# HMAC MD5 Cyber Security Project

By
Claire Gouessant
Bertrand Duhamel
Ella Witenberg

## Table of content

## Introduction

As part of our cyber security course, we have seen how we can encrypt a message. Encryption is useful in many ways. It can be not only to discuss with someone else without being understandable, but also to keep something secret to anyone else.

Our goal was to learn how we use computers to encode messages. In this project, we have implemented a MD5, a H-MAC and the used salt. MD5 is a hash function, so it allows us to encode a message. It was often used to encode passwords. H-MAC is used to hide a message by a better way than only a simple hash function. To do it, it uses a secret key and a hash function, in our case the md5. Salt is used to change the message at the beginning to add a better security.

# A cypher algorithm: the MD5

## Description and history

The MD5, which stands for Message Digest 5, algorithm is a commonly used hash method that output a 128-bit hash for any given words, sentences or document. It is mainly used for document integrity verification and checksum due to extensive vulnerabilities. For example, it is vulnerable to brute forces attacks or collision attacks.
It was designed by Ronald Rivest in 1991 to replace the previous MD4 but was quickly outperformed and is completely unsuitable for security by most current informatics security standards.

## Algorithm

The MD5 takes inputs by blocks of 512-bit to encode them. If the block is not long enough, its padded with one 1 bit and 0 bits to be divisible by 488 and the 64 remaining bits are filled with the message original length.

All variables are unsigned 32 bits and wrap modulo 2^32 when calculating
The S table represents the amount of shifting in each round per state.
The K table are constants with which the states will be modified.
The a0, b0, c0 and d0 variables are the states to begin the algorithm with.

We start by breaking the message in 16 blocks of 32-bits
The M table resulting represents the message to be hashed by block of 32-bits

The main loop works as follow where F and g are ints

```
For i from 0 to 63,
        If i is between 0 and 15
                F = (B and C) or ((not B) and D)
                g = i
        else if i is between 16 and 31
                F = (D and B) or ((not D) and C)
                g = (5*i + 1) mod 16
        else if i is between 32 and 47
                F = B xor C xor D
                g = (3*i + 5) mod 16
        else if i is between 16 and 31
                F = C xor (B or (not D))
                g = (7*i) mod 16
        end
```

F = F + A + K[i] + M[g]
A = D
D = C
C = B
B = B + left rotate (F, s[i])
End

Each A, B, C and D found this way is added to their original state and the process continues until every 512 blocks are hashed
To finish, every of the 4 a0, b0, c0 and d0 blocks are appended together to obtain the hash of the message.

## Some code explained

The following are samples of our code to illustrate the previous algorithm, specifically, our MD5 function and the problem we faced.

```
public static byte[] md5(byte[] message3)//pseudobyte
{
    //s specifies the per-round shift amounts
    int[] s = { 7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17,
    //K = sin(...)
    int[] K = {0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee, 0xf57c0faf,

    //Initialize variables:
    byte[] a1 = fromIntToByte(1732584193);//(0x67452301);   //A
    byte[] b1 = fromIntToByte(-271733879);//((int)0xefcdab89);   //B
    byte[] c1 = fromIntToByte(-1732584194);//((int)0x98badcfe);   //C
    byte[] d1 = fromIntToByte(271733878);//(0x10325476);   //D

    //Note: All variables are unsigned 32 bit and wrap modulo 2^32 when cal
    int originalLength = message3.length/8;

    //Pre-processing: adding a single 1 bit : append "1" bit to message
    // Notice: the input bytes are considered as bits strings,
    //  where the first bit is the most significant bit of the byte.[48]
    message3 = append(message3, (byte)0x01);
```

Here we start initializing the message to be hashed by converting it to bytes and padding it with a 1 in bit and then 0 until it reaches 448 mod (512) in size.
Then we added the length of the message in 64-bit to make the message in size mod (512).

The function Switch Endian is used to flip the bits to have them in the right order (little endian) for the rest of the data processing.

```
//append original length in bits mod 264 to message
byte[] originalLength3 = fromIntToByte((originalLength*8)%(256*256*256*128));// = 2^31

if(originalLength3.length != 32) System.err.println("error length: "+originalLength3.length);

message3 = append(message3, switchEndian(originalLength3));

byte[] originalLength4 = fromIntToByte((originalLength-(originalLength%(256*256*256*128)))/(256*256*256*128));

message3 = append(message3, switchEndian(originalLength4));
message3 = switchEndian(message3);
```

After this we can start processing the message by blocks of 512-bits

```
//Process the message in successive 512-bit chunks:
//for each 512-bit chunk of padded message break chunk into sixteen 32-b
for(int bitBlock = 0; bitBlock < message3.length; bitBlock += 512)
{
    //Initialize hash value for this chunk:
    byte[] A = copy(a1);
    byte[] B = copy(b1);
    byte[] C = copy(c1);
    byte[] D = copy(d1);
```

