Network Security

[Lab Manual]
2010-11

Indira Gandhi Institute of Technology
Guru Gobind Singh Indraprastha University

Table of Contents

0. Guidelines

This document titled "Network Security - Lab Manual" is written based on following guidelines:

1. Lab work is divided into 12 weeks. Each week has certain objectives to be accomplished.

2. Each week has 2 hours of Lab work.

3. Students can implement the specified algorithm using any programming Language(C, C++, Java cryptography).

5. All students are required to records their work in Project File (individual).

.

.

1. Syllabus

As taken from www.ipu.ac.in website*. ***Academics □□Academics □□Scheme Syllabus (Affiliated Institutes, w.e.f. 2005-06)***

UNIT – I

Introduction: Codes and Ciphers – Some Classifical systems – Statistical theory of cipher systems – Complexity theory of Crypto systems – Stream ciphers, Block ciphers.

Stream Ciphers: Rotor based system – shift register based systems – Design considerations for stream ciphers – Cryptanalysis of stream ciphers – Combined eneryption and encoding.

Block Ciphers – DES and variant, modes of use of DES.

UNIT – II

Public Key systems – Knacksack systems – RSK – Diffle Hellman Exchange 0 Authentication and Digital signatures, Elliptic curve based systems.

System Identification and clustering

Cryptology of speech signals – narrow band and wide band systems – analogue & digital systems of speech encryption.

UNIT – III

Network Security: Hash function – Authentication:

Protocols – Digital Signature standards.

Electronics Mail Security – PGP (Pretty Good Privacy) MIME, Data Compression technique.

IP Security: Architecture, Authentication Leader, Encapsulating security Payload – Key management.

Web Security: Secure Socket Layer & Transport Layer security, Secure electronic transactions.

Firewalls Design principle, established systems.

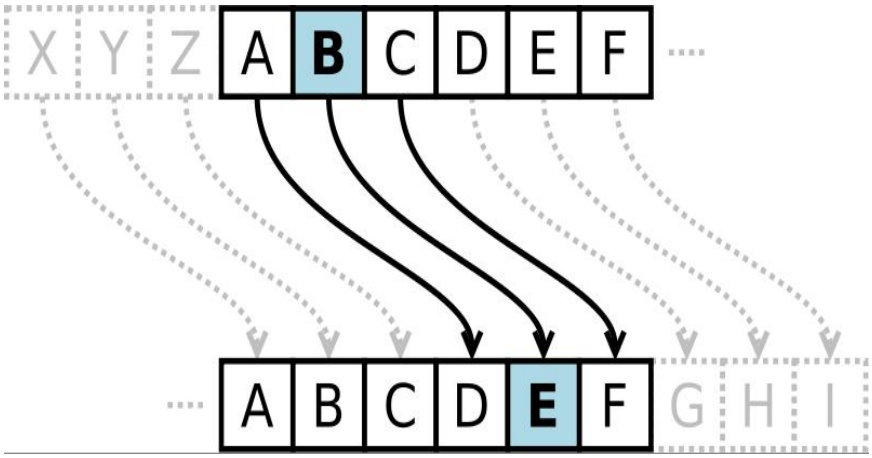
UNIT – IV

Telecommunication Network architecture, TMN management layers, Management information Model, Management servicing and functions, Structure of management information and TMN information model.

TEXT BOOKS:

1. William Stallings, “Network Security Essentials, 2nd Edition, 2002.
2. William Stallings, “Cryptography & Network Security”, 3rd Edition, 1999.

CEASER CIPHER ENCRYPTION/DECRYPTION



EXPERIMENT NO. 1

AIM: To implement Ceaser Cipher Encryption-Decryption.

Theory: In cryptography, a Caesar cipher, also known as a Caesar's cipher, the shift cipher, Caesar's code or Caesar shift, is one of the simplest and most widely known encryption techniques. It is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet. For example, with a shift of 3, A would be replaced by D, B would become E, and so on.

Example: The transformation can be represented by aligning two alphabets; the cipher alphabet is the plain alphabet rotated left or right by some number of positions. For instance, here is a Caesar cipher using a left rotation of three places (the shift parameter, here 3, is used as the key):

Plain: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher: DEFGHIJKLMNOPQRSTUVWXYZABC

When encrypting, a person looks up each letter of the message in the "plain" line and writes down the corresponding letter in the "cipher" line. Deciphering is done in reverse.

Ciphertext: WKH TXLFN EURZQ IRA MXPSV RYHU WKH ODCB GRJ
Plaintext: the quick brown fox jumps over the lazy dog

The encryption can also be represented using modular arithmetic by first transforming the letters into numbers, according to the scheme, A = 0, B = 1,..., Z = 25.Encryption of a letter *x* by a shift *n* can be described mathematically as:

$$E_n(x) = (x + n) \mod 26.$$

Decryption is performed similarly,

$$D_n(x) = (x - n) \mod 26.$$

Breaking of Ceaser's Cipher :

Obviously, it had two weaknesses. The first was that the algorithm was not particularly strong. If trial and error couldn't crack the algorithm, then some simple analysis would. If English text was being encrypted, then it would be relatively simple to compare the frequency of letters in the cipher text against the frequency of letters in standard English. Statistics would soon reveal patterns that pointed out the probable plain text letter associated with each cipher text letter. Once a single association was found the entire algorithm could be cracked. No message would be secure.

But, it worked for Caesar, and it illustrates how conventional cryptography works.

Frequency analysis :

Caesar's Cipher is so vulnerable to frequency analysis. It is because there is a one-to-one relationship

between each letter. If a sufficiently large ciphertext is given, the plaintext can be found out by frequency analysis.

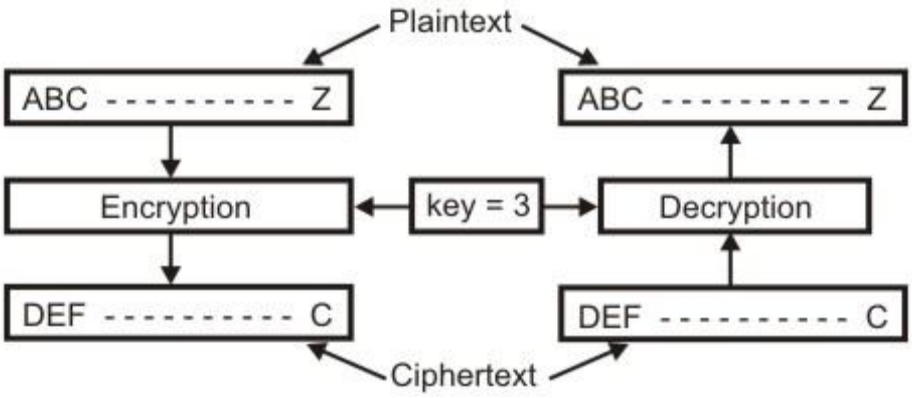
Letter	Frequency
E	0.127
T	0.097
I	0.075
A	0.073
O	0.068
N	0.067
S	0.067
R	0.064
H	0.049
C	0.045
L	0.040
D	0.031
P	0.030
Y	0.027
U	0.024
M	0.024
F	0.021
B	0.017
G	0.016
W	0.013
V	0.008
K	0.008
X	0.005
Q	0.002
Z	0.001
J	0.001

The frequency table of the letter appear in English

If there is a sufficiently large ciphertext, it would be solved by comparing the frequency of letters in the cipher text against the frequency of letters in standard English. If the frequency of the letter in the cipher text is almost the same as the frequency of letters in standard English, we can find out which letter is substituted for the letter in ciphertext. Then the message would be decrypted.

Drawback:
• Brute force attack is easy
• Try out all the 25 possible keys

Monoalphabetic Substitution



EXPERIMENT NO. 2

AIM: To implement Monoalphabetic Substitution.

THEORY:

The ciphers in this substitution section replace each letter with another letter according to the cipher alphabet. Ciphers in which the cipher alphabet remains unchanged throughout the message are called Monoalphabetic Substitution Ciphers.

In a monoalphabetic cipher, our substitution characters are a random permutation of the 26 letters of the alphabet:

• The key now is the sequence of substitution letters. In other words, the key in this case is the actual random permutation of the alphabet used.
• Note that there are 26! permutations of the alphabet. That is a number larger than 4 × 1026.

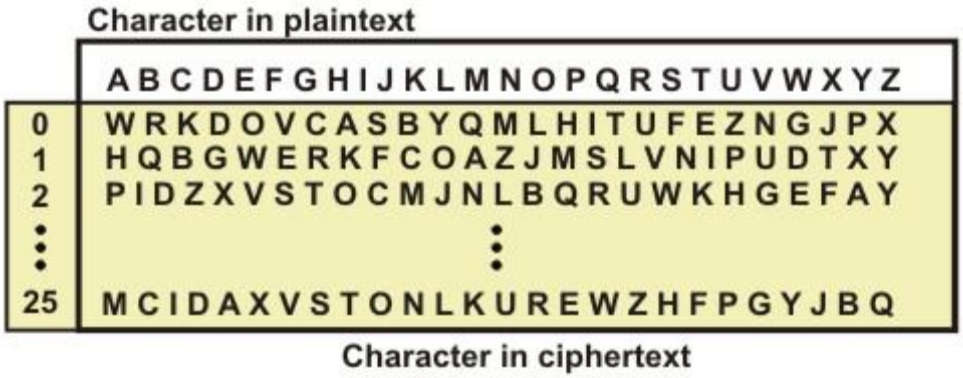
In general, an example of a monoalphabetic substitution is shown below.

***PLAINTEXT* :** a b c d e f g h i j k l m
CIPHERTEXT: Q R S K O W E I P L T U Y

***PLAINTEXT* :** n o p q r s t u v w x y z
CIPHERTEXT : A C Z M N V D H F G X J B

Drawback:
• We can make guesses by observing the relative frequency of letters in the text.
• Compare it with standard frequency distribution charts in English (say).
• Also look at the frequency of digrams and trigrams, for which tables are also available.
• Easy to break in general

Polyalphabetic Substitution



EXPERIMENT NO. 3

AIM: To implement Polyalphabetic Substitution.

