verification result = true
```

Perfect. The digital signature passed verification nicely.

Last update: 2013.

Private key and Public Key Pair Generation

This chapter provides tutorial notes and example codes on private key and public key pair generation. Topics include public key encryption algorithms, RSA, DSA and DiffieHellman; private key and public key pair generation class and sample program; RSA, DSA, and DiffieHellman key pair samples.

Conclusion:

- JDK supports 3 algorithms to generate private and public pairs: RSA, DSA, and DiffieHellman.
- JDK allows you to store private keys in files with PKCS#8 encoded format.
- JDK allows you to store public keys in files with X.509 encoded format.

Sample programs listed in this chapter have been tested with JDK 1.3 to 1.6.

Question: Key pairs are used to encrypt and decrypt data. But how to encrypt and decrypt data? Which classes and methods to use?

Private and Public Keys and Related Interfaces

This section describes private and public key pairs used in RSA and DSA encryption algorithms. JDK supports private and public keys with the java.security.Key interface.

What a private and public key pair? A private and public key pair is a private key and a public key used in a public key encryption algorithm.

Known private and public key pair generation algorithms are:

- RSA - Developed by MIT professors: Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman in 1977.
- DSA - The Digital Signature Algorithm.

- DiffieHellman

JDK supports private and pubic key pairs with 3 interfaces.

1. java.security.Key is the interface acting as a base to support common features of both private key and public key. Major methods include:

getAlgorithm() - Returns the algorithm name used to generate the key.

getEncoded() - Returns the key as a byte array in its primary encoding format, or null if this key does not support encoding.

getFormat() - Returns the name of the primary encoding format of this key, or null if this key does not support encoding.

2. java.security.PrivateKey is the interface representing a private key. It extends java.security.Key interface with no additional methods.

3. java.security.PublicKey is the interface representing a public key. It extends java.security.Key interface with no additional methods.

Last update: 2013.

KeyPair and KeyPairGenerator Classes

This section describes the KeyPair and KeyPairGenerator Classes. The KeyPairGenerator.generateKeyPair() can be used to generate a private and public key pair.

java.security.KeyPair is a final class representing a key pair (a public key and a private key). Major methods in the KeyPair class:

getPrivate() - Returns a PriviateKey object representing the private key in the key pair.

getPublic() - Returns a PublicKey object representing the public key in the key pair.

java.security.KeyPairGenerator is an abstract class providing a link to implementation classes of private and public key pair generration algorithms provided by various security package providers. Major methods in the KeyPairGenerator class:

getInstance() - Returns a KeyPairGenerator object of the specified algorithm from the implementation of the specified provider. If provider is not specified, the default implementation is used. This is a static method.

initialize() - Initializes the key pair generator with the specified key size.

generateKeyPair() - Generates a key pair and returns a KeyPair object.

getAlgorithm() - Returns the algorithm name of the current key pair generator object.

getProvider() - Returns the provider as a provider object of the current key pair generator object.

See the next section for a sample program on how to use the java.security.KeyPairGenerator class.

Last update: 2013.

Key Pair Sample Program - JcaKeyPair.java

This section provides a tutorial example on how to write simple program to generate a pair of private key and public key for the RSA or DSA algorithm.

The following sample program shows you how to invoke the key pair generation algorithms implemented by the default provider, Sun, and generate key pairs.

```
/**
 * JcaKeyPair.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.io.*;
import java.math.*;
import java.security.*;
import java.security.interfaces.*;
class JcaKeyPair {
   public static void main(String[] a) {
      if (a.length<3) {
         System.out.println("Usage:");
         System.out.println("java JcaKeyPair keySize output"
            +" algorithm");
         return;
      }
      int keySize = Integer.parseInt(a[0]);
      String output = a[1];
      String algorithm = a[2]; // RSA, DSA
      try {
         getKeys(keySize,output,algorithm);
      } catch (Exception e) {
         System.out.println("Exception: "+e);
         return;
      }
   }
   private static void getKeys(int keySize, String output,
         String algorithm) throws Exception {
      KeyPairGenerator kg = KeyPairGenerator.getInstance(algorithm);
      kg.initialize(keySize);
      System.out.println();
      System.out.println("KeyPairGenerator Object Info: ");
      System.out.println("Algorithm = "+kg.getAlgorithm());
      System.out.println("Provider = "+kg.getProvider());
      System.out.println("Key Size = "+keySize);
      System.out.println("toString = "+kg.toString());
      KeyPair pair = kg.generateKeyPair();
      PrivateKey priKey = pair.getPrivate();
      PublicKey pubKey = pair.getPublic();
      String fl = output+".pri";
      FileOutputStream out = new FileOutputStream(fl);
      byte[] ky = priKey.getEncoded();
      out.write(ky);
      out.close();
      System.out.println();
      System.out.println("Private Key Info: ");
      System.out.println("Algorithm = "+priKey.getAlgorithm());
      System.out.println("Saved File = "+fl);
      System.out.println("Size = "+ky.length);
      System.out.println("Format = "+priKey.getFormat());
      System.out.println("toString = "+priKey.toString());
      fl = output+".pub";
      out = new FileOutputStream(fl);
      ky = pubKey.getEncoded();
      out.write(ky);
      out.close();
      System.out.println();
      System.out.println("Public Key Info: ");
      System.out.println("Algorithm = "+pubKey.getAlgorithm());
      System.out.println("Saved File = "+fl);
      System.out.println("Size = "+ky.length);
      System.out.println("Format = "+pubKey.getFormat());
      System.out.println("toString = "+pubKey.toString());
   }
}
```

Note that:

- KeyPairGenerator.getInstance(algorithm) method is called to get default key pair generator of the specified algorithm.
- generateKeyPair() method is called to return a new pair of private key and public key.
- getPrivate() method is called to return the private key from the key pair.
- getPublic() method is called to return the public key from the key pair.
- getEncoded() method is called to convert the key to an encoded byte stream.

See next sections on test result of this sample program.

Last update: 2013.

DSA Private Key and Public Key Pair Sample

This section provides a tutorial example on how to run JcaKeyPair.java to generate a DSA private key and public key pair sample. Keys are stored PKCS#8 and X.509 encoding formats.

Here is the result of my first test of JcaKeyPair.java - generating a pair of DSA private key and public key. It is done with JDK 1.6.

```
javac -classpath . JcaKeyPair.java
```

```
java -cp . JcaKeyPair 512 dsa dsa

KeyPairGenerator Object Info:
Algorithm = DSA
Provider = SUN version 1.6
Key Size = 512
toString = sun.security.provider.DSAKeyPairGenerator@19b49e6

Private Key Info:
Algorithm = DSA
Saved File = dsa.pri
Size = 201
Format = PKCS#8
toString = Sun DSA Private Key
parameters:
p:
fca682ce 8e12caba 26efccf7 110e526d b078b05e decbcd1e b4a208f3 ae1617ae
01f35b91 a47e6df6 3413c5e1 2ed0899b cd132acd 50d99151 bdc43ee7 37592e17
q:
962eddcc 369cba8e bb260ee6 b6a126d9 346e38c5
g:
678471b2 7a9cf44e e91a49c5 147db1a9 aaf244f0 5a434d64 86931d2d 14271b9e
35030b71 fd73da17 9069b32e 2935630e 1c206235 4d0da20a 6c416e50 be794ca4

x:     4e3ca05f 1a902b97 46f0dec2 9d70f952 8f33ee77


Public Key Info:
Algorithm = DSA
Saved File = dsa.pub
Size = 243
Format = X.509
toString = Sun DSA Public Key
Parameters:
p:
fca682ce 8e12caba 26efccf7 110e526d b078b05e decbcd1e b4a208f3 ae1617ae
01f35b91 a47e6df6 3413c5e1 2ed0899b cd132acd 50d99151 bdc43ee7 37592e17
q:
962eddcc 369cba8e bb260ee6 b6a126d9 346e38c5
g:
678471b2 7a9cf44e e91a49c5 147db1a9 aaf244f0 5a434d64 86931d2d 14271b9e
35030b71 fd73da17 9069b32e 2935630e 1c206235 4d0da20a 6c416e50 be794ca4

y:
54e3a092 862564f4 32ff94ec dcf052d2 502f2a3b d45fcb06 a53c2b9f 2224b25a
8951fe72 d7cae350 5fb307ab de5f828c a5703417 505abc51 efd2c15a 56445bfd
```

The program seems to be working:

- Since I am not specifying the provider name, the implementation of the DSA algorithm provided in the default security package was selected. Of course, Sun is the provider of the default security package.
- The key pair generated from the generateKeyPair() method indeed has two keys, a private key and a public key.
- The private key was written to a file using PKCS#8 format, and the public key was written to another file using X.509 format.

In order to see the keys, I need to use my other program, HexWriter.java, to convert binary data to hex numbers. See chapter "Encoding Conversion" for details.

Here is how to look at DSA key files in hex numbers, 16 bytes per line:

```
javac HexWriter.java

java -cp . HexWriter dsa.pri dsa_pri.hex

type dsa_pri.hex
3081C60201003081A806072A8648CE38
040130819C024100FCA682CE8E12CABA
26EFCCF7110E526DB078B05EDECBCD1E
B4A208F3AE1617AE01F35B91A47E6DF6
3413C5E12ED0899BCD132ACD50D99151
BDC43EE737592E17021500962EDDCC36
9CBA8EBB260EE6B6A126D9346E38C502
40678471B27A9CF44EE91A49C5147DB1
A9AAF244F05A434D6486931D2D14271B
9E35030B71FD73DA179069B32E293563
0E1C2062354D0DA20A6C416E50BE794C
A4041602144E3CA05F1A902B9746F0DE
C29D70F9528F33EE77

java -cp . HexWriter dsa.pub dsa_pub.hex

type dsa_pub.hex
3081F03081A806072A8648CE38040130
819C024100FCA682CE8E12CABA26EFCC
F7110E526DB078B05EDECBCD1EB4A208
F3AE1617AE01F35B91A47E6DF63413C5
E12ED0899BCD132ACD50D99151BDC43E
E737592E17021500962EDDCC369CBA8E
BB260EE6B6A126D9346E38C502406784
71B27A9CF44EE91A49C5147DB1A9AAF2
44F05A434D6486931D2D14271B9E3503
0B71FD73DA179069B32E2935630E1C20
62354D0DA20A6C416E50BE794CA40343
00024054E3A092862564F432FF94ECDC
F052D2502F2A3BD45FCB06A53C2B9F22
24B25A8951FE72D7CAE3505FB307ABDE
5F828CA5703417505ABC51EFD2C15A56
445BFD
```

Last update: 2013.

RSA Private Key and Public Key Pair Sample

This section provides a tutorial example on how to run JcaKeyPair.java to generate a RSA private key and public key pair sample. Keys are stored PKCS#8 and X.509 encoding formats.

Now let's see the private key and public key generated by the RSA algorithm:

```
java -cp . JcaKeyPair 512 rsa rsa
```

```
KeyPairGenerator Object Info:
Algorithm = rsa
Provider = SunRsaSign version 1.5
Key Size = 512
toString = java.security.KeyPairGenerator$Delegate@a59698

Private Key Info:
Algorithm = RSA
Saved File = rsa.pri
Size = 346
Format = PKCS#8
toString = Sun RSA private CRT key, 512 bits
  modulus:          90869450415146058688797477200948425302945076773547
1740987359289561440861968860814477403774349719761641670312566894138086
6493349088794356554895149433555027
  public exponent:  65537
  private exponent: 89365058183270423953039885874475912959479623544084
4479456143566699940284657762576258282420226939967257905899144258740638
4754958587400493169361356902030209
  prime p:          10156561001330124071320723955895014468217435540658
9305284428666903702505233009
  prime q:          89468719188754548893545560595594841381237600305314
352142924213312069293984003
  prime exponent p: 93508487983621011980308809077436163233486980736420
426663592427234014400426465
  prime exponent q: 39924206061844862938366722914051164017185614552526
332124140845908593107749243
  crt coefficient:  82979745043413288095388081210478420482729140505221
184605143714377105157807297

Public Key Info:
Algorithm = RSA
Saved File = rsa.pub
Size = 94
Format = X.509
toString = Sun RSA public key, 512 bits
  modulus: 90869450415146058688797477200948425302945076773547174098735
9289561440861968860814477403774349719761641670312566894138086649334908
8794356554895149433555027
  public exponent: 65537

java -cp . HexWriter rsa.pri rsa_pri.hex

type rsa_pri.hex
30820156020100300D06092A864886F7
0D0101010500048201403082013C0201
00024100AD800FE8B7445E8B1C84527E
02899F585CFFDB3548C36EBC29FCE7AC
3E44CCC421FACBAA98475FB4764E8F3D
3EE8C1BEFB554882E6AD9ABC8404F4A4
F0E308530203010001024100AAA0BB04
9ED7BA330D4484EC300AB08EF2471D89
F5995D99E7A135260BC715A85E755563
1A89D80E55D91C898AF4DE5405A553A0
403249C4C610C503CD66DB81022100E0
8C191D98B8C1B20E6BD54E20CE60CB1E
712FB4E92DE0515BCDDEBF3CE79A7102
2100C5CD8023172DB0FE9DF0286C50BE
66312876C0869B69DBD9A847D1AC3E42
4903022100CEBBEDBBB40A163B0ACFF8
F90F7732E28F4A8233BBA3832D24AAAB
F3C1ED31E1022058444CC2DBEC02C88C
```

```
380801D5C2311E0C9D796A57DDD4427B
8A98F110D3497B022100B774DF0EF99B
53EF4B8E3B918604C0E362120BC11A2D
E8889E1872AB15965CC1

java -cp . HexWriter rsa.pub rsa_pub.hex

type rsa_pub.hex
305C300D06092A864886F70D01010105
00034B003048024100AD800FE8B7445E
8B1C84527E02899F585CFFDB3548C36E
BC29FCE7AC3E44CCC421FACBAA98475F
B4764E8F3D3EE8C1BEFB554882E6AD9A
BC8404F4A4F0E308530203010001
```

Comparing the default implementation of DSA algorithm with RSA algorithm, the DSA implementation returns all the internal parameters of the keys in the toString() method.

Last update: 2013.

DiffieHellman Private Key and Public Key Pair Sample

This section provides a tutorial example on how to run JcaKeyPair.java to generate a DiffieHellman private key and public key pair sample. Keys are stored PKCS#8 and X.509 encoding formats.

Now let's see the private key and public key generated by the DiffieHellman algorithm:

```
java -cp . JcaKeyPair 512 diff DiffieHellman

KeyPairGenerator Object Info:
Algorithm = DiffieHellman
Provider = SunJCE version 1.6
Key Size = 512
toString = java.security.KeyPairGenerator$Delegate@89fbe3

Private Key Info:
Algorithm = DH
Saved File = diff.pri
Size = 212
Format = PKCS#8
toString = SunJCE Diffie-Hellman Private Key:
x:
1cdeaf4b 7ce125ff 405952a0 f2d0808e 3dcfb444 bceaebea 438e11b5 5db2a340
970d3aa8 bd04d347 0c01bc61 7383965e
p:
fca682ce 8e12caba 26efccf7 110e526d b078b05e decbcd1e b4a208f3 ae1617ae
01f35b91 a47e6df6 3413c5e1 2ed0899b cd132acd 50d99151 bdc43ee7 37592e17
g:
678471b2 7a9cf44e e91a49c5 147db1a9 aaf244f0 5a434d64 86931d2d 14271b9e
35030b71 fd73da17 9069b32e 2935630e 1c206235 4d0da20a 6c416e50 be794ca4
l:
384

Public Key Info:
Algorithm = DH
```

```
Saved File = diff.pub
Size = 226
Format = X.509
toString = SunJCE Diffie-Hellman Public Key:
y:
47c4fb37 511b7185 c1c67cbf c90d9d64 5ee34431 64097cac ee26b779 acd54a41
296bdf75 b93b7f2b 90052d35 e26e2204 6e562b03 3519cd94 2cddebe8 f96a97e1
p:
fca682ce 8e12caba 26efccf7 110e526d b078b05e decbcd1e b4a208f3 ae1617ae
01f35b91 a47e6df6 3413c5e1 2ed0899b cd132acd 50d99151 bdc43ee7 37592e17
g:
678471b2 7a9cf44e e91a49c5 147db1a9 aaf244f0 5a434d64 86931d2d 14271b9e
35030b71 fd73da17 9069b32e 2935630e 1c206235 4d0da20a 6c416e50 be794ca4
l:
384

java -cp . HexWriter diff.pri diff_pri.hex

type diff_pri.hex
3081D102010030819706092A864886F7
0D010301308189024100FCA682CE8E12
CABA26EFCCF7110E526DB078B05EDECB
CD1EB4A208F3AE1617AE01F35B91A47E
6DF63413C5E12ED0899BCD132ACD50D9
9151BDC43EE737592E170240678471B2
7A9CF44EE91A49C5147DB1A9AAF244F0
5A434D6486931D2D14271B9E35030B71
FD73DA179069B32E2935630E1C206235
4D0DA20A6C416E50BE794CA402020180
043202301CDEAF4B7CE125FF405952A0
F2D0808E3DCFB444BCEAEBEA438E11B5
5DB2A340970D3AA8BD04D3470C01BC61
7383965E

java -cp . HexWriter diff.pub diff_pub.hex

type diff_pub.hex
3081DF30819706092A864886F70D0103
01308189024100FCA682CE8E12CABA26
EFCCF7110E526DB078B05EDECBCD1EB4
A208F3AE1617AE01F35B91A47E6DF634
13C5E12ED0899BCD132ACD50D99151BD
C43EE737592E170240678471B27A9CF4
4EE91A49C5147DB1A9AAF244F05A434D
6486931D2D14271B9E35030B71FD73DA
179069B32E2935630E1C2062354D0DA2
0A6C416E50BE794CA402020180034300
024047C4FB37511B7185C1C67CBFC90D
9D645EE3443164097CACEE26B779ACD5
4A41296BDF75B93B7F2B90052D35E26E
22046E562B033519CD942CDDEBE8F96A
97E1
```

Last update: 2013.

PKCS#8/X.509 Private/Public Encoding Standards

This chapter provides tutorial notes and example codes on PKCS#8 and X.509 key encoding standards. Topics include PKCS#8 standard for private key encoding; X.509 standard for public key encoding; PKCS8EncodedKeySpec and X509EncodedKeySpec classes to decode encoded keys; JcaKeyFactoryTest.java test program to read and convert keys back from encoded key files.

Conclusion:

- Two encoding standards, PKCS#8 and X.509, are supported by JDK.
- java.security.spec.PKCS8EncodedKeySpec class can be used to convert private keys encoded as byte strings into key spec objects.
- java.security.spec.X509EncodedKeySpec class can be used to convert public keys encoded as byte strings into key spec objects.
- java.security.KeyFactory class can be used to convert key spec objects back to private or public key objects.

Sample programs listed in this chapter have been tested with JDK 1.3 to 1.6.

What Is Key Encoding?

This section describes private and public key encoding standards: PKCS#8 is used for encoding private keys and X.509 is used for encoding public keys.

What is **Key Encoding**? Key encoding is the process of converting encryption and decryption keys, or private and public keys, into a specific encoding format for storing them in files or transmitting them to remote systems.

As you can see from the previous chapter, JDK supports two commonly used key encoding standards:

- PKCS#8 - PKCS stands for Public-Key Cryptography Standards, developed by RSA Security currently a division of EMC. PKCS#8 describes syntax for private-key information, including a private key for some public-key algorithm and a set of attributes. PKCS#8 is mainly used to encode private keys.
- X.509 - X.509 is an ITU-T standard for a public key infrastructure (PKI) for single sign-on and Privilege Management Infrastructure (PMI). X.509 specifies, amongst other things, standard formats for public key certificates, certificate revocation lists, attribute certificates, and a certification path validation algorithm.

To manage different key encoding standards, JDK offers a group of classes:

- KeyFactory - A class to convert keys between Key objects and EncodedKeySpec objects.
- EncodedKeySpec - An abstract class offers a grouping point of all sub classes that represent various key encoding standards.
- PKCS8EncodedKeySpec - A sub class of EncodedKeySpec represents the ASN.1 encoding of a private key based on PKCS#8 standard.
- X509EncodedKeySpec - A sub class of EncodedKeySpec represents the ASN.1 encoding of a public key based on X.509 standard.

See next sections on how to use these key encoding classes.

Last update: 2013.

PKCS#8 and X.509 Key Encoding Classes

This section describes 2 JDK classes: PKCS8EncodedKeySpec representing the PKCS#8 encoding standard and the X.509 encoding standard.

java.security.spec.PKCS8EncodedKeySpec - A sub class of the EncodedKeySpec class represents the ASN.1 encoding of a private key based on the PKCS#8 encoding standard. It has three methods:

- PKCS8EncodedKeySpec() - Constructs a PKCS8EncodedKeySpec object from the specified byte array that contains the PKCS#8 encoded private key.
- getEncoded() - Returns the encoded key in a byte array of this object.
- getFormat() - Returns the name of encoding used in this object.

java.security.spec.X509EncodedKeySpec - A sub class of EncodedKeySpec represents the ASN.1 encoding of a private key based on the X.509 encoding standard. It has three methods:

- X509EncodedKeySpec() - Constructs a X509EncodedKeySpec object from the specified byte array that contains the X509 encoded public key.

- getEncoded() - Returns the encoded key in a byte array of this object.
- getFormat() - Returns the name of encoding used in this object.

Last update: 2013.

java.security.KeyFactory - Reading Encoded Keys

This section describes the java.security.KeyFactory class, which allows you to convert key spec objects back to key objects. Full process of reading and converting encoded key files back to key objects are also provided.

In the previous chapter, we learned how to write keys into files as encoded byte strings generated by the java.security.Key.getEncoded() method. Now we want to learn how to read those encoded type strings back and convert them into keys using 3 classes: java.security.spec.PKCS8EncodedKeySpec, java.security.spec.X509EncodedKeySpec, and java.security.KeyFactory.

PKCS8EncodedKeySpec and X509EncodedKeySpec classes are described in the previous section. Here is a quick description of the KeyFactory class.

java.security.KeyFactory - A class to convert keys between Key objects and EncodedKeySpec objects. Major methods offered:

getInstance() - Returns a KeyFactory object of the specified algorithm and the specified security package provider. If not specified, the default security package provider will be used.

generatePrivate() - Generates a PrivateKey object based on the specified EncodedKeySpec object, and returns it.

generatePublic() - Generates a PublicKey object based on the specified EncodedKeySpec object, and returns it.

getKeySpec() - Converts a Key object to an EncodedKeySpec object, and returns it.

getAlgorithm() - Returns the algorithm name of this object.

getProvider() - Returns the provider as a provider object of this object.

To read an encoded byte string and convert it back a key, you need to follow these steps:

- Read the encoded byte string from a file into a byte array.
- Convert the byte array into a key spec object using the PKCS8EncodedKeySpec or X509EncodedKeySpec constructor.
- Then convert the key spec object into a key object using the generatePrivate() or

generatePublic() of a KeyFactory object of the correct encryption algorithm.

See the next section on how to write a sample program.

Last update: 2013.

JcaKeyFactoryTest.java - Key Factory Test Program

This section provides a tutorial example on how to write a sample program to read encoded key files into key spec objects and convert them back into key objects.

The following sample program shows you how to use the KeyFactory class to read in encoded keys from files:

```
/**
 * JcaKeyFactoryTest.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.io.*;
import java.math.*;
import java.security.*;
import java.security.interfaces.*;
import java.security.spec.*;
class JcaKeyFactoryTest {
   public static void main(String[] a) {
      if (a.length<3) {
         System.out.println("Usage:");
         System.out.println("java JcaKeyFactoryTest keySize output"
            +" algorithm");
         return;
      }
      int keySize = Integer.parseInt(a[0]);
      String output = a[1];
      String algorithm = a[2]; // RSA, DSA
      try {
         writeKeys(keySize,output,algorithm);
         readKeys(output,algorithm);
      } catch (Exception e) {
         System.out.println("Exception: "+e);
         return;
      }
   }
   private static void writeKeys(int keySize, String output,
         String algorithm) throws Exception {
      KeyPairGenerator kg = KeyPairGenerator.getInstance(algorithm);
      kg.initialize(keySize);
      System.out.println();
      System.out.println("KeyPairGenerator Object Info: ");
      System.out.println("Algorithm = "+kg.getAlgorithm());
      System.out.println("Provider = "+kg.getProvider());
      System.out.println("Key Size = "+keySize);
      System.out.println("toString = "+kg.toString());
      KeyPair pair = kg.generateKeyPair();
      PrivateKey priKey = pair.getPrivate();
      PublicKey pubKey = pair.getPublic();
      String fl = output+".pri";
      FileOutputStream out = new FileOutputStream(fl);
      byte[] ky = priKey.getEncoded();
      out.write(ky);
      out.close();
      System.out.println();
      System.out.println("Private Key Info: ");
      System.out.println("Algorithm = "+priKey.getAlgorithm());
      System.out.println("Saved File = "+fl);
      System.out.println("Length = "+ky.length);
      System.out.println("Format = "+priKey.getFormat());
      System.out.println("toString = "+priKey.toString());

      fl = output+".pub";
      out = new FileOutputStream(fl);
      ky = pubKey.getEncoded();
      out.write(ky);
      out.close();
      System.out.println();
      System.out.println("Public Key Info: ");
      System.out.println("Algorithm = "+pubKey.getAlgorithm());
      System.out.println("Saved File = "+fl);
      System.out.println("Length = "+ky.length);
      System.out.println("Format = "+pubKey.getFormat());
      System.out.println("toString = "+pubKey.toString());
   }
   private static void readKeys(String input,
         String algorithm) throws Exception {
      KeyFactory keyFactory = KeyFactory.getInstance(algorithm);
      System.out.println();
      System.out.println("KeyFactory Object Info: ");
      System.out.println("Algorithm = "+keyFactory.getAlgorithm());
      System.out.println("Provider = "+keyFactory.getProvider());
      System.out.println("toString = "+keyFactory.toString());

      String priKeyFile = input+".pri";
      FileInputStream priKeyStream = new FileInputStream(priKeyFile);
      int priKeyLength = priKeyStream.available();
      byte[] priKeyBytes = new byte[priKeyLength];
      priKeyStream.read(priKeyBytes);
      priKeyStream.close();
      PKCS8EncodedKeySpec priKeySpec
         = new PKCS8EncodedKeySpec(priKeyBytes);
      PrivateKey priKey = keyFactory.generatePrivate(priKeySpec);
      System.out.println();
      System.out.println("Private Key Info: ");
      System.out.println("Algorithm = "+priKey.getAlgorithm());
      System.out.println("Saved File = "+priKeyFile);
      System.out.println("Length = "+priKeyBytes.length);
      System.out.println("toString = "+priKey.toString());

      String pubKeyFile = input+".pub";
      FileInputStream pubKeyStream = new FileInputStream(pubKeyFile);
      int pubKeyLength = pubKeyStream.available();
      byte[] pubKeyBytes = new byte[pubKeyLength];
      pubKeyStream.read(pubKeyBytes);
      pubKeyStream.close();
      X509EncodedKeySpec pubKeySpec
         = new X509EncodedKeySpec(pubKeyBytes);
      PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);
      System.out.println();
      System.out.println("Public Key Info: ");
      System.out.println("Algorithm = "+pubKey.getAlgorithm());
      System.out.println("Saved File = "+pubKeyFile);
      System.out.println("Length = "+pubKeyBytes.length);
      System.out.println("toString = "+pubKey.toString());
   }
}
```

Note that:

- The first part, writeKeys(), is designed to generate a private key and a public key and write them to 2 separate files.
- The second part, readKeys(), is designed to read the private key and the public key back from those 2 files.
- Of course, readKeys() can also be used to read in any encoded key files as long as they use encoding formats supported by JDK.

See the next section for testing result of JcaKeyFactoryTest.java.

Last update: 2013.

Reading DSA Private and Public Key Files

This section provides a tutorial example on running the test program JcaKeyFactoryTest.java to generate, write, read and convert DSA private and public keys.

Here is the result of my first test on JcaKeyFactoryTest.java for DSA private and public keys. It is done with JDK 1.6.

```
javac -classpath . JcaKeyFactoryTest.java

java -cp . JcaKeyFactoryTest 512 dsa DSA

KeyPairGenerator Object Info:
Algorithm = DSA
Provider = SUN version 1.6
Key Size = 512
toString = sun.security.provider.DSAKeyPairGenerator@10d448

Private Key Info:
Algorithm = DSA
Saved File = dsa.pri
Length = 201
Format = PKCS#8
toString = Sun DSA Private Key
parameters:
p:
fca682ce 8e12caba 26efccf7 110e526d b078b05e decbcd1e b4a208f3 ae1617ae
01f35b91 a47e6df6 3413c5e1 2ed0899b cd132acd 50d99151 bdc43ee7 37592e17
q:
962eddcc 369cba8e bb260ee6 b6a126d9 346e38c5
g:
678471b2 7a9cf44e e91a49c5 147db1a9 aaf244f0 5a434d64 86931d2d 14271b9e
```

```
35030b71 fd73da17 9069b32e 2935630e 1c206235 4d0da20a 6c416e50 be794ca4

x:     7bede213 af9e4dea 58ec0c53 da77353a a136804b

Public Key Info:
Algorithm = DSA
Saved File = dsa.pub
Length = 244
Format = X.509
toString = Sun DSA Public Key
Parameters:
p:
fca682ce 8e12caba 26efccf7 110e526d b078b05e decbcd1e b4a208f3 ae1617ae
01f35b91 a47e6df6 3413c5e1 2ed0899b cd132acd 50d99151 bdc43ee7 37592e17
q:
962eddcc 369cba8e bb260ee6 b6a126d9 346e38c5
g:
678471b2 7a9cf44e e91a49c5 147db1a9 aaf244f0 5a434d64 86931d2d 14271b9e
35030b71 fd73da17 9069b32e 2935630e 1c206235 4d0da20a 6c416e50 be794ca4

y:
e65f49f0 e5e41323 35a63e1e c0871f48 67d512d9 b54b8c82 218edba8 e32e236e
2bf3def4 1ac14365 38427c83 0ea43b9d c3dc737e dddd0200 6151c422 e5f15905

KeyFactory Object Info:
Algorithm = DSA
Provider = SUN version 1.6
toString = java.security.KeyFactory@13e205f

Private Key Info:
Algorithm = DSA
Saved File = dsa.pri
Length = 201
toString = Sun DSA Private Key
parameters:DSA
p:
fca682ce 8e12caba 26efccf7 110e526d b078b05e decbcd1e b4a208f3 ae1617ae
01f35b91 a47e6df6 3413c5e1 2ed0899b cd132acd 50d99151 bdc43ee7 37592e17
q:
962eddcc 369cba8e bb260ee6 b6a126d9 346e38c5
g:
678471b2 7a9cf44e e91a49c5 147db1a9 aaf244f0 5a434d64 86931d2d 14271b9e
35030b71 fd73da17 9069b32e 2935630e 1c206235 4d0da20a 6c416e50 be794ca4

x:     7bede213 af9e4dea 58ec0c53 da77353a a136804b


Public Key Info:
Algorithm = DSA
Saved File = dsa.pub
Length = 244
toString = Sun DSA Public Key
    Parameters:DSA
p:
fca682ce 8e12caba 26efccf7 110e526d b078b05e decbcd1e b4a208f3 ae1617ae
01f35b91 a47e6df6 3413c5e1 2ed0899b cd132acd 50d99151 bdc43ee7 37592e17
q:
962eddcc 369cba8e bb260ee6 b6a126d9 346e38c5
g:
678471b2 7a9cf44e e91a49c5 147db1a9 aaf244f0 5a434d64 86931d2d 14271b9e
35030b71 fd73da17 9069b32e 2935630e 1c206235 4d0da20a 6c416e50 be794ca4
```

```
y:
e65f49f0 e5e41323 35a63e1e c0871f48 67d512d9 b54b8c82 218edba8 e32e236e
2bf3def4 1ac14365 38427c83 0ea43b9d c3dc737e dddd0200 6151c422 e5f15905
```

The program seems to be working:

- A pair of keys is generated using the DSA algorithm first.
- The private key is then stored in a file using the default encoding format.
- The public key is also stored in a file using the default encoding format.
- A KeyFactory object is then created with DSA algorithm using the default security package provider.
- The private key is then read into the program and decoded into PrivateKey object with the help of PKCS8EncodedKeySpec class.
- The public key is also read into the program and decoded into PublicKey object with the help of X509EncodedKeySpec class.

Last update: 2013.

Reading RSA Private and Public Key Files

This section provides a tutorial example on running the test program JcaKeyFactoryTest.java to generate, write, read and convert RSA private and public keys.

Here is the result of my second test on JcaKeyFactoryTest.java for RSA private and public keys. It is done with JDK 1.3.1.

```
java -cp . JcaKeyFactoryTest 512 rsa RSA

KeyPairGenerator Object Info:
Algorithm = RSA
Provider = SunRsaSign version 1.5
Key Size = 512
toString = java.security.KeyPairGenerator$Delegate@a59698

Private Key Info:
Algorithm = RSA
Saved File = rsa.pri
Length = 344
Format = PKCS#8
toString = Sun RSA private CRT key, 512 bits
  modulus:          97043338366949517069564373997756011362690918019854
39417581528518529957501810500158232695319631354806771122335643311613341
45991922330880714565095163873634439
  public exponent:  65537
  private exponent: 44482947462983207078242570750363738610961714271853
24100028483119843118441672388337721687858836501384242172329828256137344
14357872219950861380783503157622281
  prime p:          10113877693296832729678411411276047071299336819261
95714250675744070202186282777
```

```
  prime q:          95950674221883124976393580772166965334465293046924
608406056369328592349151507
  prime exponent p: 51357208257202083771147268467561159419224351131150
139882124964656472311148365
  prime exponent q: 74907613956824819663582421633995309739118089834030
405183140334167387871766905
  crt coefficient:  45674196662035780810780138565427379979561183703478
862392227968269998791179698

Public Key Info:
Algorithm = RSA
Saved File = rsa.pub
Length = 94
Format = X.509
toString = Sun RSA public key, 512 bits
  modulus: 97043338366949517069564373997756011362690918019854394175815
2851852995750181050015823269531963135480677112233564331161341459919223
3088071456509516387363439
  public exponent: 65537

KeyFactory Object Info:
Algorithm = RSA
Provider = SunRsaSign version 1.5
toString = java.security.KeyFactory@1e0cf70

Private Key Info:
Algorithm = RSA
Saved File = rsa.pri
Length = 344
toString = Sun RSA private CRT key, 512 bits
  modulus:          97043338366949517069564373997756011362690918019854
3941758152851852995750181050015823269531963135480677112233564331161341
4599192233088071456509516387363439
  public exponent:  65537
  private exponent: 44482947462983207078242570750363738610961714271853
2410002848311984311844167238833772168785883650138424217232982825613734
1435787221995086138078350315762281
  prime p:          10113877693296832729678411411276047071299336819261
9571425067574407020218628277
  prime q:          95950674221883124976393580772166965334465293046924
608406056369328592349151507
  prime exponent p: 51357208257202083771147268467561159419224351131150
139882124964656472311148365
  prime exponent q: 74907613956824819663582421633995309739118089834030
405183140334167387871766905
  crt coefficient:  45674196662035780810780138565427379979561183703478
862392227968269998791179698

Public Key Info:
Algorithm = RSA
Saved File = rsa.pub
Length = 94
toString = Sun RSA public key, 512 bits
  modulus:          97043338366949517069564373997756011362690918019854
3941758152851852995750181050015823269531963135480677112233564331161341
4599192233088071456509516387363439
  public exponent: 65537
```

The result confirms that my readKeys() method works perfectly.

Last update: 2013.

Cipher - Public Key Encryption and Decryption

This chapter provides tutorial notes and example codes on the cipher process. Topics include the cipher class, javax.crypto.Cipher; the cipher sample program and test results with DSA and RSA encryption algorithms.

Conclusions:

- Pairs of public keys and private keys are used for asymmetric encryption algorithms like DSA or RSA.
- The javax.crypto.Cipher class can be used to perform encryption or decryption with pairs of public keys and private keys.

javax.crypto.Cipher - The Public Key Encryption Class

This section provides a quick introduction of the cipher class, javax.crypto.Cipher, to encrypt input data with a public key.

In the previous chapter, I learned how to use classes and interfaces in the JCE (Java Cryptography Extension) package to generate and manage private and public keys. Now I am ready to learn how to use private keys to encrypt or cipher any input data using the javax.crypto.Cipher class.

javax.crypto.Cipher is a class that provides functionality of a cryptographic cipher for encryption and decryption. It has the following major methods.

getInstance() - Returns a Cipher object of the specified transformation (algorithm plus options) from the implementation of the specified provider. If provider is not specified, the default implementation is used. This is a static method.

init() - Initializes this cipher for the specified operation mode with the specified key or public key certificate. Two important operation modes are Cipher.ENCRYPT_MODE and Cipher.DECRYPT_MODE.

update() - Feeds additional input data to this cipher and generates partial output data.

doFinal() - Feeds the last part of the input data to this cipher and generates the last part of the output data.

getBlockSize() - Returns the block size of this cipher.

getAlgorithm() - Returns the algorithm name of this cipher.

getProvider() - Returns the provider as a Provider object of this cipher.

See the next section how to use the javax.crypto.Cipher class in a sample program.

Last update: 2013.

JcePublicCipher.java - Public Key Encryption Sample Program

This section provides a tutorial example on how to write a public key encryption and private key decryption program using the javax.crypto.Cipher class.

The following sample program shows you how to do encryption and decryption with a private and public key pair:

```
/**
 * JcePublicCipher.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.io.*;
import java.security.*;
import java.security.spec.*;
import javax.crypto.*;
import javax.crypto.spec.*;
class JcePublicCipher {
   public static void main(String[] a) {
      if (a.length<5) {
         System.out.println("Usage:");
         System.out.println("java JcePublicCipher algorithm mode"
            +" keyFile input output");
         return;
      }
      String algorithm = a[0];
      String mode = a[1];
      String keyFile = a[2];
      String input = a[3];
      String output = a[4];
      try {
         Key ky = readKey(algorithm, mode, keyFile);
         publicCipher(algorithm, mode, ky, input, output);
      } catch (Exception e) {
               e.printStackTrace();
         return;
      }
   }
   private static Key readKey(String algorithm, String mode,
      String input) throws Exception {
      KeyFactory keyFactory = KeyFactory.getInstance(algorithm);
      System.out.println();
      System.out.println("KeyFactory Object Info: ");
      System.out.println("Algorithm = "+keyFactory.getAlgorithm());
      System.out.println("Provider = "+keyFactory.getProvider());
      System.out.println("toString = "+keyFactory.toString());

      FileInputStream fis = new FileInputStream(input);
      int kl = fis.available();
      byte[] kb = new byte[kl];
      fis.read(kb);
      fis.close();
      Key ky = null;
      if (mode.equalsIgnoreCase("encrypt")) {
         X509EncodedKeySpec pubKeySpec
            = new X509EncodedKeySpec(kb);
         ky = keyFactory.generatePublic(pubKeySpec);
      } else if (mode.equalsIgnoreCase("decrypt")) {
         PKCS8EncodedKeySpec priKeySpec
            = new PKCS8EncodedKeySpec(kb);
         ky = keyFactory.generatePrivate(priKeySpec);
      } else
         throw new Exception("Invalid mode: "+mode);
      System.out.println();
      System.out.println("Key Object Info: ");
      System.out.println("Algorithm = "+ky.getAlgorithm());
      System.out.println("Saved File = "+input);
      System.out.println("Length = "+kl);
      System.out.println("Format = "+ky.getFormat());
      System.out.println("toString = "+ky.toString());
      return ky;
   }
   private static void publicCipher(String algorithm, String mode,
      Key ky, String input, String output) throws Exception {
      Cipher cf = Cipher.getInstance(algorithm);
      if (mode.equalsIgnoreCase("encrypt"))
         cf.init(Cipher.ENCRYPT_MODE,ky);
      else if (mode.equalsIgnoreCase("decrypt"))
         cf.init(Cipher.DECRYPT_MODE,ky);
      else
         throw new Exception("Invalid mode: "+mode);
      System.out.println();
      System.out.println("Cipher Object Info: ");
      System.out.println("Block Size = "+cf.getBlockSize());
      System.out.println("Algorithm = "+cf.getAlgorithm());
      System.out.println("Provider = "+cf.getProvider());
      System.out.println("toString = "+cf.toString());

      FileInputStream fis = new FileInputStream(input);
      FileOutputStream fos = new FileOutputStream(output);
      int bufSize = 1024;
      byte[] buf = new byte[bufSize];
      int n = fis.read(buf,0,bufSize);
      int fisSize = 0;
      int fosSize = 0;
      while (n!=-1) {
         fisSize += n;
         byte[] out = cf.update(buf,0,n);
         fosSize += out.length;
         fos.write(out);
         n = fis.read(buf,0,bufSize);
      }
      byte[] out = cf.doFinal();
      fosSize += out.length;
      fos.write(out);
      fis.close();
      fos.close();
      System.out.println();
      System.out.println("Cipher Process Info: ");
      System.out.println("Input Size = "+fisSize);
      System.out.println("Output Size = "+fosSize);
   }
}
```

Note that:

- This sample program requires a public key file to encrypt plaintext. A public key can be generated by my JcaKeyPair.java program as described in the previous chapter.
- This sample program requires a private key file to decrypt ciphertext. A private key can be generated by my JcaKeyPair.java program as described in the previous chapter.

See the next section for test result of this sample program.

Last update: 2013.

DSA Public Key Encryption Tests

This section provides the test result of DSA public key encryption and private key decryption using the javax.crypto.Cipher class.

Here is the result of my first test of JcePublicCipher.java with the DSA algorithm. It is done with JDK 1.6.0.