We start by initializing the states the basic hexadecimal (bytes) constants given by the algorithm.

```
 for(int i = 0;  i < 16; i++)
 {
     byte[] F = Ff(B, C, D);
     int g = bitBlock + (i * 32);
     byte[] K2 = fromIntToByte(K[i]);
     byte[] M = fromByteToWord(message3, g);
     F = concat(concat(F, A), concat(K2, M));
     A = D;
     D = C;
     C = B;
     B = concat(B, round(F, s[i]));
 }
```

This is the code corresponding to the first round of the algorithm. The three-next round look very similar to this with some constants changing and having each time A, B, C and D modified.

```
        a1 = concat(a1, A);
        b1 = concat(b1, B);
        c1 = concat(c1, C);
        d1 = concat(d1, D);
    }
    byte digest[] = append(append(a1, b1), append(c1, d1));
    return switchEndian(digest);
```

At the end of the four rounds, each 4 parts are concatenated to it previous self from the previous block. It is more like an [or] bit-wise operation than a concatenation.

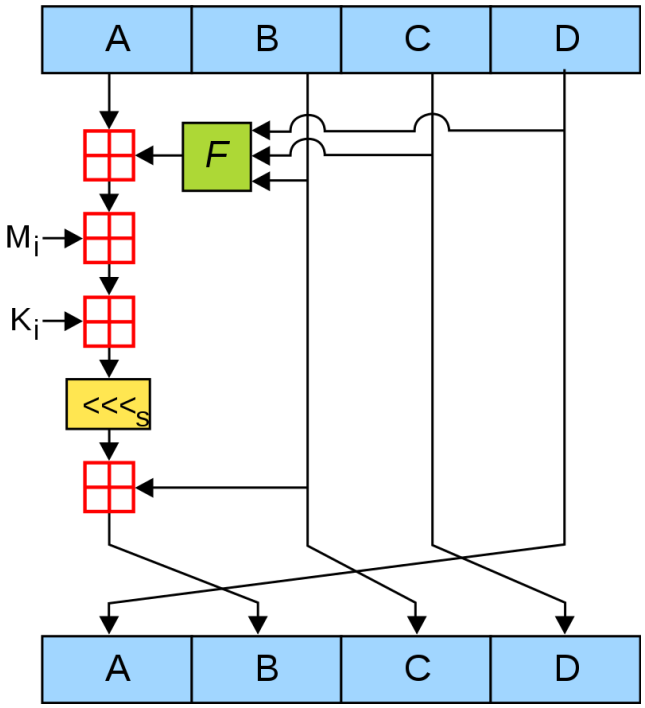
When all four blocks are processed, the resulting A, B, C and D are appended together to form the final message in bits but in big-endian. That is why we return the little-endian result to have the good result prior to the MD5 algorithm.

Here is an example of result from our MD5 function

```
 Veuillez saisir le message à hasher :
azerty
 Message : azerty

 MD5 :

ab4f63f9ac65152575886860dde480a1
```




The MD5 algorithm.

# Add some salt to your security

## Principle

The principle of salt is to add some, fixed or random, chain of letters or numbers at the end of a password and then hash it to have a different hash and thus have the password harder to recover.
With a dynamic salt, each salt is stored in the data base with the id and the password with salt hashed. When the user enters a password with his id, the salt corresponding to its id is added to the password entered, the whole thing being hashed to compare it to the hash in memory. This prevents simply stocking passwords that would be easily accessible if a hacker breached into the database, the salt making hard to for the password to be found if so since different messages can have the same hash.

## Some code explained

```
for(int i = 0; i < 100; i++)
{
    identifiants[i] = i+1 + generate((int)Math.floor(Math.random() * 10 + 1));
    salage[i] = generate((int)Math.floor(Math.random() * 10 + 1));
    passwords[i] = generate((int)Math.floor(Math.random() * 100 + 1));
}
for (int i = 0; i < 100; i++)
{
    System.out.println("Id : " + identifiants[i]);
    System.out.println("Mdp : " + passwords[i]);
    System.out.println("Mdp Hashé: " + fromByteToHex(md5(wider((passwords[i]+salage[i]).getBytes()))));
    System.out.println("Sel : " + salage[i] + "\n");
}
```

Here we one hundred ids starting with their position in the table and followed by a random number of random char. The generate function return a random alpha-numeric char and takes in parameter the number of random char wanted, here random between 1 and 11.

By the same way, the salt is generated for each id.

To finish, a password is generated the same way but with 1 to 100 random chars to complexify a bit the process.

Is then printed out, the id, the password, the hash of the salted password and the salt to verify that everything works correctly.

Everything is written on console but also saved in a .text document that should be next to the folders in the programme files. It will be overwritten each time this process is done.