THEORY: A cipher is **polyalphabetic** if a given letter of the alphabet will not always enciphered by the same ciphertext letter, and, as a consequence, cannot be described by a single set of ciphertext alphabet corresponding to a single set of plaintext alphabet.The simplest way to produce a polyalphabetic cipher is to combine different monoalphabetic ciphers. One of the problems with monoalphabetic ciphers is that the letters occur with certain frequency in a language. This frequency can be graphed for both plaintext letters and the ciphertext letters of the enciphered message, and, after some analysis, the cipher is relatively easily broken.

One of the example of the polyalphabetic ciphers is the **Vigenère cipher**. The Vigenère cipher was published in 1586 by the French diplomat Blaise de Vigenère. The basic idea of this cipher is to use a number of monoalphabetic ciphers in turn. In order to encipher by Vigenère cipher, you need two things: a *keyword* and the *Vigenère square*, below.

Key Word Letters

	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
A	**A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**
B	**B C D E F G H I J K L M N O P Q R S T U V W X Y Z A**
C	**C D E F G H I J K L M N O P Q R S T U V W X Y Z A B**
D	**D E F G H I J K L M N O P Q R S T U V W X Y Z A B C**
E	**E F G H I J K L M N O P Q R S T U V W X Y Z A B C D**
F	**F G H I J K L M N O P Q R S T U V W X Y Z A B C D E**
G	**G H I J K L M N O P Q R S T U V W X Y Z A B C D E F**
H	**H I J K L M N O P Q R S T U V W X Y Z A B C D E F G**
M	**I I J K L M N O P Q R S T U V W X Y Z A B C D E F G H**
e	**J J K L M N O P Q R S T U V W X Y Z A B C D E F G H I**
s	**K K L M N O P Q R S T U V W X Y Z A B C D E F G H I J**
s	**L L M N O P Q R S T U V W X Y Z A B C D E F G H I J K**
a	**M M N O P Q R S T U V W X Y Z A B C D E F G H I J K L**
g	**N N O P Q R S T U V W X Y Z A B C D E F G H I J K L M**
e	**O O P Q R S T U V W X Y Z A B C D E F G H I J K L M N**
	P P Q R S T U V W X Y Z A B C D E F G H I J K L M N O
L	**Q Q R S T U V W X Y Z A B C D E F G H I J K L M N O P**

e **R R S T U V W X Y Z A B C D E F G H I J K L M N O P Q**

t **S S T U V W X Y Z A B C D E F G H I J K L M N O P Q R**

t **T T U V W X Y Z A B C D E F G H I J K L M N O P Q R S**

e **U U V W X Y Z A B C D E F G H I J K L M N O P Q R S T**

r **V V W X Y Z A B C D E F G H I J K L M N O P Q R S T U**

s **W W X Y Z A B C D E F G H I J K L M N O P Q R S T U V**

X X Y Z A B C D E F G H I J K L M N O P Q R S T U V W

Y Y Z A B C D E F G H I J K L M N O P Q R S T U V W X

Z Z A B C D E F G H I J K L M N O P Q R S T U V W X Y

We write the keyword repeatedly over the text of the message until reaching the end. The rule for enciphering:

 The letter of the keyword that is above a plaintext letter determines the alphabet (i.e. the row of the square) which will be used to encipher this cleartext letter.

For example, if our keyword was "hell" and our message was "I like mathematics", then we write:

h	e	l	l	h	e		l	l	h	e	l		l	h	E	l	l
i	l	i	k	e	m	a	t	h	e	m	a	t	I	c	s		

Then, to encipher the first *i* we look it up the *h*th row of the Vigenère square to find *p*.

Another, quite mathematical example of a polyalphabetic cipher is **Hill's Algorithm**. It uses a matrix to encipher not single letters but a combination of *two letters* at a time. For example, to encipher *pr* by the matrix:

7	9
3	12

we multiply the matrix by the vector (16, 18), the positions of *p* and *r* in the alphabet.

Drawback:
• Key and the plaintext share the same frequency distribution of letters.
• The best thing would have been to use a keyword which is as large as the plaintext, and has no statistical relationship to it.

PLAY FAIR CIPHER

M	T	S	A	B
C	D	E	F	G
H	I/J	K	L	N
O	P	Q	R	U
V	Y	X	Y	Z

Figure 3 - Playfair cipher with a keyword of "MTS"

EXPERIMENT NO. 4

AIM: To implement Play fair Cipher.

THEORY:

The **Playfair cipher** or **Playfair square** is a manual symmetric encryption technique and was the first literal digraph substitution cipher. The technique encrypts pairs of letters (*digraphs*), instead of single letters as in the simple substitution cipher and rather more complex Vigenère cipher systems then in use. The Playfair is thus significantly harder to break since the frequency analysis used for simple substitution ciphers does not work with it. Frequency analysis can still be undertaken, but on the 600 possible digraphs rather than the 26 possible monographs. The frequency analysis of digraphs is possible, but considerably more difficult – and it generally requires a much larger ciphertext in order to be useful.

Decryption

The Playfair cipher uses a 5 by 5 table containing a key word or phrase. Memorization of the keyword and 4 simple rules was all that was required to create the 5 by 5 table and use the cipher.

To generate the key table, one would first fill in the spaces in the table with the letters of the keyword (dropping any duplicate letters), then fill the remaining spaces with the rest of the letters of the alphabet in order (usually omitting "Q" to reduce the alphabet to fit, other versions put both "I" and "J" in the same space). The key can be written in the top rows of the table, from left to right, or in some other pattern, such as a spiral beginning in the upper-left-hand corner and ending in the center. The keyword together with the conventions for filling in the 5 by 5 table constitute the cipher key.

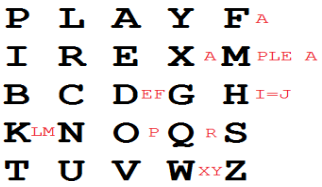
To encrypt a message, one would break the message into digraphs (groups of 2 letters) such that, for example, "HelloWorld" becomes "HE LL OW OR LD", and map them out on the key table. The two letters of the digraph are considered as the opposite corners of a rectangle in the key table. Note the relative position of the corners of this rectangle. Then apply the following 4 rules, in order, to each pair of letters in the plaintext:

1. If both letters are the same (or only one letter is left), add an "X" after the first letter. Encrypt the new pair and continue. Some variants of Playfair use "Q" instead of "X", but any uncommon monograph will do.
2. If the letters appear on the same row of your table, replace them with the letters to their immediate right respectively (wrapping around to the left side of the row if a letter in the original pair was on the right side of the row).
3. If the letters appear on the same column of your table, replace them with the letters immediately below respectively (wrapping around to the top side of the column if a letter in the original pair was on the bottom side of the column).
4. If the letters are not on the same row or column, replace them with the letters on the same row respectively but at the other pair of corners of the rectangle defined by the original pair. The order is important – the first letter of the encrypted pair is the one that lies on the same **row** as the first letter of the plaintext pair.

To decrypt, use the INVERSE (opposite) of the first 3 rules, and the 4th as-is (dropping any extra "X"s (or "Q"s) that don't make sense in the final message when finished).

Example:

Using "playfair example" as the key, (assuming I and J are interchangeable) the table becomes:

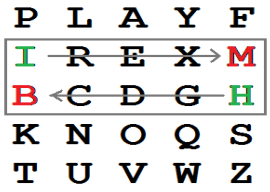


P L A Y F
I R E X M
B C D G H
K N O Q S
T U V W Z

Encrypting the message "Hide the gold in the tree stump":

HI DE TH EG OL DI NT HE TR EX ES TU MP
^

1. The pair HI forms a rectangle, replace it with BM

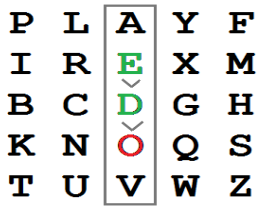


HI

Shape: Rectangle
Rule: Pick Same Rows, Opposite Corners

BM

2. The pair DE is in a column, replace it with OD

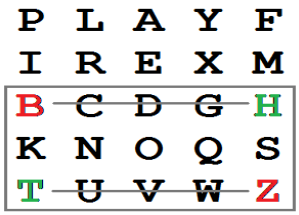


DE

Shape: Column
Rule: Pick Items Below Each Letter, Wrap to Top if Needed

OD

3. The pair TH forms a rectangle, replace it with ZB

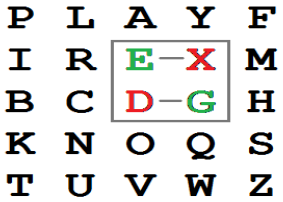


TH

Shape: Rectangle
Rule: Pick Same Rows, Opposite Corners

ZB

4. The pair EG forms a rectangle, replace it with XD

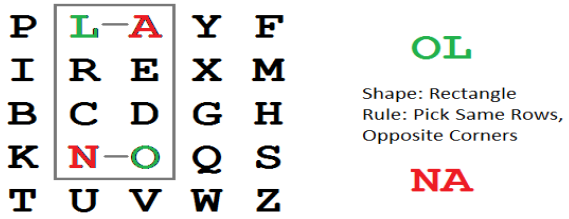


EG

Shape: Rectangle
Rule: Pick Same Rows, Opposite Corners

XD

5. The pair OL forms a rectangle, replace it with NA



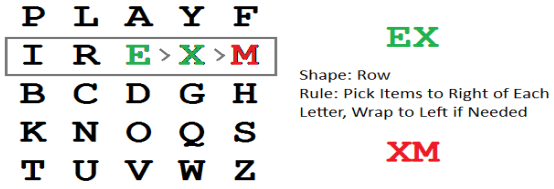
6. The pair DI forms a rectangle, replace it with BE

7. The pair NT forms a rectangle, replace it with KU

8. The pair HE forms a rectangle, replace it with DM

9. The pair TR forms a rectangle, replace it with UI

10. The pair EX (X inserted to split EE) is in a row, replace it with XM



11. The pair ES forms a rectangle, replace it with MO

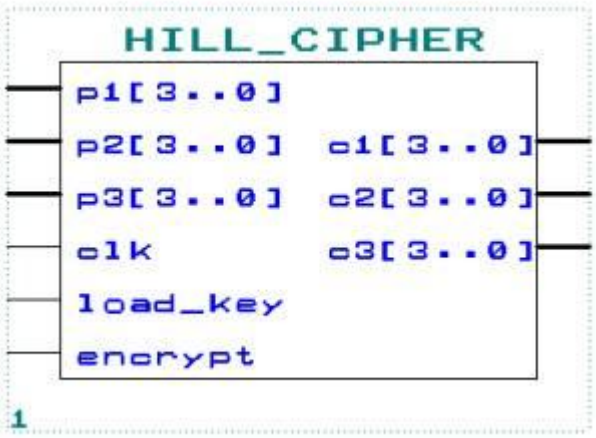
12. The pair TU is in a row, replace it with UV

13. The pair MP forms a rectangle, replace it with IF

BM OD ZB XD NA BE KU DM UI XM MO UV IF

Thus the message "Hide the gold in the tree stump" becomes "BMODZBXDNABEKUDMUIXMMOUVIF".