```
java JcePublicCipher DSA encrypt dsa.pub JcePublicCipher.java jce.cph

KeyFactory Object Info:
Algorithm = DSA
Provider = SUN version 1.6
toString = java.security.KeyFactory@42e816

Key Object Info:
Algorithm = DSA
Saved File = dsa.pub
Length = 244
Format = X.509
toString = Sun DSA Public Key
    Parameters:DSA
        p:     fca682ce 8e12caba 26efccf7 110e526d b078b05e decbcd1e
               b4a208f3 ae1617ae 01f35b91 a47e6df6 3413c5e1 2ed0899b
               cd132acd 50d99151 bdc43ee7 37592e17
        q:     962eddcc 369cba8e bb260ee6 b6a126d9 346e38c5
        g:     678471b2 7a9cf44e e91a49c5 147db1a9 aaf244f0 5a434d64
               86931d2d 14271b9e 35030b71 fd73da17 9069b32e 2935630e
               1c206235 4d0da20a 6c416e50 be794ca4
        y:     e82a134e b1f8f5ae b40c2709 ccfb9da6 314ef48c ec025a37
               0b7ed86d d35647e2 d3fb17f2 c0ed1bdf dce6ea96 de04a2f6
               8d87666e ed6a1123 588c289a 9ad741ed
```

```
Exception: java.security.NoSuchAlgorithmException: Cannot find any
provider supporting DSA
```

This tells us that DSA public key and private key are not designed for encryption. They are designed to do digital signing. DSA stands for Digital Signature Algorithm.

Last update: 2013.

RSA Public Key Encryption Tests

This section provides the test result of RSA public key encryption and private key decryption using the javax.crypto.Cipher class.

Here is the result of my first test of JcePublicCipher.java with the RSA algorithm. It is done with JDK 1.6.0.

```
java JcePublicCipher RSA encrypt rsa.pub JcePublicCipher.java jce.cph

KeyFactory Object Info:
Algorithm = rsa
Provider = SunRsaSign version 1.5
toString = java.security.KeyFactory@1fb8ee3

Key Object Info:
Algorithm = RSA
Saved File = rsa.pub
Length = 94
Format = X.509
toString = Sun RSA public key, 512 bits
  modulus: 898054043475188831003171521616804086973204086084387401880396
6115229515610960443502949757801056895070926940886672738034912224273421
8612911902966014500229283
  public exponent: 65537

Cipher Object Info:
Block Size = 0
Algorithm = rsa
Provider = SunJCE version 1.6
toString = javax.crypto.Cipher@94948a
javax.crypto.IllegalBlockSizeException: Data must not be longer tha...
        at com.sun.crypto.provider.RSACipher.a(DashoA13*..)
        at com.sun.crypto.provider.RSACipher.engineDoFinal(DashoA13*..
        at javax.crypto.Cipher.doFinal(DashoA13*..)
        at JcePublicCipher.publicCipher(JcePublicCipher.java:95)
        at JcePublicCipher.main(JcePublicCipher.java:25)
```

We have a problem here! The error says, the input plaintext must be less than 53 bytes! This is because the RSA algorithm uses PKCS1 padding schema, which can only encrypt only 53 bytes at a time if your key size is 64 bytes (512 bits). I guess that it always pads 11 bytes.

To continue the test, I created a text file with 53 bytes, RSA_Input_53_Bytes.txt. Here is the result of the encryption step of the second test:

```
java JcePublicCipher RSA encrypt rsa.pub RSA_Input_53_Bytes.txt rsa.cph
```

```
KeyFactory Object Info:
Algorithm = RSA
Provider = SunRsaSign version 1.5
toString = java.security.KeyFactory@1fb8ee3

Key Object Info:
Algorithm = RSA
Saved File = rsa.pub
Length = 94
Format = X.509
toString = Sun RSA public key, 512 bits
  modulus: 898054043475188831003171521616804086973204086084387401880396
61152295156109604435029497578010568950709269408866727380349122242734218
612911902966014500229283
  public exponent: 65537

Cipher Object Info:
Block Size = 0
Algorithm = RSA
Provider = SunJCE version 1.6
toString = javax.crypto.Cipher@94948a

Cipher Process Info:
Input Size = 53
Output Size = 64
```

Here is the result of the decryption step:

```
java JcePublicCipher RSA decrypt rsa.pri rsa.cph rsa.clr

KeyFactory Object Info:
Algorithm = RSA
Provider = SunRsaSign version 1.5
toString = java.security.KeyFactory@1fb8ee3

Key Object Info:
Algorithm = RSA
Saved File = rsa.pri
Length = 344
Format = PKCS#8
toString = Sun RSA private CRT key, 512 bits
  modulus:          89805404347518883100317152161680408697320408608438
7401880396115229515610960443502949757801056895070926940886672738034912
2242734218612911902966014500229283
  public exponent:  65537
  private exponent: 21604162609563799242559168423654627516089438975245
2077117450068704831516889727162134069384428521450311722251472165612082
8100869387183133950489977572509697
  prime p:          11254349822059181087482145123391808891111614563551
162474604711525238638222849
  prime q:          79796172828656045903942442466972784531823950142255
13047595715652869470626l667
  prime exponent p: 75694719899682109824240577825348664740727964103753
86281308285080504050l426689
  prime exponent q: 49831667079030257575272782441152252201564899331249
152768504791877105882832159
  crt coefficient:  18772505426601783312760396170055433476231007074472
732727249064343427657206138

Cipher Object Info:
```

```
Block Size = 0
Algorithm = RSA
Provider = SunJCE version 1.6
toString = javax.crypto.Cipher@94948a

Cipher Process Info:
Input Size = 64
Output Size = 53
```

Now checking the decryption output with the original text:

```

comp RSA_Input_53_Bytes.txt rsa.clr
Comparing RSA_Input_53_Bytes.txt and rsa.clr...
Files compare OK
```

Note that I was not able to test the RSA algorithm with JDK 1.4.1, because no RSA encryption implementation was included in JDK 1.4.1

Last update: 2013.

MD5 Mesasge Digest Algorithm

This chapter provides tutorial notes and example codes on the MD5 message digest algorithm. Topics include MD5 algorithm overview; using MD5 in Java, PHP, and Perl.

Conclusions:

- A message digest algorithm is a hash function that takes a bit sequence of any length and produces a bit sequence of a fixed small length.
- The output of a message digest is considered as a digital signature of the input data.
- MD5 is a message digest algorithm producing 128 bits of data.
- It uses constants derived to trigonometric Sine function.
- It loops through the original message in blocks of 512 bits, with 4 rounds of operations for each block, and 16 operations in each round.
- Most modern programming languages provides MD5 algorithm as built-in functions.

What Is MD5 Message Digest Algorithm?

This section describes what is MD5 - a message digest algorithm which takes as input a message of arbitrary length and produces as output a 128-bit 'fingerprint'.

Based on the MD5 RFC document, MD5 is message-digest algorithm, which takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" or "message digest" of the input. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given prespecified target message digest.

MD5 was designed by Ronald Rivest in 1991 to replace an earlier hash function, MD4. MD5 is more secure than MD4. However a number of weaknesses have been found in recent years. The most recent paper published in this area shows that a collision of MD5 can be found within one minute on a standard notebook PC, using a method called tunneling.

Despite its weaknesses, MD5 is widely used in digital signature processes. It's been implemented in many programming languages.

Last update: 2013.

MD5 Message Digest Algorithm Overview

This section describes the MD5 algorithm - a 5-step process of padding of '1000...', appending message length, dividing as 512-bit blocks, initializing 4 buffers, and 4-round of hashing each block.

MD5 algorithm is well described in RFC 1321 - The MD5 Message-Digest Algorithm, see http://www.ietf.org/rfc/rfc1321.txt. Below is a quick overview of the algorithm.

MD5 algorithm consists of 5 steps:

Step 1. Appending Padding Bits. The original message is "padded" (extended) so that its length (in bits) is congruent to 448, modulo 512. The padding rules are:

- The original message is always padded with one bit "1" first.
- Then zero or more bits "0" are padded to bring the length of the message up to 64 bits fewer than a multiple of 512.

Step 2. Appending Length. 64 bits are appended to the end of the padded message to indicate the length of the original message in bytes. The rules of appending length are:

- The length of the original message in bytes is converted to its binary format of 64 bits. If overflow happens, only the low-order 64 bits are used.
- Break the 64-bit length into 2 words (32 bits each).
- The low-order word is appended first and followed by the high-order word.

Step 3. Initializing MD Buffer. MD5 algorithm requires a 128-bit buffer with a specific initial value. The rules of initializing buffer are:

- The buffer is divided into 4 words (32 bits each), named as A, B, C, and D.
- Word A is initialized to: 0x67452301.
- Word B is initialized to: 0xEFCDAB89.
- Word C is initialized to: 0x98BADCFE.
- Word D is initialized to: 0x10325476.

Step 4. Processing Message in 512-bit Blocks. This is the main step of MD 5 algorithm, which loops through the padded and appended message in blocks of 512 bits each. For each input block, 4 rounds of operations are performed with 16 operations in each round. This step can be described in the following pseudo code slightly modified from the RFC 1321's version:

```
Input and predefined functions:
   A, B, C, D: initialized buffer words

   F(X,Y,Z) = (X AND Y ) OR (NOT X AND Z)
   G(X,Y,Z) = (X AND Z ) OR (Y AND NOT Z)
   H(X,Y,Z) = X XOR Y XOR Z
   I(X,Y,Z) = Y XOR (X OR NOT Z)

   T[1, 2, ..., 64]: Array of special constants (32-bit integers) as:
      T[i] = int(abs(sin(i)) * 2**32)

   M[1, 2, ..., N]: Blocks of the padded and appended message

   R1(a,b,c,d,X,s,i): Round 1 operation defined as:
      a = b + ((a + F(b,c,d) + X + T[i]) <<< s)

   R2(a,b,c,d,X,s,i): Round 1 operation defined as:
      a = b + ((a + G(b,c,d) + X + T[i]) <<< s)

   R3(a,b,c,d,X,s,i): Round 1 operation defined as:
      a = b + ((a + H(b,c,d) + X + T[i]) <<< s)

   R4(a,b,c,d,X,s,i): Round 1 operation defined as:
      a = b + ((a + I(b,c,d) + X + T[i]) <<< s)

Algorithm:
   For k = 1 to N do the following

     AA = A
     BB = B
     CC = C
     DD = D
     (X[0], X[1], ..., X[15]) = M[k] /* Divide M[k] into 16 words */

     /* Round 1. Do 16 operations. */
     R1(A,B,C,D,X[ 0], 7, 1)
     R1(D,A,B,C,X[ 1],12, 2)
     R1(C,D,A,B,X[ 2],17, 3)
     R1(B,C,D,A,X[ 3],22, 4)
     R1(A,B,C,D,X[ 4], 7, 5)
     R1(D,A,B,C,X[ 5],12, 6)
     R1(C,D,A,B,X[ 6],17, 7)
     R1(B,C,D,A,X[ 7],22, 8)
     R1(A,B,C,D,X[ 8], 7, 9)
     R1(D,A,B,C,X[ 9],12,10)
     R1(C,D,A,B,X[10],17,11)
     R1(B,C,D,A,X[11],22,12)
     R1(A,B,C,D,X[12], 7,13)
     R1(D,A,B,C,X[13],12,14)
     R1(C,D,A,B,X[14],17,15)
     R1(B,C,D,A,X[15],22,16)

     /* Round 2. Do 16 operations. */
     R2(A,B,C,D,X[ 1], 5,17)
     R2(D,A,B,C,X[ 6], 9,18)
     R2(C,D,A,B,X[11],14,19)
     R2(B,C,D,A,X[ 0],20,20)
     R2(A,B,C,D,X[ 5], 5,21)
     R2(D,A,B,C,X[10], 9,22)
     R2(C,D,A,B,X[15],14,23)
     R2(B,C,D,A,X[ 4],20,24)
     R2(A,B,C,D,X[ 9], 5,25)
     R2(D,A,B,C,X[14], 9,26)
     R2(C,D,A,B,X[ 3],14,27)
     R2(B,C,D,A,X[ 8],20,28)
     R2(A,B,C,D,X[13], 5,29)
     R2(D,A,B,C,X[ 2], 9,30)
     R2(C,D,A,B,X[ 7],14,31)
     R2(B,C,D,A,X[12],20,32)

     /* Round 3. Do 16 operations. */
     R3(A,B,C,D,X[ 5], 4,33)
     R3(D,A,B,C,X[ 8],11,34)
     R3(C,D,A,B,X[11],16,35)
     R3(B,C,D,A,X[14],23,36)
     R3(A,B,C,D,X[ 1], 4,37)
     R3(D,A,B,C,X[ 4],11,38)
     R3(C,D,A,B,X[ 7],16,39)
     R3(B,C,D,A,X[10],23,40)
     R3(A,B,C,D,X[13], 4,41)
     R3(D,A,B,C,X[ 0],11,42)
     R3(C,D,A,B,X[ 3],16,43)
     R3(B,C,D,A,X[ 6],23,44)
     R3(A,B,C,D,X[ 9], 4,45)
     R3(D,A,B,C,X[12],11,46)
     R3(C,D,A,B,X[15],16,47)
     R3(B,C,D,A,X[ 2],23,48)

     /* Round 4. Do 16 operations. */
     R4(A,B,C,D,X[ 0], 6,49)
     R4(D,A,B,C,X[ 7],10,50)
     R4(C,D,A,B,X[14],15,51)
     R4(B,C,D,A,X[ 5],21,52)
     R4(A,B,C,D,X[12], 6,53)
     R4(D,A,B,C,X[ 3],10,54)
     R4(C,D,A,B,X[10],15,55)
     R4(B,C,D,A,X[ 1],21,56)
     R4(A,B,C,D,X[ 8], 6,57)
     R4(D,A,B,C,X[15],10,58)
     R4(C,D,A,B,X[ 6],15,59)
     R4(B,C,D,A,X[13],21,60)
     R4(A,B,C,D,X[ 4], 6,61)
     R4(D,A,B,C,X[11],10,62)
     R4(C,D,A,B,X[ 2],15,63)
     R4(B,C,D,A,X[ 9],21,64)

     A = A + AA
     B = B + BB
     C = C + CC
     D = D + DD
   End of for loop

Output:
   A, B, C, D: Message digest
```

Step 5. Output. The contents in buffer words A, B, C, D are returned in sequence with low-order byte first.

Last update: 2013.

Using MD5 Message Digest in Java

This section provides a tutorial example on how to use MD5 message digest algorithm in Java. The JDK JCE package offers the MD5 algorithm through a generic message digest class, javax.security.MessageDigest.

Sun provides MD5 algorithm in Java under their JCE (Java Cryptography Extension) package, which is included in JDK 1.5.

Sun's implementation of MD5 can be accessed through a generic class called MessageDigest. Here are the main methods of MessageDigest class:

- getInstance("MD5") - Returns a message digest object represents a specific implementation of MD5 algorithm from the default provider, Sun.
- getProvider() - Returns the provider name of the current object.
- update(bytes) - Updates the input message by appending a byte array at the end.
- digest() - Performs MD5 algorithm on the current input message and returns the message digest as a byte array. This method also resets the input message to an empty byte string.
- reset() - Resets the input message to an empty byte string.

Here is a sample Java program to show you how to use the MessageDigest class to perform some tests on MD5 algorithms.

```
/**
 * JceMd5Test.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.security.*;
class JceMd5Test {
   public static void main(String[] a) {
      try {
         MessageDigest md = MessageDigest.getInstance("MD5");
         System.out.println("Message digest object info: ");
         System.out.println("   Algorithm = "+md.getAlgorithm());
         System.out.println("   Provider = "+md.getProvider());
         System.out.println("   toString = "+md.toString());

         String input = "";
         md.update(input.getBytes());
               byte[] output = md.digest();
         System.out.println();
         System.out.println("MD5(\""+input+"\") =");
         System.out.println("   "+bytesToHex(output));
         input = "abc";
         md.update(input.getBytes());
               output = md.digest();
         System.out.println();
         System.out.println("MD5(\""+input+"\") =");
         System.out.println("   "+bytesToHex(output));

         input = "abcdefghijklmnopqrstuvwxyz";
         md.update(input.getBytes());
               output = md.digest();
         System.out.println();
         System.out.println("MD5(\""+input+"\") =");
         System.out.println("   "+bytesToHex(output));

      } catch (Exception e) {
         System.out.println("Exception: "+e);
      }
   }
   public static String bytesToHex(byte[] b) {
      char hexDigit[] = {'0', '1', '2', '3', '4', '5', '6', '7',
                         '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'};
      StringBuffer buf = new StringBuffer();
      for (int j=0; j<b.length; j++) {
         buf.append(hexDigit[(b[j] >> 4) & 0x0f]);
         buf.append(hexDigit[b[j] & 0x0f]);
      }
      return buf.toString();
   }
}
```

If you run this sample program with JDK 1.5, you should get the following output:

```
Message digest object info:
   Algorithm = MD5
   Provider = SUN version 1.5
   toString = MD5 Message Digest from SUN, <initialized>

MD5("") =
   D41D8CD98F00B204E9800998ECF8427E

MD5("abc") =
   900150983CD24FB0D6963F7D28E17F72

MD5("abcdefghijklmnopqrstuvwxyz") =
   C3FCD3D76192E4007DFB496CCA67E13B
```

The output matches the testing result listed in RFC 1321.

Last update: 2013.

Using MD5 Message Digest in PHP

This section provides a tutorial example on how to use MD5 message digest algorithm in PHP. The PHP engine has a built-in function md5().

If you are interested in using MD5 in PHP, you can use the built-in function md5(). Here is a sample program showing you how to use md5() function:

```
<?php # PhpMd5Test.php
# Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
#
   $input = "";
   $output = md5($input);
   print("\n");
   print("MD5(\"".$input."\") =\n");
   print("   $output\n");

   $input = "abc";
   $output = md5($input);
   print("\n");
   print("MD5(\"".$input."\") =\n");
   print("   $output\n");

   $input = "abcdefghijklmnopqrstuvwxyz";
   $output = md5($input);
   print("\n");
   print("MD5(\"".$input."\") =\n");
   print("   $output\n");
?>
```

If you run this sample program with PHP 5, you should get:

```
MD5("") =
   d41d8cd98f00b204e9800998ecf8427e

MD5("abc") =
   900150983cd24fb0d6963f7d28e17f72

MD5("abcdefghijklmnopqrstuvwxyz") =
   c3fcd3d76192e4007dfb496cca67e13b
```

The output matches the testing result listed in RFC 1321.

Last update: 2013.

Using MD5 Message Digest in Perl

This section provides a tutorial example on how to use MD5 message digest algorithm in Perl. John Allen implemented the entire MD5 algorithm in 8 lines of Perl 5 code.

If you are interested in using MD5 in Perl, you can look a very interesting implementation by John Allen in 8 lines of Perl 5, see http://www.cypherspace.org/adam/rsa/md5.html. Here is a copy of John's code, stored in PerlMd5In8Lines.pl:

```
#!/usr/bin/perl -iH9T4C`>_-JXF8NMS^$#)4=@<,$18%"0X4!`L0%P8*#Q4``04``04#!P``
@A=unpack N4C24,unpack u,$^I;@K=map{int abs 2**32*sin$_}1..64;sub L{($x=pop)
<<($n=pop)|2**$n-1&$x>>32-$n}sub M{($x=pop)-($m=1+~0)*int$x/$m}do{$l+=$r=read
STDIN,$_,64;$r++,$_.="\x80"if$r<64&&!$p++;@W=unpack V16,$_."\0"x7;$W[14]=$l*8
if$r<57;($a,$b,$c,$d)=@A;for(0..63){$a=M$b+L$A[4+4*($_>>4)+$_%4],M&{(sub{$b&$c
|$d&~$b},sub{$b&$d|$c&~$d},sub{$b^$c^$d},sub{$c^($b|~$d)})[$z=$_/16]}+$W[($A[
20+$z]+$A[24+$z]*($_%16))%16]+$K[$_]+$a;($a,$b,$c,$d)=($d,$a,$b,$c)}$v=a;for(
@A[0..3]){$_=M$_+${$v++}}}while$r>56;print unpack(H32,pack V4,@A),"\n"
```

To test this Perl program on Windows, I did the following in a command window:

```
>copy con empty.txt
^Z
        1 file(s) copied.

>perl PerlMd5In8Lines.pl < empty.txt
d41d8cd98f00b204e9800998ecf8427e

>copy con abc.txt
abc^Z
        1 file(s) copied.

>perl PerlMd5In8Lines.pl < abc.txt
900150983cd24fb0d6963f7d28e17f72

>copy con a_to_z.txt
abcdefghijklmnopqrstuvwxyz^Z
        1 file(s) copied.

>perl PerlMd5In8Lines.pl < a_to_z.txt
c3fcd3d76192e4007dfb496cca67e13b
```

The output matches the testing result listed in RFC 1321. This proves that John's program works perfectly. Note that:

- "copy con file_name" command allows me to enter data from keyboard into a new file.
- ^Z stands for (Ctrl-Z). It sends an end-of-file signal to the "copy" command.

Last update: 2013.

SHA1 Mesasge Digest Algorithm

This chapter provides tutorial notes and example codes on the SHA1 message digest algorithm. Topics include SHA1 algorithm overview; using SHA1 in Java, PHP, and Perl.

Conclusions:

- A message digest algorithm is a hash function that takes a bit sequence of any length and produces a bit sequence of a fixed small length.
- The output of a message digest is considered as a digital signature of the input data.
- SHA1 is a message digest algorithm producing 160 bits of data.
- Most modern programming languages provides SHA1 algorithm as built-in functions.

What Is SHA1 Message Digest Algorithm?

This section describes what is SHA1 (Secure Hash Algorithm 1) - a message digest algorithm which takes as input a message of arbitrary length and produces as output a 160-bit 'fingerprint'.

SHA1 (Secure Hash Algorithm 1) is message-digest algorithm, which takes an input message of any length < 2^64 bits and produces a 160-bit output as the message digest.

Based on the SHA1 RFC document, the SHA-1 is called secure because it is computationally infeasible to find a message which corresponds to a given message digest, or to find two different messages which produce the same message digest. Any change to a message in transit will, with very high probability, result in a different message digest, and the signature will fail to verify.

The original specification of the algorithm was published in 1993 as the Secure Hash Standard, FIPS PUB 180, by US government standards agency NIST (National Institute of Standards and Technology). This version is now often referred to as "SHA0".

SHA-0 was withdrawn by the NSA shortly after publication and was superseded by the revised version, published in 1995 in FIPS PUB 180-1 and commonly referred to as "SHA1".

Last update: 2013.

SHA1 Message Digest Algorithm Overview

This section describes the SHA1 algorithm - a 6-step process of padding of '1000...', appending message length, preparing 80 process functions, preparing 80 constants, preparing 5 word buffers, processing input in 512 blocks.

SHA1 algorithm is well described in RFC 3174 - US Secure Hash Algorithm 1 (SHA1), see http://www.ietf.org/rfc/rfc3174.txt. Below is a quick overview of the algorithm.

SHA1 algorithm consists of 6 tasks:

Task 1. Appending Padding Bits. The original message is "padded" (extended) so that its length (in bits) is congruent to 448, modulo 512. The padding rules are:

- The original message is always padded with one bit "1" first.
- Then zero or more bits "0" are padded to bring the length of the message up to 64 bits fewer than a multiple of 512.

Task 2. Appending Length. 64 bits are appended to the end of the padded message to indicate the length of the original message in bytes. The rules of appending length are:

- The length of the original message in bytes is converted to its binary format of 64 bits. If overflow happens, only the low-order 64 bits are used.
- Break the 64-bit length into 2 words (32 bits each).
- The low-order word is appended first and followed by the high-order word.

Task 3. Preparing Processing Functions. SHA1 requires 80 processing functions defined as:

```
f(t;B,C,D) = (B AND C) OR ((NOT B) AND D)         ( 0 <= t <= 19)
f(t;B,C,D) = B XOR C XOR D                        (20 <= t <= 39)
f(t;B,C,D) = (B AND C) OR (B AND D) OR (C AND D)  (40 <= t <= 59)
f(t;B,C,D) = B XOR C XOR D                        (60 <= t <= 79)
```

Task 4. Preparing Processing Constants. SHA1 requires 80 processing constant words defined as:

```
K(t) = 0x5A827999         ( 0 <= t <= 19)
K(t) = 0x6ED9EBA1         (20 <= t <= 39)
K(t) = 0x8F1BBCDC         (40 <= t <= 59)
K(t) = 0xCA62C1D6         (60 <= t <= 79)
```

Task 5. Initializing Buffers. SHA1 algorithm requires 5 word buffers with the following initial values:

```
H0 = 0x67452301
```

```
   H1 = 0xEFCDAB89
   H2 = 0x98BADCFE
   H3 = 0x10325476
   H4 = 0xC3D2E1F0
```

Task 6. Processing Message in 512-bit Blocks. This is the main task of SHA1 algorithm, which loops through the padded and appended message in blocks of 512 bits each. For each input block, a number of operations are performed. This task can be described in the following pseudo code slightly modified from the RFC 3174's method 1:

```
Input and predefined functions:
   M[1, 2, ..., N]: Blocks of the padded and appended message
   f(0;B,C,D), f(1,B,C,D), ..., f(79,B,C,D): Defined as above
   K(0), K(1), ..., K(79): Defined as above
   H0, H1, H2, H3, H4, H5: Word buffers with initial values

Algorithm:
   For loop on k = 1 to N

     (W(0),W(1),...,W(15)) = M[k] /* Divide M[k] into 16 words */

     For t = 16 to 79 do:
         W(t) = (W(t-3) XOR W(t-8) XOR W(t-14) XOR W(t-16)) <<< 1

     A = H0, B = H1, C = H2, D = H3, E = H4

     For t = 0 to 79 do:
         TEMP = A<<<5 + f(t;B,C,D) + E + W(t) + K(t)
         E = D, D = C, C = B<<<30, B = A, A = TEMP
     End of for loop

     H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D, H4 = H4 + E
   End of for loop

Output:
   H0, H1, H2, H3, H4, H5: Word buffers with final message digest
```

Step 5. Output. The contents in H0, H1, H2, H3, H4, H5 are returned in sequence the message digest.

Last update: 2013.

Using SHA1 Message Digest in Java

This section provides a tutorial example on how to use SHA1 message digest algorithm in Java. The JDK JCE package offers the SHA1 algorithm through a generic message digest class, javax.security.MessageDigest.

Sun provides SHA1 algorithm in Java under their JCE (Java Cryptography Extension) package, which is included in JDK 1.5.

Sun's implementation of SHA1 can be accessed through a generic class called MessageDigest. Here are the main methods of MessageDigest class:

- getInstance("SHA1") - Returns a message digest object represents a specific implementation of SHA1 algorithm from the default provider, Sun.
- getProvider() - Returns the provider name of the current object.
- update(bytes) - Updates the input message by appending a byte array at the end.
- digest() - Performs SHA1 algorithm on the current input message and returns the message digest as a byte array. This method also resets the input message to an empty byte string.
- reset() - Resets the input message to an empty byte string.

Here is a sample Java program to show you how to use the MessageDigest class to perform some tests on SHA1 algorithms.

```
/**
 * JceSha1Test.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.security.*;
class JceSha1Test {
   public static void main(String[] a) {
      try {
         MessageDigest md = MessageDigest.getInstance("SHA1");
         System.out.println("Message digest object info: ");
         System.out.println("   Algorithm = "+md.getAlgorithm());
         System.out.println("   Provider = "+md.getProvider());
         System.out.println("   toString = "+md.toString());

         String input = "";
         md.update(input.getBytes());
               byte[] output = md.digest();
         System.out.println();
         System.out.println("SHA1(\""+input+"\") =");
         System.out.println("   "+bytesToHex(output));

         input = "abc";
         md.update(input.getBytes());
               output = md.digest();
         System.out.println();
         System.out.println("SHA1(\""+input+"\") =");
         System.out.println("   "+bytesToHex(output));

         input = "abcdefghijklmnopqrstuvwxyz";
         md.update(input.getBytes());
               output = md.digest();
         System.out.println();
         System.out.println("SHA1(\""+input+"\") =");
         System.out.println("   "+bytesToHex(output));

      } catch (Exception e) {
         System.out.println("Exception: "+e);
      }
   }
   public static String bytesToHex(byte[] b) {
      char hexDigit[] = {'0', '1', '2', '3', '4', '5', '6', '7',
                         '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'};
      StringBuffer buf = new StringBuffer();
      for (int j=0; j<b.length; j++) {
         buf.append(hexDigit[(b[j] >> 4) & 0x0f]);
         buf.append(hexDigit[b[j] & 0x0f]);
      }
      return buf.toString();
   }
}
```

If you run this sample program with JDK 1.5, you should get the following output:

```
Message digest object info:
   Algorithm = SHA1
   Provider = SUN version 1.5
   toString = SHA1 Message Digest from SUN, <initialized>


SHA1("") =
   DA39A3EE5E6B4B0D3255BFEF95601890AFD80709

SHA1("abc") =
   A9993E364706816ABA3E25717850C26C9CD0D89D

SHA1("abcdefghijklmnopqrstuvwxyz") =
   32D10C7B8CF96570CA04CE37F2A19D84240D3A89
```

The output matches the testing result listed FreeBSD libmd makefile:
http://www.opensource.apple.com/source/libmd/libmd-2/Makefile

Last update: 2013.

Using SHA1 Message Digest in PHP

This section provides a tutorial example on how to use SHA1 message digest algorithm in PHP. The PHP engine has a built-in function sha1().

If you are interested in using SHA1 in PHP, you can use the built-in function sha1(). Here is a sample program showing you how to use sha1() function:

```
<?php # PhpSha1Test.php
# Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
#
   $input = "";
   $output = sha1($input);
   print("\n");
   print("SHA1(\"".$input."\") =\n");
   print("   $output\n");

   $input = "abc";
   $output = sha1($input);
   print("\n");
   print("SHA1(\"".$input."\") =\n");
   print("   $output\n");

   $input = "abcdefghijklmnopqrstuvwxyz";
   $output = sha1($input);
   print("\n");
   print("SHA1(\"".$input."\") =\n");
   print("   $output\n");
?>
```

If you run this sample program with PHP 5, you should get:

```
SHA1("") =
   da39a3ee5e6b4b0d3255bfef95601890afd80709

SHA1("abc") =
   a9993e364706816aba3e25717850c26c9cd0d89d

SHA1("abcdefghijklmnopqrstuvwxyz") =
   32d10c7b8cf96570ca04ce37f2a19d84240d3a89
```

The output matches the testing result listed FreeBSD libmd makefile: http://www.opensource.apple.com/source/libmd/libmd-2/Makefile

Last update: 2013.

Using SHA1 Message Digest in Perl

This section provides a tutorial example on how to use SHA1 message digest algorithm in Perl. John Allen implemented the entire SHA1 algorithm in 8 lines of Perl 5 code.

If you are interested in using SHA1 in Perl, you can look a very interesting implementation by John Allen in 8 lines of perl5, see http://www.cypherspace.org/adam/rsa/sha.html. Here is a copy of John's code, stored in PerlSha1In8Lines.pl:

```
#!/usr/bin/perl -iD9T4C`>_-JXF8NMS^$#)4=L/2X?!:@GF9;MGKH8\;O-S*8L'6
@A=unpack"N*",unpack u,$^I;@K=splice@A,5,4;sub M{($x=pop)-($m=1+~0)*int$x/$m};
sub L{$n=pop;($x=pop)<<$n|2**$n-1&$x>>32-$n}@F=(sub{$b&($c^$d)^$d},$S=sub{$b^$c
^$d},sub{($b|$c)&$d|$b&$c},$S);do{$l+=$r=read STDIN,$_,64;$r++,$_.="\x80"if$r<
64&&!$p++;@W=unpack N16,$_."\0"x7;$W[15]=$l*8 if$r<57;for(16..79){push@W,L$W[$_
-3]^$W[$_-8]^$W[$_-14]^$W[$_-16],1}($a,$b,$c,$d,$e)=@A;for(0..79){$t=M&{$F[$_/
20]}+$e+$W[$_]+$K[$_/20]+L$a,5;$e=$d;$d=$c;$c=L$b,30;$b=$a;$a=$t}$v='a';@A=map{
M$_+${$v++}}@A}while$r>56;printf'%.8x'x5 ."\n",@A
```

To test this Perl program on Windows, I did the following in a command window:

```
>copy con empty.txt
^Z
        1 file(s) copied.

>perl PerlSha1In8Lines.pl < empty.txt
da39a3ee5e6b4b0d3255bfef95601890afd80709

>copy con abc.txt
abc^Z
        1 file(s) copied.

>perl PerlSha1In8Lines.pl < abc.txt
a9993e364706816aba3e25717850c26c9cd0d89d
```

```
>copy con a_to_z.txt
abcdefghijklmnopqrstuvwxyz^Z
        1 file(s) copied.

>perl PerlSha1In8Lines.pl < a_to_z.txt
32d10c7b8cf96570ca04ce37f2a19d84240d3a89
```

The output matches the testing result listed FreeBSD libmd makefile: http://www.opensource.apple.com/source/libmd/libmd-2/Makefile The output proves that John's program works perfectly. Note that:

- "copy con file_name" command allows to copy enter data from keyboard into a new file.
- ^Z stands for (Ctrl-Z). It sends an end-of-file signal to the "copy" command.

Last update: 2013.

OpenSSL Introduction and Installation

This chapter provides tutorial notes and example codes on OpenSSL. Topics include introduction of OpenSSL; installing OpenSSL on Windows systems.

Conclusions:

- OpenSSL is a toolkit for SSL (Secure Sockets Layer and TLS (Transport Layer Security).
- OpenSSL also offers a command line tool to generate and manage private and public keys.
- Intalling OpenSSL on Windows needs a pre-compiled binary version of OpenSSL.

What Is OpenSSL?

This section describes what is OpenSSL - a cryptography toolkit for SSL (Secure Sockets Layer and TLS (Transport Layer Security). It offers command line tool for generating private keys and managing public keys.

OpenSSL is a cryptography toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) network protocols and related cryptography standards required by them.

The openssl program is a command line tool for using the various cryptography functions of OpenSSL's crypto library from the shell. It can be used for:

- Creation of RSA, DH and DSA key parameters
- Creation of X.509 certificates, CSRs and CRLs
- Calculation of Message Digests
- Encryption and Decryption with Ciphers
- SSL/TLS Client and Server Tests
- Handling of S/MIME signed or encrypted mail

Last update: 2013.

Installing OpenSSL on Windows

This section provides a tutorial example on how to install OpenSSL on a Windows system.

OpenSSL is officially distributed in C source code format. This is not a problem for Unix systems where C compiler is always available. But if you have a Windows system, you will have a hard time to install OpenSSL in C source code format. What you should do is to find a pre-compiled binary version for Windows. Here is how I installed OpenSSL on my Windows system:

1. Go to http://gnuwin32.sourceforge.net/packages/openssl.htm, and download the "Setup" version of "Binaries", openssl-0.9.7c-bin.exe.

2. Double click on openssl-0.9.7c-bin.exe to install OpenSSL to \local\gnuwin32 directory.

3. Go back to the same page, download the "Setup" version of "Documentation", and install it to the same directory.

4. Open command line window, and try the following command:

```
>\local\gnuwin32\bin\openssl -help
openssl:Error: '-help' is an invalid command.

