DIFFIE HELLMAN KEY EXCHANGE PROTOCOL

ONE TIME PAD

DIFFIE HELLMAN KEY EXCHANGE PROTOCOL

- **Diffie–Hellman** key exchange is a mathematical method of securely exchanging cryptographic keys over a public channel and was one of the first public-key protocols as conceived by Ralph Merkle and named after Whitfield Diffie and Martin Hellman. [DH is one of the earliest practical examples of public key exchange implemented within the field of cryptography. Published in 1976 by Diffie and Hellman, this is the earliest publicly known work that proposed the idea of a private key and a corresponding public key.

Traditionally, secure encrypted communication between two parties required that they first exchange keys by some secure physical means, such as paper key lists transported by a trusted courier. 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 channel. This key can then be used to encrypt subsequent communications using a symmetric-key cipher.

Diffie–Hellman is used to secure a variety of Internet services. However, research published in October 2015 suggests that the parameters in use for many DH Internet applications at that time are not strong enough to prevent compromise by very well-funded attackers, such as the security services of some countries.[

- Diffie–Hellman key exchange establishes a shared secret between two parties that can be used for secret communication for exchanging data over a public network. An analogy illustrates the concept of public key exchange by using colors instead of very large numbers:
- The process begins by having the two parties, Alice and Bob, publicly agree on an arbitrary starting color that does not need to be kept secret. In this example, the color is yellow. Each person also selects a secret color that they keep to themselves – in this case, red and cyan. The crucial part of the process is that Alice and Bob each mix their own secret color together with their mutually shared color, resulting in orange-tan and light-blue mixtures respectively, and then publicly exchange the two mixed colors. Finally, each of them mixes the color they received from the partner with their own private color. The result is a final color mixture (yellow-brown in this case) that is identical to their partner's final color mixture.
- If a third party listened to the exchange, they would only know the common color (yellow) and the first mixed colors (orange-tan and light-blue), but it would be very hard for them to find out the final secret color (yellow-brown). Bringing the analogy back to a real-life exchange using large numbers rather than colors, this determination is computationally expensive. It is impossible to compute in a practical amount of time even for modern supercomputers.

Step-by-Step explanation is as follows:

<table>
<tr><th>Alice</th><th>Bob</th></tr>
<tr><td>Public Keys available = P, G</td><td>Public Keys available = P, G</td></tr>
<tr><td>Private Key Selected = a</td><td>Private Key Selected = b</td></tr>
<tr><td>Key generated =
$x = G^a mod P$</td><td>Key generated =
$y = G^b mod P$</td></tr>
<tr><td colspan="2">Exchange of generated keys takes place</td></tr>
<tr><td>Key received = y</td><td>key received = x</td></tr>
<tr><td>Generated Secret Key =
$k_a = y^a mod P$</td><td>Generated Secret Key =
$k_b = x^b mod P$</td></tr>
<tr><td colspan="2">Algebraically, it can be shown that
$k_a = k_b$</td></tr>
<tr><td colspan="2">Users now have a symmetric secret key to encrypt</td></tr>
</table>

from geekaforgeeks

```cpp
/* This program calculates the Key for two persons
using the Diffie-Hellman Key exchange algorithm using C++ */
#include <cmath>
#include <iostream>

using namespace std;

// Power function to return value of a ^ b mod P
long long int power(long long int a, long long int b,
                    long long int P)
{
    if (b == 1)
        return a;

    else
        return (((long long int)pow(a, b)) % P);
}

// Driver program
int main()
{
    long long int P, G, x, a, y, b, ka, kb;

    // Both the persons will be agreed upon the
    // public keys G and P
    P = 23; // A prime number P is taken
    cout << "The value of P : " << P << endl;

    G = 9; // A primitive root for P, G is taken
    cout << "The value of G : " << G << endl;

    // Alice will choose the private key a
    a = 4; // a is the chosen private key
    cout << "The private key a for Alice : " << a << endl;

    x = power(G, a, P); // gets the generated key

    // Bob will choose the private key b
    b = 3; // b is the chosen private key
    cout << "The private key b for Bob : " << b << endl;

    y = power(G, b, P); // gets the generated key

    // Generating the secret key after the exchange
    // of keys
    ka = power(y, a, P); // Secret key for Alice
    kb = power(x, b, P); // Secret key for Bob
    cout << "Secret key for the Alice is : " << ka << endl;

    cout << "Secret key for the Bob is : " << kb << endl;

    return 0;
}
// This code is contributed by Pranay Arora
```

one-time pad

In cryptography, the **one-time pad** (**OTP**) is an encryption technique that cannot be cracked, but requires the use of a single-use pre-shared key that is larger than or equal to the size of the message being sent. In this technique, a plaintext is paired with a random secret key (also referred to as *a one-time pad*). Then, each bit or character of the plaintext is encrypted by combining it with the corresponding bit or character from the pad using modular addition.