HILL CIPHER ENCYPTION-DECRYPTION



EXPERIMENT NO. 5

AIM: To implement Hill Cipher Encyption-Decryption .

THEORY:

In classical cryptography, the **Hill cipher** is a polygraphic substitution cipher based on linear algebra

Operation

Each letter is first encoded as a number. Often the simplest scheme is used: A = 0, B =1, ..., Z=25, but this is not an essential feature of the cipher. A block of *n* letters is then considered as a vector of n dimensions, and multiplied by an n × n matrix, modulo 26. (If one uses a larger number than 26 for the modular base, then a different number scheme can be used to encode the letters, and spaces or punctuation can also be used.) The whole matrix is considered the cipher key, and should be random provided that the matrix is invertible in $\mathbb{Z}_{26}^n$(to ensure decryption is possible). A Hill cipher is another way of working out the equation of a matrix.

Consider the message 'ACT', and the key below (or GYBNQKURP in letters):

$$\begin{pmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{pmatrix}$$

Since 'A' is 0, 'C' is 2 and 'T' is 19, the message is the vector:

$$\begin{pmatrix} 0 \\ 2 \\ 19 \end{pmatrix}$$

Thus the enciphered vector is given by:

$$\begin{pmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{pmatrix} \begin{pmatrix} 0 \\ 2 \\ 19 \end{pmatrix} = \begin{pmatrix} 67 \\ 222 \\ 319 \end{pmatrix} \equiv \begin{pmatrix} 15 \\ 14 \\ 7 \end{pmatrix} \pmod{26}$$

which corresponds to a ciphertext of 'POH'. Now, suppose that our message is instead 'CAT', or:

$$\begin{pmatrix} 2 \\ 0 \\ 19 \end{pmatrix}$$

This time, the enciphered vector is given by:

$$\begin{pmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{pmatrix} \begin{pmatrix} 2 \\ 0 \\ 19 \end{pmatrix} \equiv \begin{pmatrix} 31 \\ 216 \\ 325 \end{pmatrix} \equiv \begin{pmatrix} 5 \\ 8 \\ 13 \end{pmatrix} \pmod{26}$$

which corresponds to a ciphertext of 'FIN'. Every letter has changed. The Hill cipher has achieved Shannon's diffusion, and an n-dimensional Hill cipher can diffuse fully across n symbols at once.

Decryption

In order to decrypt, we turn the ciphertext back into a vector, then simply multiply by the inverse matrix of the key matrix (IFKVIVVMI in letters). (There are standard methods to calculate the inverse matrix; see matrix inversion for details.) We find that in $\mathbb{Z}_{26}^n$ the inverse matrix of the one in the previous example is:

$$\begin{pmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{pmatrix}^{-1} \equiv \begin{pmatrix} 8 & 5 & 10 \\ 21 & 8 & 21 \\ 21 & 12 & 8 \end{pmatrix} \pmod{26}$$

Taking the previous example ciphertext of 'POH', we get:

$$\begin{pmatrix} 8 & 5 & 10 \\ 21 & 8 & 21 \\ 21 & 12 & 8 \end{pmatrix} \begin{pmatrix} 15 \\ 14 \\ 7 \end{pmatrix} \equiv \begin{pmatrix} 260 \\ 574 \\ 539 \end{pmatrix} \equiv \begin{pmatrix} 0 \\ 2 \\ 19 \end{pmatrix} \pmod{26}$$

which gets us back to 'ACT', just as we hoped.

We have not yet discussed one complication that exists in picking the encrypting matrix. Not all matrices have an inverse (see invertible matrix). The matrix will have an inverse if and only if its determinant is not zero, and does not have any common factors with the modular base. Thus, if we work modulo 26 as above, the determinant must be nonzero, and must not be divisible by 2 or 13. If the determinant is 0, or has common factors with the modular base, then the matrix cannot be used in the Hill cipher, and another matrix must be chosen (otherwise it will not be possible to decrypt). Fortunately, matrices which satisfy the conditions to be used in the Hill cipher are fairly common.

For our example key matrix:

$$\begin{vmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{vmatrix} \equiv 6(16\cdot 15 - 10\cdot 17) - 24(13\cdot 15 - 10\cdot 20) + 1(13\cdot 17 - 16\cdot 20) \equiv 441 \equiv 25 \pmod{26}$$

So, modulo 26, the determinant is 25. Since this has no common factors with 26, this matrix can be used for the Hill cipher.

The risk of the determinant having common factors with the modulus can be eliminated by making the modulus prime. Consequently a useful variant of the Hill cipher adds 3 extra symbols (such as a space, a period and a question mark) to increase the modulus to 29.

Example:

Let

$$K = \begin{vmatrix} 3 & 3 \\ 2 & 5 \end{vmatrix}$$

and suppose the plaintext message is HELP. Then this plaintext is represented by two pairs

$$HELP \rightarrow \begin{vmatrix} H \\ E \end{vmatrix}, \begin{vmatrix} L \\ P \end{vmatrix} \rightarrow \begin{vmatrix} 7 \\ 4 \end{vmatrix}, \begin{vmatrix} 11 \\ 15 \end{vmatrix}$$

Then we compute

$$\begin{vmatrix} 3 & 3 \\ 2 & 5 \end{vmatrix} \begin{vmatrix} 7 \\ 4 \end{vmatrix} = \begin{vmatrix} 7 \\ 8 \end{vmatrix}, \begin{vmatrix} 3 & 3 \\ 2 & 5 \end{vmatrix} \begin{vmatrix} 11 \\ 15 \end{vmatrix} = \begin{vmatrix} 0 \\ 19 \end{vmatrix}$$

and continue encryption as follows:

$$\begin{vmatrix} 7 \\ 8 \end{vmatrix}, \begin{vmatrix} 0 \\ 19 \end{vmatrix} \rightarrow \begin{vmatrix} H \\ I \end{vmatrix}, \begin{vmatrix} A \\ T \end{vmatrix}$$

The matrix K is invertible, hence K^{-1} exists such that $KK^{-1} = K^{-1}K - I_2$. To implement decrypting, we compute

$$K^{-1} = 9^{-1} \begin{vmatrix} 5 & 23 \\ 24 & 3 \end{vmatrix} = 3 \begin{vmatrix} 5 & 23 \\ 24 & 3 \end{vmatrix} = \begin{vmatrix} 15 & 17 \\ 20 & 9 \end{vmatrix}$$

$$HIAT \rightarrow \begin{vmatrix} H \\ I \end{vmatrix}, \begin{vmatrix} A \\ T \end{vmatrix} \rightarrow \begin{vmatrix} 7 \\ 8 \end{vmatrix}, \begin{vmatrix} 0 \\ 19 \end{vmatrix}$$

Then we compute

$$\begin{vmatrix} 15 & 17 \\ 20 & 9 \end{vmatrix} \begin{vmatrix} 7 \\ 8 \end{vmatrix} = \begin{vmatrix} 7 \\ 4 \end{vmatrix}, \begin{vmatrix} 15 & 17 \\ 20 & 9 \end{vmatrix} \begin{vmatrix} 0 \\ 19 \end{vmatrix} = \begin{vmatrix} 11 \\ 15 \end{vmatrix}$$

Therefore

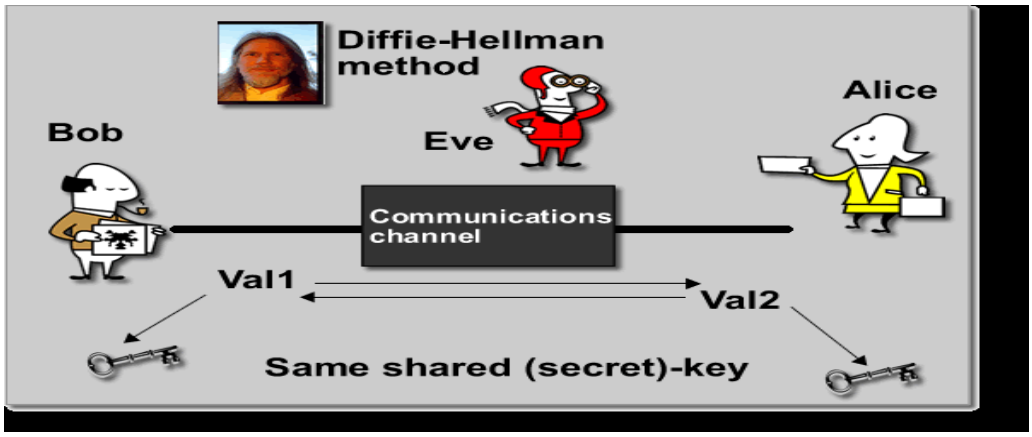
$$\begin{vmatrix} 7 \\ 4 \end{vmatrix}, \begin{vmatrix} 11 \\ 15 \end{vmatrix} \rightarrow \begin{vmatrix} H \\ E \end{vmatrix}, \begin{vmatrix} L \\ P \end{vmatrix} \rightarrow HELP.$$

Security Issue:

Unfortunately, the basic Hill cipher is vulnerable to a known-plaintext attack because it is completely linear. An opponent who intercepts n^2 plaintext/ciphertext character pairs can set up a linear system which can (usually) be easily solved; if it happens that this system is indeterminate, it is only necessary to add a few more plaintext/ciphertext pairs. Calculating this solution by standard linear algebra algorithms then takes very little time.