Standard commands
asn1parse      ca             ciphers        crl            crl2pkcs7
dgst           dh             dhparam        dsa            dsaparam
enc            engine         errstr         gendh          gendsa
genrsa         nseq           ocsp           passwd         pkcs12
pkcs7          pkcs8          rand           req            rsa
rsautl         s_client       s_server       s_time         sess_id
smime          speed          spkac          verify         version
x509
......
```

If you see the list of commands printed by OpenSSL, you know that your installation is done correctly.

Last update: 2013.

OpenSSL Generating and Managing RSA Keys

This chapter provides tutorial notes and example codes on managing RSA keys with OpenSSL. Topics include generating new RSA keys; viewing existing RSA keys; encrypting RSA keys with DES algorithm to protect them with passwords.

Conclusions:

- "openssl genrsa -out key_file" command allows you to generate new RSA keys.
- "openssl rsa -in key_file -text" command allows you to view existing RSA keys.
- "openssl genrsa -des3 -out key_file" command allows you to generate new RSA keys and encrypt them with DES algorithm.
- "openssl rsa -in key_file -des -out encrypted_key_file" command allows you to encrypt existing RSA keys.

Generating New RSA Key Pairs

This section provides a tutorial example on how to generate a new pair of RSA priviate key and public key using the OpenSSL command line tool.

RSA is an asymmetric encryption algorithm developed in 1977 that use a pair of private key and public key. RSA is the initials of the developers of the RSA algorithm: Ron Rivest, Adi Shamir, and Leonard Adleman. Today, RSA is probably the most used the encryption algorithm for the Internet communication.

Here is how to use OpenSSL to generate a new pair of RSA private key and public key:

```
>openssl genrsa -out herong_rsa.key
Loading 'screen' into random state - done
Generating RSA private key, 512 bit long modulus
..++++++++++++
................++++++++++++
e is 65537 (0x10001)

>type herong_rsa.key
-----BEGIN RSA PRIVATE KEY-----
MIIBOwIBAAJBANoK3b+0NV1xrFLjsKFPLrxMReu3ezPxLjDWjktakq9gDGM5WUvI
CSENry/M1h2AhxGSxQluy4b1ynzBGWeO320CAwEAAQJAbQQn0NSKJflSvnLG+i/7
3vuHrg4j1FmOza5IoNZdJr9DyESMC+prebZkAFM2EW+ZLZy2JiEIqdDz79VAVRzs
ZQIhAPBvHYEWxCIcSYn8aG7o2lyY5/EB1gvwgAfSdWFlemUDAiEA6CijmKOX1WRd
```

```
KPf9g52Tpxk4TZzdjIpcbvYR7znIZs8CIBDxI3kXK5bju2LXwFwgWFKyC5X19Sk+
NydV8yN7zRYVAiEAni7CeUhONfmyeC2wsLL3Xg2TDV7qnc3QeVJ0mdl3MIUCIQCo
o0AdFXm789FfHuB+mVIKNtBLTAQNaMuXz6lXl7Ib7Q==
-----END RSA PRIVATE KEY-----
```

Note that:

- The output says generating private key. But it is actually generating a pair of private key and public key.
- The key pair is saved in an encoded format called PEM.

Last update: 2013.

Viewing Components of RSA Keys

This section provides a tutorial example on how to view different components of a pair of RSA private key and public key using the OpenSSL command line tool.

Here is how to see the components of a RSA key:

```
>openssl rsa -in herong_rsa.key -text
Private-Key: (512 bit)
modulus:
    00:da:0a:dd:bf:b4:35:5d:71:ac:52:e3:b0:a1:4f:
    2e:bc:4c:45:eb:b7:7b:33:f1:2e:30:d6:8e:4b:5a:
    92:af:60:0c:63:39:59:4b:c8:09:21:0d:af:2f:cc:
    d6:1d:80:87:11:92:c5:09:6e:cb:86:f5:ca:7c:c1:
    19:67:8e:df:6d
publicExponent: 65537 (0x10001)
privateExponent:
    6d:04:27:d0:d4:8a:25:f9:52:be:72:c6:fa:2f:fb:
    de:fb:87:ae:0e:23:d4:59:8e:cd:ae:48:a0:d6:5d:
    26:bf:43:c8:44:8c:0b:ea:6b:79:b6:64:00:53:36:
    11:6f:99:2d:9c:b6:26:21:08:a9:d0:f3:ef:d5:40:
    55:1c:ec:65
prime1:
    00:f0:6f:1d:81:16:c4:22:1c:49:89:fc:68:6e:e8:
    da:5c:98:e7:f1:01:d6:0b:f0:80:07:d2:75:61:65:
    7a:65:03
prime2:
    00:e8:28:a3:98:a3:97:d5:64:5d:28:f7:fd:83:9d:
    93:a7:19:38:4d:9c:dd:8c:8a:5c:6e:f6:11:ef:39:
    c8:66:cf
exponent1:
    10:f1:23:79:17:2b:96:e3:bb:62:d7:c0:5c:20:58:
    52:b2:0b:95:f5:f5:29:3e:37:27:55:f3:23:7b:cd:
    16:15
exponent2:
    00:9e:2e:c2:79:48:4e:35:f9:b2:78:2d:b0:b0:b2:
    f7:5e:0d:93:0d:5e:ea:9d:cd:d0:79:52:74:99:d9:
    77:30:85
coefficient:
    00:a8:a3:40:1d:15:79:bb:f3:d1:5f:1e:e0:7e:99:
    52:0a:36:d0:4b:4c:04:0d:68:cb:97:cf:a9:57:97:
    b2:1b:ed
```

```
writing RSA key
-----BEGIN RSA PRIVATE KEY-----
MIIBOwIBAAJBANoK3b+0NV1xrFLjsKFPLrxMReu3ezPxLjDWjktakq9gDGM5WUvI
CSENry/M1h2AhxGSxQluy4b1ynzBGWeO320CAwEAAQJAbQQn0NSKJflSvnLG+i/7
3vuHrg4j1FmOza5IoNZdJr9DyESMC+prebZkAFM2EW+ZLZy2JiEIqdDz79VAVRzs
ZQIhAPBvHYEWxCIcSYn8aG7o2lyY5/EB1gvwgAfSdWFlemUDAiEA6CijmKOX1WRd
KPf9g52Tpxk4TZzdjIpcbvYR7znIZs8CIBDxI3kXK5bju2LXwFwgWFKyC5X19Sk+
NydV8yN7zRYVAiEAni7CeUhONfmyeC2wsLL3Xg2TDV7qnc3QeVJ0mdl3MIUCIQCo
o0AdFXm789FfHuB+mVIKNtBLTAQNaMuXz6lXl7Ib7Q==
-----END RSA PRIVATE KEY-----
```

Last update: 2013.

Encrypting RSA Keys

This section provides a tutorial example on how to store RSA keys encrypted with password protection.

RSA keys are pairs of private key and public key. The private key should not be share with anyone else. So it's strongly recommended to store the RSA keys in an encrypted form with a password to protect them.

Here is how to use OpenSSL to generate a RSA key pair and encrypt it using the DES3 algorithm:

```
>openssl genrsa -des3 -out herong_rsa.key
Loading 'screen' into random state - done
Generating RSA private key, 2048 bit long modulus
...............................................+++
................................................................+++
e is 65537 (0x10001)
Enter pass phrase for user.key: mykey
Verifying - Enter pass phrase for user.key: mykey

>type herong_rsa.key
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,61523E68F580569D

MC5FNOEYflddyBF2orcTtzHSUpzrevcxZpbUU827hdmgDEoZKg54VVd9xGnxnodm
hq1LSenM1AxWO5Bzzmxr3WCiz94xPxNSUd/1f9eonaVZT7CaVzn533lj3G26uvtO
OyNXyBcb/kUGHXeCeGU322lB1p8gShOC/P9ip/wQvlR2yaSQGc4xKwON7O4dAvtM
rfoV0YJNCiK/tbK/5YBykMTYBsIAqJRmKKyAILd38dd0/lDTmLDxn2SEmMmuGjTC
yYEXZfW1PJn+gELSD1xysQ58wVtBXvdbQWG0RJYukseLurZABSyz4Lvg8fUboBAJ
42DFO101aaCWR/uuZefNPbPzBWrdh2w+ptqxWTKTOTYoqgrW15VdRE/4YH1N8R8a
wbzOS1oDbiRWH2WXcJ+E1dxh4UEoGuNkCV8W3nmVTvE=
-----END RSA PRIVATE KEY-----
```

Perfect. This extra parameter "-des3" triggers the "genrsa" command to generate a key pair and encrypt it immediately with DES-EDE3-CBC algorithm.

The above command is good for encrypting the key pair at the time of generation. How about encrypt a key pair previously generated? This could be done by the "rsa" command:

```
>openssl genrsa -out herong_rsa.key
Loading 'screen' into random state - done
Generating RSA private key, 512 bit long modulus
....++++++++++++
.++++++++++++
e is 65537 (0x10001)

>type herong_rsa.key
-----BEGIN RSA PRIVATE KEY-----
MIIBOwIBAAJBAKnmGcbuiAGG2XKek5LbVwF7AoT8HuNXXip7KyWevbrFlSxJWSjf
pmeGJo7/Nsw6hFwor28RyAy1wsW5BNYOXdECAwEAAQJAQsEsI6OZQLBRQ93Wsf8I
goZoiQPexwiO8TYPz+o9NeLELOzhYHiKuzOO5c2oVYXTSgM9IMCCo35fkzOlTdyj
oQIhANe/bnRWtO+7gSbcqmINtFW12pbkgzQ+SlQxp7HSNL3FAiEAyZjTrFGKlG9k
Ub4EcNFkWjIzOM/vHifYdmB/ZO9ZzJ0CIHEnEYMqxpLFQKNlMGdk0KPzUMW666VG
1iz6Lf1xRgARAiEAplhZiR27iKGlmKF/TowpDxfPFjjVaP+d6IfVdrbdVS0CIQCY
OHLGbU3QZn2VjSUH/BF4kP7cEPDngxbYiZ2+f2D77Q==
-----END RSA PRIVATE KEY-----

>openssl rsa -in herong_rsa.key -des -out herong_rsa_des.key
writing RSA key
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:

>type herong_rsa_des.key
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-CBC,C386818044590B20

10JtM182aDIEMcGLGHXC51woLVdcsYWAAp0tCI1NKJRy/ZBKQLs7gzgGD9ZFBA3D
eZ0W7CVT226yDNSAq/3G+st1cR3kfFmxO3cfT8DHKV4zJVLSRrKfklURp0SdfaB6
LLpbdz9OSwxYphVTBTQAaeLYBipZhyV5BJZeQH40b5S3SclHid5Bn3SaxmFIgRCp
X07GQkiVU+KLhW4Q2v7uV7qU/dlym7WAsxlw4vEw9EhLw2RTPGEC0IaTzPtgWnsE
wQcvS0gDg5C8sP/rpHdQcZFCqpt4+n9M/p1Ciz1d0DNYRefvZnmf9w/z02oT3KY+
nJxrL6kh2kYVUOQKSwlA4Swtt4lPy6gimg+1xG96+BnrG803FYQ23rlusCThg+yw
lHpltupnF9YW38dParIlLsxMxFRhRc8qNZSAwnBHP78=
-----END RSA PRIVATE KEY-----
```

Last update: 2013.

OpenSSL Managing Certificates

This chapter provides tutorial notes and example codes on managing certificates with OpenSSL. Topics include introduction of certificate, generating self-signed certificate; viewing internal components of certificates.

Conclusions:

- A certificate is a public key of a subject signed by an issuer.
- A self-signed certificate is a certificate that the issuer signed his own public key.
- "openssl req -new -key key_file -x509 -out certificate_file" command allows you to generate a self-signed certificate based on a private key file.
- "openssl x509 -in certificate_file -noout -text" command allows you to view different components of a certificate.

What Is a Certificate?

This section describes what is a certificate - A digitally signed statement from the issuer saying that the public key of the subject has some specific values.

Certificate: A digitally signed statement from the issuer saying that the public key of the subject has some specific values.

The above definition is copied from the JDK 1.3.1 documentation. It has a couple of important terms:

- "signed statement" - The certificate must be signed by the issuer with a digital signature.
- "issuer" - The person or organization who is issuing this certificate.
- "public key" - The public key of a key pair selected by the subject.
- "subject" - The person or organization who owns the public key.

X.509 Certificate - A certificate written in X.509 standard format. X.509 standard was introduction in 1988. It requires a certificate to have the following information:

- Version - X.509 standard version number.

- Serial Number - A sequence number given to each certificate.
- Signature Algorithm Identifier - Name of the algorithm used to sign this certificate by the issuer
- Issuer Name - Name of the issuer.
- Validity Period - Period during which this certificate is valid.
- Subject Name - Name of the owner of the public key.
- Subject Public Key Information - The public key and its related information.

Last update: 2013.

Generating Self-Signed Certificates

This section provides a tutorial example on how to generate a self-signed certificate for yourself with the OpenSSL command line tool.

A self-signed certificate is a certificate that the "issuer" is the "subject" himself. In other word, a seft-signed certificate is a certificate where the "issuer" signs his own public key with his private key.

If you want to generate a self-signed certificate for yourself, here what you to need to do:

- Enter your own name as the "subject".
- Provide your public key.
- Sign it with your private key.
- Put everything in the X.509 format.

That sounds like a lot of work. But OpenSSL can do everything for you in one shot with the "req" command. Before we try the "req" command, we need to make sure that you have the "openssl.cnf" installed on your local system. If you don't, go find a copy on the Web. If you can not find it, send me an email. I will send you my copy. Here is how the "openssl.cnf" looks like:

```
domain                  = some.com
dir                     = .

####################################################################
[ ca ]
default_ca      = CA_default            # The default ca section

####################################################################
[ CA_default ]

certs           = $dir/ssl.crt          # Where the issued certs are
crl_dir         = $dir/ssl.crl          # Where the issued crl are k
```

```
database        = $dir/.index.txt       # database index file.
new_certs_dir   = $dir/.issued          # default place for new cert

...
```

Here is the command to generated a self-signed certificate based on a RSA key pair file, herong_rsa_des.key, generated previously:

```

>openssl req -new -key herong_rsa_des.key -x509 -out herong.crt
   -config openssl.cnf

Enter pass phrase for herong_rsa_des.key:
You are about to be asked to enter information that will be
incorporated into your certificate request.
What you are about to enter is what is called a Distinguished
Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) []:CN
State or Province Name (full name) []:PN
Locality Name (eg, city) []:LN
Organization Name (eg, company) []:ON
Organizational Unit Name (eg, section) []:UN
Common Name (eg, YOUR name) []:Herong Yang
Email Address []:.

>type herong.crt
-----BEGIN CERTIFICATE-----
MIICUTCCAfugAwIBAgIBADANBgkqhkiG9w0BAQQFADBXMQswCQYDVQQGEwJDTjEL
MAkGA1UECBMCUE4xCzAJBgNVBAcTAkNOMQswCQYDVQQKEwJPTjELMAkGA1UECxMC
VU4xFDASBgNVBAMTC0hlcm9uZyBZYW5nMB4XDTA1MDcxNTIxMTk0N1oXDTA1MDgx
NDIxMTk0N1owVzELMAkGA1UEBhMCQ04xCzAJBgNVBAgTAlBOMQswCQYDVQQHEwJD
TjELMAkGA1UEChMCT04xCzAJBgNVBAsTAlVOMRQwEgYDVQQDEwtIZXJvbmcgWWFu
ZzBcMA0GCSqGSIb3DQEBAQUAA0sAMEgCQQCp5hnG7ogBhtlynpOS21cBewKE/B7j
V14qeyslnr26xZUsSVko36ZnhiaO/zbMOoRcKK9vEcgMtcLFuQTWDl3RAgMBAAGj
gbEwga4wHQYDVR0OBBYEFFXI70krXeQDxZgbaCQoR4jUDncEMH8GA1UdIwR4MHaA
FFXI70krXeQDxZgbaCQoR4jUDncEoVukWTBXMQswCQYDVQQGEwJDTjELMAkGA1UE
CBMCUE4xCzAJBgNVBAcTAkNOMQswCQYDVQQKEwJPTjELMAkGA1UECxMCVU4xFDAS
BgNVBAMTC0hlcm9uZyBZYW5nggEAMAwGA1UdEwQFMAMBAf8wDQYJKoZIhvcNAQEE
BQADQQA/ugzBrjjK9jcWnDVfGHlk3icNRq0oV7Ri32z/+HQX67aRfgZu7KWdI+Ju
Wm7DCfrPNGVwFWUQOmsPue9rZBgO
-----END CERTIFICATE-----
```

Note that:

- My information, as both "issuer" and "subject", is entered from the keyboard.
- My public key that is included in the certificate is supplied from my RSA key pair file, herong_rsa_des.key. See the previous chapter on how to generate a RSA key pair file.
- My private key that is used to sign the certificate is also supplied from my RSA key pair file, herong_rsa_des.key. But the private key itself will not be included in the certificate. So don't be afraid of send the self-signed certificate to others.
- The certificate is saved in an encoded format called PEM.

Last update: 2013.

Viewing Components of Certificates

This section provides a tutorial example on how to view different components in certificate with the OpenSSL command line tool.

Here is how to see the components of a certificate:

```
>openssl x509 -in herong.crt -noout -text
Certificate:
 Data:
  Version: 3 (0x2)
  Serial Number: 0 (0x0)
  Signature Algorithm: md5WithRSAEncryption
  Issuer: C=CN, ST=PN, L=LN, O=ON, OU=UN, CN=Herong Yang
  Validity
   Not Before: Jul 15 02:19:47 2002 GMT
   Not After : Aug 14 02:19:47 2002 GMT
  Subject: C=CN, ST=PN, L=LN, O=ON, OU=UN, CN=Herong Yang
  Subject Public Key Info:
   Public Key Algorithm: rsaEncryption
   RSA Public Key: (512 bit)
    Modulus (512 bit):
     00:a9:e6:19:c6:ee:88:01:86:d9:72:9e:93:92:db:
     57:01:7b:02:84:fc:1e:e3:57:5e:2a:7b:2b:25:9e:
     bd:ba:c5:95:2c:49:59:28:df:a6:67:86:26:8e:ff:
     36:cc:3a:84:5c:28:af:6f:11:c8:0c:b5:c2:c5:b9:
     04:d6:0e:5d:d1
    Exponent: 65537 (0x10001)
  X509v3 extensions:
   X509v3 Subject Key Identifier:
    55:C8:EF:49:2B:5D:E4:03:C5:98:1B:68:24:28:47:88:D4:0E:77:04
   X509v3 Authority Key Identifier:
    keyid:55:C8:EF:49:2B:5D:E4:03:C5:98:1B:68:24:28:47:88:D4:0E:77:04
    DirName:/C=CN/ST=PN/L=LN/O=ON/OU=UN/CN=Herong Yang
    serial:00
   X509v3 Basic Constraints:
    CA:TRUE
 Signature Algorithm: md5WithRSAEncryption
  3f:ba:0c:c1:ae:38:ca:f6:37:16:9c:35:5f:18:79:64:de:27:
  0d:46:ad:28:57:b4:62:df:6c:ff:f8:74:17:eb:b6:91:7e:06:
  6e:ec:a5:9d:23:e2:6e:5a:6e:c3:09:fa:cf:34:65:70:15:65:
  10:3a:6b:0f:b9:ef:6b:64:18:0e
```

This certificate tells us that:

- The subject is "C=CN, ST=PN, L=LN, O=ON, OU=UN, CN=Herong Yang"
- The subject's public key is included in it.
- The issuer is "C=CN, ST=PN, L=LN, O=ON, OU=UN, CN=Herong Yang". The issuer is identical to the subject, because this is a self-signed certificate.
- The certificate is valid for one month.

- The certificate is signed by the issuer with the signature at the end.

Last update: 2013.

OpenSSL Generating and Signing CSR

This chapter provides tutorial notes and example codes on CSR (Certificate Signing Request) handling with OpenSSL. Topics include introduction of certificate signing process; generating CSR; signing CSR.

Conclusions:

- A CA (Certificate Authority) is an organization trusted by both communication partners.
- To ask CA to sign your public key, you need to send it to the CA as CSR (Certificate Signing Request)
- "openssl req -new -key key_file -out csr_file" command allows you to generate a CSR for the specified key.
- "openssl req -in csr_file -noout -text" command allows you to view different components of a CSR.
- "openssl x509 -req -in csr_file -CA ca_certificate -CAkey ca_key -out signed_certificate" command allows you to sign someone else's CSR with your key and self-signed certificate.

Why Certificates Need to Be Signed by CA?

This section describes why public keys need to be signed a CA (Certificate Authority). You communication partner can trust the CA, not you.

In the previous chapter, we learned how to put your own public key in a certificate and sign it by your own private key to make it as a self-signed certificate.

Of course, you can send your self-signed certificate to your communication partner and start to use it to encrypt the communication data. However, this only works if your communication partner knows you and trusts your digital signature.

In the case where you communication partner can not trust you directly, what you can do is to send your public key to a certificate authority (CA) and ask them to sign it for you. To do this, you need to put your public key into a certificate signing request (CSR), and mail it to a CA. The CA will verify the request and put your public key in a certificate and sign it with CA's private key.

When your partner receives your public key signed by a CA, he can validate the signature with the CA's public key. If the validation is ok, he can then trust your public key.

Here is a simple diagram that illustrates the certificate signing and validation process:

```
               Your public key
You ---- Certificate signing request ---> CA
                                          | |
                                          | |Sign
                                          | |
        Your public key + CA signature    | v
You <----- Certificate signed by CA --------
|                                         |
|Send                                     |Send
|                                         |
v               CA's public key           v
Partner <-- Self-signed certificate ------
|
|Verify your certificate with CA's public key
|to trust your public key in the certificate
|
v
OK
```

Last update: 2013.

Generating Certificate Signing Request (CSR)

This section provides a tutorial example on how to generate a CSR (Certificate Signing Request) for your public key with OpenSSL.

In order to send your public key to a CA for signing, you need to put the public key in a file called certificate signing request (CSR). Here is how to use the "req" command to do this:

```
>openssl req -new -key herong_rsa_des.key -out herong.csr
   -config openssl.cnf

Enter pass phrase for herong_rsa_des.key:
You are about to be asked to enter information that will be incorp...
into your certificate request.
What you are about to enter is what is called a Distinguished Name...
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) []:CN
State or Province Name (full name) []:PN
Locality Name (eg, city) []:LN
Organization Name (eg, company) []:ON
Organizational Unit Name (eg, section) []:UN
Common Name (eg, YOUR name) []:Herong Yang
Email Address []:.
```

```
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:myreq
An optional company name []:

>type herong.csr
-----BEGIN CERTIFICATE REQUEST-----
MIIBETCBvAIBADBXMQswCQYDVQQGEwJDTjELMAkGA1UECBMCUE4xCzAJBgNVBAcT
AkNOMQswCQYDVQQKEwJPTjELMAkGA1UECxMCVU4xFDASBgNVBAMTC0hlcm9uZyBZ
YW5nMFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAKnmGcbuiAGG2XKek5LbVwF7AoT8
HuNXXip7KyWevbrFlSxJWSjfpmeGJo7/Nsw6hFwor28RyAy1wsW5BNYOXdECAwEA
AaAAMA0GCSqGSIb3DQEBBAUAA0EALE+d7H514HyQXu2CgwXYDvqZRngFLZFdGxQN
6AtEXXV+eC2c+URNBcmoF3oghJdPqZv7D1nZ7EBf20XSWzioQA==
-----END CERTIFICATE REQUEST-----
```

Note that the certificate is also saved in an encoded format called PEM.

Last update: 2013.

Viewing Components of Certificate Signing Request

This section provides a tutorial example on how to view different components of a CSR (Certificate Signing Request) for your public key with OpenSSL.

Here is how to see the components of a certificate signing request:

```
>openssl req -in herong.csr -noout -text -config openssl.cnf
Certificate Request:
    Data:
        Version: 0 (0x0)
        Subject: C=CN, ST=PN, L=LN, O=ON, OU=UN, CN=Herong Yang
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (512 bit)
                Modulus (512 bit):
                    00:a9:e6:19:c6:ee:88:01:86:d9:72:9e:93:92:db:
                    57:01:7b:02:84:fc:1e:e3:57:5e:2a:7b:2b:25:9e:
                    bd:ba:c5:95:2c:49:59:28:df:a6:67:86:26:8e:ff:
                    36:cc:3a:84:5c:28:af:6f:11:c8:0c:b5:c2:c5:b9:
                    04:d6:0e:5d:d1
                Exponent: 65537 (0x10001)
        Attributes:
            challengePassword        :myreq
    Signature Algorithm: md5WithRSAEncryption
        80:be:77:39:65:0f:24:db:70:c1:76:e3:b6:c7:99:a5:c7:af:
        ae:98:5a:73:98:f8:60:f1:65:08:a9:f7:df:6f:bd:77:aa:f7:
        bb:0b:f2:0d:71:6e:ad:ee:52:5a:2b:a7:2a:c0:fd:0e:4c:8f:
        c1:43:18:58:0b:10:03:e0:e5:a3
```

Some interesting notes here:

- The request is signed with my private key. I don't see any need for this.
- My "challengePassword" is displayed in plain text. What's the value of this password, if every one can see it?

Last update: 2013.

Signing a Certificate Signing Request

This section provides a tutorial example on how to sign someone else's certificate signing request with your self-signed certificate.

Even though I am not a well established CA, but I can still use OpenSSL to sign somebody else's certificate. The following process shows you how Herong Yang signs John Smith's certificate:

```
>echo generating a key pair for John
>openssl genrsa -out john_rsa.key
Loading 'screen' into random state - done
Generating RSA private key, 512 bit long modulus
..................++++++++++++
.++++++++++++
e is 65537 (0x10001)

>echo generating the certificate signing request for John
>openssl req -new -key john_rsa.key -out john.csr
   -config openssl.cnf

You are about to be asked to enter information that will be incorp...
into your certificate request.
What you are about to enter is what is called a Distinguished Name...
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) []:CN
State or Province Name (full name) []:PN
Locality Name (eg, city) []:LN
Organization Name (eg, company) []:ON
Organizational Unit Name (eg, section) []:UN
Common Name (eg, YOUR name) []:John Smith
Email Address []:.

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:.
An optional company name []:.

>echo signing John's request with Herong's certificate and key
>openssl x509 -req -in john.csr -CA herong.crt
   -CAkey herong_rsa_des.key -out john.crt

Loading 'screen' into random state - done
Signature ok
subject=/C=CN/ST=PN/L=LN/O=ON/OU=UN/CN=John Smith
Getting CA Private Key
Enter pass phrase for herong_rsa_des.key:

>echo looking at John's certificate
>openssl x509 -in john.crt -noout -text
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number: 5 (0x5)
        Signature Algorithm: md5WithRSAEncryption
        Issuer: C=CN, ST=PN, L=CN, O=ON, OU=UN, CN=Herong Yang
        Validity
            Not Before: Jul 17 03:10:39 2002 GMT
            Not After : Aug 16 03:10:39 2002 GMT
        Subject: C=CN, ST=PN, L=LN, O=ON, OU=UN, CN=John Smith
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (512 bit)
                Modulus (512 bit):
                    00:d4:a4:be:ce:2d:be:88:56:ef:d3:de:13:15:33:
                    59:84:ea:08:fe:bc:c8:70:93:30:c0:c4:c5:de:e3:
                    65:e8:98:e1:15:12:27:d4:00:69:6e:22:fa:c3:72:
                    4a:75:a6:d8:66:dc:ec:12:f6:92:94:09:3c:3a:61:
                    69:47:99:b3:91
                Exponent: 65537 (0x10001)
    Signature Algorithm: md5WithRSAEncryption
        57:a5:9f:93:8e:f8:69:cd:9b:70:ff:f5:fc:78:e3:f6:da:70:
        b9:5d:d6:a8:ac:ae:76:41:13:04:99:28:97:55:9b:5e:94:c7:
        c5:59:26:77:33:cb:67:aa:1c:d5:0e:b7:de:33:73:b1:f6:3a:
        0b:c2:d9:6a:5b:f1:d1:ab:60:9b
```

This is nice. Now I can sign anyone's certificate, and become a CA! All I need is my RSA key pair, herong_rsa_des.key, my self-signed certificate, herong.crt, and the "x509" command.

Last update: 2013.

OpenSSL Validating Certificate Path

This chapter provides tutorial notes and example codes on certificate path validation with OpenSSL. Topics include introduction of certificate path; certificate path validation rules; generating and validating a certificate path.

Conclusions:

- A certificate path is a list of certificates in which the issuer of next certificate is the subject of the previous certificate.
- "openssl verify -CAfile first_certificate -untrusted middle_certificates last_certificate" command allows you to validate a certificate path.

What Is a Certification Path?

This section describes what is a certificate path or certificate chain - An ordered list of certificates where the subject entity of one certificate is identical to the issuing entity of the next certificate.

Certification Path: Also called Certificate Chain. An ordered list of certificates where the subject entity of one certificate is identical to the issuing entity of the next certificate.

A certification path can also be defined as an ordered list of certificates where the issuing entity of one certificate can be identified as the subject entity of the previous certificate. But the first certificate has to be a special one, because there is no previous certificate to identify the issuing entity. The first certificate must be a self-signed certificate, where the issuing entity is the same as the subject entity.

For example, the following diagram shows you a certification path:

```
Certificate 1
   Issuer: Herong Yang
   Subject: Herong Yang

Certificate 2
   Issuer: Herong Yang
   Subject: John Smith
```

```
Certificate 3
   Issuer: John Smith
   Subject: Bill White

Certificate 4
   Issuer: Bill White
   Subject: Tom Bush
```

Last update: 2013.

Certification Path Validation Rules

This section describes verification rules of a certificate path - The issuer's digital signature must match the subject's public key of the previous certificate in the path.

A certification path needs to be validated. Here are the validation rules:

- The first certificate must be self-signed. Its issuer must be recognized as a certificate authority (CA).
- The issuer of any certificate, except the first one, must be "identical" to the subject of the previous certificate.
- "identical" means that issuer's digital signature can verified by the subject's public key in the previous certificate.

OpenSSL offers a nice tool, the "verify" command, to validate a certification path. Here is the syntax of the "verify" command:

```
verify -CAfile first.crt -untrusted all_middle.crt last.crt
```

- "first.crt" is the first certificate of the path. It should be self-signed certificate.
- "last.crt" is the last certificate of the path.
- "all_middle.crt" is a collection of all middle certificates. If certificates are store in PEM format, you can join them into a collection in any text editor.

Last update: 2013.

Creating a Certificate Path with OpenSSL

This section provides a tutorial example on how to create multiple certificates to form a certificate path for testing purpose.

Here is a testing scenario I followed to generate some certificates with different issuers and subjects. See previous notes if you have trouble generating keys and signing certificates.

1. Generating a self-signed certificate for Herong, herong.crt:

```
>echo Generating keys for Herong
>openssl genrsa -des3 -out herong_rsa.key
...

>echo Generating a self-signed certificate for Herong
>openssl req -new -key herong_rsa.key -x509 -out herong.crt
   -config openssl.cnf
...
```

2. Generating a certificate for John and signed by Herong, john.crt:

```
>echo Generating keys for John
>openssl genrsa -des3 -out john_rsa.key
...

>echo Generating a certificate signing request for John
>openssl req -new -key john_rsa.key -out john.csr -config openssl.cnf
...

>echo Signing a John's request by Herong's key
>openssl x509 -req -in john.csr -CA herong.crt -CAkey herong_rsa.key
   -out john.crt -set_serial 3
...
```

3. Generating a certificate for Bill and signed by John, bill.crt:

```
>echo Generating keys for Bill
>openssl genrsa -des3 -out bill_rsa.key
...

>echo Generating a certificate signing request for Bill
>openssl req -new -key bill_rsa.key -out bill.csr -config openssl.cnf
...

>echo Signing a Bill's request by John's key
>openssl x509 -req -in bill.csr -CA john.crt -CAkey john_rsa.key
   -out bill.crt -set_serial 7
...
```

4. Generating a certificate for Tom and signed by Bill, tom.crt:

```
>echo Generating keys for Tom
>openssl genrsa -des3 -out tom_rsa.key
...

>echo Generating a certificate signing request for Bill
>openssl req -new -key tom_rsa.key -out tom.csr -config openssl.cnf
...

>echo Signing a Tom's request by Bill's key
>openssl x509 -req -in tom.csr -CA bill.crt -CAkey bill_rsa.key
   -out tom.crt -set_serial 11
...
```

Ok. 4 certificates are enough to do some interesting tests with the "openssl verify" command. See the next section for testing result.

Last update: 2013.

Validating a Certificate Path with OpenSSL

This section provides a tutorial example on how to perform validation of a certificate path with the 'openssl verify' command.

With 4 certificates created in the previous section, we are ready to test the "openssl verify" command:

1. Verify the shortest certification path, one certificate only:

```
>openssl verify herong.crt
herong.crt: /C=CN/ST=PN/L=LN/O=ON/OU=UN/CN=Herong Yang
error 18 at 0 depth lookup:self signed certificate
OK

>openssl verify -CAfile herong.crt herong.crt
herong.crt: OK
OK

>openssl verify john.crt
john.crt: /C=CN/ST=PN/L=LN/O=ON/OU=UN/CN=John Smith
error 20 at 0 depth lookup:unable to get local issuer certificate

>openssl verify -CAfile john.crt john.crt
john.crt: /C=CN/ST=PN/L=LN/O=ON/OU=UN/CN=John Smith
error 20 at 0 depth lookup:unable to get local issuer certificate
```

Note that:

- You will get an OK with an error, when validating a self-signed certificate without specifying it as the CA certificate.
- You will get a perfect OK, when validating a self-signed certificate with the CA certificate specified as itself.
- You will get an error, when validating a non self-signed certificate with or without specifying it as the CA certificate.

2. Verify certification paths of two certificates:

```
>openssl verify -CAfile herong.crt john.crt
john.crt: OK

>openssl verify -CAfile herong.crt bill.crt
bill.crt: /C=CN/ST=PN/L=LN/O=ON/OU=UN/CN=Bill White
error 20 at 0 depth lookup:unable to get local issuer certificate

>openssl verify -CAfile john.crt bill.crt
bill.crt: /C=CN/ST=PN/L=LN/O=ON/OU=UN/CN=John Smith
error 2 at 1 depth lookup:unable to get issuer certificate
```

Note that:

- Test 1: Perfect.
- Test 2: Path broken at 0 depth. Could not find the issuer on bill.crt.
- Test 3: Path broken at 1 depth. Could not find the issuer on john.crt.

3. Verify certification paths of many certificates:

```
>openssl verify -CAfile herong.crt -untrusted john.crt bill.crt
bill.crt: OK

>openssl verify -CAfile herong.crt -untrusted bill.crt tom.crt
tom.crt: /C=CN/ST=PN/L=LN/O=ON/OU=UN/CN=Bill Gate
error 20 at 1 depth lookup:unable to get local issuer certificate

>copy john.crt+bill.crt all.crt
>openssl verify -CAfile herong.crt -untrusted all.crt tom.crt
tom.crt: OK
```

Note that:

- Test 1: Perfect.
- Test 2: Path broken at 1 depth. Could not find the issuer on bill.crt.
- Test 3: Perfect. Look at how I join two certificates file together with the DOS command "copy".

Last update: 2013.

"keytool" and "keystore" from JDK

This chapter provides tutorial notes and example codes on 'keytool' and 'keystore'. Topics include introduction of 'keytool' and 'keystore'; using 'keytool' to generate private keys, generate self-signed certificates, export and import certificates.

Conclusions:

- A "keystore" is a database file introduced in JDK (Java Development Kit) to store your own private keys and public key certificates you received from other people.
- "keytool" is command line tool provided in JDK to manage "keystore" files.
- A key entry in "keystore" contains a private key and a self-signed certificate of the public key.
- A certificate entry in "keystore" contains a signed public key.
- "keytool -genkey" command can be used to generate a pair of private key and public key.
- "keytool -export" command can be used to export a certificate from "keystore" into a certificate file.
- "keytool -import" command can be used to import a certificate from a certificate file into "keystore".
- "keytool -certreq" command can be used to generate a CSR (Certificate Signing Request) for the public key stored in a key entry.
- "keytool" does not provide any way to export a private key to a key file.
- "keytool" does not provide any way to sign a CSR (Certificate Signing Request).

Certificates and Certificate Chains

This section describes what is a certificate and what is a certificate path. A certificate is a digitally signed statement from the issuer saying that the public key of the subject has some specific value.

Certificate: A digitally signed statement from the issuer saying that the public key of the subject has some specific value.

The above definition is copied from the JDK 1.3.1 documentation. It has a couple of important terms:

- "signed statement" - The certificate must be signed by the issuer with a digital signature.
- "issuer" - The person or organization who is issuing this certificate.
- "public key" - The public key of a key pair selected by the subject.
- "subject" - The person or organization who owns the public key.

X.509 Certificate - A certificate written in X.509 standard format. X.509 standard was introduction in 1988. It requires a certificate to have the following information:

- Version - X.509 standard version number.
- Serial Number - A sequence number given to each certificate.
- Signature Algorithm Identifier - Name of the algorithm used to sign this certificate by the issuer
- Issuer Name - Name of the issuer.
- Validity Period - Period during which this certificate is valid.
- Subject Name - Name of the owner of the public key.
- Subject Public Key Information - The public key and its related information.

How can you get a certificate for your own public key?

- Requesting it from a Certificate Authority (CA), like VeriSign, Thawte or Entrust.
- Doing it yourself - using tools like JDK "keytool" to generate a self-signed certificate.

Certificate Chain: A series of certificates that one certificate signs the public key of the issuer of the next certificate. Usually the top certificate (the first certificate) is self-signed, where issuer signed its own public key.

Last update: 2013.

What Is "keystore"?

This section describes what is a 'keystore' file - A database file introduced in JDK (Java Development Kit) to store your own private keys and public key certificates you received from other people.

"keystore" - A database file introduced in JDK (Java Development Kit) to store your own private keys and public key certificates you received from other people. JDK offers "java.security.KeyStore" class for you to manage "keystore" files in your Java programs. JDK also offers "keytool" program for you to manage "keystore" files with command lines.

"keystore" files support following features:

- A "keystore" file may support different types implemented by different security package providers. The default type is called JKS implemented by Sun Microsystems.
- A "keystore" file may have two types of entries: key entries for private keys and certificate entries for public key certificates.
- A key entry contains a private key and its self-signed certificate.
- A certificate entry contains a certificate.
- Every entry has a unique alias name.
- A "keystore" file is protected by a password for opening the file.
- Each key entry has its own password to give its private key an extra level of protection.

Last update: 2013.

"keytool" - Command Line Tool

This section describes the 'keytool' command tool provided in JDK. 'keytool' can be used to generate and manage private keys and certificates stored in 'keystore' files.

"keytool" is command line tool introduced in JDK 1.2 to manage keys and certificates stored in "keystore" database files. "keytool" replaces the same functions offered by "javakey" in JDK 1.1.

"keytool" uses this command line syntax "keytool sub-command options". "keytool" provided in JDK 1.3.1 supports these sub-commands:

- "-genkey": Generates a new key pair and stores it as a key entry in the keystore.
- "-list": Lists all entries in the keystore.
- "-export": Exports the certificate of the specified key entry or certificate entry out of the keystore to a certificate file.
- "-printcert": Prints summary information of a certificate from a certificate file.

- "-import": Imports the certificate from a certificate file as a certificate entry into the keystore.
- "-certreq": Generates a CSR (Certificate Signing Request) for the public key in a key entry.
- "-keyclone": Creates a new key entry by copying an existing key entry.
- "-selfcert": Replaces the certificate in a key entry with a new self-signed certificate.
- "-delete": Deletes the entry of the specified alias name.

Last update: 2013.

Generating Private Keys

This section provides a tutorial example on how to use the 'keytool -genkey' command to generate a pair of private key and public key. This command also generates a self-signed certificate from the key pair.

The "keytool -genkey" command can be used to generate a pair of private key and public key. It can also be used to generate a self-signed certificate from the key pair.

In the first example, I want to try the "-genkey" command option using JDK 1.3.1:

```
keytool -genkey -alias my_home -keystore herong.jks
Enter keystore password:  HerongJKS
What is your first and last name?
  [Unknown]:  Herong Yang
What is the name of your organizational unit?
  [Unknown]:  My Unit
What is the name of your organization?
  [Unknown]:  My Home
What is the name of your City or Locality?
  [Unknown]:  My City
What is the name of your State or Province?
  [Unknown]:  My State
What is the two-letter country code for this unit?
  [Unknown]:  US
Is <CN=Herong Yang, OU=My Unit, O=My Home, L=My City, ST=My State,
   C=US> correct?
  [no]:  yes
Enter key password for <my_home>
        (RETURN if same as keystore password):  My1stKey
```

Based on the documentation, the above example command did the following for me:

- Create a "keystore" file, herong.jks, in JKS format, with a password of "HerongJKS".
- Generate a pair of private key and public key for me using the default implementation of the default security package.
- Generate a certificate chain with a single self-signed certificate of my public key.
- Insert a key entry into the keystore with my private key and the certificate chain.

- The key entry is protected by a password of "My1stKey".

The following command shows that we do have a key entry in the keystore file:

```
keytool -list -keystore herong.jks -storepass HerongJKS

Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry:

my_home, Sat Jun 1 07:15:16 EDT 2002, keyEntry,
Certificate fingerprint
   (MD5): BE:D2:AF:4E:A7:44:13:08:16:4C:68:3B:D1:99:79:55
```

Last update: 2013.

Exporting and Import Certificates

This section provides a tutorial example on how to use the 'keytool -export' and 'keytool -import' commands to export and import the self-signed certificate from a key entry in a 'keystore' file.

In the second example, I want to export the certificate stored in the key entry to a certificate file, then import it back into the keystore as certificate entry:

```
keytool -export -alias my_home -file my_home.crt -keystore herong.jks
   -storepass HerongJKS
Certificate stored in file <my_home.crt>

keytool -printcert -file my_home.crt
Owner: CN=Herong Yang, OU=My Unit, O=My Home, L=My City, ST=My Sta...
Issuer: CN=Herong Yang, OU=My Unit, O=My Home, L=My City, ST=My St...
Serial number: 407928a4
Valid from: Sat Jun 1 07:14:44 EDT 2002 until: Sat Aug 31 07:14:44...
Certificate fingerprints:
   MD5:  BE:D2:AF:4E:A7:44:13:08:16:4C:68:3B:D1:99:79:55
   SHA1: AE:67:0C:C5:21:5C:F6:6F:45:33:9E:FB:8E:50:EA:32:32:D1:92:BB

keytool -import -alias my_home_crt -file my_home.crt
   -keystore herong.jks -storepass HerongJKS
Certificate already exists in keystore under alias <my_home>
Do you still want to add it? [no]:  yes
Certificate was added to keystore

keytool -list -keystore herong.jks -storepass HerongJKS

Keystore type: jks
Keystore provider: SUN

Your keystore contains 2 entries:

my_home_crt, Sat Jun 1 12:25:46 EDT 2004, trustedCertEntry,
Certificate fingerprint (MD5): BE:D2:AF:4E:A7:44:13:08:16:4C:68:3B...
my_home, Sat Jun 1 07:15:16 EDT 2002, keyEntry,
Certificate fingerprint (MD5): BE:D2:AF:4E:A7:44:13:08:16:4C:68:3B...
```

Looking good so far:

- The "-export" command option exports the self-signed certificate of my public key into a file, my_home.crt.
- The "-printcert" command option prints out summary information of a certificate stored in a file in X.509 format. As you can see from the print out, I am the issuer and the owner of this certificate.
- The "-import" command option imports the certificate from the certificate file back into the keystore under different alias, my_home_crt.

Certificates can also be exported in a printable format: based on RFC 1421 specification, using the BASE64 encoding algorithm.