The resulting ciphertext will be impossible to decrypt or break if the following four conditions are met:

1. The key must be at least as long as the plaintext.
2. The key must be random (uniformly distributed in the set of all possible keys and independent of the plaintext), entirely sampled from a non-algorithmic, chaotic source such as a hardware random number generator; patternless, according to Gregory Chaitin definition.It is not sufficient for OTP keys to pass statistical randomness tests as such tests cannot measure entropy, and the number of bits of entropy must be at least equal to the number of bits in the plaintext. For example, using an algorithm to generate seemingly-random data would allow an attacker to try to crack the random number generation algorithm. Truly random data by definition cannot be predicted.
3. The key must never be reused in whole or in part.
4. The key must be kept completely secret by the communicating parties.

- It has also been mathematically proven that any cipher with the property of perfect secrecy must use keys with effectively the same requirements as OTP keys.[Digital versions of one-time pad ciphers have been used by nations for critical diplomatic and military communication, but the problems of secure key distribution make them impractical for most applications.
- The "pad" part of the name comes from early implementations where the key material was distributed as a pad of paper, allowing the current top sheet to be torn off and destroyed after use. For concealment the pad was sometimes so small that a powerful magnifying glass was required to use it. The KGB used pads of such size that they could fit in the palm of a hand,or in a walnut shell To increase security, one-time pads were sometimes printed onto sheets of highly flammable nitrocellulose, so that they could easily be burned after use.

Advantages

- One-Time Pad is the only algorithm that is truly unbreakable and can be used for low-bandwidth channels requiring very high security(ex. for military uses).

Disadvantages

- There is the practical problem of making large quantities of random keys. Any heavily used system might require millions of random characters on a regular basis.
- For every message to be sent, a key of equal length is needed by both sender and receiver. Thus, a mammoth key distribution problem exists.

The assignment is as follows:

A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9
K	L	M	N	O	P	Q	R	S	T
10	11	12	13	14	15	16	17	18	19
U	V	W	X	Y	Z				
20	21	22	23	24	25				

```
using namespace std;
// Method 1
// Returning encrypted text
string stringEncryption(string text, string key)
{

    // Initializing cipherText
    string cipherText = "";

    // Initialize cipher array of key length
    // which stores the sum of corresponding no.'s
    // of plainText and key.
    int cipher[key.length()];

    for (int i = 0; i < key.length(); i++) {
        cipher[i] = text.at(i) - 'A' + key.at(i) - 'A';
    }

    // If the sum is greater than 25
    // subtract 26 from it
    // and store that resulting value
    for (int i = 0; i < key.length(); i++) {
        if (cipher[i] > 25) {
            cipher[i] = cipher[i] - 26;
        }
    }

    // Converting the no.'s into integers

    // Convert these integers to corresponding
    // characters and add them up to cipherText
    for (int i = 0; i < key.length(); i++) {
        int x = cipher[i] + 'A';
        cipherText += (char)x;
    }

    // Returning the cipherText
    return cipherText;
}

// Method 2
// Returning plain text
static string stringDecryption(string s, string key)
{
    // Initializing plain text
    string plainText = "";

    // Initializing integer array of key length
    // which stores difference
    // of corresponding no.'s of
    // each character of cipherText and key
    int plain[key.length()];

    // Running for loop for each character
    // subtracting and storing in the array
    for (int i = 0; i < key.length(); i++) {
        plain[i] = s.at(i) - 'A' - (key.at(i) - 'A');
    }
```

```
    }

    // If the difference is less than 0
    // add 26 and store it in the array.
    for (int i = 0; i < key.length(); i++) {
        if (plain[i] < 0) {
            plain[i] = plain[i] + 26;
        }
    }

    // Converting int to corresponding char
    // add them up to plainText
    for (int i = 0; i < key.length(); i++) {
        int x = plain[i] + 'A';
        plainText += (char)x;
    }

    // Returning plainText
    return plainText;
}

// Method 3
// Main driver method
int main()
{
    // Declaring plain text
    string plainText = "Hello";

    // Declaring key
    string key = "MONEY";

    // Converting plain text to toUpperCase
    // function call to stringEncryption
    // with plainText and key as parameters
    for (int i = 0; i < plainText.length(); i++) {
        // convert plaintext to uppercase
        plainText[i] = toupper(plainText[i]);
    }
    for (int i = 0; i < key.length(); i++) {
        // convert key to uppercase
        key[i] = toupper(key[i]);
    }
    string encryptedText = stringEncryption(plainText, key);

    // Printing cipher Text
    cout << "Cipher Text - " << encryptedText << endl;

    // Calling above method to stringDecryption
    // with encryptedText and key as parameters

    cout << "Message - "
         << stringDecryption(encryptedText, key);

    return 0;
}
```