While matrix multiplication alone does not result in a secure cipher it is still a useful step when combined with other non-linear operations, because matrix multiplication can provide diffusion. For example, an appropriately chosen matrix can guarantee that small differences before the matrix multiplication will result in large differences after the matrix multiplication. Some modern ciphers use indeed a matrix multiplication step to provide diffusion. For example, the MixColumns step in AES is a matrix multiplication. The function *g* in Twofish is a combination of non-linear S-boxes with a carefully chosen matrix multiplication (MDS). Recently, some publications tried to make the Hill cipher secure.

DIFFIE HELLMAN KEY EXCHANGE



EXPERIMENT NO. 6

AIM: To implement Diffie Hellman Key Exchange.

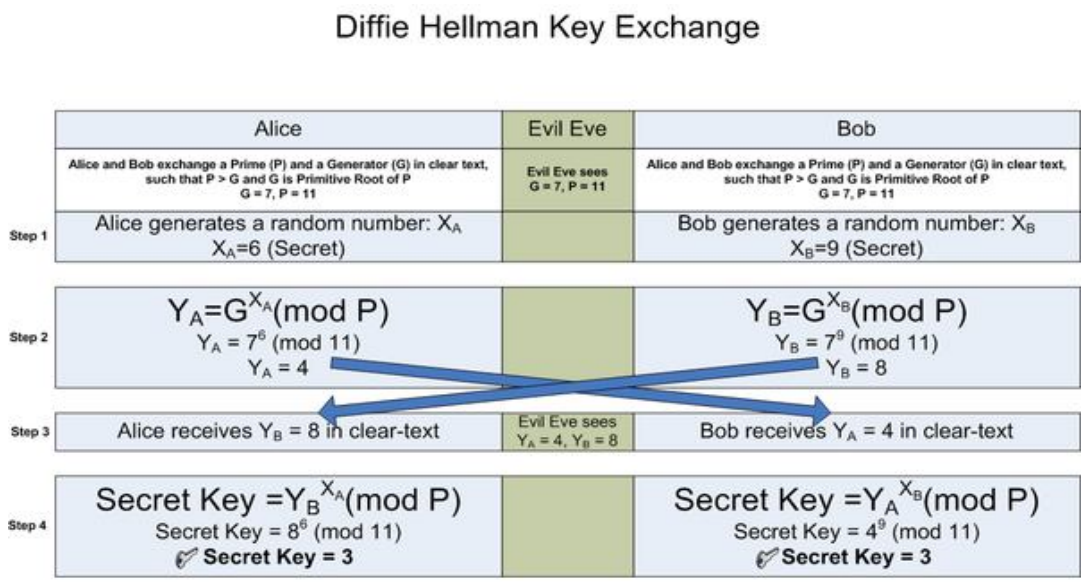
THEORY:

Diffie–Hellman key exchange (**D–H**)[nb 1] is a specific method of exchanging keys. It is one of the earliest practical examples of Key exchange implemented within the field of cryptography. The Diffie–Hellman key exchange method allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure communications channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher.

Description:

Diffie–Hellman establishes a shared secret that can be used for secret communications by exchanging data over a public network.

Here is an explanation which includes the encryption's mathematics:



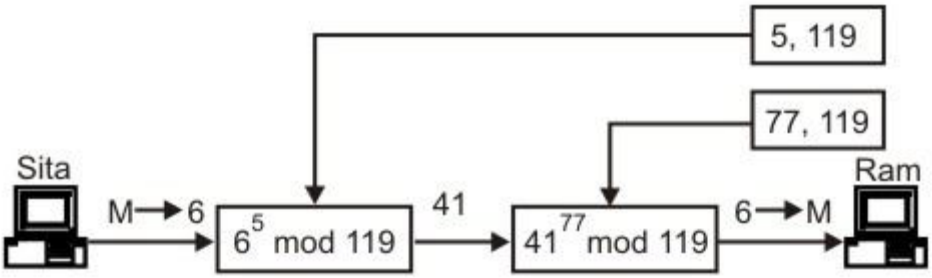
The simplest, and original, implementation of the protocol uses the multiplicative group of integers modulo *p*, where *p* is prime and *g* is primitive root mod *p*.

Security

The protocol is considered secure against eavesdroppers if *G* and *g* are chosen properly. The eavesdropper ("Eve") would have to solve the Diffie–Hellman problem to obtain g^{ab}. This is currently considered difficult. An efficient algorithm to solve the discrete logarithm problem would make it easy to compute *a* or *b* and solve the Diffie–Hellman problem, making this and many other public key cryptosystems insecure.

The Diffie–Hellman exchange by itself does not provide authentication of the communicating parties and is thus vulnerable to a man-in-the-middle attack. A person in the middle may establish two distinct Diffie–Hellman key exchanges, one with Alice and the other with Bob, effectively masquerading as Alice to Bob, and vice versa, allowing the attacker to decrypt (and read or store) then re-encrypt the messages passed between them. A method to authenticate the communicating parties to each other is generally needed to prevent this type of attack.

RSA ENCRYPTION-DECRYPTION



EXPERIMENT NO. 7

AIM: To implement RSA Encryption-Decryption.

THEORY:

In cryptography, **RSA** (which stands for Rivest, Shamir and Adleman who first publicly described it) is an algorithm for public-key cryptography. It is the first algorithm known to be suitable for signing as well as encryption, and was one of the first great advances in public key cryptography. RSA is widely used in electronic commerce protocols, and is believed to be secure given sufficiently long keys and the use of up-to-date implementations.

The RSA algorithm involves three steps: key generation, encryption and decryption.

Key generation

RSA involves a **public key** and a **private key.** The public key can be known to everyone and is used for encrypting messages. Messages encrypted with the public key can only be decrypted using the private key. The keys for the RSA algorithm are generated the following way:

1. 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. Prime integers can be efficiently found using a primality test.
2. Compute $n = pq$.
   - n is used as the modulus for both the public and private keys
3. Compute $\varphi(n) = (p-1)(q-1)$, where φ is Euler's totient function.
4. Choose an integer e such that $1 < e < \varphi(n)$ and $\gcd(e,\varphi(n)) = 1$, i.e. e and $\varphi(n)$ are coprime.
   - e is released as the public key exponent.
   - e having a short bit-length and small Hamming weight results in more efficient encryption - most commonly 0x10001 = 65537. However, small values of e (such as 3) have been shown to be less secure in some settings.[4]
5. Determine $d = e^{-1} \bmod \varphi(n)$; i.e. d is the multiplicative inverse of e mod $\varphi(n)$.
   - This is often computed using the extended Euclidean algorithm.
   - d is kept as the private key exponent.

The **public key** consists of the modulus n and the public (or encryption) exponent e. The **private key** consists of the private (or decryption) exponent d which must be kept secret.

Notes:

- An alternative, used by PKCS#1, is to choose d matching $de \equiv 1 \bmod \lambda$ with $\lambda = \operatorname{lcm}(p-1, q-1)$, where lcm is the least common multiple. Using λ instead of $\varphi(n)$ allows more choices for d. λ can also be defined using the Carmichael function, $\lambda(n)$.
- The ANSI X9.31 standard prescribes, IEEE 1363 describes, and PKCS#1 allows, that p and q match additional requirements: be strong primes, and be different enough that Fermat factorization fails.

Encryption

Alice transmits her public key (n,e) to Bob and keeps the private key secret. Bob then wishes to send message **M** to Alice.

He first turns **M** into an integer $0 < m < n$ by using an agreed-upon reversible protocol known as a padding scheme. He then computes the ciphertext c corresponding to

$$c = m^e \pmod{n}.$$

This can be done quickly using the method of exponentiation by squaring. Bob then transmits c to Alice.

Decryption

Alice can recover m from c by using her private key exponent d via computing

$$m = c^d \pmod{n}.$$

Given m, she can recover the original message **M** by reversing the padding scheme.

(In practice, there are more efficient methods of calculating c^d using the pre computed values below.)

Security:

The security of the RSA cryptosystem is based on two mathematical problems: the problem of factoring large numbers and the RSA problem. Full decryption of an RSA ciphertext is thought to be infeasible on the assumption that both of these problems are hard, i.e., no efficient algorithm exists for solving them. Providing security against *partial* decryption may require the addition of a secure padding scheme.

3-DES ENCRYPTION-DECRYPTION.

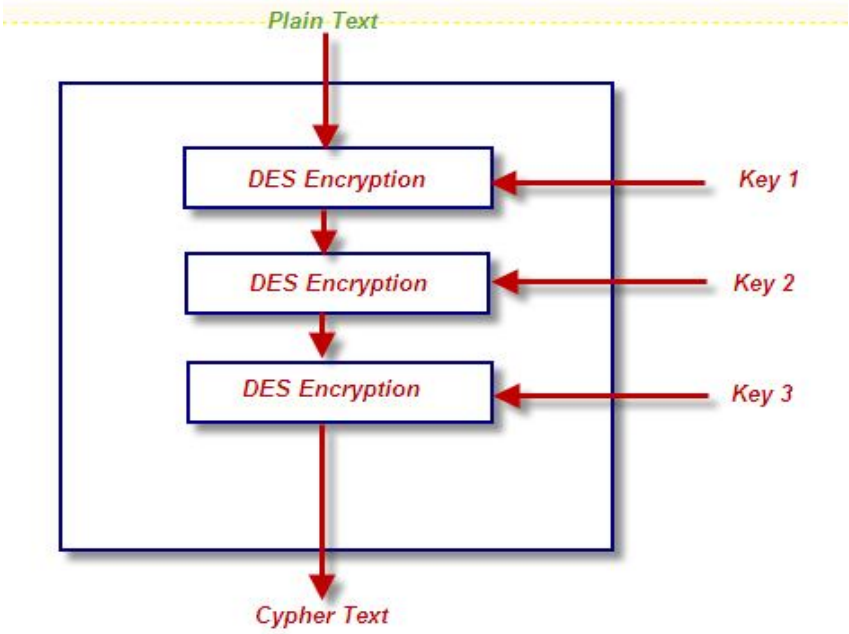


Figure: Working Of Triple DES Algorithm

EXPERIMENT NO. 8

AIM: To implement TRIPLE-DES Encryption-Decryption.

THEORY:

Data Encryption Standard (DES)

– The most widely used encryption scheme.
– Known as the Data Encryption Algorithm (DEA).
– It is a block cipher.
- The plaintext is 64-bits in length.
- The key is 56-bits in length.
- Longer plaintexts are processed in 64-bit blocks.