```
keytool -export -alias my_home_crt -file my_home.rfc -rfc
   -keystore herong.jks -storepass HerongJKS
Certificate stored in file <my_home.rfc>

type my_home.rfc
-----BEGIN CERTIFICATE-----
MIIDDTCCAssCBEB5KKQwCwYHKoZIzjgEAwUAMGwxCzAJBgNVBAYTAlVTMREwDwYDVQ...
dGF0ZTEQMA4GA1UEBxMHTXkgQ2l0eTEQMA4GA1UEChMHTXkgSG9tZTEQMA4GA1UECx...
dDEUMBIGA1UEAxMLSGVyb25nIFlhbmcwHhcNMDQwNDExMTExNDQ0WhcNMDQwNzEwMT...
MQswCQYDVQQGEwJVUzERMA8GA1UECBMITXkgU3RhdGUxEDAOBgNVBAcTB015IENpdH...
BAoTB015IEhvbWUxEDAOBgNVBAsTB015IFVuaXQxFDASBgNVBAMTC0hlcm9uZyBZYW...
ASwGByqGSM44BAEwggEfAoGBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2US...
WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1...
+4P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAhUAl2BQjxUjC8yykrmCou...
gYEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgL...
FhO3zwkyjMim4TwWeotUfI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkI...
BpKLZl6Ae1UlZAFMO/7PSSoDgYUAAoGBAJdQsMlIf1nh4T/HZvVeltsrTGED118CkG...
ygy53OLwrSK+6ptJpXP8tPMn9YFVJ3eigJrMTaZvGyd40WRiYM6Woyj3T4H73LEKLD...
QeNYOAm8cp3l9ZQkNnmIA1P6CRR43EeAmdTUlK8y6RWTsrOiJMdDMAsGByqGSM44BA...
AhQ4zAUOPWe1wdiwye9XDsVPcKS1xwIUWTdok6RIeeCMRIytKwcTOo7/qpM=
-----END CERTIFICATE-----
```

Last update: 2013.

Generating CSR (Certificate Signing Request)

This section provides a tutorial example on how to use the 'keytool -certreq' command to generate a CSR (Certificate Signing Request) for the public key stored in a key entry.

In the third example, I want to generate a CSR (Certificate Signing Request) for the public key stored in key entry. The CSR can be then sent to a CA (Certificate Authority) and singed by the CA:

```
keytool.exe -certreq -alias my_home  -keypass My1stKey
   -keystore herong.jks -storepass HerongJKS -file my_home.csr

type my_home.csr
```

```
-----BEGIN NEW CERTIFICATE REQUEST-----
MIICcTCCAi4CAQAwbDELMAkGA1UEBhMCVVMxETAPBgNVBAgTCE15IFN0YXRlMRAwDg...
eSBDaXR5MRAwDgYDVQQKEwdNeSBIb21lMRAwDgYDVQQLEwdNeSBVbml0MRQwEgYDVQ...
bmcgWWFuZzCCAbcwggEsBgcqhkjOOAQBMIIBHwKBgQD9f1OBHXUSKVLfSpwu7OTn9h...
Hj+AtlEmaUVdQCJR+1k9jVj6v8X1ujD2y5tVbNeBO4AdNG/yZmC3a5lQpaSfn+gEex...
t8Yb+DtX58aophUPBPuD9tPFHsMCNVQTWhaRMvZ1864rYdcq7/IiAxmd0UgBxwIVAJ...
spK5gqLrhAvwWBz1AoGBAPfhoIXWmz3ey7yrXDa4V7l5lK+7+jrqgvlXTAs9B4JnUV...
cQcQgYC0SRZxI+hMKBYTt88JMozIpuE8FnqLVHyNKOCjrh4rs6Z1kW6jfwv6ITVi8f...
8b6oUZCJqIPf4VrlnwaSi2ZegHtVJWQBTDv+z0kqA4GEAAKBgEqtTokXJtvReDbz4k...
c8g/SfJVaY+ZhTOa+H7aRKLzECRivEJe0uLEh8DNMPnZvbR3xVGdLNGOOCPXZt77QX...
AFWmwqWYELhNVgN5d4gSREK+hUmWTkkb6zHyhVUkwSIy0WDJGRAfgI4RaKnWxQzgZX...
KoZIzjgEAwUAAzAAMC0CFQCE8UhMAke7738nWDPG1GUIEncfiwIUZWRAW/PN+GGLH8...
b2g=
-----END NEW CERTIFICATE REQUEST-----
```

Last update: 2013.

Cloning Certificates with New Identities

This section provides a tutorial example on how to use the 'keytool -keyclone' and 'keytool -selfcert' commands to clone an existing key entry and self-sign it with a new identity.

In the fourth example, I want to create a new key entry with the same key pair of an existing key entry, but change the identity information:

```
keytool -keyclone -alias my_home -dest my_copy -keypass My1stKey
   -new My2ndKey -keystore herong.jks -storepass HerongJKS

keytool -selfcert -alias my_copy -keypass My2ndKey
   -dname "cn=Herong Yang, ou=My Unit 2, o=My Organization 2, c=US"
   -keystore herong.jks -storepass HerongJKS

keytool -export -alias my_copy -file my_copy.crt -keystore herong.jks
   -storepass HerongJKS
Certificate stored in file <my_copy.crt>

keytool -printcert -file my_copy.crt
Owner: CN=Herong Yang, OU=My Unit 2, O=My Organization 2, C=US
Issuer: CN=Herong Yang, OU=My Unit 2, O=My Organization 2, C=US
Serial number: 40798b4f
Valid from: Sat Jun 1 14:15:43 EDT 2002 until: Sat Aug 31 14:15:43...
Certificate fingerprints:
         MD5:  4A:E4:D9:BC:E9:8C:50:27:6C:00:59:76:D1:14:05:79
         SHA1: FA:F5:30:78:22:3B:52:28:0D:41:24:0B:CA:CC:6F:D4:0E:...
```

Last update: 2013.

"OpenSSL" Signing CSR Generated by "keytool"

This chapter provides tutorial notes and example codes on using 'OpenSSL' to act as CA (Certificate Authority). Topics include setting up 'OpenSSL' as CA; using 'keytool' to generate CA private key; using 'keytool' to generate CSR (Certificate Signing Request); using 'OpenSSL' to sign CSR, using 'keytool' to import signed certificate into 'keystore' files.

After I published my notes on "OpenSSL" and "keytool", many viewers have asked questions about the compatibility between those two tools. In this chapter, I will test how "OpenSSL" can be used to sign CSR generated by "keytool".

Conclusions:

- "OpenSSL" is a nice tool to help you became a CA (Certificaet Authority) to sign other people's CSR (Certificate Signing Request).
- "keytool" can not be used to sign certificates. But it can generate key pairs and CSR.
- "OpenSSL" can generate 2048-bit keys. "keytool" can only generate upto 1024-bit keys.
- CSR generated by "keytool" is compatible for "OpenSSL" to sign it into a certificate.
- Certificates generated by "OpenSSL" is compatible for "keytool" to import into 'keystore' files.

"OpenSSL" Acting as a CA (Certificate Authority)

This section provides a tutorial example on how to prepare OpenSSL to be used as CA (Certificate Authority) to sign other people's CSR (Certificate Signing Request).

If I want to act as a CA (Certificate Authority), I must have a tool to sign other people's CSR (Certificate Signing Request). As we learned from previous chapters, "keytool" can not sign CSR. So I must use "OpenSSL" to act as a CA.

Here is a list of things I need to do with "OpenSSL" as a CA:

- Install "OpenSSL" installed properly and create configuration file, openssl.cnf.
- Created a CA private key file with 2048-bit keys, and protect it with a password.
- Self-sign my CA public key certificate, which can be given out to anyone.

Let's check the "OpenSSL" installation first. If "OpenSSL" was installed at \local\GnuWin32\, the following command should report back the version number:

```
>\local\gnuwin32\bin\openssl version

OpenSSL 0.9.7c 30 Sep 2003

>set path=<old_path>;\local\gnuwin32\bin\
```

To save typing time, I added \local\gnuwin32\bin\ to the PATH environment variable as shown above.

Then look at the configuration file, openssl.cnf, which is needed only when I self-signing my CA public certificate. I can use the openssl.cnf as is without any changes. But if I want to put in my CA distinguished name information to save time when self-signing my CA public certificate, I can add these settings to openssl.cnf:

```
countryName_default              = CA
stateOrProvinceName_default      = HY State
localityName_default             = HY City
0.organizationName_default       = HY Company
organizationalUnitName_default   = HY Unit
commonName_default               = Herong Yang
emailAddress_default             = herongyang.com
```

So I am ready to generated my CA private key as described in the next section.

Last update: 2013.

"OpenSSL" Generating CA's Private Key

This section provides a tutorial example on how to use OpenSSL to generate a RSA private key of 2048 bit long with OpenSSL. This key will be used as the CA's private key and must stored securely in a file with password protection.

As a CA, I must have a good private key (2048 bit long) and I must store it securely in a file. This can be done with a single OpenSSL command "openssl genrsa" as shown in the following command window session:

```
>openssl genrsa -out herong.key -des 2048

Loading 'screen' into random state - done
Generating RSA private key, 2048 bit long modulus
..........................................................+++
.......................+++
e is 65537 (0x10001)
Enter pass phrase for herong.key: keypass
Verifying - Enter pass phrase for herong.key: keypass

>type herong.key

-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-CBC,65D5EB070215060E

l5d+v8KbkPwCgFlrsMD+5FpVsldoCZPeCjMrrWzyym5wIKHKqVltfAjO...
Z2JBXMGxAV4kGky0Jca6/Kxb/tWAmJci0YaDXO0EjwTxeulj9MVL5t3m...
...
bHbZhRs6lUxB4PNosjkhZrgPzYRD7A3EmVqo6ZLXeYO2I2lVIe7k+Lec...
-----END RSA PRIVATE KEY-----
```

Notes about what I did here:

- "genrsa" command is used to generate a pair of private key and public key using RSA algorithm.
- "-out herong.key" tells openssl to store the private key in a file called herong.key.
- "-des" option is used to encrypt my private key file "herong.key" with DES algorithm.
- "2048" used to force openssl to generate keys a length of 2048 bits. The default length is 512. Longer keys give better protection.
- "-passout keypass" is not used on the command line, because it does not work (See the test below). So I entered it when "OpenSSL" prompted for it.
- "type herong.key" is Windows command to shows the content of "herong.key".

Error message received when "-passout keypass" is used:

```
>openssl genrsa -out test.key -des -passout keypass 2048
Invalid password argument "keypass"
Error getting password
```

Recently, someone emailed me the correct syntax of the "-passout" option: "-passout pass:<password>". Here is an example of how to use the "-passout" option correctly:

```
>openssl genrsa -out test.key -des -passout pass:keypass 2048
Loading 'screen' into random state - done
Generating RSA private key, 2048 bit long modulus
.....................................................................
```

```
...........+++
............................+++
e is 65537 (0x10001)
```

Want to see some details about my private key? Run the "openssl rsa" command as shown below:

```
>openssl rsa -in herong.key -text
Enter pass phrase for herong.key: keypass
Private-Key: (2048 bit)
modulus:
    00:ba:a3:a2:d1:ab:9b:9f:26:e6:b5:79:b4:52:11:
    ...
publicExponent: 65537 (0x10001)
privateExponent:
    00:9d:62:da:2d:57:3a:2f:36:5d:bc:d0:f9:97:6f:
    ...
exponent1:
    27:6c:ec:a2:b8:78:5f:55:67:b9:47:eb:3e:25:5d:
    ...
exponent2:
    00:a9:17:88:e5:d7:63:c3:7b:f8:6f:57:78:de:53:
    ...
coefficient:
    30:f5:86:b6:81:ad:1d:35:2c:1a:c1:ba:b9:d9:ab:
    ...
writing RSA key
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEAuqOi0aubnybmtXm0UhE47JSDcCrc/NGtfbtJdk+9...
    ...
-----END RSA PRIVATE KEY-----
```

Now I am ready to generate a self-signed public key certificate based on my private key file as described in the section below.

Last update: 2013.

"OpenSSL" Self-Signing CA's Public Key

This section provides a tutorial example on how to use OpenSSL to generate a self-signed certificate for your public key as CA certificate.

As you know, my key file actually contains a pair of keys: my private key and my public key. My private key will be used only by myself to sign any documents. My public key will be used by whoever receives the document signed by me to verify the signature.

To give out my public key, I need to be put it into a certificate with my name, and signed by my own private key. This process is call generating a self-signed public key certificate. OpenSSL can do this in a single command "openssl req -new -x509" as shown in the following command window session:

```
>openssl req -new -key herong.key -x509 -days 3650
   -out herong.crt -config openssl.cnf
```

```
Enter pass phrase for herong.key: keypass
You are about to be asked to enter information that will be
incorporated into your certificate request.
What you are about to enter is what is called a Distinguished
Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [CA]:
State or Province Name (full name) [HY State]:
Locality Name (eg, city) [HY City]:
Organization Name (eg, company) [HY Company]:
Organizational Unit Name (eg, section) [HY Unit]:
Common Name (eg, YOUR name) [Herong Yang]:
Email Address [herongyang.com]:

>type herong.crt

-----BEGIN CERTIFICATE-----
MIIElzCCA3+gAwIBAgIBADANBgkqhkiG9w0BAQQFADCBkzELMAkGA1UE...
...
KqbxyZS65093ifrC0kmfNCY3cq+vBqdMvpV9
-----END CERTIFICATE-----
```

Here is what happened:

- "req" command is used to generate a certificate signing request or self-signed certificate.
- "-new" option is used to prompt for certificate "subject" information.
- "-key herong.key" option is used to specify my key file containing my private key and public key. Password will be prompted.
- "-x509" option is used to tell "req" to generate self-signed certificate.
- "-days 3650" option is used to make the self-signed certificate valid for 3650 days, about 10 years.
- "-out herong.crt" option is used to tell "req" to store the self-signed certificate in a file called "herong.crt".
- "-config openssl.cnf" option is used to specify the configuration file.
- When you are prompted for distinguished name information, just press Enter key to take the default values.
- "type herong.crt" is Windows command to show the content of "herong.crt".

Want to see some details about my self-signed certificate? Run the "openssl rsa" as shown below:

```
>openssl x509 -in herong.crt -noout -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 0 (0x0)
        Signature Algorithm: md5WithRSAEncryption
        Issuer: C=CA, ST=HY State, L=HY City, O=HY Company,
OU=HY Unit, CN=Herong Yang/emailAddress=herongyang.com
        Validity
            Not Before: Apr 1 14:07:29 2007 GMT
            Not After : Mar 29 14:07:29 2017 GMT
        Subject: C=CA, ST=HY State, L=HY City, O=HY Company,
OU=HY Unit, CN=Herong Yang/emailAddress=herongyang.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (2048 bit)
                Modulus (2048 bit):
                    00:ba:a3:a2:d1:ab:9b:9f:26:e6:b5:79:...
                    ...
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                36:7C:F4:4A:A4:9B:C9:B5:C5:F7:09:3F:31:1...
            X509v3 Authority Key Identifier:
                keyid:36:7C:F4:4A:A4:9B:C9:B5:C5:F7:09:3...
                DirName:/C=CA/ST=HY State/L=HY City
/O=HY Company/OU=HY Unit/CN=Herong Yang
/emailAddress=herongyang.com
                serial:00

            X509v3 Basic Constraints:
                CA:TRUE
    Signature Algorithm: md5WithRSAEncryption
        aa:40:06:c0:cb:28:74:b1:1e:c2:a2:89:4f:8d:1e:9c:...
        ...
```

Notice that a default serial number, 0, is used when self-signing your own CA public key certificate.

As a CA, now I have my private key and my public key certificate. I am ready to sign anything. The next section describes how someone else can use "keytool" to generated a public key and ask me to sign it.

Note that if you use the "openssl req -new -x509" command without the "-config openssl.cnf" option, "OpenSSL" will give you an error like this:

```
>openssl req -new -key herong.key -x509 -out herong.crt
-days 3650
Unable to load config info
Enter pass phrase for herong.key:
unable to find 'distinguished_name' in config
problems making Certificate Request
2252:error:0E06D06A:configuration file routines:
NCONF_get_string:no conf or environment
variable:conf_lib.c:325:
```

Last update: 2013.

"keytool" Generating Maria's Private Key

This section provides a tutorial example on how to use 'keytool' to generate a pair of private key and public key.

In this section, let's assume that Maria is using the "keytool". She wants to have her own private key to sign documents. But she needs her public key certificate to be signed by me, Herong, because other people trust me instead of Maria.

So Maria starts to generate her own private key and store it in a "keystore" file. This can be done by a single "keytool -genkeypair" command as shown in the following command session:

```
>java -version

java version "1.6.0_01"
Java(TM) SE Runtime Environment (build 1.6.0_01-b06)
Java HotSpot(TM) Client VM (build 1.6.0_01-b06, mixed mode,
  sharing)

>keytool -genkeypair -alias maria_key -keysize 1024
   -keystore maria.jks -storepass jkspass -keypass keypass
What is your first and last name?
  [Unknown]:  Maria Teresa
What is the name of your organizational unit?
  [Unknown]:  Maria Unit
What is the name of your organization?
  [Unknown]:  Maria Company
What is the name of your City or Locality?
  [Unknown]:  Maria City
What is the name of your State or Province?
  [Unknown]:  Maria State
What is the two-letter country code for this unit?
  [Unknown]:  AT
Is CN=Maria Teresa, OU=Maria Unit, O=Maria Company, L=Maria City,
ST=Maria State, C=AT correct?
  [no]:  yes
```

Here is what Maria did:

- "java -version" command is used to check the Java version.
- "keytool -genkeypair" command is used to generated a key pair: Maria's private key and Maria's public key.
- "-keystore maria.jks" option specifies the keystore file name to hold the key pair.
- "-alias maria_key" option specifies the entry name of the key pair in the keystore file, because keystore file can hold multiple key and certificate entries.
- "-keysize 1024" option specifies the key size to be 1024 bits. To bad, "keytool" can not generated 2048-bit keys.
- "-storepass jkspass" option specifies a password to protect the keystore file.
- "-keypass keypass" option specifies a password to protect "maria_key" entry in the keystore file.

- "keytool" also requires Maria's identification information (distinguished name) to be entered at the time of generating the key pair, which is not really needed for generating a key pair. Distinguished name only needed when generating certificates.

Want to confirm that Maria's key pair is in the keystore file? Try this command:

```
>keytool -list -keystore maria.jks -storepass jkspass

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry

maria_key, Apr 1, 2007, PrivateKeyEntry,
Certificate fingerprint (MD5): 54:5A:E8:77:30:82:B4:EB:C...
```

Now Maria is ready to generate a CSR (Certificate Sign Request) to ask me as a CA to sign it as described in the next section.

Last update: 2013.

"keytool" Generating Maria's CSR

This section provides a tutorial example on how to use 'keytool' to generate a CSR (Certificate Signing Request) containing a public key to ask a CA to sign it.

Maria can now use "keytool" to generate a CSR (Certificate Signing Request) containing her public and ask me as a CA to sign it for her. To do this, she needs to run one "keytool -certreq" command as shown below:

```
>keytool -certreq -alias maria_key -keypass keypass
   -keystore maria.jks -storepass jkspass -file maria.csr

>type maria.csr
-----BEGIN NEW CERTIFICATE REQUEST-----
MIICgTCCAj4CAQAwfDELMAkGA1UEBhMCQVQxFDASBgNVBAgTC01hcmlh...
...
ah8gcsGwrIvlEJCJBra1HzsK
-----END NEW CERTIFICATE REQUEST-----
```

Notes on what Maria did:

- "keytool -certreq" command is used to generated a CSR (Certificate Sign Request) based on the given key pair.
- "-alias maria_key" option specifies the entry in the keystore file where to get the key pair.
- "-keystore maria.jks" option specifies the keystore file.
- "-file maria.csr" option specifies the file name where the CSR will be stored.

- "type maria.csr" command shows the content of "maria.csr"

Normally, the distinguished name of the owner of the key pair should be asked when generating a CSR. But "keytool" has already asked and stored the distinguished name when generating the key pair.

Now Maria send her CSR file, maria.csr, to me. I will sign her CSR file into a public key certificate as described in the next section.

Last update: 2013.

"OpenSSL" Signing Maria's CSR

This section provides a tutorial example on how to use 'OpenSSL' to sign a CSR (Certificate Signing Request) generated by 'keytool' with CA's private key.

When I got Maria's CSR (Certificate Signing Request), maria.csr, I can sign it with my CA private key with the "openssl x509 -req" command as shown in the command session below:

```
>openssl x509 -req -in maria.csr -CA herong.crt
   -CAkey herong.key -out maria.crt -days 365
   -CAcreateserial -CAserial herong.seq

Loading 'screen' into random state - done
Signature ok
subject=/C=AT/ST=Maria State/L=Maria City/O=Maria Company
/OU=Maria Unit/CN=Maria Teresa
Getting CA Private Key
Enter pass phrase for herong.key: keypass

>type maria.crt
-----BEGIN CERTIFICATE-----
MIIEGTCCAwECAQIwDQYJKoZIhvcNAQEEBQAwgZMxCzAJBgNVBAYTAkhZ...
...
k7R7Q4bN2eDWX9eiUid6VuJefLx3S1HlyVLwBlR1t4zqUZUeZxVEhqf6...
-----END CERTIFICATE-----
```

Cool. CSR generated by "keytool" is compatible with "OpenSSL". Here are some notes on what I did:

- "openssl x509 -req" command signs a CSR (Certificate Sign Request) with my private key and public key certificate.
- "-req" option specifies the entry in the keystore file where to get the key pair.
- "-in maria.csr" option specifies the CSR file received from someone else.
- "-CA herong.crt" option specifies my public key certificate file.
- "-CAkey herong.key" option specifies my private key file. Password will be prompted.

- "-days 365" option specifies that the signed certificate is good for 365 days, about 1 year.
- "-out maria.crt" option specifies the file name to store Maria's public key certificate signed by me.
- "-CAcreateserial" option tells "OpenSSL" to created a serial number file, if it has not been created. The serial number value will start with 1. It will be inserted into the resulting certificate.
- "-CAserial herong.seq" option specifies the serial number file name.
- "type maria.crt" command displays the content of "maria.crt".

Do you want to see some detail information about Maria's public key certificate? Try this command "openssl x509":

```
>openssl x509 -in maria.crt -noout -text

Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number: 1 (0x1)
        Signature Algorithm: md5WithRSAEncryption
        Issuer: C=HY, ST=HY State, L=HY City, O=HY Company,
OU=HY Unit, CN=Herong Yang/emailAddress=herongyang.com
        Validity
            Not Before: Apr 1:57:05 2007 GMT
            Not After : Mar 31 17:57:05 2008 GMT
        Subject: C=AT, ST=Maria State, L=Maria City,
O=Maria Company, OU=Maria Unit, CN=Maria Teresa
        Subject Public Key Info:
            Public Key Algorithm: dsaEncryption
            DSA Public Key:
                pub:
                    0a:aa:91:a7:4e:36:39:4b:95:5e:fb:99:...
                    ...
                    79:30:3a:fe:40:38:71:71
                P:
                    00:fd:7f:53:81:1d:75:12:29:52:df:4a:...
                    ...
                    f2:22:03:19:9d:d1:48:01:c7
                Q:
                    00:97:60:50:8f:15:23:0b:cc:b2:92:b9:...
                    84:0b:f0:58:1c:f5
                G:
                    00:f7:e1:a0:85:d6:9b:3d:de:cb:bc:ab:...
                    ...
                    25:64:01:4c:3b:fe:cf:49:2a
    Signature Algorithm: md5WithRSAEncryption
        00:9e:25:92:ce:33:b1:00:fc:a1:ef:b8:70:d9:97:aa:...
        ...
        fa:c0:68:6c
```

The detailed information of the certificate seems to be good. The issuer is me, Herong Yang. The subject is Maria Teresa. The expiration is one year later.

What needs to happen next are:

- I need to return the signed certificate of Maria's public key back to the Maria. She can give this certificate to other people now and tell them that it is signed by Herong Yang.
- I need to give a copy of my CA self-signed public key certificate to Maria also. She can use my certificate to verify my signature on her certificate.
- Maria needs to import both certificates into her keystore file. See the next section for details.

Last update: 2013.

"OpenSSL" Managing Serial Numbers when Signing CSR

This section provides a tutorial example on how to manage serial number when using 'OpenSSL' to sign a CSR (Certificate Signing Request) generated by 'keytool' with CA's private key.

If I use the "openssl x509 -req" command without providing serial number options, "OpenSSL" will give me an error like this:

```
>openssl x509 -req -in maria.csr -CA herong.crt
-CAkey herong.key -out maria.crt -days 365

Loading 'screen' into random state - done
Signature ok
subject=/C=AT/ST=Maria State/L=Maria City/O=Maria Company
/OU=Maria Unit/CN=Maria Teresa
Getting CA Private Key
Enter pass phrase for herong.key: keypass
herong.srl: No such file or directory
2744:error:02001002:system library:fopen:No such file or directory:
bss_file.c:276:fopen('herong.srl','rb')
2744:error:20074002:BIO routines:FILE_CTRL:system lib:bss_file.c:278:
```

"OpenSSL" will try to open a file named "herong.srl". The error message is not clear at all. It does not say that "herong.srl" is the serial number file. There are 3 ways to supply a serial number to the "openssl x509 -req" command:

- Create a text file named as "herong.srl" and put a number in the file.
- Use the "-set_serial n" option to specify a number each time.
- Use the "-CAcreateserial -CAserial herong.seq" option to let "OpenSSL" to create and manage the serial number.

Last update: 2013.

"keytool" Importing CA's Own Certificate

This section provides a tutorial example on how to import CA's certificate generated by 'OpenSSL' into a 'keystore' file using 'keytool'

When Maria receives my CA self-signed public key certificate file, she needs to imported it into her keystore file with the "keytool -importcert" command as shown below:

```
>keytool -importcert -alias herong_crt -keypass keypass
-file herong.crt -keystore maria.jks -storepass jkspass

Owner: EMAILADDRESS=herongyang.com, CN=Herong Yang,
OU=HY Unit, O=HY Company, L=HY City, ST=HY State, C=HY
Issuer: EMAILADDRESS=herongyang.com, CN=Herong Yang,
OU=HY Unit, O=HY Company, L=HY City, ST=HY State, C=HY
Serial number: 0
Valid from: Sun Apr 1:42:10 EDT 2007
until: Wed Mar 29 23:42:10 EDT 2017
Certificate fingerprints:
         MD5:  2D:95:8D:5F:0F:4A:9B:CC:A2:69:61:F6:22:AE...
         SHA1: 1F:BB:C7:78:97:AC:C8:BF:7B:A4:88:DF:B5:62...
         Signature algorithm name: MD5withRSA
         Version: 3

Extensions:

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
...

[EMAILADDRESS=herongyang.com, CN=Herong Yang, OU=HY Unit,
O=HY Company, L=HY City, ST=HY State, C=HY]
SerialNumber: [    00]
]

Trust this certificate? [no]:  yes
Certificate was added to keystore
```

Notes on what Maria did:

- "keytool -importcert" commands imports a certificate into a keystore file.
- "-alias herong_crt" option specifies a new entry name for the imported certificate.
- "-keypass keypass" option specifies a password to protect this new entry.
- "-file herong.crt" option specifies the file name of certificate to be imported.
- "-keystore maria.jks -storepass jkspass" option specifies the keystore file name and its password.
- Maria entered "yes" when "keytool" asked to trust this certificate or not.

Want to see if the certificate was imported correctly or not? Try this "keytool -list" command:

```
>keytool -list -keystore maria.jks -storepass jkspass
```

```
Keystore type: JKS
Keystore provider: SUN

Your keystore contains 2 entries

maria_key, Apr 1, 2007, PrivateKeyEntry,
Certificate fingerprint (MD5): 54:5A:E8:77:30:82:B4:EB:C...
herong_crt, Apr 1, 2007, trustedCertEntry,
Certificate fingerprint (MD5): C1:6C:FE:38:F7:0F:71:23:3...
```

As we can see, my CA certificate was imported ok and stored as a "trustedCertEntry". So certificates generated by "OpenSSL" is compatible with "keytool" certificate format.

If Maria made a mistake when import my certificate, she could use this command to delete my certificate from her keystore file:

```
>keytool -delete -alias herong_crt -keystore maria.jks
-storepass jkspass
```

Now Maria is ready to import her own public key certificate signed by me as described in the next section.

Last update: 2013.

""keytool" Importing Maria's Certificate Signed by CA

This section provides a tutorial example on how to import a certificate signed by a CA using 'OpenSSL' into a 'keystore' file using 'keytool'

After importing CA's certificate (Herong's certificate), Maria should import her own certificate which was signed by the CA (Herong) using the "keytool -importcert" command as shown below:

```
>keytool -importcert -alias maria_crt -keypass keypass
-file maria.crt -keystore maria.jks -storepass jkspass

Certificate was added to keystore
```

The command was the same one used to import CA's certificate. But this time, "keytool" did not ask Maria to trust this certificate or not. It looks like "keytool" did a validation and found that Maria's certificate was signed by a trusted certificate, herong_crt, in the keystore.

Want to see what's in the keystore file now? Try this "keytool -list" command:

```
>keytool -list -keystore maria.jks -storepass jkspass

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 3 entries
```

```
maria_key, Apr 1, 2007, PrivateKeyEntry,
Certificate fingerprint (MD5): 54:5A:E8:77:30:82:B4:EB:C...
herong_crt, Apr 1, 2007, trustedCertEntry,
Certificate fingerprint (MD5): C1:6C:FE:38:F7:0F:71:23:3...
maria_crt, Apr 1, 2007, trustedCertEntry,
Certificate fingerprint (MD5): 5B:AB:DC:62:6E:F4:F4:96:5...
```

By now, Maria has everything she needs in her 'keystore' file to perform a digital signature on anything. Receiver of her digital signature can trust her signature, because I, as a CA, signed her certificate.

Last update: 2013.

Migrating Keys from "keystore" to "OpenSSL" Key Files

This chapter provides tutorial notes and example codes on migrating keys from 'keystore' files to 'OpenSSL' key files. Topics include generating keys in 'keystore' files; dumping keys from 'keystore' files; converting binary key files to PEM format; reading keys with OpenSSL.

Another type of questions I received is related to moving keys from "keytool" keystore files to "OpenSSL" key files. Since "keytool" does not support key exporting function, I wrote a Java program to dump keys out of the keystore file. In this chapter, I recorded some testing scenarios to find a way to move keys from "keytool" keystore files to "OpenSSL" key files.

Conclusions:

- "keytool -genkeypair" command does two things: generating a DSA key pair and generating the self-signed certificate.
- "keytool -exportcert" command only exports the self-signed certificate from a PrivateKeyEntry in a keystore.
- "DumpKey.java" program dumps the key pair into a binary PKCS#8 format from a PrivateKeyEntry in a keystore.
- "openssl enc" command can be used to perform Base64 encoding on PKCS#8 files.
- PEM format requires a header line and footer line in the Base64 encoded file.
- Key pairs generated with "keytool" are compatible with "OpenSSL".

No "keytool" Command to Export Keys

This section describes all sub-commands supported by the 'keytool' provided in JDK 1.6. There is not 'keytool' sub-command to export keys stored in 'keystore' files.

To figure out how to use "keytool" to export keys (pairs of private keys and public keys), not certificates, out of "keystore" files, I re-examined all sub-commands supported by the "keytool" tool provided in JDK 1.6.

The JDK 1.6 manual gives me the following list of sub-commands, total of 13:

```
keytool usage:

-certreq
        Generating CSR from a key pair entry

-changealias
        Renaming an entry in the keystore file

-delete
        Deleting an entry in the keystore file

-exportcert
        Exporting a certificate entry

-genkeypair
        Generating a new key pair entry

-genseckey
        Generating a secret key entry

-help
        Displaying help information

-importcert
        Importing a certificate into the keystore file

-importkeystore
        Importing all entries from another keystore file

-keypasswd
        Changing the password for an existing entry

-list
        Display all entry names

-printcert
        Print a certificate file

-storepasswd
        Changing the keystore file password
```

This confirms that as of JDK 1.6, keys stored in the key entry in "keystore" files can not be exported into key files using the "keytool" tool.

Last update: 2013.

"keytool -genkeypair" Generating PrivateKeyEntry

This section provides a tutorial example on how to generate a private and public key pair using the 'keytool -genkeypair' command. It stores the key pair in a 'PrivateKeyEntry' in a 'keystore' file.

To prepare my test on how to export private and public key pairs out of "keystore" files, I need to generate a pair of keys first in a "keystore" file with the "keytool -genkeypair" command. What I did was recorded below:

```
>java -version

java version "1.6.0_01"
Java(TM) SE Runtime Environment (build 1.6.0_01-b06)
Java HotSpot(TM) Client VM (build 1.6.0_01-b06, mixed mode,
  sharing)

>keytool -genkeypair -alias herong_key -keypass keypass
-keysize 1024 -keystore herong.jks -storepass jkspass

What is your first and last name?
  [Unknown]:  Herong Yang
What is the name of your organizational unit?
  [Unknown]:  Herong Unit
What is the name of your organization?
  [Unknown]:  Herong Company
What is the name of your City or Locality?
  [Unknown]:  Herong City
What is the name of your State or Province?
  [Unknown]:  Herong State
What is the two-letter country code for this unit?
  [Unknown]:  CA
Is CN=Herong Yang, OU=Herong Unit, O=Herong Company,
L=Herong City, ST=Herong State, C=CA correct?
  [no]:  yes

>keytool -list -keystore herong.jks -storepass jkspass

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry

herong_key, Apr 1, 2007, PrivateKeyEntry,
Certificate fingerprint (MD5): 0C:54:AE:99:4E:3D:F7:A9:7...
```

I am not going to explain all the command options used above, because they were explained in previous chapters. As you can see the private and public key pair is stored in a "PrivateKeyEntry".

In the next section, I tried to use "keytool -exportcert" to export this "PrivateKeyEntry".

Last update: 2013.

"keytool -exportcert" Exporting PrivateKeyEntry

This section provides a tutorial example on how to export a 'PrivateKeyEntry' stored in a 'keystore' file using the 'keytool -exportcert' command.

After generating my key pair with the "keytool -genkeypair" command, I got a PrivateKeyEntry inside the keystore file, herong.jks. So I tried to export it using the "keytool -exportcert" command as shown in the following command session:

```
>keytool -exportcert -alias herong_key -keypass keypass
-keystore herong.jks -storepass jkspass -file herong.crt
-rfc

Certificate stored in file <herong.crt>

>type herong.crt
-----BEGIN CERTIFICATE-----
MIIDODCCAvagAwIBAgIERqplETALBgcqhkjOOAQDBQAwfzELMAkGA1UE...
...
Cgfs2kXj/IQCFDC5GT5IrLTIFxAyPUo1tJo2DPkK
-----END CERTIFICATE-----
```

Cool. A certificate was exported. I am not going to explain all the command options used above, because they were explained in previous chapters. But I want to mention this "-rfc" option:

- "-rfc" tells "keytool" to write the output certificate in "Base 64 encoding" form described in "RFC 1421 Certificate Encoding Standard".

Without "-rfc" option, "keytool" will output certificate in a binary form, which will be very hard to transfer.

I got this certificate exported from the PrivateKeyEntry of my key pair. What is in this certificate? I will try to use "keytool -printcert" command to look into this certificate in the next section.

Last update: 2013.

"keytool -printcert" Printing Certificate Details

This section provides a tutorial example on how to print details of the certificate exported by 'keytool -exportcert' command using the 'keytool -printcert' command.

With the "keytool -exportcert" command, I got a certificate, herong.crt, exported from the PrivateKeyEntry of my key pair. Now I want see some details of this certificate with the "keytool -printcert" command as shown below:

```
>keytool -printcert -file herong.crt

Owner: CN=Herong Yang, OU=Herong Unit, O=Herong Company,
L=Herong City, ST=Herong State, C=CA
Issuer: CN=Herong Yang, OU=Herong Unit, O=Herong Company,
L=Herong City, ST=Herong State, C=CA
```