# Improvement with HMAC

## Description

In cryptography, a keyed-hash message authentication code (HMAC) is a specific type of message authentication code (MAC) involving a cryptographic hash function and a secret cryptographic key. It may be used to simultaneously verify both the data integrity and the authentication of a message.
It can be use with many other algorithms than he MD5, SHA-1 or SHA-256 for instance.

## Algorithm

The HMAC Algorithm is simple:
It takes in input a message and a key. It processes the data by blocks of 64-bits that is the block size for the rest of this explanation.

If the key is longer than the block size, it will be hashed by the MD5 algorithm in our case since it outputs a 64-bit hash. If it is not long enough, 0 in bits are added as a pad to make it the right length.

The key is then [xor] bit-wise twice with two hex decimal values on each byte that can be contained in the block size. The two values are 0x5c and 0x36.
This makes the ipad and the opad parts.

To finish the ipad, opad and message in bits are hashed as follow:
Hmac = md5 (opad || md5(ipad || message) where || is a concatenation.

Here is an example of salt and HMAC

```
 ---- Projet HMAC MD5 Claire Gouessant et Bertrand Duhamel ----
 ---- Projet HMAC MD5     M1 IFM         et     M1 IRV     ----

 Faites un choix
 1. Hasher le message en utilisant le MD5
 2. Afficher 100 messages aléatoires hasher en MD5 avec un identifiant aléatoire
 3. Afficher 100 messages aléatoires hasher en MD5 avec un salage aléatoire avec un identifiant aléatoire
 4. Hasher le message en utilisant le HMAC MD5
 5. Saisir un mot de passe et une clé, résultat: MD5, MD5+sel, HMAC, HMAC+sel
 6. quitter

5
 Veuillez saisir le message à hasher :
a
 Veuillez saisir la clé :
cle
 Message : a

 --- MD5 généré par notre programme ---

0cc175b9c0f1b6a831c399e269772661

Mdp Hashé avec sel:
8e1b3a9d744a88e8bd529516ba90d49d

Sel :
C8gFqg

Hmac: 2202fd99dc05a635c5ea96228cad243f
```

## Some code explained

```
public static String hmac(String message, String key)
{
    int blockSize = 64;

    byte[] Key;
    Key = wider(key.getBytes());


    //Keys longer than blockSize are shortened by hashing them
    if (Key.length > blockSize*8) Key = md5(Key);
    //Keys shorter than blockSize are padded to blockSize by padding with zeros on the right
    else if (Key.length < blockSize*8) Key = Pad(Key, blockSize);

    byte[] OKey = wider(key_Pad(blockSize, false));
    byte[] IKey = wider(key_Pad(blockSize, true));
    byte[] o_key_pad = xor(Key, OKey);//xor [0x5c * blockSize]   //Outer padded key
    byte[] i_key_pad = xor(Key, IKey);//xor [0x36 * blockSize]   //Inner padded key

    String Hmac = fromByteToHex(md5(append(o_key_pad, md5(append(i_key_pad, wider(message.getBytes()))))));
    return Hmac;
}
```

Here is our function HMAC that takes a message and a key in parameter.

Wider is a function to work with “pseudo bytes” containing precisely bits as we wanted and not the hexadecimal value of bytes that would normally appear. This is done for us to ensure that we had to good bytes for the process.

Key_pad creates the 0x36 pattern in opad and the 0x5c pattern in ikey before being [xor] with the key.

The fromByteToHex function is made to ensure a good conversion since we problems with the common .getHexString from java.

Aside from this, everything works exactly like the described algorithm.

## Conclusion

As we have seen before MD5 is not recommended to encode password nowadays because it doesn't resist efficiently to brute force attack. However, different kind of encryption combined give a better protection to our data. Even if these methods of encryption are not recommended nowadays, there was very used years ago.

To improve our program, we may should use integer instead of byte[] as other programs we have seen because it is more adapted to the MD5 function. But what we have done with arrays give more fluidity and is more practical in some other aspect.

We have taken a long time to do this project because it was difficult to check our data. To do the H-MAC we do not guess any code to compare with our solution when it was not working well.

But we have finally finished the MD5, with and without salt as well as the H-MAC.

## Annexes

Source pseudo code: Wikipedia

We wanted to tell that the two-following function where copied from the net to help us with our program.

```
public static String generate(int length)
{
    String chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890";
    String pass = "";
    for(int x=0;x<length;x++)
    {
       int i = (int)Math.floor(Math.random() * 62);
       pass += chars.charAt(i);
    }
    return pass;
}

public static String makeString(byte[] hash)
{
    StringBuilder hashString = new StringBuilder();
    for (int i = 0; i < hash.length; i++)
    {
        String hex = Integer.toHexString(hash[i]);
        if (hex.length() == 1)
        {
            hashString.append('0');
            hashString.append(hex.charAt(hex.length() - 1));
        }
        else
            hashString.append(hex.substring(hex.length() - 2));
    }
    return hashString.toString();
}
```