Triple DES

In cryptography, **Triple DES** (**3DES**) is the common name for the Triple Data Encryption Algorithm (TDEA or Triple DEA) block cipher, which applies the Data Encryption Standard (DES) cipher algorithm three times to each data block. Because of the availability of increasing computational power, the key size of the original DES cipher was becoming subject to brute force attacks; Triple DES was designed to provide a relatively simple method of increasing the key size of DES to protect against such attacks, without designing a completely new block cipher algorithm.

Algorithm:

Triple DES uses a "key bundle" which comprises three DES keys, K_1, K_2 and K_3, each of 56 bits (excluding parity bits). The encryption algorithm is:

$$\text{ciphertext} = E_{K3}(D_{K2}(E_{K1}(\text{plaintext})))$$

I.e., DES encrypt with K_1, DES *decrypt* with K_2, then DES encrypt with K_3.

Decryption is the reverse:

$$\text{plaintext} = D_{K1}(E_{K2}(D_{K3}(\text{ciphertext})))$$

I.e., decrypt with K_3, *encrypt* with K_2, then decrypt with K_1.

Each triple encryption encrypts one block of 64 bits of data.

In each case the middle operation is the reverse of the first and last. This improves the strength of the algorithm when using keying option 2, and provides backward compatibility with DES with keying option 3.

Keying options:

The standards define three keying options:

- Keying option 1: All three keys are independent.
- Keying option 2: K_1 and K_2 are independent, and $K_3 = K_1$.
- Keying option 3: All three keys are identical, i.e. $K_1 = K_2 = K_3$.

Keying option 1 is the strongest, with $3 \times 56 = 168$ independent key bits.

Keying option 2 provides less security, with $2 \times 56 = 112$ key bits. This option is stronger than simply DES encrypting twice, e.g. with K_1 and K_2, because it protects against meet-in-the-middle attacks.

Keying option 3 is equivalent to DES, with only 56 key bits. This option provides backward compatibility with DES, because the first and second DES operations cancel out. It is no longer recommended by the National Institute of Standards and Technology (NIST), and is not supported by ISO/IEC 18033-3.

Security Issue:

In general Triple DES with three independent keys (keying option 1) has a key length of 168 bits (three 56-bit DES keys), but due to the meet-in-the-middle attack the effective security it provides is only 112 bits. Keying option 2 reduces the key size to 112 bits. However, this option is susceptible to certain chosen-plaintext or known-plaintext attacks and thus it is designated by NIST to have only 80 bits of security.

The best attack known on keying option 1 requires around 2^{32} known plaintexts, 2^{113} steps, 2^{90} single DES encryptions, and 2^{88} memory (the paper presents other tradeoffs between time and memory). This is not currently practical and NIST considers keying option 1 to be appropriate through 2030.If the attacker seeks to discover any one of many cryptographic keys, there is a memory-efficient attack which will discover one of 2^{28} keys, given a handful of chosen plaintexts per key and around 2^{84} encryption operations.

CASE STUDY: Digital Signature

A **digital signature** or **digital signature scheme** is a mathematical scheme for demonstrating the authenticity of a digital message or document. A valid digital signature gives a recipient reason to believe that the message was created by a known sender, and that it was not altered in transit. Digital signatures are commonly used for software distribution, financial transactions, and in other cases where it is important to detect forgery or tampering.

Digital signatures employ a type of asymmetric cryptography. For messages sent through a nonsecure channel, a properly implemented digital signature gives the receiver reason to believe the message was sent by the claimed sender. Digital signatures are equivalent to traditional handwritten signatures in many respects; properly implemented digital signatures are more difficult to forge than the handwritten type. Digital signature schemes in the sense used here are cryptographically based, and must be implemented properly to be effective. Digital signatures can also provide non-repudiation, meaning that the signer cannot successfully claim they did not sign a message, while also claiming their private key remains secret; further, some non-repudiation schemes offer a time stamp for the digital signature, so that even if the private key is exposed, the signature is valid nonetheless. Digitally signed messages may be anything representable as a bitstring: examples include electronic mail, contracts, or a message sent via some other cryptographic protocol.

A digital signature scheme typically consists of three algorithms:

- A *key generation* algorithm that selects a *private key* uniformly at random from a set of possible private keys. The algorithm outputs the private key and a corresponding *public key*.
- A *signing* algorithm that, given a message and a private key, produces a signature.
- A *signature verifying* algorithm that, given a message, public key and a signature, either accepts or rejects the message's claim to authenticity.

Two main properties are required. First, a signature generated from a fixed message and fixed private key should verify the authenticity of that message by using the corresponding public key. Secondly, it should be computationally infeasible to generate a valid signature for a party who does not possess the private key.

Uses of digital signatures:

Below are some common reasons for applying a digital signature to communications:

Authentication

Although messages may often include information about the entity sending a message, that information may not be accurate. Digital signatures can be used to authenticate the source of messages. When ownership of a digital signature secret key is bound to a specific user, a valid signature shows that the message was sent by that user. The importance of high confidence in sender authenticity is especially obvious in a financial context. For example, suppose a bank's branch office sends instructions to the central office requesting a change in the balance of an account. If the central

office is not convinced that such a message is truly sent from an authorized source, acting on such a request could be a grave mistake.

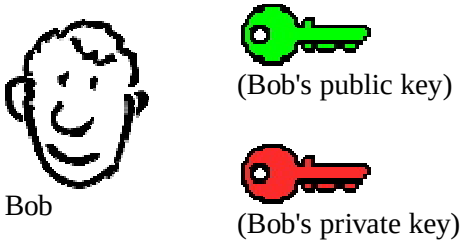
Integrity

In many scenarios, the sender and receiver of a message may have a need for confidence that the message has not been altered during transmission. Although encryption hides the contents of a message, it may be possible to *change* an encrypted message without understanding it. (Some encryption algorithms, known as nonmalleable ones, prevent this, but others do not.) However, if a message is digitally signed, any change in the message after signature will invalidate the signature. Furthermore, there is no efficient way to modify a message and its signature to produce a new message with a valid signature, because this is still considered to be computationally infeasible by most cryptographic hash functions.

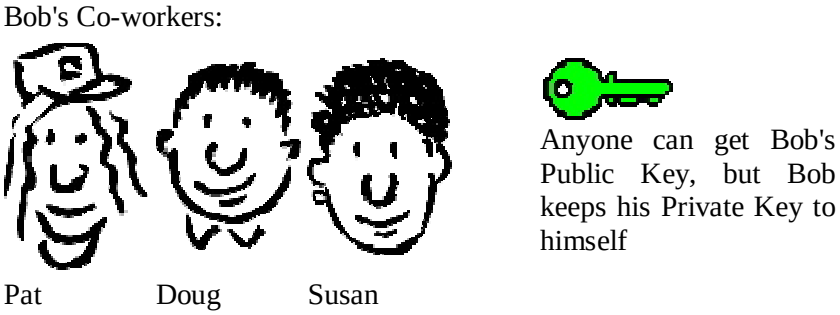
Non-repudiation

Non-repudiation, or more specifically *non-repudiation of origin*, is an important aspect of digital signatures. By this property an entity that has signed some information cannot at a later time deny having signed it. Similarly, access to the public key only does not enable a fraudulent party to fake a valid signature. This is in contrast to symmetric systems, where both sender and receiver share the same secret key, and thus in a dispute a third party cannot determine which entity was the true source of the information.

Example: What is a Digital Signature?



Bob has been given two keys. One of Bob's keys is called a Public Key, the other is called a Private Key.



Bob's Public key is available to anyone who needs it, but he keeps his Private Key to himself. Keys are used to encrypt information. Encrypting information means "scrambling it up", so that only a

person with the appropriate key can make it readable again. Either one of Bob's two keys can encrypt data, and the other key can decrypt that data.

Susan (shown below) can encrypt a message using Bob's Public Key. Bob uses his Private Key to decrypt the message. Any of Bob's co-workers might have access to the message Susan encrypted, but without Bob's Private Key, the data is worthless.

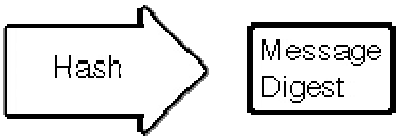
"Hey Bob, how about lunch at Taco Bell. I hear they have free refills!"

HNFmsEm6Un
BejhhyCGKOK
JUxhiygSBCEiC
0QYIh/Hn3xgiK
BcyLK1UcYiY
lxx2lCFHDC/A

HNFmsEm6Un
BejhhyCGKOK
JUxhiygSBCEiC
0QYIh/Hn3xgiK
BcyLK1UcYiY
lxx2lCFHDC/A

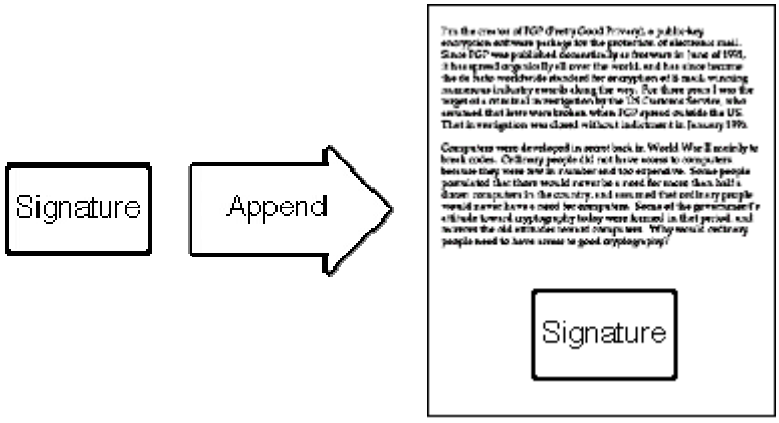
"Hey Bob, how about lunch at Taco Bell. I hear they have free refills!"

With his private key and the right software, Bob can put digital signatures on documents and other data. A digital signature is a "stamp" Bob places on the data which is unique to Bob, and is very difficult to forge. In addition, the signature assures that any changes made to the data that has been signed can not go undetected.