```
Serial number: 46aa6511
Valid from: Sun Apr 1 17:35:13 EDT 2007
until: Sat Jun 30 17:35:13 EDT 2007
Certificate fingerprints:
         MD5:  0C:54:AE:99:4E:3D:F7:A9:79:1A:93:83:0F:EF...
         SHA1: CA:23:1C:D4:F9:74:84:4C:16:F7:E7:AB:B1:08...
         Signature algorithm name: SHA1withDSA
         Version: 3
```

OK. Now I know that:

- This certificate, herong.crt, is a self-signed certificate of my public key.
- The certificate is valid for 90 days only.
- Command "keytool -exportcert" will not export the key pair itself.

After this test, I read the Java manual again. It explains what exactly "keytool -genkeypair" does clearly: "Generates a key pair (a public key and associated private key). Wraps the public key into an X.509 v3 self-signed certificate, which is stored as a single-element certificate chain. This certificate chain and the private key are stored in a new keystore entry identified by alias."

So the "PrivateKeyEntry" in the keystore file has two components: my key pair and my self-signed public key certificate.

The "keytool -exportcert" command only exports the self-signed certificate. The key pair will not be exported.

The next question is that could this certificate generated by "keytool" be viewed by "OpenSSL"? See the next section for answers.

Last update: 2013.

"openssl x509" Viewing Certificate Details

This section provides a tutorial example on how to view details of the certificate exported by 'keytool -exportcert' command using the 'openssl x509' command.

From previous section, I got my self-signed certificate generated by "keytool". The certificate is stored in RFC 1421 format. Now I want to try to view this certificate with "OpenSSL x509" command as shown below:

```
>openssl x509 -in herong.crt -noout -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1185572113 (0x46aa6511)
        Signature Algorithm: dsaWithSHA1
        Issuer: C=CA, ST=Herong State, L=Herong City,
O=Herong Company, OU=Herong Unit, CN=Herong Yang
        Validity
```

```
            Not Before: Apr 1 21:35:13 2007 GMT
            Not After : Jun 30 21:35:13 2007 GMT
        Subject: C=CA, ST=Herong State, L=Herong City,
O=Herong Company, OU=Herong Unit, CN=Herong Yang
        Subject Public Key Info:
            Public Key Algorithm: dsaEncryption
            DSA Public Key:
                pub:
                    00:b0:61:2b:c1:88:0e:19:66:58:37:b5:...
                    ...
                P:
                    00:fd:7f:53:81:1d:75:12:29:52:df:4a:...
                    ...
                Q:
                    00:97:60:50:8f:15:23:0b:cc:b2:92:b9:...
                    ...
                G:
                    00:f7:e1:a0:85:d6:9b:3d:de:cb:bc:ab:...
                    ...
   Signature Algorithm: dsaWithSHA1
       30:2c:02:14:6c:21:f3:43:b5:4f:d5:3d:2e:23:89:45:0...
       ...
```

What I learned from this test:

- Certificates generated by "keytool" are compatible with "OpenSSL".
- "openssl x509" is provides much more certificate details than "keytool -printcert" command.
- "keytool -genkeypair" uses DSA algorithm as the default to generated the private key and public key pair.

Last update: 2013.

"DumpKey.java" Dumping Private Keys Out of "keystore"

This section provides a tutorial example on how to dump private key and public key pair output of a 'PrivateKeyEntry' in a 'keystore' file using a Java program, DumpKey.java.

Since "keytool" can not be used to export the private and public key pair out of a 'keystore' file, I wrote the following Java program, DumpKey.java, to do this job:

```
/* DumpKey.java
 * Copyright (c) 2013 by Dr. Herong Yang, herongyang.com
 */
import java.io.*;
import java.security.*;
public class DumpKey {
   static public void main(String[] a) {
      if (a.length<5) {
         System.out.println("Usage:");
         System.out.println(
            "java DumpKey jks storepass alias keypass out");
         return;
      }
      String jksFile = a[0];
      char[] jksPass = a[1].toCharArray();
      String keyName = a[2];
      char[] keyPass = a[3].toCharArray();
      String outFile = a[4];

      try {
         KeyStore jks = KeyStore.getInstance("jks");
         jks.load(new FileInputStream(jksFile), jksPass);
         Key key = jks.getKey(keyName, keyPass);
         System.out.println("Key algorithm: "+key.getAlgorithm());
         System.out.println("Key format: "+key.getFormat());
         System.out.println("Writing key in binary form to "
            +outFile);

         FileOutputStream out = new FileOutputStream(outFile);
         out.write(key.getEncoded());
         out.close();
      } catch (Exception e) {
         e.printStackTrace();
         return;
      }
   }
}
```

Notes on DumpKey.java:

- The first step is to load the keystore file into "jks". I am assuming that the keystore type is "jks", which is the default type used by "keytool". Another type is "pkcs12".
- The second step is to obtain the key from the specified key entry name. I am assuming that the specified entry is a PrivateKeyEntry, which contains two components: the key and the self-signed certificate.
- The last step is to dump the key in the default encoding format. Note that the encoding format is still in a binary form.
- Converting the output in Base64 encoding is not done, because JDK does not offer any Base64 classes.

I tried my DumpKey.java program with my key pair stored in herong.jks as show below:

```
>javac DumpKey.java

>java DumpKey herong.jks jkspass herong_key keypass herong_bin.key
Key algorithm: DSA
Key format: PKCS#8
Writing key in binary form to herong_bin.key
```

Excellent. I got my key pair dumped out of the keystore file into a binary PKCS#8 format.

Now I am ready to test my private and public key pair with "OpenSSL" as shown in the next section.

Last update: 2013.

"openssl enc" Converting Keys from Binary to PEM

This section provides a tutorial example on how to convert a private and public key pair stored in binary PKCS#8 format into PEM (Privacy Enhanced Mail) format with the 'openssl enc' command.

Using my DumpKey.java program, I managed to get a private and public key pair dumped out of the "keytool" keystore file into herong_bin.key. My DumpKey.java program told me that this is a DSA key pair stored in binary PKCS#8 format.

I tried to view herong_bin.key as is with the "openssl dsa" command:

```
>openssl dsa -in herong_bin.key -text

read DSA key
unable to load Key
2228:error:0906D06C:PEM routines:PEM_read_bio:no start line:
pem_lib.c:632:Expecting: ANY PRIVATE KEY
```

Looks like "openssl dsa" command only understand PEM (Privacy Enhanced Mail) format which requires the key to be encoded in Base64 format. This can be done in two steps. First, use "openssl enc" command as shown below:

```
>openssl enc -in herong_bin.key -out herong.key -a

>type herong.key
MIIBSwIBADCCASwGByqGSM44BAEwggEfAoGBAP1/U4EddRIpUt9KnC7s5Of2EbdS
...
g9/hWuWfBpKLZl6Ae1UlZAFMO/7PSSoEFgIUSVbo98XAZDN9RZoZ+li3kIKVEbk=
```

The last step to make my herong.key file to meet PEM format standard is to add a header line and a footer line with a text editor:

```
-----BEGIN PRIVATE KEY-----
MIIBSwIBADCCASwGByqGSM44BAEwggEfAoGBAP1/U4EddRIpUt9KnC7s5Of2EbdS
...
g9/hWuWfBpKLZl6Ae1UlZAFMO/7PSSoEFgIUSVbo98XAZDN9RZoZ+li3kIKVEbk=
-----END PRIVATE KEY-----
```

Now I got my private and public key pair converted from a binary format to the PEM format in the file called herong.key. Remember my key pair was generated by "keytool".

The next thing I want to do is view this key pair with the "openssl dsa" command as described in the next section.

Last update: 2013.

"openssl dsa" Viewing Private and Public Key Pair

This section provides a tutorial example on how to view a private and public key pair stored in PEM format using the 'openssl dsa' command. The key pair was originally generated by the 'keytool -genkeypair' command.

After going through so much trouble of dumping the key pair out of the keystore file, encoding it with Base64, and making it to meet PEM file standard, finally I can view it with the "openssl dsa" command now:

```
>openssl dsa -in herong.key -text
read DSA key
Private-Key: (1024 bit)
priv:
    49:56:e8:f7:c5:c0:64:33:7d:45:9a:19:fa:58:b7:
    90:82:95:11:b9
pub:
    00:b0:61:2b:c1:88:0e:19:66:58:37:b5:bc:0f:78:
    88:f7:79:b5:fa:6c:cb:6c:b2:86:44:d8:b2:15:13:
    e3:09:dd:9c:5a:52:02:4a:fb:1c:30:e8:2b:b5:45:
    8f:88:5a:57:a9:1f:c0:b8:3d:1c:a1:a9:6f:20:76:
    a7:c0:eb:5e:df:bf:87:84:14:02:53:d5:87:c9:3a:
    13:9d:e8:45:4f:c6:2d:48:44:e2:80:18:63:e4:40:
    a1:f0:2c:e4:d2:86:34:8d:dd:93:92:42:1a:19:1d:
    0e:44:f6:a2:6d:94:1a:ed:d7:d2:94:7e:0b:26:88:
    a4:cb:c4:88:5b:56:49:2e:80
P:
    00:fd:7f:53:81:1d:75:12:29:52:df:4a:9c:2e:ec:
    e4:e7:f6:11:b7:52:3c:ef:44:00:c3:1e:3f:80:b6:
    51:26:69:45:5d:40:22:51:fb:59:3d:8d:58:fa:bf:
    c5:f5:ba:30:f6:cb:9b:55:6c:d7:81:3b:80:1d:34:
    6f:f2:66:60:b7:6b:99:50:a5:a4:9f:9f:e8:04:7b:
    10:22:c2:4f:bb:a9:d7:fe:b7:c6:1b:f8:3b:57:e7:
    c6:a8:a6:15:0f:04:fb:83:f6:d3:c5:1e:c3:02:35:
    54:13:5a:16:91:32:f6:75:f3:ae:2b:61:d7:2a:ef:
    f2:22:03:19:9d:d1:48:01:c7
Q:
    00:97:60:50:8f:15:23:0b:cc:b2:92:b9:82:a2:eb:
    84:0b:f0:58:1c:f5
G:
    00:f7:e1:a0:85:d6:9b:3d:de:cb:bc:ab:5c:36:b8:
    57:b9:79:94:af:bb:fa:3a:ea:82:f9:57:4c:0b:3d:
    07:82:67:51:59:57:8e:ba:d4:59:4f:e6:71:07:10:
    81:80:b4:49:16:71:23:e8:4c:28:16:13:b7:cf:09:
    32:8c:c8:a6:e1:3c:16:7a:8b:54:7c:8d:28:e0:a3:
    ae:1e:2b:b3:a6:75:91:6e:a3:7f:0b:fa:21:35:62:
    f1:fb:62:7a:01:24:3b:cc:a4:f1:be:a8:51:90:89:
    a8:83:df:e1:5a:e5:9f:06:92:8b:66:5e:80:7b:55:
    25:64:01:4c:3b:fe:cf:49:2a
writing DSA key
-----BEGIN DSA PRIVATE KEY-----
MIIBvAIBAAKBgQD9f1OBHXUSKVLfSpwu7OTn9hG3UjzvRADDHj+AtlEmaUVdQCJR
+1k9jVj6v8X1ujD2y5tVbNeBO4AdNG/yZmC3a5lQpaSfn+gEexAiwk+7qdf+t8Yb
+DtX58aophUPBPuD9tPFHsMCNVQTWhaRMvZ1864rYdcq7/IiAxmd0UgBxwIVAJdg
UI8VIwvMspK5gqLrhAvwWBz1AoGBAPfhoIXWmz3ey7yrXDa4V7l5lK+7+jrqgvlX
TAs9B4JnUVlXjrrUWU/mcQcQgYC0SRZxI+hMKBYTt88JMozIpuE8FnqLVHyNKOCj
rh4rs6Z1kW6jfwv6ITVi8ftiegEkO8yk8b6oUZCJqIPf4VrlnwaSi2ZegHtVJWQB
TDv+z0kqAoGBALBhK8GIDhlmWDe1vA94iPd5tfpsy2yyhkTYshUT4wndnFpSAkr7
HDDoK7VFj4haV6kfwLg9HKGpbyB2p8DrXt+/h4QUAlPVh8k6E53oRU/GLUhE4oAY
Y+RAofAs5NKGNI3dk5JCGhkdDkT2om2UGu3X0pR+CyaIpMvEiFtWSS6AAhRJVuj3
```

```
xcBkM31Fmhn6WLeQgpURuQ==
-----END DSA PRIVATE KEY-----
```

Wonderful! Here is what I learned from this exercise:

- Private and public key pairs generated by "keytool" can be used by "OpenSSL".
- Private and public key pairs stored in "keystore" files can be dumped out. But I have to write a Java program, DumpKey.java, to do this.
- My DumpKey.java program does not write the key in PEM format. Extra steps are needed to convert the dumped binary keys to in PEM format.

My private and public key pair, herong.key, dumped and converted from the "keytool" keystore file is now ready to be used by "OpenSSL" for signing any documents.

Last update: 2013.

Certificate X.509 Standard and DER/PEM Formats

This chapter provides tutorial notes and example codes on certificate content standard and file formats. Topics include X.509 standard for certificate content; DER encoding for certificate binary file format; PEM encoding for certificate text file format; exchanging certificates in DER and PEM formats between 'OpenSSL' and 'keytool'.

Conclusions:

- X.509 is an international that defines the contents of digital certificates
- DER (Distinguished Encoding Rules) is a data object encoding schema that can be used to encode certificate objects into binary files.
- PEM (Privacy Enhanced Mail) is an encrypted email encoding schema that can be borrowed to encode certificate DER files into text files.
- A certificate PEM file is really a certificate DER file encoded with Base64 algorithm.
- "keytool" supports both PEM and DER certificate formats.
- "OpenSSL" supports both PEM and DER certificate formats.

X.509 Certificate Standard

This section describes the X.509 certificate standard - An international standard that defines what should be included in a digital certificate.

X.509 is an international standard for what should be included in a digital certificate. Here is the definition from webpedia.com:

A widely used standard for defining digital certificates. X.509 (Version 1) was first issued in 1988 as a part of the ITU X.500 Directory Services standard. When X.509 was revised in 1993, two more fields were added resulting in the Version 2 format. These two additional fields support directory access control. X.509 Version 3 defines the format for certificate extensions used to store additional information regarding the certificate holder and to define certificate usage. Collectively, the term X.509 refers to the latest published version, unless the version number is stated.

X.509 is published as ITU recommendation ITU-T X.509 (formerly CCITT X.509) and ISO/IEC/ITU 9594-8 which defines a standard certificate format for public key certificates and certification validation. With minor differences in dates and titles, these publications provide identical text in the defining of public-key and attribute certificates.

My understanding of X.509 is that a certificate is required to have the following information:

- Version - X.509 standard version number.
- Serial Number - A sequence number given to each certificate.
- Signature Algorithm Identifier - Name of the algorithm used to sign this certificate by the issuer
- Issuer Name - Name of the issuer.
- Validity Period - Period during which this certificate is valid.
- Subject Name - Name of the owner of the public key.
- Subject Public Key Information - The public key and its related information.

The content structure of a Version 3 X.509 certificate should look like this:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1185572113 (0x46aa6511)
        Signature Algorithm: dsaWithSHA1
        Issuer: C=CA, ST=Herong State, L=Herong City, ...
        Validity
            Not Before: Apr 1 21:35:13 2007 GMT
            Not After : Jun 30 21:35:13 2007 GMT
        Subject: C=CA, ST=Herong State, L=Herong City, ...
        Subject Public Key Info:
            Public Key Algorithm: dsaEncryption
            DSA Public Key:
                pub:
                    00:b0:61:2b:c1:88:0e:19:66:58:37:b5:...
                    ...
                P:
                    00:fd:7f:53:81:1d:75:12:29:52:df:4a:...
                    ...
                Q:
                    00:97:60:50:8f:15:23:0b:cc:b2:92:b9:...
                    ...
                G:
                    00:f7:e1:a0:85:d6:9b:3d:de:cb:bc:ab:...
                    ...
   Signature Algorithm: dsaWithSHA1
       30:2c:02:14:6c:21:f3:43:b5:4f:d5:3d:2e:23:89:45:0...
       ...
```

X.509 defines how a certificate contents should be written. But it does not define how certificate contents should be encoded to store in files.

Two commonly used encoding schemas are used to store X.509 certificates in files, DER and PEM, as described in next sections.

Last update: 2013.

What Is DER (Distinguished Encoding Rules) Encoding?

This section describes the DER (Distinguished Encoding Rules) - A binary format of encoding a data value of any data types including nested data structures.

What Is DER? DER (Distinguished Encoding Rules) is one of ASN.1 encoding rules defined in ITU-T X.690, 2002, specification. ASN.1 encoding rules can be used to encode any data object into a binary file.

The basic encoding rule of DER is that a data value of all data types shall be encoded as four components in the following order:

- Identifier octets
- Length octets
- Contents octets
- End-of-contents octets

Of course, complex data types are supported to create data values of nested structures like this example:

```
{
 name {givenName "John",initial "P",familyName "Smith"},
 title "Director",
 number 51,
 dateOfHire "19710917",
 nameOfSpouse {givenName "Mary",initial "T",familyName "Smith"},
 children
  {
   {
    name {givenName "Ralph",initial "T",familyName "Smith"},
    dateOfBirth "19571111"
   },
   {
    name {givenName "Susan",initial "B",familyName "Jones"},
    dateOfBirth "19590717"
   }
  }
 }
```

As you can see from this example, DER is flexible enough to encode almost any data object.

The full specification of DER is in ITU-T X.690 - Information technology # ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER) .

DER is used as the most popular encoding format to store X.509 certificates in files. Those certificate DER files are binary files, which can not be viewed with text editors. But they can be processed by application without any problems.

DER encoded certificate files are supported by almost all applications. "OpenSSL" and "keytool" support DER encoded certificate files with no problem. See other sections below for examples of certificate files saved in DER encoding.

Last update: 2013.

What Is PEM (Privacy Enhanced Mail) Encoding?

This section describes the PEM (Privacy Enhanced Mail) Encoding - An encoding schema defined in RFC 1421-1424, 1993, specification to encode an email with encryptions into a text message of printable characters.

What Is PEM? PEM (Privacy Enhanced Mail) is an encoding schema defined in RFC 1421-1424, 1993, specification to encode an email with encryptions into a text message of printable characters.

A PEM encoded email has 5 elements in this order:

- A pre-boundary line of "-----BEGIN PRIVACY-ENHANCED MESSAGE-----".
- Header Portion
- Blank Line
- Text Portion
- A post-boundary line of "-----END PRIVACY-ENHANCED MESSAGE-----".

Here is an example of PEM encoded email message:

```
-----BEGIN PRIVACY-ENHANCED MESSAGE-----
Proc-Type: 4,ENCRYPTED
```

```
Content-Domain: RFC822
DEK-Info: DES-CBC,BFF968AA74691AC1
Originator-Certificate:
 MIIBlTCCAScCAWUwDQYJKoZIhvcNAQECBQAwUTELMAkGA1UEBhMCVVMxIDAeBgNV
 BAoTF1JTQSBEYXRhIFNlY3VyaXR5LCBJbmMuMQ8wDQYDVQQLEwZCZXRhIDExDzAN
 BgNVBAsTBk5PVEFSWTAeFw05MTA5MDQxODM4MTdaFw05MzA5MDMxODM4MTZaMEUx
 CzAJBgNVBAYTAlVTMSAwHgYDVQQKExdSU0EgRGF0YSBTZWN1cml0eSwgSW5jLjEU
 MBIGA1UEAxMLVGVzdCBVc2VyIDEwWTAKBgRVCAEBAgICAANLADBIAkEAwHZHl7i+
 yJcqDtjJCowzTdBJrdAiLAnSC+CnnjOJELyuQiBgkGrgIh3j8/x0fM+YrsyF1u3F
 LZPVtzlndhYFJQIDAQABMA0GCSqGSIb3DQEBAgUAA1kACKr0PqphJYw1j+YPtcIq
 iWlFPuN5jJ79Khfg7ASFxskYkEMjRNZV/HZDZQEhtVaU7Jxfzs2wfX5byMp2X3U/
 5XUXGx7qusDgHQGs7Jk9W8CW1fuSWUgN4w==
Key-Info: RSA,
 I3rRIGXUGWAF8js5wCzRTkdhO34PTHdRZY9Tuvm03M+NM7fx6qc5udixps2Lng0+
 wGrtiUm/ovtKdinz6ZQ/aQ==
Issuer-Certificate:
 MIIB3DCCAUgCAQowDQYJKoZIhvcNAQECBQAwTzELMAkGA1UEBhMCVVMxIDAeBgNV
 BAoTF1JTQSBEYXRhIFNlY3VyaXR5LCBJbmMuMQ8wDQYDVQQLEwZCZXRhIDExDTAL
 BgNVBAsTBFRMQ0EwHhcNOTEwOTAxMDgwMDAwWhcNOTIwOTAxMDc1OTU5WjBRMQsw
 CQYDVQQGEwJVUzEgMB4GA1UEChMXUlNBIERhdGEgU2VjdXJpdHksIEluYy4xDzAN
 BgNVBAsTBkJldGEgMTEPMA0GA1UECxMGTk9UQVJZMHAwCgYEVQgBAQICArwDYgAw
 XwJYCsnp6lQCxYykNlODwutF/jMJ3kL+3PjYyHOwk+/9rLg6X65B/LD4bJHtO5XW
 cqAz/7R7XhjYCm0PcqbdzoACZtIlETrKrcJiDYoP+DkZ8k1gCk7hQHpbIwIDAQAB
 MA0GCSqGSIb3DQEBAgUAA38AAICPv4f9Gx/tY4+p+4DB7MV+tKZnvBoy8zgoMGOx
 dD2jMZ/3HsyWKWgSF0eH/AJB3qr9zosG47pyMnTf3aSy2nBO7CMxpUWRBcXUpE+x
 EREZd9++32ofGBIXaialnOgVUn0OzSYgugiQ077nJLDUj0hQehCizEs5wUJ35a5h
MIC-Info: RSA-MD5,RSA,
 UdFJR8u/TIGhfH65ieewe2lOW4tooa3vZCvVNGBZirf/7nrgzWDABz8w9NsXSexv
 AjRFbHoNPzBuxwmOAFeA0HJszL4yBvhG
Recipient-ID-Asymmetric:
 MFExCzAJBgNVBAYTAlVTMSAwHgYDVQQKExdSU0EgRGF0YSBTZWN1cml0eSwgSW5j
 LjEPMA0GA1UECxMGQmV0YSAxMQ8wDQYDVQQLEwZOT1RBUlk=,
 66
Key-Info: RSA,
 O6BS1ww9CTyHPtS3bMLD+L0hejdvX6Qv1HK2ds2sQPEaXhX8EhvVphHYTjwekdWv
 7x0Z3Jx2vTAhOYHMcqqCjA==

qeWlj/YJ2Uf5ng9yznPbtD0mYloSwIuV9FRYx+gzY+8iXd/NQrXHfi6/MhPfPF3d
jIqCJAxvld2xgqQimUzoS1a4r7kQQ5c/Iua4LqKeq3ciFzEv/MbZhA==
-----END PRIVACY-ENHANCED MESSAGE-----
```

As you can see those streams of printable characters, Base64 encoding is used wherever is needed to convert a byte stream into a text stream.

For more details of PEM encoding, read these RFCs:

- RFC 1421 Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures .
- RFC 1422 Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management .
- RFC 1423 Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers .
- RFC 1424 Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services .

Last update: 2013.

Certificate in PEM Format

This section describes what is certificate PEM (Privacy Enhanced Mail) format - A format uses PEM idea to convert a certificate in DER format to a message of printable characters. The conversion uses the Base64 encoding.

As you can see from the previous section, when applications generate X.509 certificates, they usually write them as DER files. Since DER files are binary files, they can not be included directly into emails.

In order to transfer certificate DER files easily in emails, the PEM encoding idea described in previous section was borrowed to encode a DER file into a text message of printable characters with the help of Base64 encoding. The result is a certificate file in PEM format with 3 elements in this order:

- A pre-boundary line of "-----BEGIN CERTIFICATE-----".
- Base64 encoding output of DER encoded certificate
- A post-boundary line of "-----END CERTIFICATE-----".

Here is an example of PEM encoded certificate:

```
-----BEGIN CERTIFICATE-----
MIICUTCCAfugAwIBAgIBADANBgkqhkiG9w0BAQQFADBXMQswCQYDVQQGEwJDTjEL
MAkGA1UECBMCUE4xCzAJBgNVBAcTAkNOMQswCQYDVQQKEwJPTjELMAkGA1UECxMC
VU4xFDASBgNVBAMTC0hlcm9uZyBZYW5nMB4XDTA1MDcxNTIxMTk0N1oXDTA1MDgx
NDIxMTk0N1owVzELMAkGA1UEBhMCQ04xCzAJBgNVBAgTAlBOMQswCQYDVQQHEwJD
TjELMAkGA1UEChMCT04xCzAJBgNVBAsTAlVOMRQwEgYDVQQDEwtIZXJvbmcgWWFu
ZzBcMA0GCSqGSIb3DQEBAQUAA0sAMEgCQQCp5hnG7ogBhtlynpOS21cBewKE/B7j
V14qeyslnr26xZUsSVko36ZnhiaO/zbMOoRcKK9vEcgMtcLFuQTWDl3RAgMBAAGj
gbEwga4wHQYDVR0OBBYEFFXI70krXeQDxZgbaCQoR4jUDncEMH8GA1UdIwR4MHaA
FFXI70krXeQDxZgbaCQoR4jUDncEoVukWTBXMQswCQYDVQQGEwJDTjELMAkGA1UE
CBMCUE4xCzAJBgNVBAcTAkNOMQswCQYDVQQKEwJPTjELMAkGA1UECxMCVU4xFDAS
BgNVBAMTC0hlcm9uZyBZYW5nggEAMAwGA1UdEwQFMAMBAf8wDQYJKoZIhvcNAQEE
BQADQQA/ugzBrjjK9jcWnDVfGHlk3icNRq0oV7Ri32z/+HQX67aRfgZu7KWdI+Ju
Wm7DCfrPNGVwFWUQOmsPue9rZBgO
-----END CERTIFICATE-----
```

Because PEM uses printable characters only, PEM is used as the most popular encoding format to store X.509 certificates to transfer them through emails.

PEM encoded certificate files are supported by almost all applications. "OpenSSL" and "keytool" support PEM encoded certificate files with no problem. See other sections below for test notes.

Last update: 2013.

"keytool" Exporting Certificates in DER and PEM

This section provides a tutorial example on how to export certificates in DER and PEM format using the 'keytool -exportcert' command.

My first test was about "keytool" exporting certificates in DER and PEM formats. This was done as:

- Using "keytool -genkeypair" to generated a key pair and a self-sign certificate in a keystore file.
- Using "keytool -exportcert" to export the certificate in DER format.
- Using "keytool -exportcert -rfc" to export the certificate in PEM format.

The test session was recorded below:

```
>keytool -genkeypair -keysize 1024 -alias herong_key
-keypass keypass -keystore herong.jks -storepass jkspass

What is your first and last name?
  [Unknown]:  Herong Yang
What is the name of your organizational unit?
  [Unknown]:  Herong Unit
What is the name of your organization?
  [Unknown]:  Herong Company
What is the name of your City or Locality?
  [Unknown]:  Herong City
What is the name of your State or Province?
  [Unknown]:  Herong State
What is the two-letter country code for this unit?
  [Unknown]:  CA
Is CN=Herong Yang, OU=Herong Unit, O=Herong Company, L=Herong City,
ST=Herong State, C=CA correct?
  [no]:  yes

>keytool -exportcert -alias herong_key -keypass keypass
-keystore herong.jks -storepass jkspass -file keytool_crt.der

Certificate stored in file <keytool_crt.der>

>keytool -exportcert -alias herong_key -keypass keypass
-keystore herong.jks -storepass jkspass -rfc -file keytool_crt.pem

Certificate stored in file <keytool_crt.pem>
```

Note that "keytool -exportcert" command uses DER format by default. The "-rfc" option is to change it to PEM (RFC 1421) format.

Now I got one certificate generated by "keytool" and stored in two encoding files: keytool_crt.der and keytool_crt.pem. How can I verify that they are really using DER and PEM formats? I used "OpenSSL" to try to view them as described in the next section.

Last update: 2013.

"OpenSSL" Viewing Certificates in DER and PEM

This section provides a tutorial example on how to use 'OpenSSL' to view certificates in DER and PEM formats generated by the 'keytool -exportcert' command.

One way to verify if "keytool" did export my certificate using DER and PEM formats correctly or not is to use "OpenSSL" to view those certificate files. To do this, I used the "openssl x509" command to view keytool_crt.der and keytool_crt.pem:

```
>openssl x509 -in keytool_crt.pem -inform pem -noout -text

Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1185636568 (0x46ab60d8)
        Signature Algorithm: dsaWithSHA1
        Issuer: C=CA, ST=Herong State, L=Herong City, ...
        ...

>openssl x509 -in keytool_crt.der -inform der -noout -text

Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1185636568 (0x46ab60d8)
        Signature Algorithm: dsaWithSHA1
        Issuer: C=CA, ST=Herong State, L=Herong City, ...
        O=Herong Company, OU=Heron
        ...
```

Cool. "OpenSSL" can read certificates in DER and PEM formats generated by "keytool". What I learned so far:

- "keytool" can generate self-signed X5.09 version 3 certificates.
- "keytool" can export certificates with DER and PEM formats.
- "OpenSSL" can read certificates generated by "keytool" in both DER and PEM formats.

Last update: 2013.

"OpenSSL" Generating Certificates in DER and PEM

This section provides a tutorial example on how to generate certificates in DER and PEM formats using 'OpenSSL'.

After tested how "keytool" can be used to export certificates in DER and PEM formats, I decided to try with "OpenSSL" to see if it can generate certificates in DER and PEM formats or not.

What I did was to:

- Run "openssl genrsa" to generate a RSA key pair.
- Run "openssl req -new -x509" to generate a self-signed certificate and stored it in PEM format.
- Run "openssl x509" to convert the certificate from PEM encoding to DER format.

The test session was recorded below:

```
>openssl genrsa -out herong.key -des 1024

Loading 'screen' into random state - done
Generating RSA private key, 1024 bit long modulus
.........................++++++
...................++++++
e is 65537 (0x10001)
Enter pass phrase for herong.key: keypass
Verifying - Enter pass phrase for herong.key: keypass

>openssl req -new -x509 -key herong.key -out openssl_crt.pem
-outform pem -config openssl.cnf

Enter pass phrase for herong.key: keypass
You are about to be asked to enter information that will be
incorporated into your certificate request.
What you are about to enter is what is called a Distinguished
Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [CA]:
State or Province Name (full name) [HY State]:
Locality Name (eg, city) [HY City]:
Organization Name (eg, company) [HY Company]:
Organizational Unit Name (eg, section) [HY Unit]:
Common Name (eg, YOUR name) [Herong Yang]:
Email Address [herongyang.com]:

>openssl x509 -in openssl_crt.pem -inform pem
-out openssl_crt.der -outform der
```

Now I got one certificate generated by "OpenSSL" and stored in two files: openssl_crt.der and openssl_crt.pem. How can I verify that they are really using DER and PEM formats? I used "keytool" to try to view them as described in the next section.

Last update: 2013.

"keytool" Viewing Certificates in DER and PEM

This section provides a tutorial example on how to use 'keytool' to view certificates in DER and PEM formats generated by 'OpenSSL'.

One way to verify the certificate in DER and PEM formats generated by "OpenSSL" is to view it with the "keytool -printcert" command:

```
>keytool -printcert -file openssl_crt.pem

Owner: EMAILADDRESS=herongyang.com, CN=Herong Yang, OU=HY Unit, ...
Issuer: EMAILADDRESS=herongyang.com, CN=Herong Yang, OU=HY Unit, ...
Serial number: 0
Valid from: Sun Apr 1 13:02:22 EDT 2007 until: ...
Certificate fingerprints:
         MD5:  BF:B8:3A:19:E5:05:CE:CA:8C:F7:05:FA:FE:51:A6:EC
         SHA1: F7:C7:2A:57:73:5E:CE:E5:73:09:13:35:FB:91:CF:27:...
         Signature algorithm name: MD5withRSA
         Version: 3

Extensions:
...

>keytool -printcert -file openssl_crt.der

Owner: EMAILADDRESS=herongyang.com, CN=Herong Yang, OU=HY Unit, ...
Issuer: EMAILADDRESS=herongyang.com, CN=Herong Yang, OU=HY Unit, ...
Serial number: 0
Valid from: Sun Apr 1 13:02:22 EDT 2007 until: ...
Certificate fingerprints:
         MD5:  BF:B8:3A:19:E5:05:CE:CA:8C:F7:05:FA:FE:51:A6:EC
         SHA1: F7:C7:2A:57:73:5E:CE:E5:73:09:13:35:FB:91:CF:27:...
         Signature algorithm name: MD5withRSA
         Version: 3

Extensions:
...
```

Cool. "keytool" can read certificates in DER and PEM formats generated by "OpenSSL". What I learned so far:

- "OpenSSL" can generate self-signed X5.09 version 3 certificates.
- "OpenSSL" can write certificates with DER and PEM formats.
- "keytool" can read certificates generated by "OpenSSL" in both DER and PEM formats.

Last update: 2013.

"keytool" Importing Certificates in DER and PEM

This section provides a tutorial example on how to use 'keytool' to import certificates in DER and PEM formats generated by 'OpenSSL' into 'keystore' files.

I also tried to import the certificate generated by "OpenSSL" into "keytool" keystore files. The "keytool -importcert" command had no trouble reading the certificate in both PEM and DER formats. My command session is recorded here:

```
>keytool -importcert -file openssl_crt.pem
-keystore herong.jks -storepass jkspass
-alias openssl_crt_pem -keypass keypass

Owner: EMAILADDRESS=herongyang.com, CN=Herong Yang, OU=HY Unit, ...
Issuer: EMAILADDRESS=herongyang.com, CN=Herong Yang, OU=HY Unit, ...
Serial number: 0
Valid from: Sun Apr 1 13:02:22 EDT 2007 until: ...
Certificate fingerprints:
         MD5:  BF:B8:3A:19:E5:05:CE:CA:8C:F7:05:FA:FE:51:A6:EC
         SHA1: F7:C7:2A:57:73:5E:CE:E5:73:09:13:35:FB:91:CF:27:...
         Signature algorithm name: MD5withRSA
         Version: 3

Extensions:
...
Trust this certificate? [no]:  yes
Certificate was added to keystore

>keytool -importcert -file openssl_crt.der
-keystore herong.jks -storepass jkspass
-alias openssl_crt_der -keypass keypass

Certificate already exists in keystore under alias <openssl_crt_pem>
Do you still want to add it? [no]:  yes
Certificate was added to keystore

>keytool -list -keystore herong.jks -store
pass jkspass

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 3 entries

openssl_crt_der, Apr 1, 2007, trustedCertEntry,
Certificate fingerprint (MD5): BF:B8:3A:19:E5:05:CE:CA:8C:F7:05:...
openssl_crt_pem, Apr 1, 2007, trustedCertEntry,
Certificate fingerprint (MD5): BF:B8:3A:19:E5:05:CE:CA:8C:F7:05:...
herong_key, Apr 1, 2007, PrivateKeyEntry,
Certificate fingerprint (MD5): 5B:44:F1:D7:3D:9F:9E:15:5B:D1:25:...
```

Wonderful! There was no trouble at all for "keytool" to import my self-signed certificate generated by "OpenSSL" into the keystore file in both DER and PEM formats.

Last update: 2013.

Migrating Keys from "OpenSSL" Key Files to "keystore"

This chapter provides tutorial notes and example codes on migrating keys from 'OpenSSL' key files to 'keystore' files. Topics include generating a 'OpenSSL' key file; generating a self-signed certificate; merging a key file with its self-signed certificate into a PKCS#12 file; importing PKCS#12 files into 'kestore' files.

Conclusions:

- PKCS#8 is designed as the Private-Key Information Syntax Standard. It defines what should be included in a private key file.
- PKCS#12 is designed as the Personal Information Exchange Syntax Standard. It defines how to package a private key and its self-signed certificates into a single file.
- "openssl pkcs8" command can be used to read and write private keys in PKCS#8 format.
- "openssl pkcs12" command can be used to read and write PKCS#12 files.
- "keytool -importkeystore" command can be used to convert contents between PKCS#12 files and JKS files.

What Is PKCS#8?

This section describes what is PKCS#8 - One of the PKCS (Public Key Cryptography Standards) used to store a single private key. A PKCS#8 file can be encrypted with a password to protect the private key.

PKCS#8 is one of the PKCS (Public Key Cryptography Standards) devised and published by

RSA Security. PKCS#8 is designed as the Private-Key Information Syntax Standard. It is used to store private keys.

PKCS#8 standard actually has two versions: non-encrypted and encrypted.

The non-encrypted PKCS#8 version defines the following syntax for a private key:

```
PrivateKeyInfo ::= SEQUENCE {
  version Version,

  privateKeyAlgorithm PrivateKeyAlgorithmIdentifier,
  privateKey PrivateKey,
  attributes [0] IMPLICIT Attributes OPTIONAL }

Version ::= INTEGER

PrivateKeyAlgorithmIdentifier ::= AlgorithmIdentifier

PrivateKey ::= OCTET STRING

Attributes ::= SET OF Attribute
```

The encrypted PKCS#8 version defines the following syntax:

```
EncryptedPrivateKeyInfo ::= SEQUENCE {
  encryptionAlgorithm EncryptionAlgorithmIdentifier,
  encryptedData EncryptedData }

EncryptionAlgorithmIdentifier ::= AlgorithmIdentifier

EncryptedData ::= OCTET STRING
```

Java SE "keytool" does not support exporting private keys in PKCS#8 format directly. But you can use my "DumpKey.java" to do this as described in another chapter of this book.

"OpenSSL" does not support exporting private keys in PKCS#8 format directly. It writes private keys in its own format referred as a private key traditional format. But it offers the "openssl pkcs8" command to convert private keys files from traditional format to pkcs#8 back and forth.

When writing a private key in PKCS#8 format in a file, it needs to stored in either DER encoding or PEM encoding. DER and PEM encodings are describes in other chapters in this book.

Visit PKCS page at rsa.comto read more about PKCS#8.

Last update: 2013.

What Is PKCS#12?

This section describes what is PKCS#12 - One of the PKCS (Public Key Cryptography Standards) used to store a private key and its self-signed certificate together as a single file.

PKCS#12 is one of the PKCS (Public Key Cryptography Standards) devised and published by RSA Security. PKCS#12 is designed as the Personal Information Exchange Syntax Standard.

PKCS#12 can be used in the same way as JKS (Java KeyStore) to store a private key and its self-signed certificate together in a single file. In fact, the Java SE "keytool" supports two keystore types: "jks" and "pkcs12".

When you use "OpenSSL" to generate private keys and certificates, they are stored as individual separate files. But "OpenSSL" does offer the "openssl pkcs12" command to merge private keys and certificates into a PKCS#12 file.

The "openssl pkcs12" command is very important if you want exchange private keys and certificates between "keytool" and "OpenSSL". Read other sections to see my tutorial notes on this.

Visit PKCS page at rsa.comto read more about PKCS#8.

Last update: 2013.

"openssl genrsa" Generating Private Key

This section provides a tutorial example on how to generate a RSA private key with the 'openssl genrsa' command. The key file can be then converted to DER or PEM encoding with or without DES encryption.

To understand better about PKCS#8 private key format, I started with "OpenSSL" to generate a RSA private key (it's really a private and public key pair). The "openssl genrsa" command can only store the key in the traditional format. But it offers various encryptions as options.

In the following test, I tried to use:

- "openssl genrsa" to generate a RSA private key and store it in the traditional format with DER encoding, but no encryption.
- "openssl rsa" to convert the key file format to traditional with PEM encoding, but no encryption.
- "openssl rsa" to convert the key file format to traditional with DER encoding and encryption.
- "openssl rsa" to convert the key file format to traditional with PEM encoding and encryption.

My command session was recorded as blow:

```
>rem traditional format, PEM encoding, no encryption
>openssl genrsa -out openssl_key.pem 1024

Loading 'screen' into random state - done
Generating RSA private key, 1024 bit long modulus
```

```
.......++++++
........................................++++++
e is 65537 (0x10001)

>rem traditional format, DER encoding, no encryption
>openssl rsa -in openssl_key.pem -inform pem
-out openssl_key.der -outform der

writing RSA key

>rem traditional format, PEM encoding, DES encryption
>openssl rsa -in openssl_key.pem -inform pem
-out openssl_key_des.pem -outform pem -des

writing RSA key
Enter PEM pass phrase: keypass
Verifying - Enter PEM pass phrase: keypass

>rem traditional format, DER encoding, DES encryption
>openssl rsa -in openssl_key.pem -inform pem
-out openssl_key_des.der -outform der -des

writing RSA key
```

All commands were executed as expected except the last one. The traditional format with DER encoding seems not able to apply the DES encryption.

Anyway, I got my RSA private key stored in OpenSSL traditional format with 3 flavors:

```

04/01/2007  09:55 AM               608 openssl_key.der
04/01/2007  09:52 AM               887 openssl_key.pem
04/01/2007  10:01 AM               958 openssl_key_des.pem
```

Now I am ready to my private key to PKCS#8 format as described in the next section.

Last update: 2013.

"openssl pkcs8" Converting Keys to PKCS#8 Format

This section provides a tutorial example on how to convert a private key file from the traditional format into PKCS#8 format using the 'openssl pkcs8' command. Keys can still be encoded with DER or PEM with or without DES encryption in PKCS#8 format.

Once I have my private key stored in the traditional format, I can use the "openssl pkcs8" command to convert it into PKCS#8 format. My plan was to try to do the following:

- "openssl pkcs8 -topk8" to convert the key file format to PKCS#8 with PEM encoding, but no encryption.
- "openssl pkcs8 -topk8" to convert the key file format to PKCS#8 with DER encoding, but no encryption.
- "openssl pkcs8 -topk8" to convert the key file format to PKCS#8 with PEM encoding and

encryption.

- "openssl pkcs8 -topk8" to convert the key file format to PKCS#8 with DER encoding and encryption.

My command session was recorded as blow:

```
>rem PKCS#8 format, PEM encoding, no encryption
>openssl pkcs8 -topk8 -in openssl_key.pem -inform pem
-out openssl_key_pk8.pem -outform pem -nocrypt

>rem PKCS#8 format, DER encoding, no encryption
>openssl pkcs8 -topk8 -in openssl_key.pem -inform pem
-out openssl_key_pk8.der -outform der -nocrypt

>rem PKCS#8 format, PEM encoding, encrypted
>openssl pkcs8 -topk8 -in openssl_key.pem -inform pem
-out openssl_key_pk8_enc.pem -outform pem

Enter Encryption Password: keypass
Verifying - Enter Encryption Password: keypass
Loading 'screen' into random state - done

>rem PKCS#8 format, DER encoding, encrypted
>openssl pkcs8 -topk8 -in openssl_key.pem -inform pem
-out openssl_key_pk8_enc.der -outform der

Enter Encryption Password: keypass
Verifying - Enter Encryption Password: keypass
Loading 'screen' into random state - done
```

All commands executed as expected this time. I got my RSA private key stored in OpenSSL traditional format and PKCS#8 format in 7 flavors:

```
04/01/2007  09:55 AM               608 openssl_key.der
04/01/2007  09:52 AM               887 openssl_key.pem
04/01/2007  10:01 AM               958 openssl_key_des.pem
04/01/2007  10:29 AM               634 openssl_key_pk8.der
04/01/2007  10:28 AM               916 openssl_key_pk8.pem
04/01/2007  11:53 AM               677 openssl_key_pk8_enc.der
04/01/2007  10:29 AM               993 openssl_key_pk8_enc.pem
```

Now the question is how to verify them? Looks like there no easy tool to do this. I will leave this task later by writing a Java program to verify them.

Last update: 2013.

"openssl pkcs12" Merging Key with Certificate

This section provides a tutorial example on how to merge a private key and its self-signed certificate into a single PKCS#12 file, with can be then encoded as PEM and encrypted with DES.

PKCS#12 (Personal Information Exchange Syntax Standard) defines how a private key and its related certificates should be stored in single file. In this section, I want to try the following:

- Use "openssl reg -new -x509" command to create a self-signed certificate with my private key.
- Use "openssl pkcs12 -export" command to merge my private key and my certificate into a PKCS#12 file.
- Use "openssl pkcs12" command to parse a PKCS#12 file into an encrypted PEM file.

My command session was recorded as blow:

```
>rem self-signed certificate in X509 format, PEM encoding
>openssl req -new -x509 -key openssl_key.pem -keyform pem
-out openssl_crt.pem -outform pem -config openssl.cnf

You are about to be asked to enter information that will be
incorporated into your certificate request.
What you are about to enter is what is called a Distinguished
Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [CA]:
State or Province Name (full name) [HY State]:
Locality Name (eg, city) [HY City]:
Organization Name (eg, company) [HY Company]:
Organizational Unit Name (eg, section) [HY Unit]:
Common Name (eg, YOUR name) [Herong Yang]:
Email Address [herongyang.com]:

>rem key and certificate merged in PKCS#12 format
>openssl pkcs12 -export -inkey openssl_key.pem -in openssl_crt.pem
-out openssl_key_crt.p12 -name openssl_key_crt

Loading 'screen' into random state - done
Enter Export Password: p12pass
Verifying - Enter Export Password:

>rem encrypt the PKCS#12 file
>openssl pkcs12 -in openssl_key_crt.p12 -out openssl_key_crt_enc.pem

Enter Import Password: p12pass
MAC verified OK
Enter PEM pass phrase: keypass
Verifying - Enter PEM pass phrase: keypass
```

Notes on the commands and options I used:

- "openssl req -new -x509" command generates a self-signed certificate based on the given private and public key pair.
- "openssl pkcs12 -export" command merges the private and public key pair with its self-signed certificate into a PKCS#12 file.

- "-inkey openssl_key.pem" option specifies the private and public key pair in PEM encoded file.
- "-in openssl_crt.pem" option specifies the self-signed certificate in PEM encoded file.
- "-out openssl_key_crt.p12" option specifies the output PKCS#12 file name.
- "-name openssl_key_crt" option specifies a name for the key pair and the certificate in the PKCS#12 file.
- "openssl pkcs12" command without "-export" option parses a PKCS#12 file as input.

The result is very nice. My private key and my self-signed certificate are stored in single files now:

- openssl_key_crt.p12 - PKCS#12 file, encrypted, binary form.
- openssl_key_crt_enc.pem - PEM encoded and encrypted private key and PEM encoded certificate in one file.

Want to see the file structure of openssl_key_crt_enc.pem? Here it is:

```
>type openssl_key_crt_enc.pem

Bag Attributes
    localKeyID: B5 BA 41 DE E6 FE 22 70 D7 C8 C8 55 76 E6 AF 92 6B...
subject=/C=CA/ST=HY State/L=HY City/O=HY Company/OU=HY Unit/CN=Her...
issuer=/C=CA/ST=HY State/L=HY City/O=HY Company/OU=HY Unit/CN=Hero...
-----BEGIN CERTIFICATE-----
MIIDgzCCAuygAwIBAgIBADANBgkqhkiG9w0BAQQFADCBjjELMAkGA1UEBhMCQ0Ex
...
joy2xMaAryTrfoyUyqL10TusG3MeoXnHl4u4F5mLbQgr13CYHjdp
-----END CERTIFICATE-----
Bag Attributes
    localKeyID: B5 BA 41 DE E6 FE 22 70 D7 C8 C8 55 76 E6 AF 92 6B...
Key Attributes: <No Attributes>
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,5845E016B16C7803

xo6pJ9madEbOB9SAQgIGC3GeZ7xDqHZJm6RkquOju23dSxzzetR2u/PPtnQ82hK0
...
7DSeQRZg3a1TTwQXwYXCqHdc2qLzISH/C4ERqm7EqJ2PCsEe7GSfmA==
-----END RSA PRIVATE KEY-----
```

openssl_key_crt_enc.pem looks like a concatenated file of the key PEM file and certificate PEM file.

Now I have the final PKCS#12 file with my private key and certificate. I can verify it with Java SE "keytool" command as described in the next section.

Last update: 2013.

"keytool -list" Verifying PKCS#12 Files

This section provides a tutorial example on how to merge a private key and its self-signed certificate into a single PKCS#12 file, with can be then encoded as PEM and encrypted with DES.

Since Java SE "keytool" command support PKCS#12 files, I want to try it with my PKCS#12 file, openssl_key_crt.p12, created by "OpenSSL" with the following tests:

- Use "keytool -list" command to display what's in the PKCS#12 file.
- "keytool -exportcert" command only exports the self-signed certificate from a PrivateKeyEntry in a keystore.

My command session was recorded as blow:

```
>keytool -list -keystore openssl_key_crt.p12 -storetype pkcs12
-storepass p12pass

Keystore type: PKCS12
Keystore provider: SunJSSE

Your keystore contains 1 entry

openssl_key_crt, Jul 29, 2007, PrivateKeyEntry,
Certificate fingerprint (MD5): 1D:D4:AC:96:53:25:9F:1A:D0:A7:46:6C...

>keytool -exportcert -keystore openssl_key_crt.p12 -storetype pkcs12
-storepass p12pass -alias openssl_key_crt
-file keytool_openssl_crt.pem -rfc

Certificate stored in file <keytool_openssl_crt.pem>
```

Notes on the commands and options I used:

- "keytool -list" command lists what's in the keystore file.
- "-keystore openssl_key_crt.p12" option specifies the keystore file, a PKCS#12 file generated by "OpenSSL".
- "-storetype pkcs12" option specifies the type of the keystore file, "jks" or "pkcs12".
- "-storepass p12pass" option specifies the password to open the keystore file.
- "keytool -exportcert" command exports the self-signed certificate out of the keystore file.
- "-rfc" option tells keytool to write the certificate file with PEM (RFC1421) encoding.

The tests were successful and helped me to learn that:

- The PKCS#12 file generated by "OpenSSL" does meet the PKCS#12 standard.

- "OpenSSL" and "keytool" can share keystore files in PKCS#12 format.

As an exercise, you can open "openssl_crt.pem" and "keytool_openssl_crt.pem". They should contain the same Base64 encoded strings.

In the next section, I want to try to convert the PKCS#12 file to a JKS (Java KeyStore) file.

Last update: 2013.

"keytool -importkeystore" Importing PKCS#12 Files

This section provides a tutorial example on how to import a private key stored in a PKCS#12 file into a JKS (Java KeyStore) file with the 'keytool -importkeystore' command.

Since Java uses JKS (Java KeyStore) as the keystore file type, I want to try to convert my PKCS#12 file, openssl_key_crt.p12, to a JKS file with the "keystore -importkeystore" command:

```
>keytool -importkeystore -srckeystore openssl_key_crt.p12
-srcstoretype pkcs12 -srcstorepass p12pass -srcalias openssl_key_crt
-destkeystore openssl_key_crt.jks -deststoretype jks
-deststorepass jkspass

>keytool -list -keystore openssl_key_crt.jks -storetype jks
-storepass jkspass

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry

openssl_key_crt, Jul 29, 2007, PrivateKeyEntry,
Certificate fingerprint (MD5): 1D:D4:AC:96:53:25:9F:1A:D0:A7:46:6C...
```

There was no trouble at all. "keytool -importkeystore" command has a lots of options. But they are very easy to understand.

Last update: 2013.

Summary - Migrating "OpenSSL" Keys to "keystore"

This section describes high level steps on how to migrate a private key generated by 'OpenSSL' into a JKS (Java KeyStore) file. The key step is to merge the private key with its self-signed certificate into a PKCS#12 file.

As a summary, I want offer some notes here about migrating private keys from "OpenSSL" files to "keytool" keystore files:

There is no easy way to just migrate the private keys from "OpenSSL" key files directly to "keytool" keystore files.

"openssl pkcs12 -export" command should be used to combine the private key file and the self-signed certificate file in a PKCS#12 file.

"keytool" can use the PKCS#12 file directly with the "-storetype pkcs12" open.

"keytool -importkeystore" command should be used to convert the PKCS#12 file into a JKS (Java KeyStore) file.

Last update: 2013.

Summary - Migrating "keystore" Keys to "OpenSSL"

This section describes high level steps on how to migrate a private key generated in a JKS (Java KeyStore) file to an 'OpenSSL' key file. The key step is to convert a JKS file into a PKCS#12 file with 'keytool'.

Once we know that "keytool" supports PKCS#12 files, we can also use PKCS#12 files to migrate private keys from "keytool" keystore files "OpenSSL" key files. Here are my notes on how to do this:

"keytool -importkeystore" command should be used to convert a JKS (Java KeyStore) file into a PKCS#12 file.

"openssl pkcs12" command should be used to split the private key file out of the PKCS#12 file.

"openssl pkcs12" command should be used to split the certificate file out of the PKCS#12 file.

If you are tired of using PKCS#12 files, of course you can use my "DumpKey.java" program to dump the private key out of a JKS file and use it directly with OpenSSL. See the previous chapter for more information on DumpKey.java.

Last update: 2013.

Using Certificates in IE (Internet Explorer)

This chapter provides tutorial notes and example codes on using certificates in IE 9. Topics include why Web browsers need certificates; viewing the certificate from an https Web server; installing the certificate from a server; exporting a certificate from IE 9 to a certificate file; importing a certificate into IE 9.

Conclusions:

- Web browsers need certificate from Web servers, if they are using https (SSL protocol) to encrypt messages to secure the communication.
- IE 9 allows you to view server's certificate through the lock icon.
- IE 9 allows you to install server's certificate into certificate stores.
- IE 9 allows you to export and import certificates in certificate files in DER and PEM formats.

Why Using Certificates with Web Browsers?

This section describes why Web browsers needs to use certificates - A SSL enabled Web server requires your Web browser to use certificates to encrypt messages between the browser and the server.

In previous parts of the book, we have learned how to generate certificates. Now let's see how we can use certificates with Web browsers.

The first question we need to ask is why do we need use certificates with a Web browser? The answer is that many Web sites supports SSL (Secure Socket Layer), which encrypts every message sent from and received by the Web browser. Encryption added by SSL makes your communication with the Web server secure, because no systems other than your browser and the final server can understand those encrypted messages.

When a Web browser reaches a SSL enabled Web site (URL starts with https://), the server will send a certificate, called server certificate, the Web browser. The subject of the server certificate represents the server. The browser is expected to trust the server certificate, or validate it with a trusted certification path.

A SSL enabled Web site may also ask the Web browser to send back a certificate, called client certificate, to identify the client, so that the server can validate the client.

So a Web browser needs to have certificates for two purposes:

- To form a certification path to validate the server certificate.
- To identify client, if the server wants to do client certification.

In next sections, we will look at how two popular Web browsers, IE (Internet Explorer) and Firefox uses certificates.

Last update: 2013.

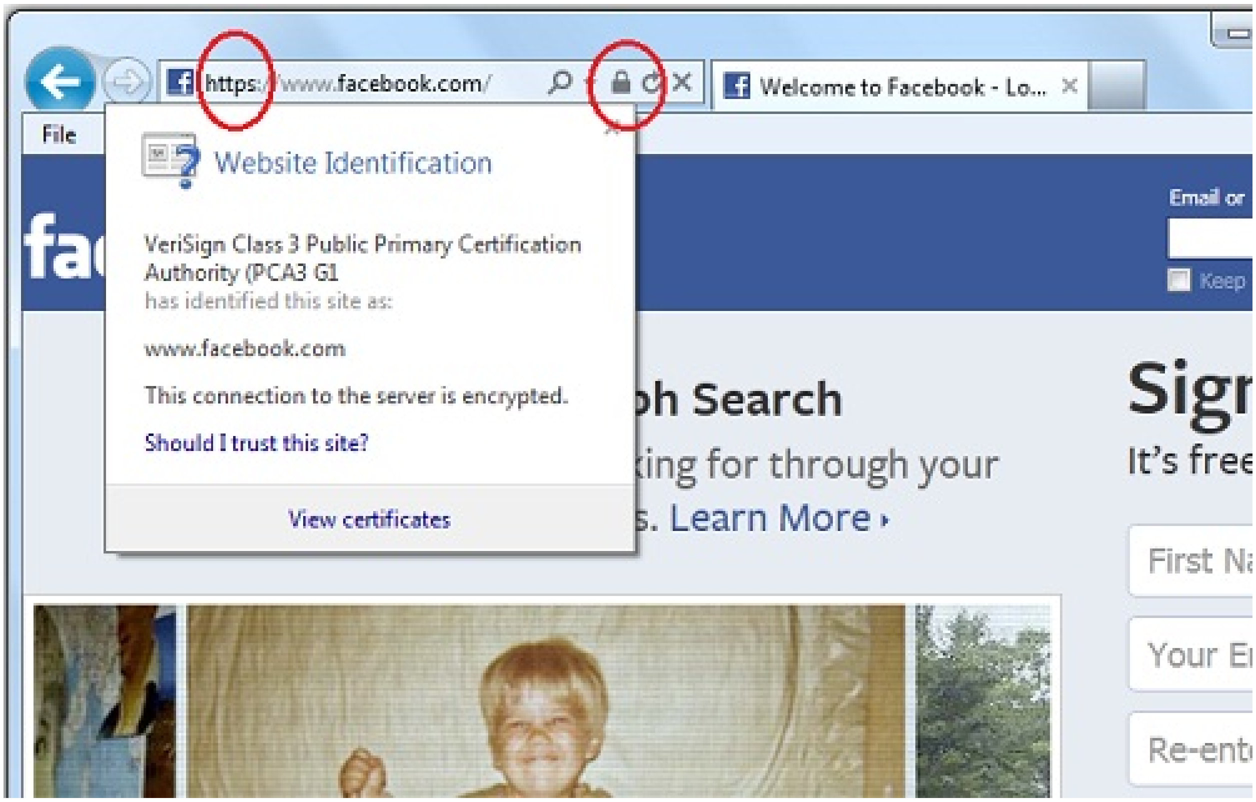
Visiting a "https" Web Site with IE

This section describes how IE (Internet Explorer) 9 shows a lock icon when you visit an 'https' Web site to provide you more security related information.

As I mentioned in the previous section, if a Web site wants to encrypt communication messages with your browser, it will enable the SSL protocol and use "https" as part of their Web address.

If you go to a Web site that provides online services, I am sure that you will see "https" in the Web address field starting on the log in page. This indicates that Web site uses SSL protocol to encrypt all information you send and receive on this server.

If you are using IE (Internet Explorer) 9, it will display an extra lock icon next to the Web address field when you are connecting to an "https" Web site. The picture below shows the "https" Web address and the lock icon when you use IE 9 to visit the Facebook log in page: "https://www.facebook.com":



Website Identification

VeriSign Class 3 Public Primary Certification Authority (PCA3 G1
has identified this site as:

www.facebook.com

This connection to the server is encrypted.

Should I trust this site?

If you click the lock icon, IE will provide you a summary of security related information about this Web site. For https://www.facebook.com Web site, you will get something like:

```
VeriSign Class 3 Public Primary Certification Authority (PCA3 G1
has identified this site as: www.facebook.com
This connection to the server is encrypted.

View certificate
```

More security related information will be provided if you click the "View certificates" button. See next sections.

Last update: 2013.

Viewing Certificate Details

This section provides a tutorial example on how to view certificate details when visiting an 'https' Web site in IE 9.

When you visit a "https" Web server, it will send its certificate to your browser. Server's certificate is needed by the browser for these 2 tasks:

- Your browser must validate the certificate to determine that the Web site can be trusted or not before doing any further communications.

- Your browser must use the public key in the certificate to help secure the communication messages sent and received.

Normally, your browser will do these 2 tasks automatically without your interaction. You don't need to know where is the server certificate and what's in the certificate.

But since I am interested to learn more about "https" communication, I want to see the server certificate. Here is what did on IE 9 to see details of the server certificate.

1. Run IE 9 and go to https://login.yahoo.com and wait for the log in page to be loaded.

2. Click the lock icon at the end of the Web address field. A small pop up windows shows up.

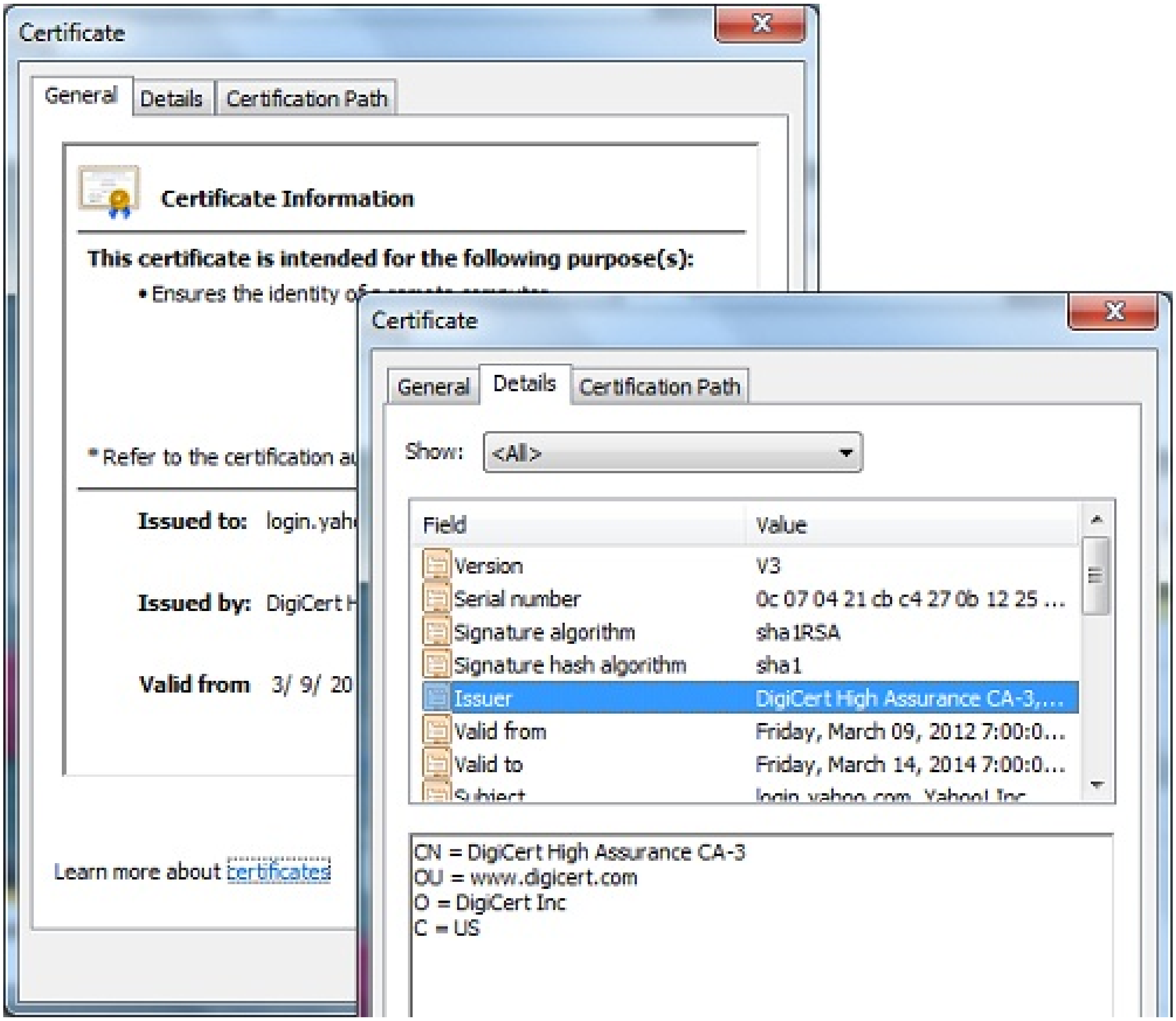
3. Click the "View certificates" link on the pop up window. The Certificate dialog box shows up. The General tab tells me this information:

```
This certificate is intended for the following purpose(s):
- Ensures the identify of a remote computer

Issued to: login.yahoo.com
Issued by: DigiCert High Assurance CA-3
Valid from 3/ 9/ 2012 to 3/ 14/ 2014
```

4. If you click the Details tab, you will details of this certificate:

```
Version                 V3
Serial number           0c 07 04 ...
Signature algorithm     sha1RSA
Signature has algorithm sha1
Issuer                  DigiCert High Assurance CA-3
Valid from              Friday, March 09, 2012 07:00:00 PM
Valid to                Friday, March 14, 2014 07:00:00 PM
Subject                 login.yahoo.com, Yahoo! Inc.
Public key              RSA (2048 Bits)
Authority Key Identifier KeyID=50 ea 73 ...
Subject Key Identifier     2a aa f2 ...
Enhanced Key Usage      Server Authentication (1.3.6.1.5.5.7.3.1)
                        Client Authentication (1.3.6.1.5.5.7.3.2)
Key Usage               Digital Signature, Key Encipherment (a0)
Thumbprint algorithm    sha1
Thumbprint              68 22 14 ...
```

Cool. Now I see details of a real certificate for commercial uses. The picture below shows you steps to reach certificate details:



Certificate Information

This certificate is intended for the following purpose(s):

- Ensures the identity of

* Refer to the certification a

Issued to: login.yah

Issued by: DigiCert H

Valid from 3/ 9/ 20

Learn more about certificates

Show: <All>

Field	Value
Version	V3
Serial number	0c 07 04 21 cb c4 27 0b 12 25 ...
Signature algorithm	sha1RSA
Signature hash algorithm	sha1
Issuer	DigiCert High Assurance CA-3,...
Valid from	Friday, March 09, 2012 7:00:0...
Valid to	Friday, March 14, 2014 7:00:0...
Subject	login.yahoo.com, Yahoo! Inc...

CN = DigiCert High Assurance CA-3
OU = www.digicert.com
O = DigiCert Inc
C = US

Last update: 2013.

Viewing Certificate Path

This section provides a tutorial example on how to view certificate path when visiting an 'https' Web site in IE 9. The top certificate in a certificate path is the CA certificate, which is trusted automatically.

As we learned from previous chapters, in order to validate a certificate, a certificate path must be established to link the certificate to a CA (Certificate Authority) certificate.

Here is what I did to find the certificate path for https://login.yahoo.com Web site.

1. Run IE 9 and go to https://login.yahoo.com and wait for the log in page to be loaded.

2. Click the lock icon at the end of the Web address field. A small pop up windows shows up.

3. Click the "View certificates" link on the pop up window. The Certificate dialog box shows up.

4. Click the "Certificate Path" tab. A certificate path with 4 certificates shows up:

```
GTE CyberTrust Global Root
 |- DigiCert High Assurance EV Root CA
     |- DigiCert High Assurance CA-3
         |- login.yahoo.com
```

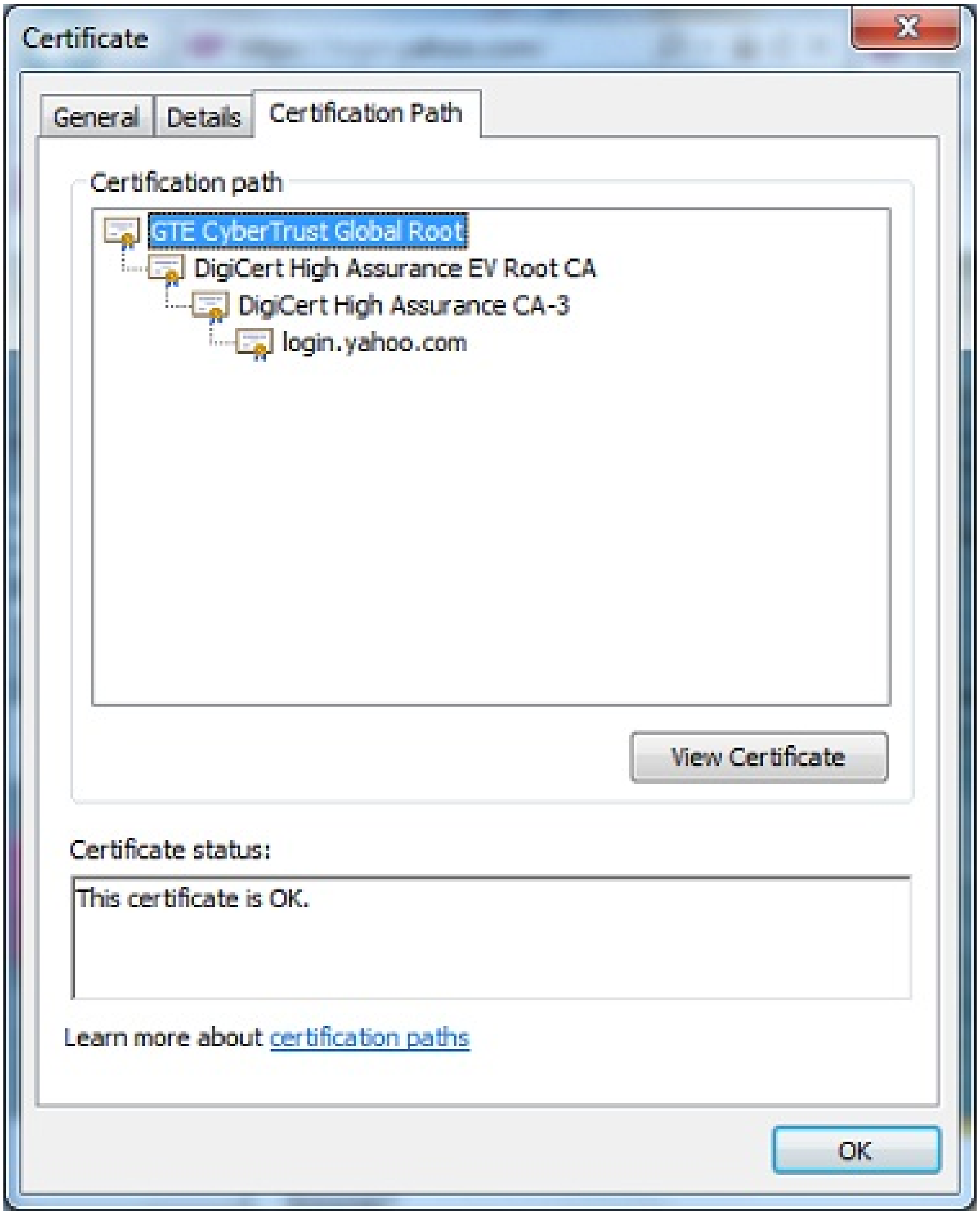
5. Double click on "GTE CyberTrust Global Root". Another Certificate dialog box shows up providing information on the GTE CyberTrust certificate, which is also the CA certificate, because it is the top certificate in the certificate path.

```
This certificate is intended for the following purpose(s):
- Ensures the identify of a remote computer

Issued to: GTE CyberTrust Global Root
Issued by: GTE CyberTrust Global Root
Valid from 8/ 12/ 1998 to 8/ 13/ 2018
```

It is interesting to see that:

- This certificate is valid for 20 years! Can we trust this CA for that long time?
- The CA certificate is a self-signed certificate.

The picture below shows you the certificate path of login.yahoo.com:



GTE CyberTrust Global Root
DigiCert High Assurance EV Root CA
DigiCert High Assurance CA-3
login.yahoo.com

Certificate status:

This certificate is OK.

Learn more about certification paths

Last update: 2013.

Installing Certificate Permanently in IE

This section provides a tutorial example on how to install the certificate provided by an 'https' Web site in IE 9. The certificate can be installed into a certificate store automatically selected by IE.

Normally, when a browser visits a "https" Web server, it will get the certificate from the server and use it for one web session only. When you close the browser, the certificate will be removed from memory. It will not be stored on computer permanently.

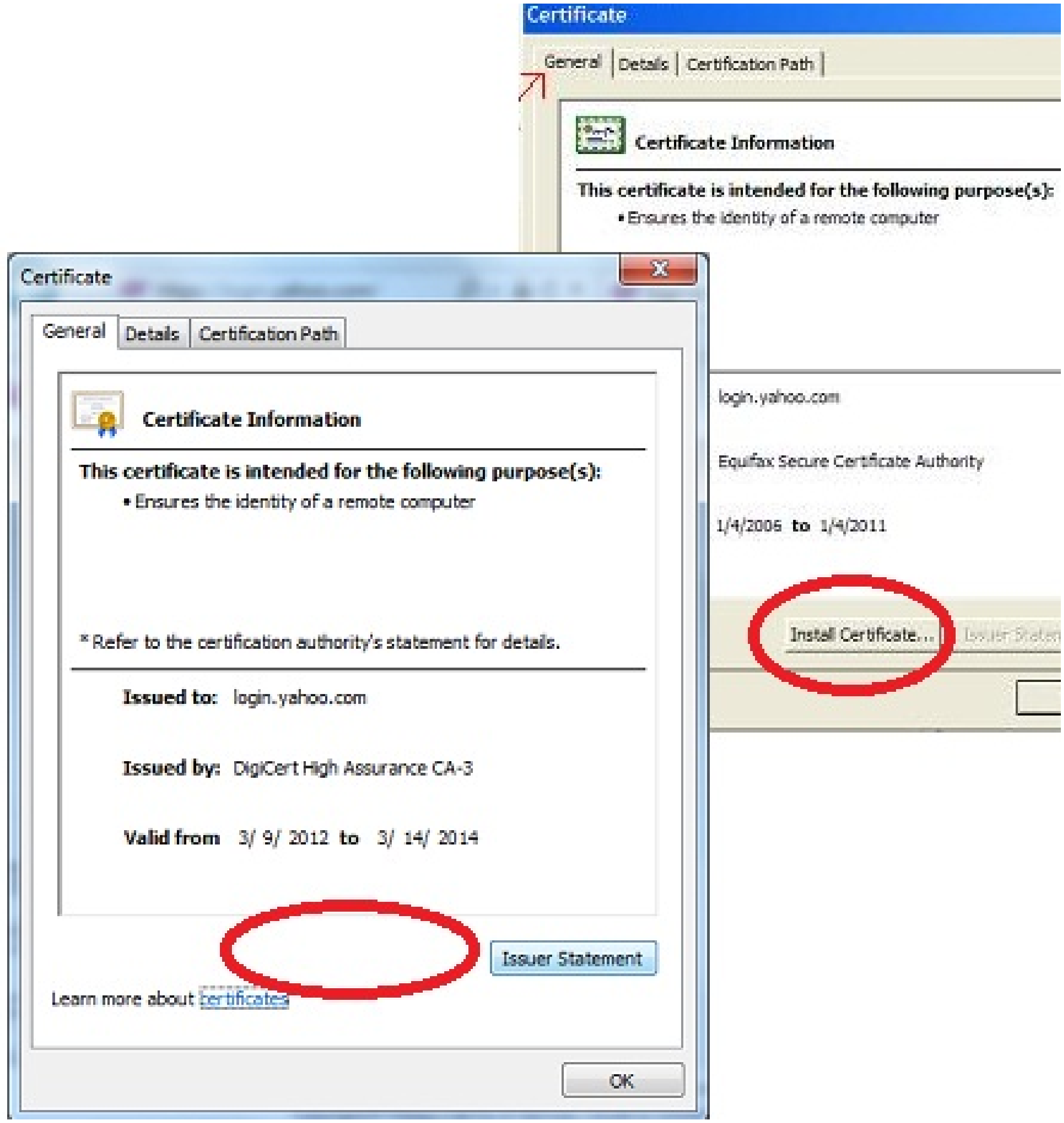
However, if you want to keep a copy of the certificate on the computer, you can install it in steps shown below:

1. Run IE 9 and go to https://login.yahoo.com and wait for the log in page to be loaded.

2. Click the lock icon at the end of the Web address field. A small pop up windows shows up.

3. Click the "View certificates" link on the pop up window. The Certificate dialog box shows up.

But the "Install Certificate" button is not supported any more! I cannot install this certificate into the certificate store on my local computer in IE 9. See the comparison between IE 9 and IE 7 below:



Certificate Information

This certificate is intended for the following purpose(s):

- Ensures the identity of a remote computer

login.yahoo.com

Equifax Secure Certificate Authority

1/4/2006 to 1/4/2011

Certificate Information

This certificate is intended for the following purpose(s):

- Ensures the identity of a remote computer

* Refer to the certification authority's statement for details.

Issued to: login.yahoo.com

Issued by: DigiCert High Assurance CA-3

Valid from 3/ 9/ 2012 to 3/ 14/ 2014

So are we losing a nice feature in IE 9? The answer is not really. This "Install Certificate" button is supported only in the "Administrator" mode now.

4. Ok. Close IE browser. Then right-click the IE icon and select "Run as administrator". Repeat all steps mentioned above and click the "Install Certificate" button. The Certificate Import Wizard starts.

5. Click the "Next" button. The Certificate Store step shows up with these fields:

```
Certificate stores are system areas where certificates are kept.
```

```
Windows can automatically select a certificate store, or you can
specify a location for
   (.) Automatically select the certificate store based on the type
       of certificate
   ( ) Place all certificates in the following store
```

6. Let's use the "Automatically select..." option and click the "Next" button. The confirmation step shows up.

7. Click the "Finish" button. The certificate will be installed into a certificate store somewhere on your computer. We will try to find it back in the next tutorial.

Last update: 2013.