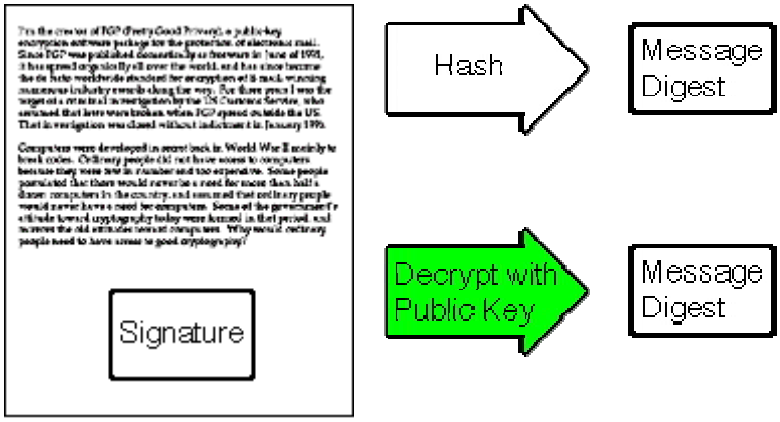


To sign a document, Bob's software will crunch down the data into just a few lines by a process called "hashing". These few lines are called a message digest. (It is not possible to change a message digest back into the original data from which it was created.)

Bob's software then encrypts the message digest with his private key. The result is the digital signature.



Finally, Bob's software appends the digital signature to document. All of the data that was hashed has been signed.



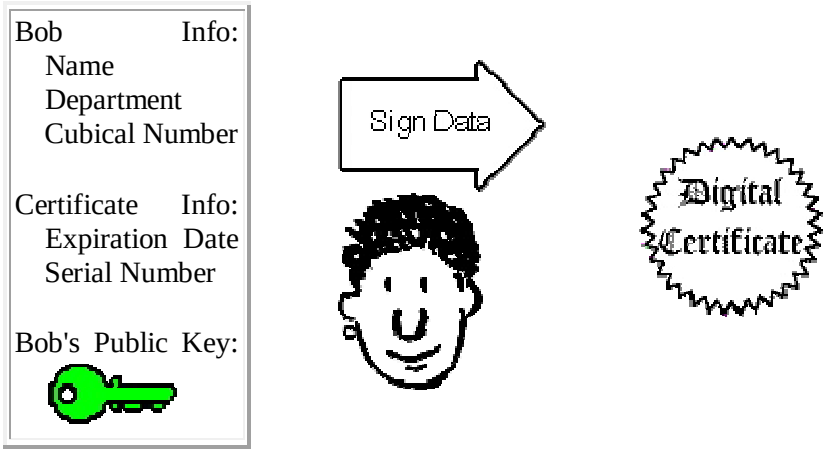
Bob now passes the document on to Pat.

First, Pat's software decrypts the signature (using Bob's public key) changing it back into a message digest. If this worked, then it proves that Bob signed the document, because only Bob has his private key. Pat's software then hashes the document data into a message digest. If the message digest is the same as the message digest created when the signature was decrypted, then Pat knows that the signed data has not been changed.

Plot complication...

Doug (our disgruntled employee) wishes to deceive Pat. Doug makes sure that Pat receives a signed message and a public key that appears to belong to Bob. Unbeknownst to Pat, Doug deceitfully sent a key pair he created using Bob's name. Short of receiving Bob's public key from him in person, how can Pat be sure that Bob's public key is authentic?

It just so happens that Susan works at the company's certificate authority center. Susan can create a digital certificate for Bob simply by signing Bob's public key as well as some information about Bob.



Now Bob's co-workers can check Bob's trusted certificate to make sure that his public key truly belongs to him. In fact, no one at Bob's company accepts a signature for which there does not exist a certificate generated by Susan. This gives Susan the power to revoke signatures if private keys are compromised, or no longer needed. There are even more widely accepted certificate authorities that certify Susan.

Let's say that Bob sends a signed document to Pat. To verify the signature on the document, Pat's software first uses Susan's (the certificate authority's) public key to check the signature on Bob's certificate. Successful de-encryption of the certificate proves that Susan created it. After the certificate is de-encrypted, Pat's software can check if Bob is in good standing with the certificate authority and that all of the certificate information concerning Bob's identity has not been altered.

Pat's software then takes Bob's public key from the certificate and uses it to check Bob's signature. If Bob's public key de-encrypts the signature successfully, then Pat is assured that the signature was created using Bob's private key, for Susan has certified the matching public key. And of course, if the signature is valid, then we know that Doug didn't try to change the signed content.

CASE STUDY: Java Security Features / Mat Lab Security Features

Why security?

Java's security model is one of the language's key architectural features that make it an appropriate technology for networked environments. Security is important because networks provide a potential avenue of attack to any computer hooked to them. This concern becomes especially strong in an environment in which software is downloaded across the network and executed locally, as is done with Java applets, for example. Because the class files for an applet are automatically downloaded when a user goes to the containing Web page in a browser, it is likely that a user will encounter applets from untrusted sources. Without any security, this would be a convenient way to spread viruses. Thus, Java's security mechanisms help make Java suitable for networks because they establish a needed trust in the safety of network-mobile code.

Java's security model is focused on protecting users from hostile programs downloaded from untrusted sources across a network. To accomplish this goal, Java provides a customizable "sandbox" in which Java programs run. A Java program must play only inside its sandbox. It can do anything within the boundaries of its sandbox, but it can't take any action outside those boundaries. The sandbox for untrusted Java applets, for example, prohibits many activities, including:

- Reading or writing to the local disk
- Making a network connection to any host, except the host from which the applet came
- Creating a new process
- Loading a new dynamic library and directly calling a native method

By making it impossible for downloaded code to perform certain actions, Java's security model protects the user from the threat of hostile code.

Java Security Overview

1 Introduction

The Java™ platform was designed with a strong emphasis on security. At its core, the Java language itself is type-safe and provides automatic garbage collection, enhancing the robustness of application code. A secure class loading and verification mechanism ensures that only legitimate Java code is executed.

The initial version of the Java platform created a safe environment for running potentially untrusted code, such as Java applets downloaded from a public network. As the platform has grown and widened its range of deployment, the Java security architecture has correspondingly evolved to support an increasing set of services. Today the architecture includes a large set of application programming interfaces (APIs), tools, and implementations of commonly-used security algorithms, mechanisms, and protocols. This provides the developer a comprehensive security framework for writing applications, and also provides the user or administrator a set of tools to securely manage applications.

The Java security APIs span a wide range of areas. Cryptographic and public key infrastructure (PKI) interfaces provide the underlying basis for developing secure applications. Interfaces for performing authentication and access control enable applications to guard against unauthorized access to protected resources.

The APIs allow for multiple interoperable implementations of algorithms and other security services. Services are implemented in *providers*, which are plugged into the Java platform via a standard interface that makes it easy for applications to obtain security services without having to know anything about their implementations. This allows developers to focus on how to integrate security into their applications, rather than on how to actually implement complex security mechanisms.

The Java platform includes a number of providers that implement a core set of security services. It also allows for additional custom providers to be installed. This enables developers to extend the platform with new security mechanisms.

This paper gives a broad overview of security in the Java platform, from secure language features to the security APIs, tools, and built-in provider services, highlighting key packages and classes where applicable. Note that this paper is based on Java™ SE version 6.

2 Java Language Security and Bytecode Verification

The Java language is designed to be type-safe and easy to use. It provides automatic memory management, garbage collection, and range-checking on arrays. This reduces the overall programming burden placed on developers, leading to fewer subtle programming errors and to safer, more robust code.

In addition, the Java language defines different access modifiers that can be assigned to Java classes, methods, and fields, enabling developers to restrict access to their class implementations as appropriate. Specifically, the language defines four distinct access levels: private, protected, public, and, if unspecified, package. The most open access specifier is public access is allowed to anyone. The most restrictive modifier is private access is not allowed outside the particular class in which the private member (a method, for example) is defined. The protected modifier allows access to any subclass, or to other classes within the same package. Package-level access only allows access to classes within the same package.

A compiler translates Java programs into a machine-independent bytecode representation. A bytecode verifier is invoked to ensure that only legitimate bytecodes are executed in the Java runtime. It checks that the bytecodes conform to the Java Language Specification and do not violate Java language rules or namespace restrictions. The verifier also checks for memory management violations, stack underflows or overflows, and illegal data typecasts. Once bytecodes have been verified, the Java runtime prepares them for execution.

3 Basic Security Architecture

The Java platform defines a set of APIs spanning major security areas, including cryptography, public key infrastructure, authentication, secure communication, and access control. These APIs allow developers to easily integrate security into their application code. They were designed around the following principles:

1. Implementation independence Applications do not need to implement security themselves. Rather, they can request security services from the Java platform. Security services are implemented in providers (see below), which are plugged into the Java platform via a standard interface. An application may rely on multiple independent providers for security functionality.
2. Implementation interoperability Providers are interoperable across applications. Specifically, an application is not bound to a specific provider, and a provider is not bound to a specific application.
3. Algorithm extensibility The Java platform includes a number of built-in providers that implement a basic set of security services that are widely used today. However, some

applications may rely on emerging standards not yet implemented, or on proprietary services. The Java platform supports the installation of custom providers that implement such services.

Security Providers

The java.security.Provider class encapsulates the notion of a security provider in the Java platform. It specifies the provider's name and lists the security services it implements. Multiple providers may be configured at the same time, and are listed in order of preference. When a security service is requested, the highest priority provider that implements that service is selected.

Applications rely on the relevant getInstance method to obtain a security service from an underlying provider. For example, message digest creation represents one type of service available from providers. (Section 4 discusses message digests and other cryptographic services.) An application invokes the getInstance method in the java.security.MessageDigest class to obtain an implementation of a specific message digest algorithm, such as MD5.

MessageDigest md = MessageDigest.getInstance("MD5");

The program may optionally request an implementation from a specific provider, by indicating the provider name, as in the following:

MessageDigest md = MessageDigest.getInstance("MD5", "ProviderC");

Figures 1 and 2 illustrate these options for requesting an MD5 message digest implementation. Both figures show three providers that implement message digest algorithms. The providers are ordered by preference from left to right (1-3). In Figure 1, an application requests an MD5 algorithm implementation without specifying a provider name. The providers are searched in preference order and the implementation from the first provider supplying that particular algorithm, ProviderB, is returned. In Figure 2, the application requests the MD5 algorithm implementation from a specific provider, ProviderC. This time the implementation from that provider is returned, even though a provider with a higher preference order, ProviderB, also supplies an MD5 implementation.