Managing Certificates in Certificate Stores

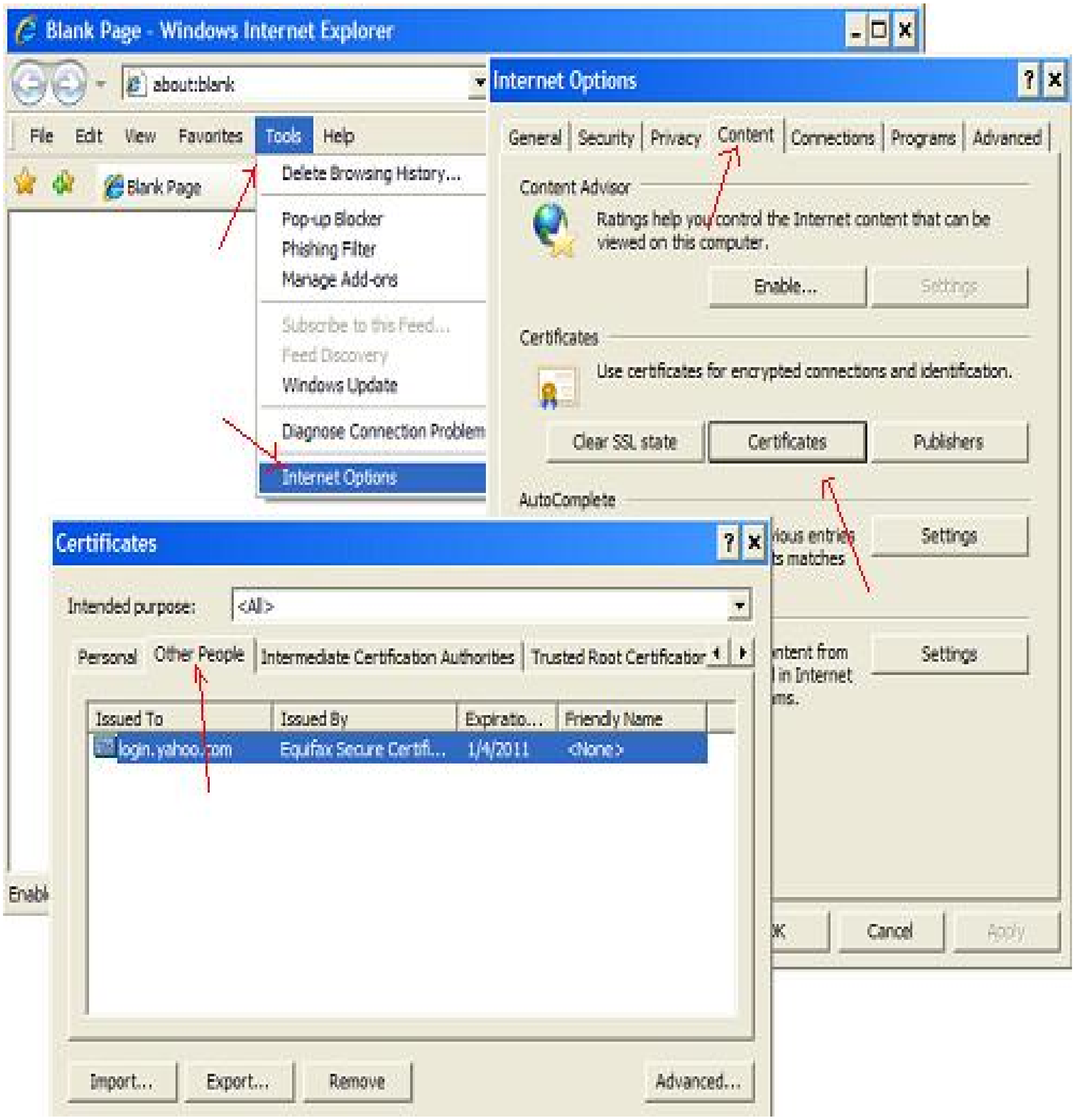
This section provides a tutorial example on how to use IE 9 to manage certificates installed in certificate stores.

In the previous tutorial, I let IE 9 installed the login.yahoo.com certificate into a certificate store automatically selected by IE 9. Now I want to know how to use IE 9 to manage certificates installed in certificate stores.

1. Run IE 9 in administrator mode and click the "Options" > "Internet Options" menu. The Internet Options dialog box shows up.

2. Click the "Content" tab and the "Certificates" button. The Certificates dialog box shows up.

3. Click the "Other People" tab. The login.yahoo.com certificate shows up in the list:


Blank Page - Windows Internet Explorer
about:blank
File Edit View Favorites Tools Help
Blank Page
Delete Browsing History...
Pop-up Blocker
Phishing Filter
Manage Add-ons
Subscribe to this Feed...
Feed Discovery
Windows Update
Diagnose Connection Problem
Internet Options
Internet Options
General Security Privacy Content Connections Programs Advanced
Content Advisor
Ratings help you control the Internet content that can be viewed on this computer.
Enable... Settings
Certificates
Use certificates for encrypted connections and identification.
Clear SSL state Certificates Publishers
AutoComplete
Settings
Settings
OK Cancel Apply
Certificates
Intended purpose: <All>
Personal Other People Intermediate Certification Authorities Trusted Root Certification
Import... Export... Remove Advanced...

Issued To	Issued By	Expiratio...	Friendly Name
login.yahoo.com	Equifax Secure Certifi...	1/4/2011	<None>

Now we know that IE 9 installed login.yahoo.com certificate in the "Other People" certificate store. If you click other tabs, you will see other certificates in different certificate stores:

- Personal - For my own certificates.
- Other People - For other people's certificates.
- Intermediate Certification Authorities - For certificates from non-root CAs.

- Trusted Root Certification Authorities - For certificates from root CAs.
- Trusted Publishers - For trusted self-signed certificates.
- Untrusted Publishers - For untrusted self-signed certificates.

Last update: 2013.

Exporting Certificates Out of IE

This section provides a tutorial example on how to use IE to export certificates installed in certificate stores into certificate files in DER and PEM formats.

In previous tutorial, we learned how to use IE 9 to access certificates installed in certificate stores. Now let's see if we can export a certificate out of a certificate store into a certificate file.

1. Run IE 9 and click the "Options" > "Internet Options" menu. The Internet Options dialog box shows up.

2. Click the "Content" tab and the "Certificates" button. The Certificates dialog box shows up.

3. Click "Trusted Root Certification Authorities" tab. You should see a list of certificates that come with the IE installation. All certificates listed on this tab are automatically trusted.

4. Select "VeriSign Trust Network" with expiration date of 8/1/2028, and click the "Export" button. The Certificate Export Wizard shows up.

5. Click the "Next" button. The Export File Format step shows up with these options:

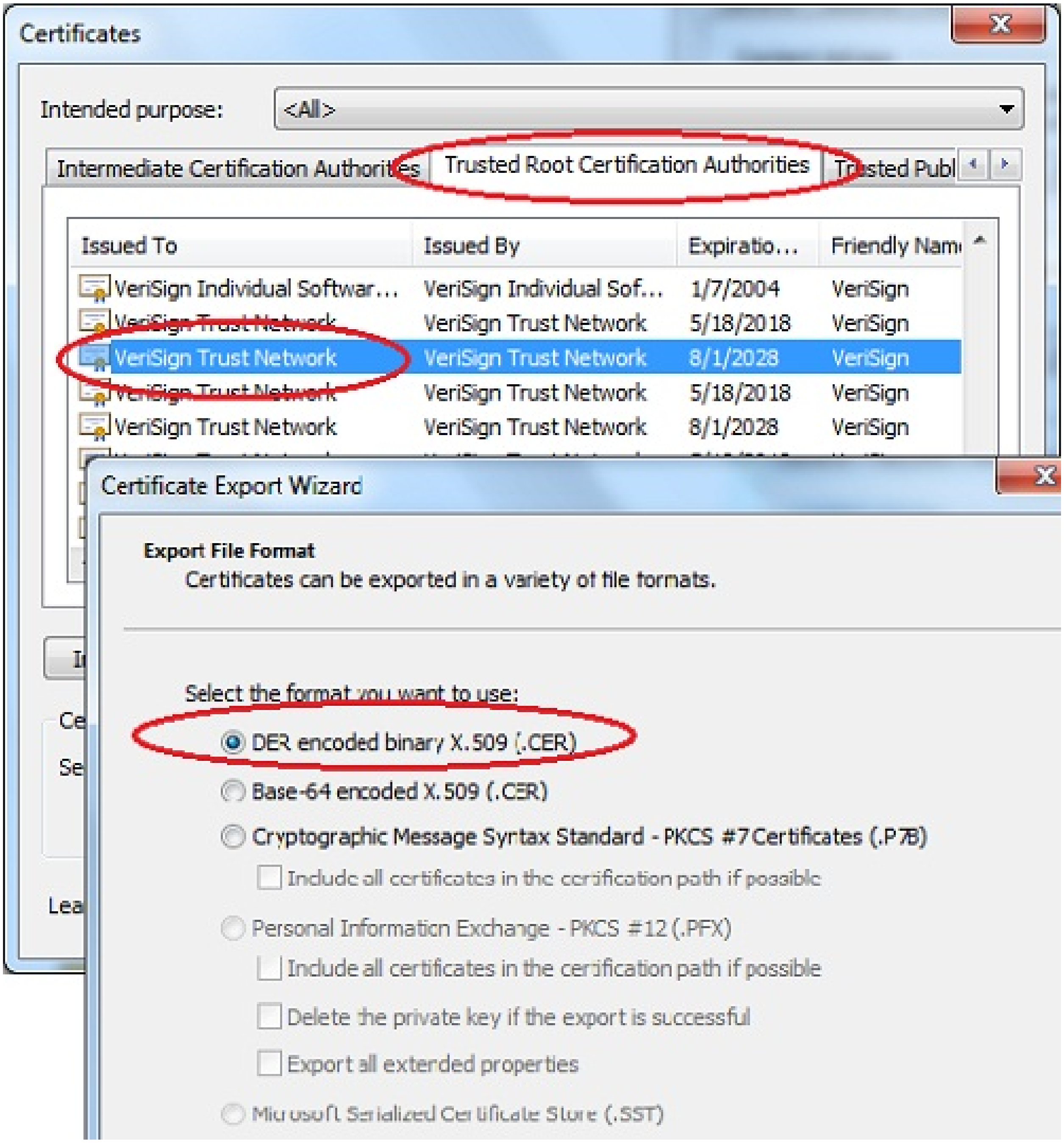
```
Certificate can be exported in a variety of file formats.

Select the format you want to use:
   (.) DER encoded binary X.509 (.CER)
   ( ) Base-64 encoded X.509 (.CER)
   ( ) Cryptographic Message Syntax Standard
       - PKCS #7 Certificates (.P7B)
```

6. Select "DER encoded binary X.509 (.CER)" as the export file format, and click the "Next" button. The File to Export step shows up.

7. Enter a file name: \verisign1.cer, and click the "Next" button. The confirmation step shows up.

8. Click the "Finish" button. The selected certificate will be exported into the specified file.

The picture below shows you some steps of the exporting process:


Certificates
Intermediate Certification Authorities
Trusted Root Certification Authorities
Trusted Publ
Certificate Export Wizard

Intended purpose: <All>

Issued To	Issued By	Expiratio...	Friendly Name
VeriSign Individual Softwar...	VeriSign Individual Sof...	1/7/2004	VeriSign
VeriSign Trust Network	VeriSign Trust Network	5/18/2018	VeriSign
VeriSign Trust Network	VeriSign Trust Network	8/1/2028	VeriSign
VeriSign Trust Network	VeriSign Trust Network	5/18/2018	VeriSign
VeriSign Trust Network	VeriSign Trust Network	8/1/2028	VeriSign

Export File Format

Certificates can be exported in a variety of file formats.

Select the format you want to use:

● DER encoded binary X.509 (.CER)

○ Base-64 encoded X.509 (.CER)

○ Cryptographic Message Syntax Standard - PKCS #7 Certificates (.P7B)

□ Include all certificates in the certification path if possible

○ Personal Information Exchange - PKCS #12 (.PFX)

□ Include all certificates in the certification path if possible

□ Delete the private key if the export is successful

□ Export all extended properties

○ Microsoft Serialized Certificate Store (.SST)

In the next tutorial, we will view the contents of the certificate from VeriSign Trust Network stored in a DER file, \verisign1.cer.

Last update: 2013.

OpenSSL Viewing Certificates Exported from IE

This section provides a tutorial example on how to use OpenSSL to view contents of a certificate file exported from IE 9.

In the previous tutorial, we successfully exported the certificate from VeriSign Trust Network to a DER file, \verisign1.cer. Now I want to view it's contents with OpenSSL.

```
>openssl x509 -in \verisign1.cer -inform DER -noout -text
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number:
            b9:2f:60:cc:88:9f:a1:7a:46:09:b8:5b:70:6c:8a:af
        Signature Algorithm: sha1WithRSAEncryption

        Issuer: C=US, O=VeriSign, Inc., OU=Class 3 Public Primary Cer
tification Authority - G2, OU=(c) 1998 VeriSign, Inc. - For authorize
d use only, OU=VeriSign Trust Network
        Validity
            Not Before: May 18 00:00:00 1998 GMT
            Not After : Aug  1 23:59:59 2028 GMT
        Subject: C=US, O=VeriSign, Inc., OU=Class 3 Public Primary Cer
tification Authority - G2, OU=(c) 1998 VeriSign, Inc. - For authorized
 use only, OU=VeriSign Trust Network
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:a7:88:01:21:74:2c:e7:1a:03:f0:98:e1:97:3c:
                    0f:21:08:f1:9c:db:97:e9:9a:fc:c2:04:06:13:be:
                    5f:52:c8:cc:1e:2c:12:56:2c:b8:01:69:2c:cc:99:
                    1f:ad:b0:96:ae:79:04:f2:13:39:c1:7b:98:ba:08:
                    2c:e8:c2:84:13:2c:aa:69:e9:09:f4:c7:a9:02:a4:
                    42:c2:23:4f:4a:d8:f0:0e:a2:fb:31:6c:c9:e6:6f:
                    99:27:07:f5:e6:f4:4c:78:9e:6d:eb:46:86:fa:b9:
                    86:c9:54:f2:b2:c4:af:d4:46:1c:5a:c9:15:30:ff:
                    0d:6c:f5:2d:0e:6d:ce:7f:77
                Exponent: 65537 (0x10001)
    Signature Algorithm: sha1WithRSAEncryption
        72:2e:f9:7f:d1:f1:71:fb:c4:9e:f6:c5:5e:51:8a:40:98:b8:
        68:f8:9b:1c:83:d8:e2:9d:bd:ff:ed:a1:e6:66:ea:2f:09:f4:
        ca:d7:ea:a5:2b:95:f6:24:60:86:4d:44:2e:83:a5:c4:2d:a0:
        d3:ae:78:69:6f:72:da:6c:ae:08:f0:63:92:37:e6:bb:c4:30:
        17:ad:77:cc:49:35:aa:cf:d8:8f:d1:be:b7:18:96:47:73:6a:
        54:22:34:64:2d:b6:16:9b:59:5b:b4:51:59:3a:b3:0b:14:f4:
        12:df:67:a0:f4:ad:32:64:5e:b1:46:72:27:8c:12:7b:c5:44:
        b4:ae
```

Very nice.

- This is a self-signed certificate from VeriSign Inc., valid until year 2028.
- The certificate file format is DER, not PEM. You need to use "-inform DER" with the OpenSSL command.

Of course, we can export the certificate out of IE, and save it in PEM format. Just select "Base-64 encoded X.509 (.CER)" as the export file format.

Last update: 2013.

Importing CA Certificate into IE

This section provides a tutorial example on how to use IE 9 to import a CA certificate into the 'Trusted Root Certification Authorities' certificate store.

After exporting certificates from IE 9, I want to test the certificate import function of IE 9. First I want to import my own self-signed certificate, herong.crt, generated in previous tutorials as a CA certificate into IE 9.

1. Run IE 9 and click the "Options" > "Internet Options" menu. The Internet Options dialog box shows up.

2. Click the "Content" tab and the "Certificates" button. The Certificates dialog box shows up.

3. Click the "Trusted Root Certification Authorities" tab, and click the "Import..." button. The Certificate Import Wizard shows up.

4. Click the "Next" button. The File to Import step shows up.

5. Use the "Browse" button to find and select herong.crt. Then click the "Next" button. The Certificate Store step shows up.

6. Keep the default certificate store selection: "Trusted Root Certificate Authorities", and click the "Next" button. The confirmation step shows up.

7. Click the "Finish" button. The Security Warning message shows up.

```
You are about to install a certificate from a certificate authority
(CA) claiming to represent: Herong Yang

Windows cannot validate that the certificate is actually from "Herong
Yang". You should confirm its origin by contacting "Herong Yang".
The following number will assist you in this process:

Thumbprint (sha1): 59FB31E9...

Warning: If you install this root certificate, Windows will
automatically trust any certificate issued by this CA. Installing
a certificate with an unconfirmed thumbprint is a security risk.
If you click "Yes" you acknowledge this risk.

Do you want to install this certificate?
```

8. Click the "Yes" button. My self-signed certificate will be installed as a trusted root certificate.

Last update: 2013.

Importing Certificate Path into IE

This section provides a tutorial example on how to use IE 9 to import a certificate path into certificate stores.

Next, I want to import some other certificates signed by me into IE 9 certificate stores to form a certificate path:

- herong.crt - self-signed by Herong.
- john.crt - signed by Herong.
- bill.crt - signed by John.
- tom.crt - signed by Bill.

Repeat the import process described in the previous tutorial to import john.crt, bill.crt, and tom.crt into the "Intermediate Certificate Authorities" certificate store.

Now if you view the "Tom Bush" certificate in IE 9, the Certification Path will display a certificate path of 4 certificates:

```
Herong Yang
 |- John Smith
     |- Bill White
         |- Tom Bush
```

Last update: 2013.

Using Certificates in Firefox

This chapter provides tutorial notes and example codes on using certificates in Firefox 18. Topics include why Web browsers need certificates; viewing the certificate from an https Web server; exporting a certificate from Firefox 18 to a certificate file; importing a CA certificate into Firefox 18.

Conclusions:

- Web browsers need certificate from Web servers, if they are using https (SSL protocol) to encrypt messages to secure the communication.
- Firefox 18 allows you to view server's certificate through the lock icon.
- Firefox 18 allows you to export and import certificates in certificate files in DER and PEM formats.
- I can't find a way to import a full certificate path into Firefox 18.

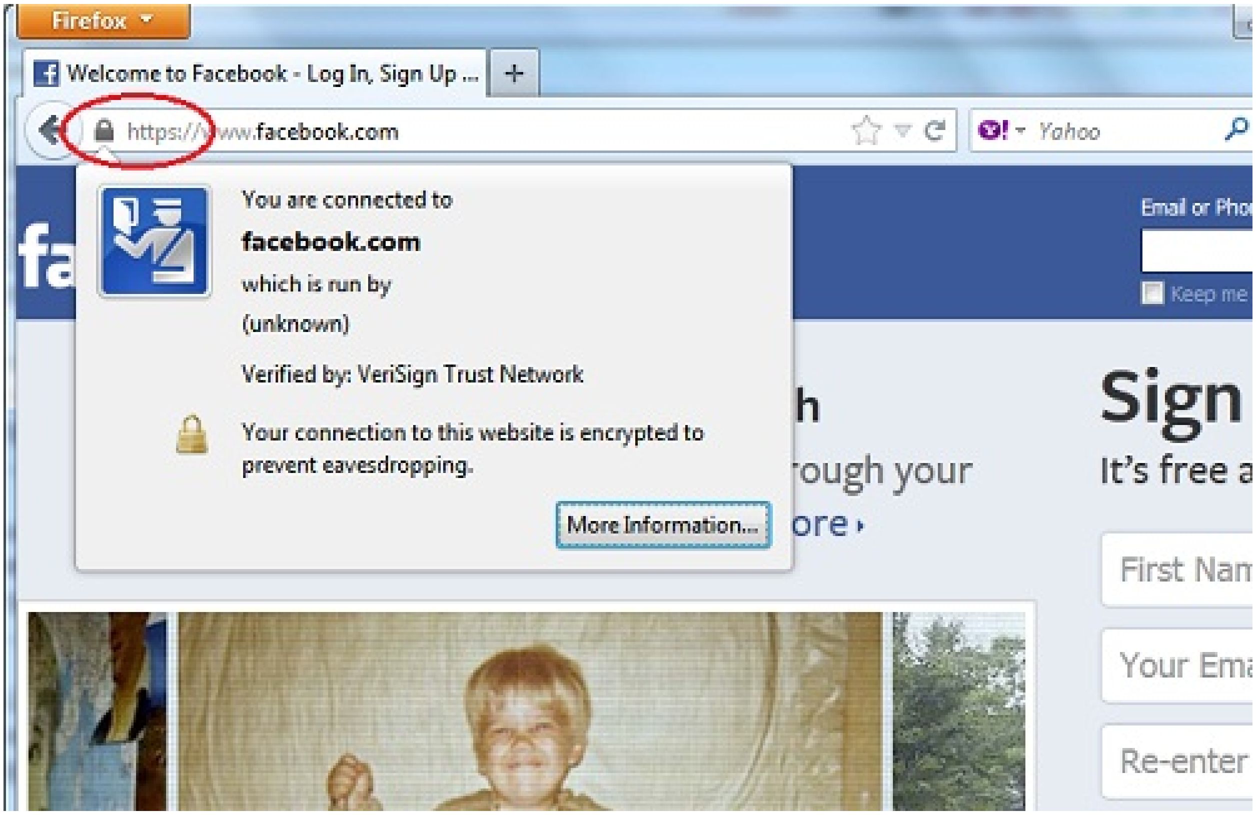
Visiting a "https" Web Site with Firefox

This section describes how Firefox 18 shows a lock icon when you visit an 'https' Web site to provide you more security related information.

As we learned in the previous chapter, if a Web site wants to encrypt communication messages with your browser, it will enable the SSL protocol and use "https" as part of their Web address.

If you go to a Web site that provides online services, I am sure that you will see "https" in the Web address field starting on the log in page. This indicates that Web site uses SSL protocol to encrypt all information you send and receive on this server.

If you are using Firefox 18, it will display an extra lock icon in front of the Web address field when you are visiting a "https" Web site. The picture below shows the "https" Web address and the lock icon when you use Firefox 18 to visit the Facebook log in page: "https://www.facebook.com":



You are connected to
facebook.com
which is run by
(unknown)

Verified by: VeriSign Trust Network

Your connection to this website is encrypted to prevent eavesdropping.

If you click the lock icon, Firefox will provide you a summary of security related information about this Web site. For https://www.facebook.com Web site, you will get something like:

```
You are connected to "facebook.com" which is run by (unknown)
Verified by: VeriSign Trust Network
Your connection to this website is encrypted to prevent eavesdropping.
[More Information...]
```

More security related information will be provided if you click the "More Information" button. See next sections.

Last update: 2013.

Viewing Certificate Details

This section provides a tutorial example on how to view certificate details when visiting an 'https' Web site in Firefox 18.

When you visit a "https" Web server, it will send its certificate to your browser. Server's certificate is needed by the browser for these 2 tasks:

- Your browser must validate the certificate to determine that the Web site can be trusted or not before doing any further communications.
- Your browser must use the public key in the certificate to help secure the communication messages sent and received.

Normally, your browser will do these 2 tasks automatically without your interaction. You don't need to know where is the server certificate and what's in the certificate.

But since I am interested to learn more about "https" communication, I want to see the server certificate. Here is what did on Firefox 18 to see details of the server certificate.

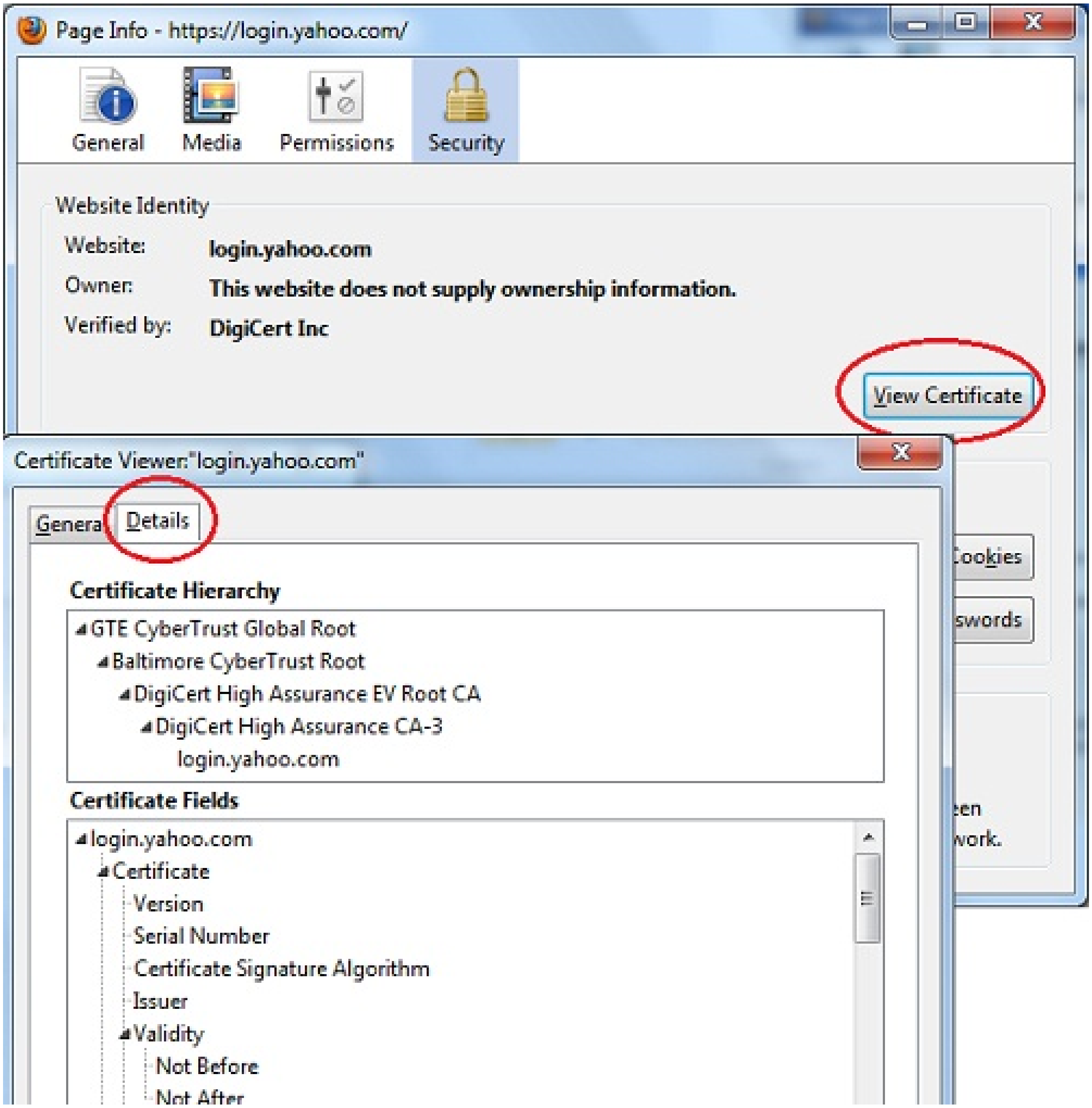
1. Run Firefox 18 and go to https://login.yahoo.com and wait for the log in page to be loaded.

2. Click the lock icon in front of the Web address. The Page Info dialog box shows up with the Security tab selected.

3. Click the "View Certificate" button. The Certificate Viewer dialog box shows up. The General tab tells me this information:

```
This certificate has been verified for the following uses:
 SSL Server Certificate

Issued to
Common Name (CN)   login.yahoo.com
Organization (O)   Yahoo! Inc.
...

Issued by
Common Name (CN)   DigiCert Assurance CA-3
Organization (O)   DigiCert Inc
...

Validity
Issued On          1/9/2012
Expires On         3/14/2014

Fingerprints
SHA1 Fingerprint   68:22:...
MD5 Fingerprint    94:28:...
```

4. If you click the Details tab, you will see more information about this certificate. See the picture below:



Website Identity

Website: login.yahoo.com

Owner: This website does not supply ownership information.

Verified by: DigiCert Inc

Certificate Hierarchy

```
GTE CyberTrust Global Root
  Baltimore CyberTrust Root
    DigiCert High Assurance EV Root CA
      DigiCert High Assurance CA-3
        login.yahoo.com
```

Certificate Fields

```
login.yahoo.com
  Certificate
    Version
    Serial Number
    Certificate Signature Algorithm
    Issuer
    Validity
      Not Before
      Not After
```

Cool. Now I see details of a real certificate for commercial uses.

While on the Details tab of the Certificate Viewer, you can also:

- Look at each certificate of the Certificate Hierarchy.
- Export any certificate in the Certificate Hierarchy to a certificate file.
- Installing certificates into the browser is not supported.

Last update: 2013.

Managing Certificates in Certificate Stores

This section provides a tutorial example on how to access and manage certificates pre-installed Firefox 18.

In the previous tutorial, I learned how to view the certificate from the server, while contacting an https (SSL) Web server. Now I want to know how to access pre-installed certificates in Firefox.

1. Run Firefox 18, and go to the "Tools" > "Options..." menu. The Options dialog box shows up.

2. Click the "Advanced" icon, then click the "Encryption" tab. Encryption options shows up:

```
Protocols
[x] Use SSL 3.0      [x] Use TSL 1.0

Certificates
When a server requests my personal certificates:
( ) Select one automatically      (.) Ask me every time
```

3. Click the "View Certificates" button. The Certificate Manager dialog box shows up.

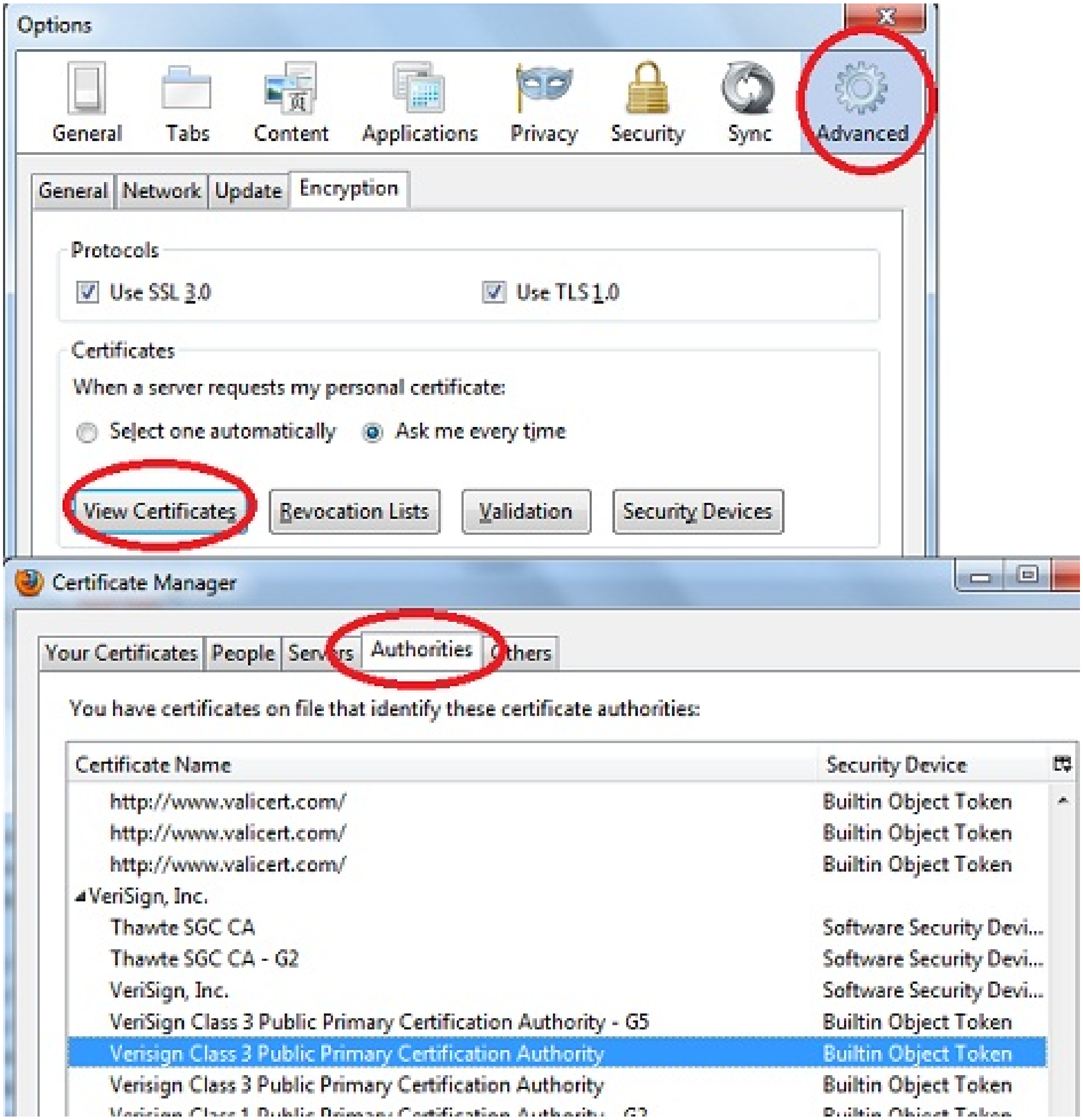
4. Go to the Authorities tab.

5. Select "VeriSign Class 3 Public Primary Certification Authority", and click "View" button. The Certificate Viewer dialog box shows up.

Now we know that Firefox 18 does have pre-installed certificates. If you click other tabs, you will see some other certificates in different categories:

- Your Certificates - For my own certificates.
- People - For other people's certificates.
- Servers - For certificates from non CAs.
- Authorities - For certificates from CAs.
- Others - For other certificates.

The picture below shows you how to access the Certificate Manager in Firefox 18:



☑ Use SSL 3.0 ☑ Use TLS 1.0

When a server requests my personal certificate:

○ Select one automatically ● Ask me every time

You have certificates on file that identify these certificate authorities:

Certificate Name	Security Device
http://www.valicert.com/	Builtin Object Token
http://www.valicert.com/	Builtin Object Token
http://www.valicert.com/	Builtin Object Token
◢VeriSign, Inc.	
Thawte SGC CA	Software Security Devi...
Thawte SGC CA - G2	Software Security Devi...
VeriSign, Inc.	Software Security Devi...
VeriSign Class 3 Public Primary Certification Authority - G5	Builtin Object Token
Verisign Class 3 Public Primary Certification Authority	Builtin Object Token
Verisign Class 3 Public Primary Certification Authority	Builtin Object Token
Verisign Class 1 Public Primary Certification Authority - G3	Builtin Object Token

Last update: 2013.

Exporting Certificates Out of Firefox

This section provides a tutorial example on how to export a certificate from Firefox into a certificate file in DER and PEM formats.

In previous tutorial, we learned how to access certificates installed in Firefox. Now let's see if we can export a certificate out of Firefox.

1. Repeat steps listed in the previous tutorial until you see the Certificate Manager dialog box.

2. Go to the Authorities tab and select "VeriSign Class 3 Public Primary Certification Authority".

3. Click "Export" button. The Save Certificate To File dialog box shows up. The "Save as type" field supports these certificate file types with "X.509 Certificate (PEM)(*.crt;*.pem)" as the default:

```
X.509 Certificate (PEM)(*.crt;*.pem)
X.509 Certificate with chain (PEM)(*.crt;*.pem)
X.509 Certificate (DER)(*.der)
X.509 Certificate (PKCS#7)(*.p7c)
X.509 Certificate with chain (PKCS#7)(*.p7c)
```

4. Enter a file name: \verisign3.crt, and click the "Save" button. The selected certificate will be exported into the specified file.

In the next tutorial, we will view the contents of the certificate from "VeriSign Class 3 Public Primary Certification Authority" stored in a PEM file, \verisign3.crt.

Last update: 2013.

OpenSSL Viewing Certificates Exported from Firefox

This section provides a tutorial example on how to use OpenSSL to view contents of a certificate file exported from Firefox.

In the previous tutorial, we successfully exported the certificate from "VeriSign Class 3 Public Primary Certification Authority" to a DER file, \verisign3.crt. Now I want to view its contents with OpenSSL.

```
>openssl x509 -in \verisign3.crt -inform PEM -noout -text
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number:
            3c:91:31:cb:1f:f6:d0:1b:0e:9a:b8:d0:44:bf:12:be
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=US, O=VeriSign, Inc., OU=Class 3 Public Primary ...
Authority
        Validity
            Not Before: Jan 29 00:00:00 1996 GMT
            Not After : Aug  2 23:59:59 2028 GMT
        Subject: C=US, O=VeriSign, Inc., OU=Class 3 Public Primary...
 Authority
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:c9:5c:59:9e:f2:1b:8a:01:14:b4:10:df:04:40:
                    db:e3:57:af:6a:45:40:8f:84:0c:0b:d1:33:d9:d9:
                    11:cf:ee:02:58:1f:25:f7:2a:a8:44:05:aa:ec:03:
                    1f:78:7f:9e:93:b9:9a:00:aa:23:7d:d6:ac:85:a2:
                    63:45:c7:72:27:cc:f4:4c:c6:75:71:d2:39:ef:4f:
                    42:f0:75:df:0a:90:c6:8e:20:6f:98:0f:f8:ac:23:
                    5f:70:29:36:a4:c9:86:e7:b1:9a:20:cb:53:a5:85:
                    e7:3d:be:7d:9a:fe:24:45:33:dc:76:15:ed:0f:a2:
                    71:64:4c:65:2e:81:68:45:a7
                Exponent: 65537 (0x10001)
    Signature Algorithm: sha1WithRSAEncryption
        10:72:52:a9:05:14:19:32:08:41:f0:c5:6b:0a:cc:7e:0f:21:
        19:cd:e4:67:dc:5f:a9:1b:e6:ca:e8:73:9d:22:d8:98:6e:73:
        03:61:91:c5:7c:b0:45:40:6e:44:9d:8d:b0:b1:96:74:61:2d:
        0d:a9:45:d2:a4:92:2a:d6:9a:75:97:6e:3f:53:fd:45:99:60:
        1d:a8:2b:4c:f9:5e:a7:09:d8:75:30:d7:d2:65:60:3d:67:d6:
        48:55:75:69:3f:91:f5:48:0b:47:69:22:69:82:96:be:c9:c8:
        38:86:4a:7a:2c:73:19:48:69:4e:6b:7c:65:bf:0f:fc:70:ce:
        88:90
```

Very nice.

- This is a self-signed certificate from VeriSign Inc., valid until year 2028.
- The certificate file format is PEM, not DER. You need to use "-inform PEM" with the OpenSSL command.

Of course, we can export the certificate out of Firefox, and save it in other formats. Just select a different one from the supported file type list.

Last update: 2013.

Importing CA Certificate into Firefox

This section provides a tutorial example on how to import a CA certificate into Firefox in the 'Authorities' category.

After exporting certificates from Firefox 18, I want to test the certificate import function of Firefox 18. First I want to import my own self-signed certificate, herong.crt, generated in previous tutorials as a CA certificate into Firefox 18.

1. Repeat steps listed in the previous tutorial until you see the Certificate Manager dialog box.

2. Click the "Your Certificates" tab and click the "Import" button. The File Name to Restore dialog box shows up.

3. Change the selection in "Files of type" from "PKCS12 Files" to "All Files" and select "herong.crt".

4. Click the "Open" button. Surprisingly, the Password Entry Dialog box shows up with this message: "Please enter the password that was used to encrypt this certificate backup." Why

Firefox is asking for password to import a certificate?

It looks like only certificates stored in PKCS12 format can be imported into the "Your Certificates" category. I need to select another category to do the import.

5. Click the "People" tab and click the "Import" button. The "Select File containing somebody's Email certificate to import" dialog box shows up.

6. Keep the select in "Files of type" as "Certificate Files" and select "herong.crt".

7. Click the "Open" button. Another surprise, an alert message shows up: "This certificate can't be verified and will not be imported. The certificate issuer might be unknown or untrusted, the certificate might have expired or been revoked, or the certificate might not have been approved."

By looking at those certificate category names, I think my CA certificate should be imported into "Authorities" category.

8. Click the "Authorities" tab and click the "Import" button. The "Select File containing Server certificate to import" dialog box shows up.

9. Keep the select in "Files of type" as "Certificate Files" and select "herong.crt".

10. Click the "Open" button. The "Downloading Certificate" dialog box shows up with these options:

```
You have been asked to trust a new Certificate Authority (CA).

Do you want to trust "Herong Yang" for the following purposes?
[ ] Trust this CA to identify Web sites.
[ ] Trust this CA to identify email users.
[ ] Trust this CA to identify software developers.

Before trusting this CA for any purpose, you should examine its
certificate and its policy and procedures (if available).
```

11. Check all checkboxes and click the OK button. My certificate will be imported into Firefox as a trusted CA certificate.

Last update: 2013.

Importing Certificate Path into Firefox

This section provides a tutorial example of trying to import a certificate path into Firefox. But it failed..

Next, I want to import some other certificates signed by me into Firefox to form a certificate path:

- herong.crt - self-signed by Herong.
- john.crt - signed by Herong.
- bill.crt - signed by John.
- tom.crt - signed by Bill.

1. Repeat steps listed in the previous tutorial until you see the Certificate Manager dialog box.

2. Click the "People" tab and click the "Import" button. The "Select File containing somebody's Email certificate to import" dialog box shows up.

3. Keep the select in "Files of type" as "Certificate Files" and select "john.crt".

4. Click the "Open" button. No validation errors, because the issuer, Herong Yang, has a CA certificate in Firefox now.

But John's certificate is not showing up in the "People" category. If check the "Others" category. Surprisingly, John's certificate is listed there!

5. Click the "People" tab and click the "Import" button. The "Select File containing somebody's Email certificate to import" dialog box shows up.

6. Keep the select in "Files of type" as "Certificate Files" and select "bill.crt".

7. Click the "Open" button. An alert message shows up: "This certificate can't be verified and will not be imported. The certificate issuer might be unknown or untrusted, the certificate might have expired or been revoked, or the certificate might not have been approved."

The message is clear. The issuer of Bill's certificate is "John Smith", whose certificate is in Firefox, but not trusted.

Looks like there is no way to import a full certificate path into Firefox 18.

Last update: 2013.

Using Certificates in Google Chrome

This chapter provides tutorial notes and example codes on using certificates in Chrome 24. Topics include viewing the certificate from an https Web server; viewing and managing trusted certificates saved in certificate stores shared with IE 9.

Conclusions:

- Web browsers need certificate from Web servers, if they are using https (SSL protocol) to encrypt messages to secure the communication.
- Chrome 24 allows you to view server's certificate through the lock icon in front of the Web address.
- Chrome 24 shares the same certificate stores with IE (Internet Explorer) 9 on Windows systems.
- Chrome 24 uses the same certificate tool as IE 9 to export and import certificates to and from certificate files.

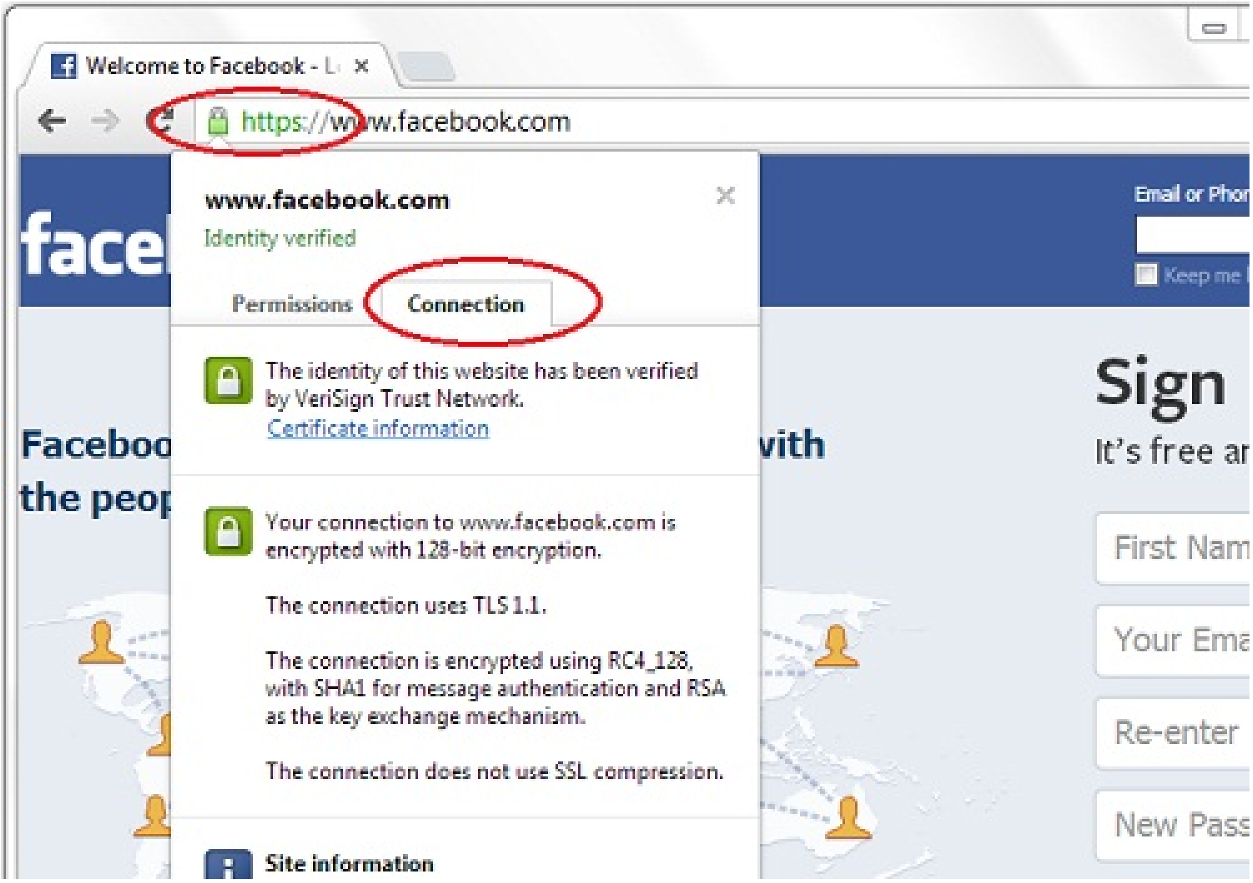
Visiting a "https" Web Site with Chrome

This section describes how Google Chrome 24 shows a lock icon when you visit an 'https' Web site to provide you more security related information.

Google Chrome is becoming a popular Web browser now. Let's take a quick look at how Chrome uses public key certificates.

If you go to a Web site that provides online services, I am sure that you will see "https" in the Web address field starting on the log in page. This indicates that Web site uses SSL protocol to encrypt all information you send and receive on this server.

If you are using Google Chrome 24, it will display an extra lock icon in front of the Web address field when you are connecting to an "https" Web site. The picture below shows the "https" Web address and the lock icon when you use Chrome 24 to visit the Facebook log in page: "https://www.facebook.com":



www.facebook.com
Identity verified

The identity of this website has been verified by VeriSign Trust Network.
Certificate information

Your connection to www.facebook.com is encrypted with 128-bit encryption.

The connection uses TLS 1.1.

The connection is encrypted using RC4_128, with SHA1 for message authentication and RSA as the key exchange mechanism.

The connection does not use SSL compression.

Site information

If you click the lock icon, then the "Connection" tab, Chrome will provide you a summary of security related information about this Web site. For https://www.facebook.com Web site, you will get something like:

```
www.facebook.com
Identity verified

The identify of this website has been verified by VeriSign Trust
Network. "Certificate information" link

Your connection to www.facebook.com is encrypted with 128-bit
encryption.

The connection is encrypted using RC4_128 with SHA1 for message
authentication and RSA as the key exchange mechanism.

The connection does now use SSL compression
```

More security related information will be provided if you click the "Certificate information" link. See next sections.

Last update: 2013.

Viewing Certificate Details

This section provides a tutorial example on how to view certificate details when visiting an 'https' Web site in Chrome 24.

When you visit a "https" Web server, it will send its certificate to your browser. Server's certificate is needed by the browser for these 2 tasks:

- Your browser must validate the certificate to determine that the Web site can be trusted or not before doing any further communications.
- Your browser must use the public key in the certificate to help secure the communication messages sent and received.

Normally, your browser will do these 2 tasks automatically without your interaction. You don't need to know where is the server certificate and what's in the certificate.

But since I am interested to learn more about "https" communication, I want to see the server certificate. Here is what did on Chrome 24 to see details of the server certificate.

1. Run Chrome 24 and go to https://login.yahoo.com and wait for the log in page to be loaded.

2. Click the lock icon at the end of the Web address field. A small pop up windows shows up.

3. Click the "Connection" tab and then the "Certificate information" link on the pop up window. The Certificate dialog box shows up. The General tab tells me this information:

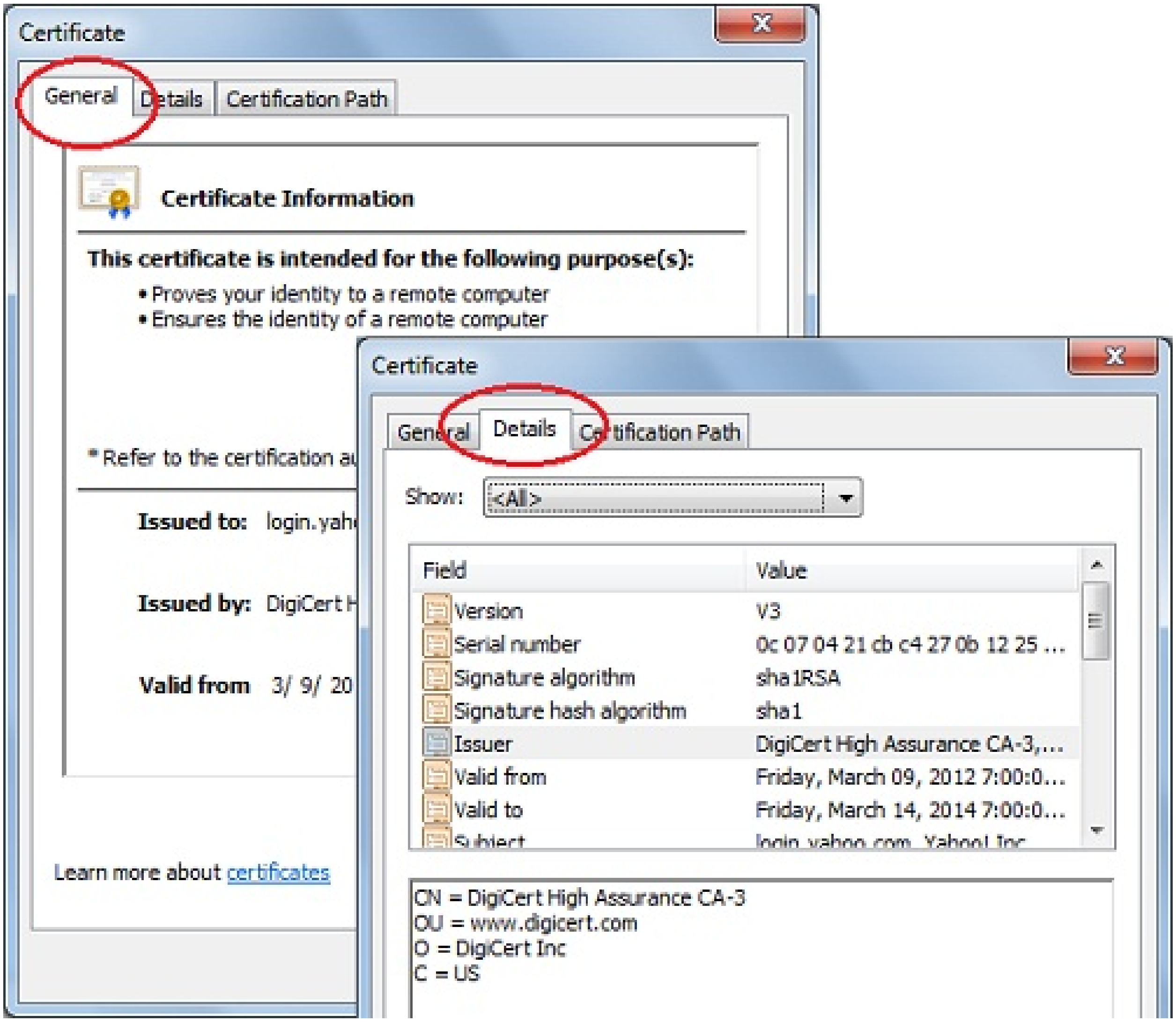
```
This certificate is intended for the following purpose(s):
- Proves your identity to a remote computer
- Ensures the identify of a remote computer

Issued to: login.yahoo.com
Issued by: DigiCert High Assurance CA-3
Valid from 3/ 9/ 2012 to 3/ 14/ 2014
```

4. If you click the Details tab, you will details of this certificate:

```
Version                V3
Serial number          0c 07 04 ...
Signature algorithm    sha1RSA
Signature has algorithm sha1
Issuer                 DigiCert High Assurance CA-3
Valid from             Friday, March 09, 2012 07:00:00 PM
Valid to               Friday, March 14, 2014 07:00:00 PM
Subject                login.yahoo.com, Yahoo! Inc.
Public key             RSA (2048 Bits)
Authority Key Identifier KeyID=50 ea 73 ...
Subject Key Identifier   2a aa f2 ...
Enhanced Key Usage     Server Authentication (1.3.6.1.5.5.7.3.1)
                       Client Authentication (1.3.6.1.5.5.7.3.2)
Key Usage              Digital Signature, Key Encipherment (a0)
```

```
Thumbprint algorithm  sha1
Thumbprint            68 22 14 ...
```

Cool. Now I see details of a real certificate for commercial uses. The picture below shows you steps to reach certificate details:



This certificate is intended for the following purpose(s):

- Proves your identity to a remote computer
- Ensures the identity of a remote computer

* Refer to the certification au

Issued to: login.yah

Issued by: DigiCert H

Valid from 3/ 9/ 20

Field	Value
Version	V3
Serial number	0c 07 04 21 cb c4 27 0b 12 25 ...
Signature algorithm	sha1RSA
Signature hash algorithm	sha1
Issuer	DigiCert High Assurance CA-3,...
Valid from	Friday, March 09, 2012 7:00:0...
Valid to	Friday, March 14, 2014 7:00:0...
Subject	login.yahoo.com, Yahoo! Inc...

CN = DigiCert High Assurance CA-3
OU = www.digicert.com
O = DigiCert Inc
C = US

Notice that the "Certificate" dialog box used in Chrome is identical to "Certificate" dialog gox used in IE (Internet Explorer) 9. See IE 9 tutorials for information provided on the "Certificate Path" tab.

Last update: 2013.

Installing Certificate Permanently in Chrome - Not Supported

This section provides a tutorial example on trying to install the certificate provided by an 'https'

Web site in Chrome 24. But I don't see any button or link that allows to install the server's certificate into the browser.

Normally, when a browser visits a "https" Web server, it will get the certificate from the server and use it for one web session only. When you close the browser, the certificate will be removed from memory. It will not be stored on computer permanently.

However, if you want to keep a copy of the certificate on the computer, you can try these steps shown below:

1. Run Chrome 24 and go to https://login.yahoo.com and wait for the log in page to be loaded.

2. Click the lock icon at the end of the Web address field. A small pop up windows shows up.

3. Click the "Connection" tab and then the "Certificate information" link on the pop up window. The Certificate dialog box shows up. The Certificate dialog box shows up.

But the "Install Certificate" button is not supported on the "General" tab! May be I need to run Chrome in administrator mode?

4. Close Chrome browser. Then right-click the Chrome icon and select "Run as administrator". Repeat all steps mentioned above. I still don't see the "Install Certificate" button.

So, Chrome does not allow my to install the certificate from the website into the browser.

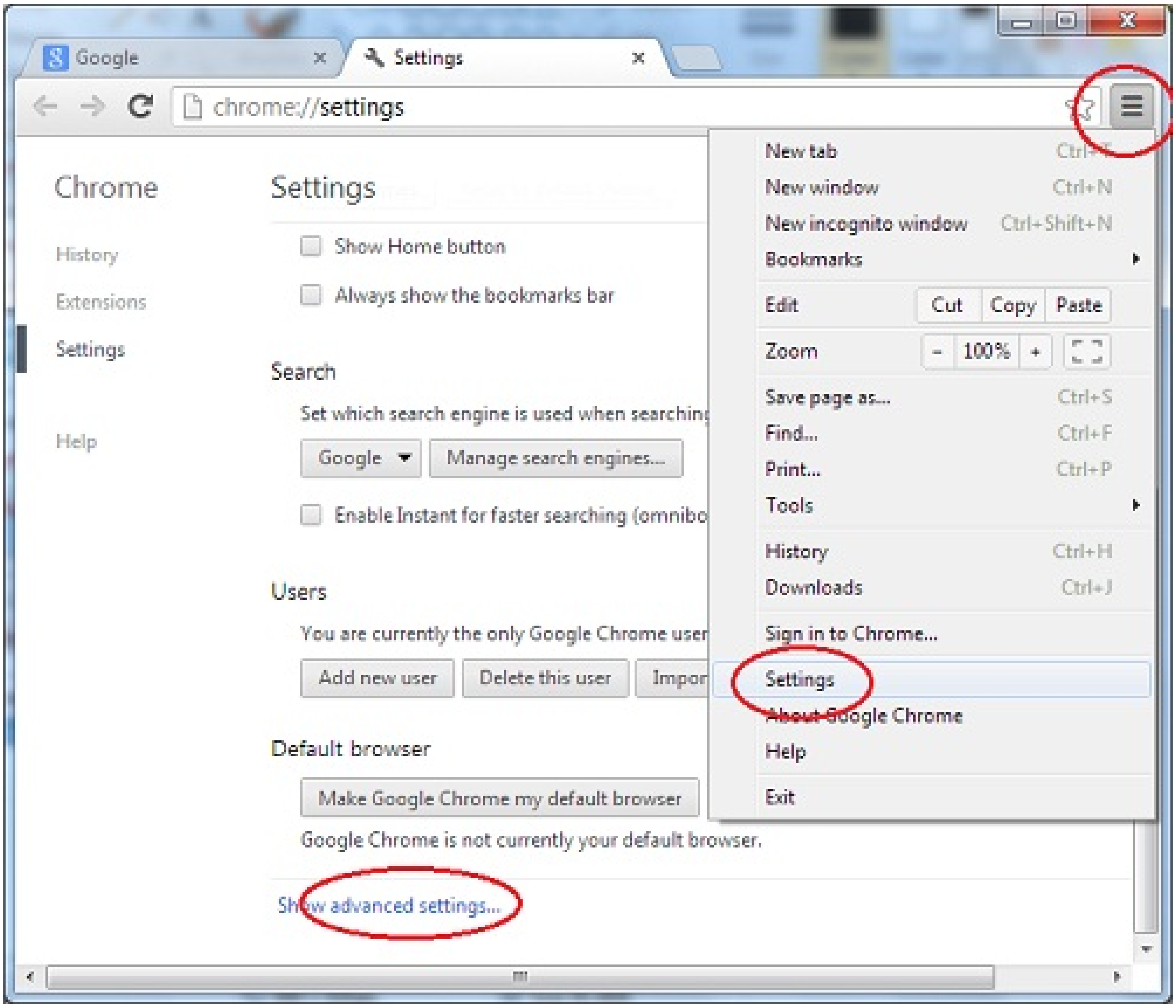
Last update: 2013.

Managing Certificates in Certificate Stores

This section provides a tutorial example on how to use Chrome 24 to manage certificates installed in certificate stores.

Now I want to know how Chrome 24 manages certificates that are already installed on the local Windows system.

1. Run Chrome 24 and click the menu icon. Then select "Settings" from the pop up menu. The Chrome settings page shows up.



Set which search engine is used when searchin

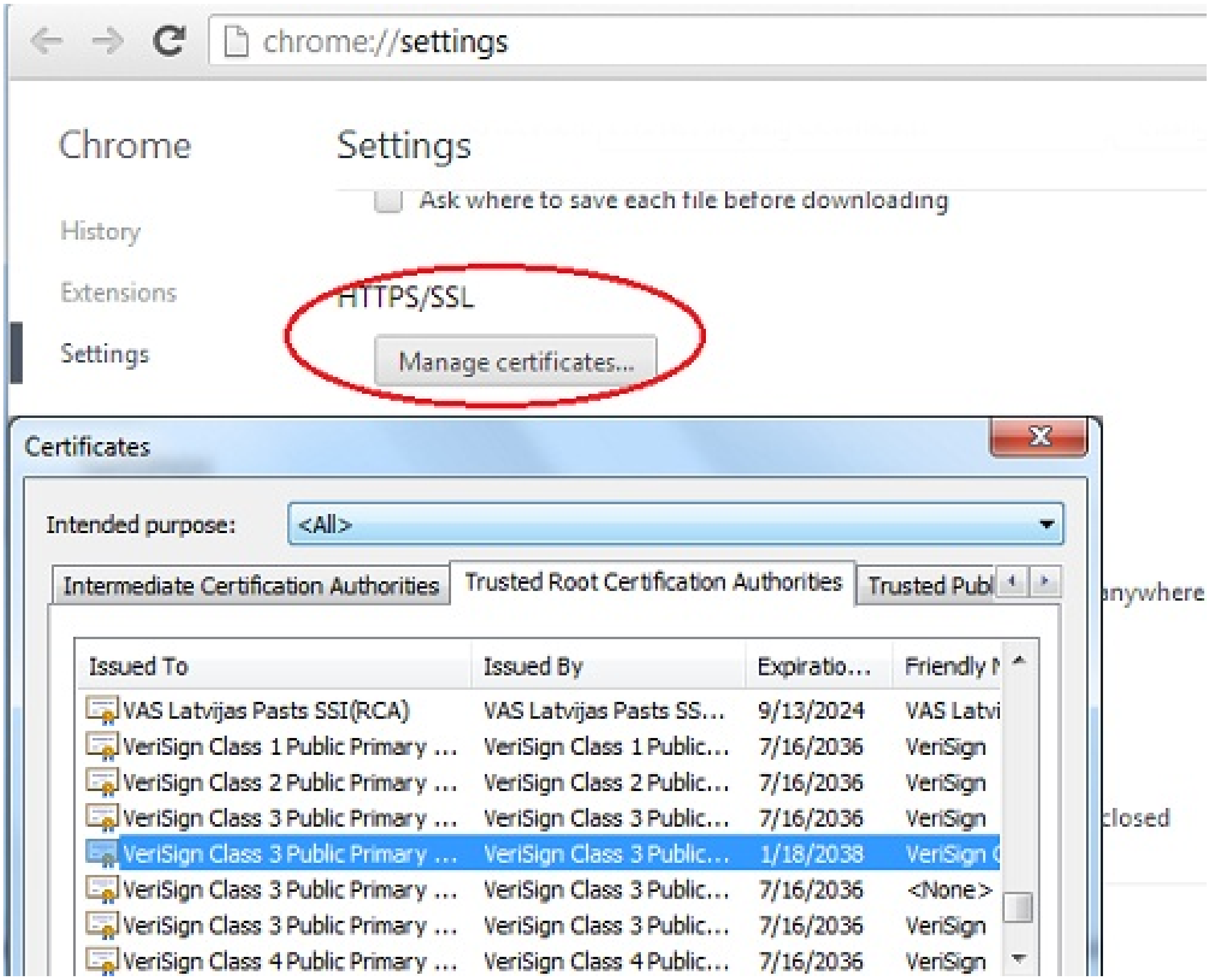
Enable Instant for faster searching (omnibo

You are currently the only Google Chrome user

Google Chrome is not currently your default browser.

2. Click the "Show advanced settings..." link. The scroll down to the HTTPS/SSL section.

3. Click the "Manage certificates..." button in the HTTPS/SSL section. The Certificates dialog box shows up.



Issued To	Issued By	Expiratio...	Friendly N
VAS Latvijas Pasts SSI(RCA)	VAS Latvijas Pasts SS...	9/13/2024	VAS Latvi
VeriSign Class 1 Public Primary ...	VeriSign Class 1 Public...	7/16/2036	VeriSign
VeriSign Class 2 Public Primary ...	VeriSign Class 2 Public...	7/16/2036	VeriSign
VeriSign Class 3 Public Primary ...	VeriSign Class 3 Public...	7/16/2036	VeriSign
VeriSign Class 3 Public Primary ...	VeriSign Class 3 Public...	1/18/2038	VeriSign C
VeriSign Class 3 Public Primary ...	VeriSign Class 3 Public...	7/16/2036	<None>
VeriSign Class 3 Public Primary ...	VeriSign Class 3 Public...	7/16/2036	VeriSign
VeriSign Class 4 Public Primary ...	VeriSign Class 4 Public...	7/16/2036	VeriSign

As you can see, the "Certificates" dialog box in Chrome is the same as IE. This tells me that Chrome share same certificate stores with IE and uses the same tools to export and import certificates to and from certificate files. See IE tutorials for more information.

Last update: 2013.

Outdated Tutorials

This chapter contains some outdated tutorial notes and example codes from previous versions of this book.

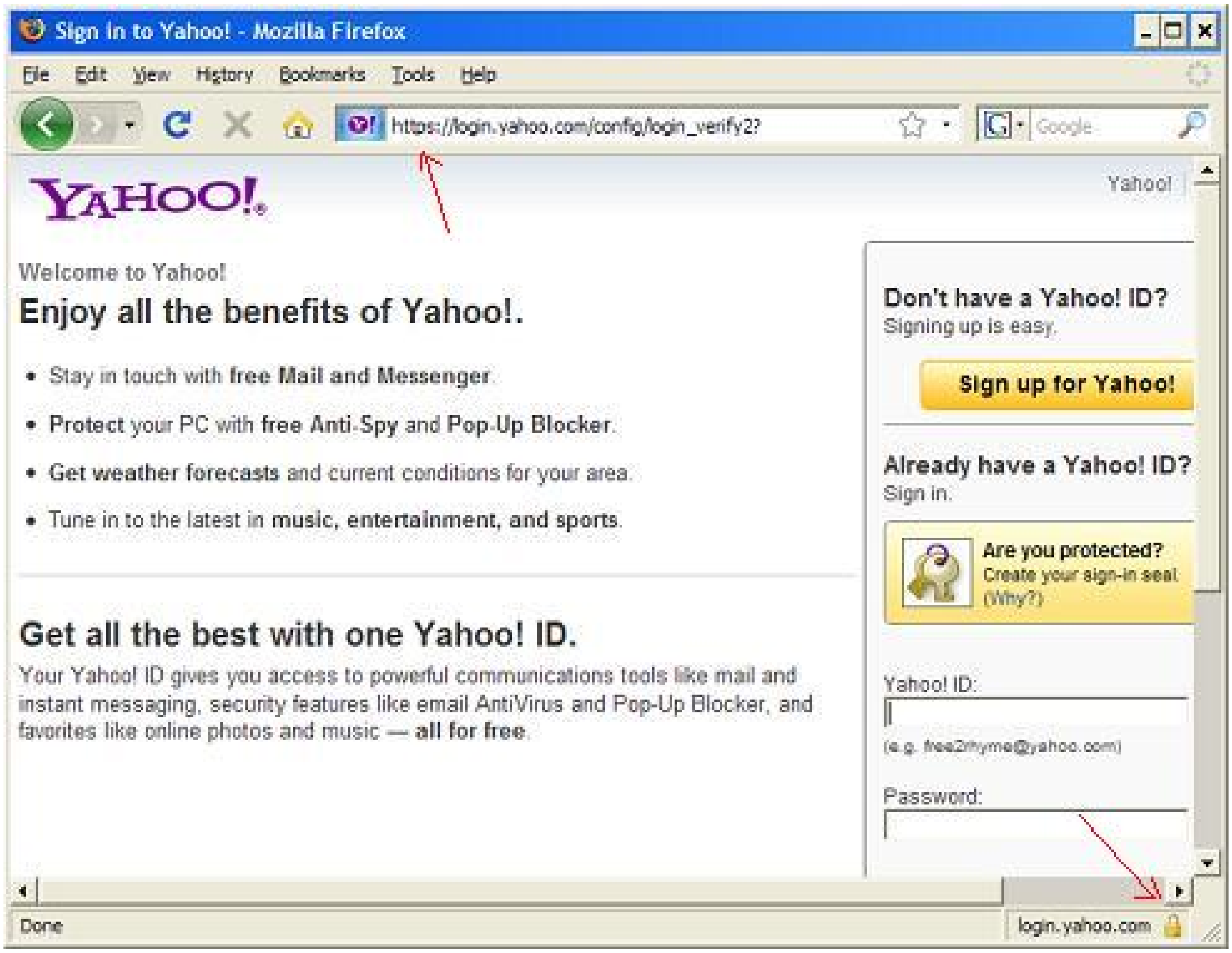
Outdated: Visiting an "https" Web Site with Firefox 3

This section describes how Firefox 3 shows a lock icon when you visit an 'https' Web site to provide you more security related information.

As we learned in the previous chapter, if a Web site wants to encrypt communication messages with your browser, it will enable the SSL protocol and use "https" as part of their Web address.

If you go to a Web site that provides online services, I am sure that you will see "https" in the Web address field starting on the log in page. This indicates that Web site uses SSL protocol to encrypt all information you send and receive on this server.

If you are using Firefox 3, it will display an extra lock icon at the bottom left corner. when you are connecting to an "https" Web site. The picture below shows the "https" Web address and the lock icon when you use Firefox 3 to visit the Yahoo log in page: "https://login.yahoo.com":



Sign in to Yahoo! - Mozilla Firefox
File Edit View History Bookmarks Tools Help
https://login.yahoo.com/config/login_verify2?
Google
YAHOO!
Yahoo!
Done
login.yahoo.com

Welcome to Yahoo!

Enjoy all the benefits of Yahoo!.

- Stay in touch with free Mail and Messenger.
- Protect your PC with free Anti-Spy and Pop-Up Blocker.
- Get weather forecasts and current conditions for your area.
- Tune in to the latest in music, entertainment, and sports.

Get all the best with one Yahoo! ID.

Your Yahoo! ID gives you access to powerful communications tools like mail and instant messaging, security features like email AntiVirus and Pop-Up Blocker, and favorites like online photos and music — all for free.

Don't have a Yahoo! ID?
Signing up is easy.

Sign up for Yahoo!

Already have a Yahoo! ID?
Sign in.

Are you protected?
Create your sign-in seal
(Why?)

Yahoo! ID:
(e.g. free2rhyme@yahoo.com)

Password:

If you click the lock icon, IE will provide you more information on security related to this Web site.

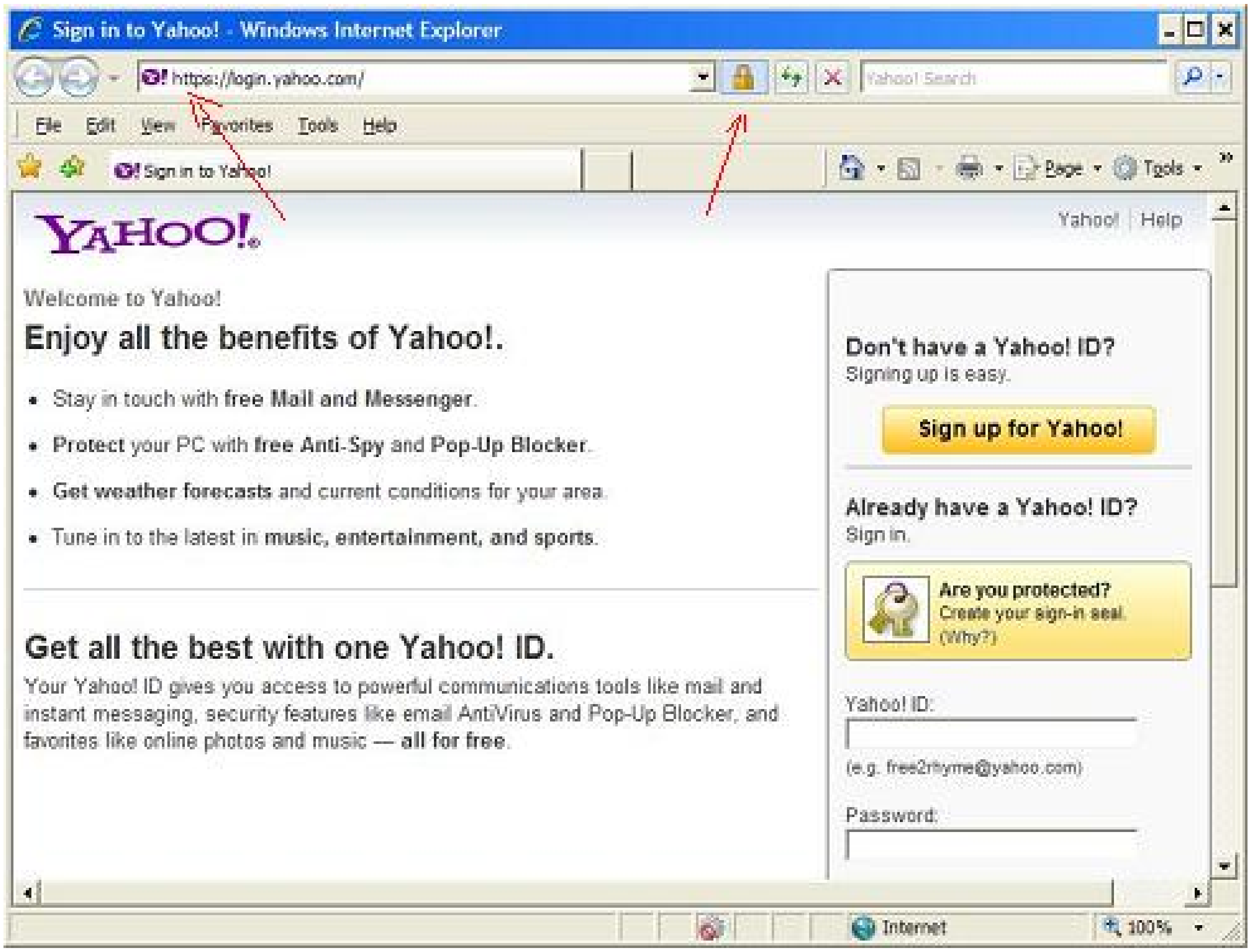
Outdated: Visiting an "https" Web Site with IE 7

This section describes how IE (Internet Explorer) 7 shows a lock icon when you visit an 'https' Web site to provide you more security related information.

As I mentioned in the previous section, if a Web site wants to encrypt communication messages with your browser, it will enable the SSL protocol and use "https" as part of their Web address.

If you go to a Web site that provides online services, I am sure that you will see "https" in the Web address field starting on the log in page. This indicates that Web site uses SSL protocol to encrypt all information you send and receive on this server.

If you are using IE (Internet Explorer) 7, it will display an extra lock icon next to the Web address field when you are connecting to an "https" Web site. The picture below shows the "https" Web address and the lock icon when you use IE 7 to visit the Yahoo log in page: "https://login.yahoo.com":



Welcome to Yahoo!

Enjoy all the benefits of Yahoo!.

- Stay in touch with free Mail and Messenger.
- Protect your PC with free Anti-Spy and Pop-Up Blocker.
- Get weather forecasts and current conditions for your area.
- Tune in to the latest in music, entertainment, and sports.

Get all the best with one Yahoo! ID.

Your Yahoo! ID gives you access to powerful communications tools like mail and instant messaging, security features like email AntiVirus and Pop-Up Blocker, and favorites like online photos and music — all for free.

Don't have a Yahoo! ID?
Signing up is easy.

Already have a Yahoo! ID?
Sign in.

Are you protected?
Create your sign-in seal.
(Why?)

Yahoo! ID:

(e.g. free2rhyme@yahoo.com)

Password:

If you click the lock icon, IE will provide you more information on security related to this Web site.

References

List of reference materials used in this book.

- *Handbook of Applied Cryptography*, A. Megezes, P. van Oorschot, and S. Vanstone, CRC Press, 1996, http://www.cacr.math.uwaterloo.ca/hac
- *Federal Information Processing Standards Publication 1981 Guidelines for Implementing and Using the NBS Data Encryption Standard*, 1981 http://www.itl.nist.gov/fipspubs/fip74.htm
- *DES Modes of Operation*, 1980 http://www.itl.nist.gov/fipspubs/fip81.htm
- *The DES Algorithm Illustrated*, J. Orlin Grabbe, http://www.orlingrabbe.com/des.htm
- *PKCS #5: Password-Based Encryption Standard*, 1993, An RSA Laboratories Technical Note, ftp://ftp.rsasecurity.com/pub/pkcs/doc/pkcs-5.doc
- *Java Cryptography Architecture API Specification & Reference*, 2004, http://java.sun.com/j2se/1.5.0/docs/guide/security/CryptoSpec.html
- *SourceForge.net: mcrypt - Encryption command and library*, http://sourceforge.net/projects/mcrypt
- *Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)*, Bruce Schneier, December 1993, http://www.schneier.com/paper-blowfish-fse.html
- *The Blowfish Algorithm*, Asian School of Cyber Laws, http://www.asianlaws.org/infosec/library/algo/blowfish.pdf
- *Encryption Using the Blowfish Algorithm*, Kevin Hackett, 1998, http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/1998f/blowfish_encryption
- *Coder's Lagoon - Homepage of BlowfishJ*, Markus Hahn, http://come.to/hahn
- *Primer on Public Key Encryption*, MyCrypto.net, http://www.mycrypto.net/encryption/encryption_public.html
- *Public Key Encryption*, Charlie Fletcher, http://www.krellinst.org/UCES/archive/modules/charlie/pke/
- *OpenSSL: The Open Source Toolkit for SSL/TLS*, The OpenSSL Project, http://www.openssl.org/
- *Herong's Tutorial Notes on Data Encoding*, Dr. Herong Yang, http://www.geocities.com/herong_yang/data/
- *PEM, Privacy Enhanced Mail*, http://www.networksorcery.com/enp/data/pem.htm
- *RFC 1421 - Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption*

and Authentication Procedures, February 1993, http://www.faqs.org/rfcs/rfc1421.html

- *RFC 3548 - The Base16, Base32, and Base64 Data Encodings*, July 2003, http://www.faqs.org/rfcs/rfc3548.html
- *MD5 Homepage (unofficial),* http://userpages.umbc.edu/~mabzug1/cs/md5/md5.html
- *RFC 1321 - The MD5 Message-Digest Algorithm*, April 1992, http://www.ietf.org/rfc/rfc1321.txt
- *MD5 in 8 lines of perl5*, John Allen, http://www.cypherspace.org/adam/rsa/md5.html
- *MD5*, Wikipedia, http://en.wikipedia.org/wiki/MD5
- *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD*, Xiaoyun Wang, Dengguo Feng, Xuejia Lai and Hongbo Yu, http://eprint.iacr.org/2004/199
- *FIPS PUB 180-1 - Secure Hash Algorithm*, 1993 May 11, http://www.itl.nist.gov/fipspubs/fip180-1.htm
- *FIPS PUB 186-3 - Digital Signature Standard (DSS)*, 2009, http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf
- *RFC 1321 - US Secure Hash Algorithm 1 (SHA1)*, September 2001 http://www.ietf.org/rfc/rfc3174.txt
- *SHA in 8 lines of perl5*, John Allen, http://www.cypherspace.org/adam/rsa/sha.html
- *SHA hash functions*, Wikipedia, http://en.wikipedia.org/wiki/SHA-1
- *Collision Search Attacks on SHA1*, Xiaoyun Wang. Yiqun Lisa Yin and Hongbo Yu, http://theory.lcs.mit.edu/~yiqun/shanote.pdf
- *PKCS #1: RSA Cryptography Standard*, EMC, http://www.rsa.com/rsalabs/node.asp?id=2125
- *The RSA Algorithm*, Evgeny Milanov, http://www.math.washington.edu/~morrow/336_09/papers/Yevgeny.pdf