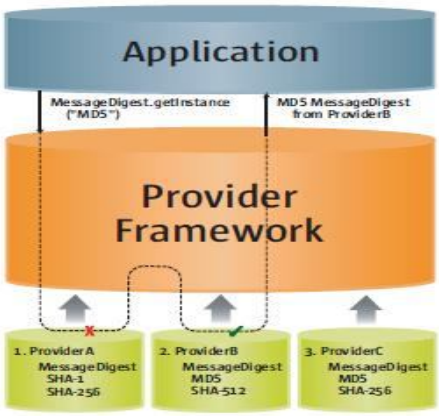


Figure 1 Provider searching

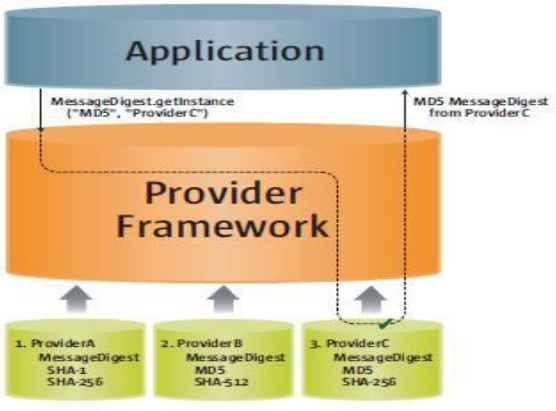


Figure 2 Specific provider requested

The Java platform implementation from Sun Microsystems includes a number of pre-configured default providers that implement a basic set of security services that can be used by applications. Note that other vendor implementations of the Java platform may include different sets of providers that

encapsulate vendor-specific sets of security services. When this paper mentions built-in default providers, it is referencing those available in Sun's implementation.

The sections below on the various security areas (cryptography, authentication, etc.) each include descriptions of the relevant services supplied by the default providers. A table in Appendix C summarizes all of the default providers.

File Locations

Certain aspects of Java security mentioned in this paper, including the configuration of providers, may be customized by setting security properties. You may set security properties statically in the security properties file, which by default is the *java.security* file in the *lib/security* directory of the directory where the Java™ Runtime Environment (JRE) is installed. Security properties may also be set dynamically by calling appropriate methods of the Security class (in the java.security package).

The tools and commands mentioned in this paper are all in the ~jre/bin directory, where ~jre stands for the directory in which the JRE is installed. The *cacerts* file mentioned in Section 5 is in ~jre/lib/security.

4 Cryptography

The Java cryptography architecture is a framework for accessing and developing cryptographic functionality for the Java platform. It includes APIs for a large variety of cryptographic services, including

- Message digest algorithms
- Digital signature algorithms
- Symmetric bulk encryption
- Symmetric stream encryption
- Asymmetric encryption
- Password-based encryption (PBE)
- Elliptic Curve Cryptography (ECC)
- Key agreement algorithms
- Key generators
- Message Authentication Codes (MACs)
- (Pseudo-)random number generators

For historical (export control) reasons, the cryptography APIs are organized into two distinct packages. The java.security package contains classes that are *not* subject to export controls (like Signature and MessageDigest). The javax.crypto package contains classes that are subject to export controls (like Cipher and KeyAgreement).

The cryptographic interfaces are provider-based, allowing for multiple and interoperable cryptography implementations. Some providers may perform cryptographic operations in software; others may perform the operations on a hardware token (for example, on a smartcard device or on a hardware cryptographic accelerator). Providers that implement export-controlled services must be digitally signed.

The Java platform includes built-in providers for many of the most commonly used cryptographic algorithms, including the RSA and DSA signature algorithms, the DES, AES, and ARCFOUR encryption algorithms, the MD5 and SHA-1 message digest algorithms, and the Diffie-Hellman key agreement algorithm. These default providers implement cryptographic algorithms in Java code.

The Java platform also includes a built-in provider that acts as a bridge to a native PKCS#11 (v2.x) token. This provider, named SunPKCS11, allows Java applications to seamlessly access cryptographic services located on PKCS#11-compliant tokens.

5 Public Key Infrastructure

Public Key Infrastructure (PKI) is a term used for a framework that enables secure exchange of information based on public key cryptography. It allows identities (of people, organizations, etc.) to be bound to digital certificates and provides a means of verifying the authenticity of certificates. PKI encompasses keys, certificates, public key encryption, and trusted Certification Authorities (CAs) who generate and digitally sign certificates.

The Java platform includes API and provider support for X.509 digital certificates and certificate revocation lists (CRLs), as well as PKIX-compliant certification path building and validation. The classes related to PKI are located in the java.security and java.security.cert packages.

Key and Certificate Storage

The Java platform provides for long-term persistent storage of cryptographic keys and certificates via key and certificate stores. Specifically, the java.security.KeyStore class represents a *key store*, a secure repository of cryptographic keys and/or trusted certificates (to be used, for example, during certification path validation), and the java.security.cert.CertStore class represents a *certificate store*, a public and potentially vast repository of unrelated and typically untrusted certificates. A CertStore may also store CRLs.

KeyStore and CertStore implementations are distinguished by types. The Java platform includes the standard *PKCS11* and *PKCS12* key store types (whose implementations are compliant with the corresponding PKCS specifications from RSA Security), as well as a proprietary file-based key store type called *JKS* (which stands for "Java Key Store").

The Java platform includes a special built-in JKS key store, *cacerts*, that contains a number of certificates for well-known, trusted CAs. The keytool documentation (see the security features documentation link in Section 9) lists the certificates included in *cacerts*.

The SunPKCS11 provider mentioned in the "Cryptography" section (Section 4) includes a *PKCS11* KeyStore implementation. This means that keys and certificates residing in secure hardware (such as a smartcard) can be accessed and used by Java applications via the KeyStore API. Note that smartcard keys may not be permitted to leave the device. In such cases, the java.security.Key object reference returned by the KeyStore API may simply be a reference to the key (that is, it would not contain the actual key material). Such a Key object can only be used to perform cryptographic operations on the device where the actual key resides.

The Java platform also includes an *LDAP* certificate store type (for accessing certificates stored in an LDAP directory), as well as an in-memory *Collection* certificate store type (for accessing certificates managed in a java.util.Collection object).

PKI Tools

There are two built-in tools for working with keys, certificates, and key stores:

keytool is used to create and manage key stores. It can

- Create public/private key pairs

- Display, import, and export X.509 v1, v2, and v3 certificates stored as files
- Create self-signed certificates
- Issue certificate (PKCS#10) requests to be sent to CAs
- Import certificate replies (obtained from the CAs sent certificate requests)
- Designate public key certificates as trusted

The **jarsigner** tool is used to sign JAR files, or to verify signatures on signed JAR files. The Java ARchive (JAR) file format enables the bundling of multiple files into a single file. Typically a JAR file contains the class files and auxiliary resources associated with applets and applications. When you want to digitally sign code, you first use keytool to generate or import appropriate keys and certificates into your key store (if they are not there already), then use the **jar** tool to place the code in a JAR file, and finally use the jarsigner tool to sign the JAR file. The jarsigner tool accesses a key store to find any keys and certificates needed to sign a JAR file or to verify the signature of a signed JAR file. Note: **jarsigner** can optionally generate signatures that include a timestamp. Systems (such as Java Plug-in) that verify JAR file signatures can check the timestamp and accept a JAR file that was signed while the signing certificate was valid rather than requiring the certificate to be current. (Certificates typically expire annually, and it is not reasonable to expect JAR file creators to re-sign deployed JAR files annually.)

6 Authentication

Authentication is the process of determining the identity of a user. In the context of the Java runtime environment, it is the process of identifying the user of an executing Java program. In certain cases, this process may rely on the services described in the "Cryptography" section (Section 4).

The Java platform provides APIs that enable an application to perform user authentication via pluggable login modules. Applications call into the LoginContext class (in the javax.security.auth.login package), which in turn references a configuration. The configuration specifies which login module (an implementation of the javax.security.auth.spi.LoginModule interface) is to be used to perform the actual authentication.

Since applications solely talk to the standard LoginContext API, they can remain independent from the underlying plug-in modules. New or updated modules can be plugged in for an application without having to modify the application itself. Figure 3 illustrates the independence between applications and underlying login modules:

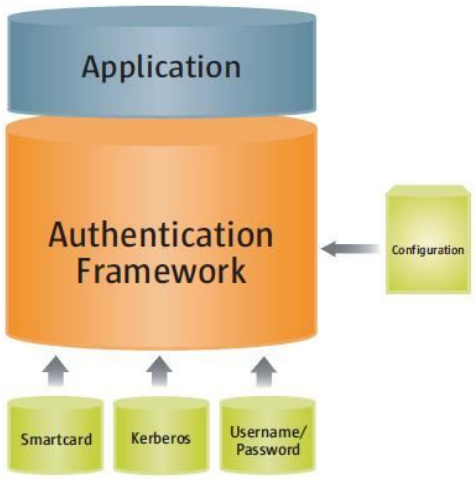


Figure 3 Authentication login modules plugging into the authentication framework

CASE STUDY: AUTHENTICATION IN KERBEROS

Kerberos is a computer network authentication protocol, which allows nodes communicating over a non-secure network to prove their identity to one another in a secure manner. Its designers aimed primarily at a client–server model, and it provides mutual authentication — both the user and the server verify each other's identity. Kerberos protocol messages are protected against eavesdropping and replay attacks.

Kerberos builds on symmetric key cryptography and requires a trusted third party, and optionally may use public-key cryptography by utilizing asymmetric key cryptography during certain phases of authentication.

Kerberos uses as its basis the symmetric Needham-Schroeder protocol. It makes use of a trusted third party, termed a key distribution center (KDC), which consists of two logically separate parts: an Authentication Server (AS) and a Ticket Granting Server (TGS). Kerberos works on the basis of "tickets" which serve to prove the identity of users.

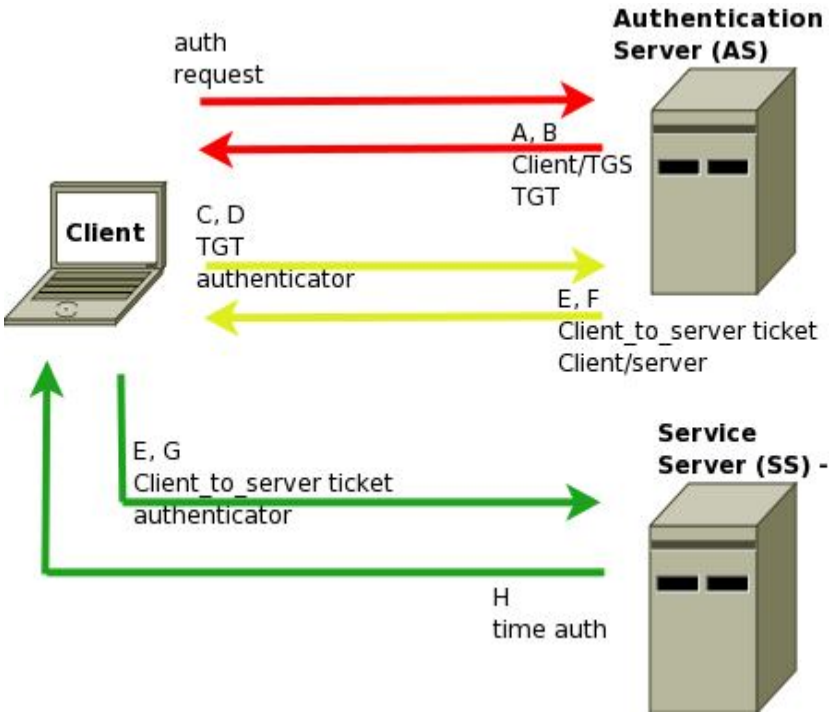
The KDC maintains a database of secret keys; each entity on the network — whether a client or a server — shares a secret key known only to itself and to the KDC. Knowledge of this key serves to prove an entity's identity. For communication between two entities, the KDC generates a session key which they can use to secure their interactions. The security of the protocol relies heavily on participants maintaining loosely synchronized time and on short-lived assertions of authenticity called *Kerberos tickets*.

Description

The following is an intuitive description. The client authenticates itself to the Authentication Server and receives a ticket (All tickets are time-stamped). It then contacts the Ticket Granting Server, and using the ticket it demonstrates its identity and asks for a service. If the client is eligible for the service, then the Ticket Granting Server sends another ticket to the client. The client then contacts the Service Server, and using this ticket it proves that it has been approved to receive the service.

A simplified and more detailed description of the protocol follows. The following abbreviations are used:

- AS = Authentication Server
- SS = Service Server
- TGS = Ticket-Granting Server
- TGT = Ticket Granting Ticket



Kerberos negotiations:

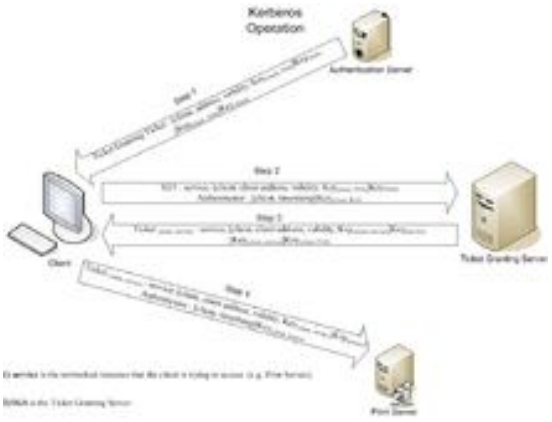
The client authenticates to the AS **once** using a long-term *shared secret* (e.g. a password) and receives a TGT from the AS. Later, when the client wants to contact some SS, it can **(re)use** this ticket to get additional tickets from TGS, for SS, without resorting to using the shared secret. These tickets can be used to prove authentication to SS.

The phases are detailed below.

User Client-based Logon

1. A user enters a username and password on the client machine.
2. The client performs a one-way function (hash usually) on the entered password, and this becomes the secret key of the client/user.

Client Authentication



Kerberos Protocol

1. The client sends a cleartext message of the user ID to the AS requesting services on behalf of the user. (Note: Neither the secret key nor the password is sent to the AS.) The AS generates the secret key by hashing the password of the user found at the database (e.g. Active Directory in Windows Server).
2. The AS checks to see if the client is in its database. If it is, the AS sends back the following two messages to the client:
    - Message A: *Client/TGS Session Key* encrypted using the secret key of the client/user.
    - Message B: *Ticket-to Get-Ticket* (which includes the client ID, client network address, ticket validity period, and the *client/TGS session key*) encrypted using the secret key of the TGS.
3. Once the client receives messages A and B, it attempts to decrypt message A with the secret key generated from the password entered by the user. If the user entered password does not match the password in the AS database, the client's secret key will be different and thus unable to decrypt message A. With a valid password and secret key the client decrypts message A to obtain the *Client/TGS Session Key*. This session key is used for further communications with the TGS. (Note: The client cannot decrypt Message B, as it is encrypted using TGS's secret key.) At this point, the client has enough information to authenticate itself to the TGS.

Client Service Authorization

1. When requesting services, the client sends the following two messages to the TGS:
    - Message C: Composed of the TGT from message B and the ID of the requested service.
    - Message D: Authenticator (which is composed of the client ID and the timestamp), encrypted using the *Client/TGS Session Key*.
2. Upon receiving messages C and D, the TGS retrieves message B out of message C. It decrypts message B using the TGS secret key. This gives it the "client/TGS session key". Using this key, the TGS decrypts message D (Authenticator) and sends the following two messages to the client:
    - Message E: *Client-to-server ticket* (which includes the client ID, client network address, validity period and *Client/Server Session Key*) encrypted using the service's secret key.
    - Message F: *Client/server session key* encrypted with the *Client/TGS Session Key*.

Client Service Request

1. Upon receiving messages E and F from TGS, the client has enough information to authenticate itself to the SS. The client connects to the SS and sends the following two messages:
    - Message E from the previous step (the *client-to-server ticket*, encrypted using service's secret key).
    - Message G: a new Authenticator, which includes the client ID, timestamp and is encrypted using *client/server session key*.
2. The SS decrypts the ticket using its own secret key to retrieve the *Client/Server Session Key*. Using the sessions key, SS decrypts the Authenticator and sends the following message to the client to confirm its true identity and willingness to serve the client:
    - Message H: the timestamp found in client's Authenticator plus 1, encrypted using the *Client/Server Session Key*.

3. The client decrypts the confirmation using the *Client/Server Session Key* and checks whether the timestamp is correctly updated. If so, then the client can trust the server and can start issuing service requests to the server.
4. The server provides the requested services to the client.

Drawbacks

- Single point of failure: It requires continuous availability of a central server. When the Kerberos server is down, no one can log in. This can be mitigated by using multiple Kerberos servers and fallback authentication mechanisms.
- Kerberos has strict time requirements, which means the clocks of the involved hosts must be synchronized within configured limits. The tickets have a time availability period and if the host clock is not synchronized with the Kerberos server clock, the authentication will fail. The default configuration per MIT requires that clock times are no more than five minutes apart. In practice Network Time Protocol daemons are usually used to keep the host clocks synchronized.
- The administration protocol is not standardized and differs between server implementations. Password changes are described in RFC 3244.
- Since all authentication is controlled by a centralized KDC, compromise of this authentication infrastructure will allow an attacker to impersonate any user.

APPENDIX A

1. Basic Vocabulary of Classical Encryption

Plaintext: This is what you want to encrypt.

Ciphertext: The encrypted output

Enciphering or encryption: The process by which plaintext is converted into ciphertext.
Encryption algorithm: The sequence of data processing steps that go into transforming plaintext into ciphertext. Various parameters used by an encryption algorithm are derived from a secret key. In classical cryptography for commercial and other civilian applications, the encryption algorithm is made public.

Secret key: A secret key is used to set some or all of the various parameters used by the encryption algorithm. The important thing to note is that the same secret key is used for encryption and decryption in classical cryptography. It is for this reason that classical cryptography is also referred to as symmetric key cryptography.

Deciphering or decryption: Recovering plaintext from cipher-text
Decryption algorithm: The sequence of data processing steps that go into transforming ciphertext back into plaintext. Various parameters used by a decryption algorithm are derived from the same secret key that was used in the encryption algorithm. In classical cryptography for commercial and other civilian applications, the decryption algorithm is made public.

Cryptography: The many schemes available today for encryption and decryption.

Cryptographic system: Any single scheme for encryption .

Cipher: A cipher means the same thing as a “cryptographic system”.

Block cipher: A block cipher processes a block of input data at a time and produces a ciphertext block of the same size.

Stream cipher: A stream cipher encrypts data on the fly, usually one byte at at time.

Cryptanalysis: Means “breaking the code”. Cryptanalysis relies on a knowledge of the encryption algorithm (that for civilian applications should be in the public domain) and some knowledge of the possible structure of the plaintext (such as the structure of a typical inter-bank financial transaction) for a partial or full reconstruction of the plaintext from ciphertext. Additionally, the goal is to also infer the key for decryption of future messages.
The precise methods used for cryptanalysis depend on whether the “attacker” has just a piece of ciphertext, or pairs of plaintext and ciphertext, how much structure is possessed by the plaintext, and how much of that structure is known to the attacker.
All forms of cryptanalysis for classical encryption exploit the fact that some aspect of the structure of plaintext may survive in the ciphertext.

Brute-force attack: When encryption and decryption algorithms are publicly available, as they generally are, a brute-force attack means trying every possible key on a piece of ciphertext until an intelligible translation into plaintext is obtained.

key space: The total number of all possible keys that can be used in a cryptographic system. For example, DES uses a 56-bit key. So the key space is of size 256, which is approximately the same as 7.2 × 1016.

Cryptology: Cryptography and cryptanalysis together constitute the area of cryptology.

2. Building Blocks of Classical Encryption Techniques

- Two building blocks of all classical encryption techniques are substitution and transposition.
    - Substitution means replacing an element of the plaintext with an element of ciphertext.
    - Transposition means rearranging the order of appearance of the elements of the plaintext.
    - Transposition is also referred to as permutation.