



This is the Myplace service for the 2024/25 session. For the past session, please go to [classes 2023/24](#).



[Dashboard](#) / [My classes](#) / [CS808](#) / Week 2 (w/c 30th September): Introducing Cryptography / [2.10: Article: Making Block Ciphers Usable- Messag...](#)



NH

CS808: Computer Security Fundamentals

2.10: Article: Making Block Ciphers Usable- Message Authentication Codes

✓ Done: View

One limitation of ciphers, even when making use of block ciphers with unique initialisation vectors, is that an attacker could flip a bit (or more) of the initialisation vector or ciphertext. This is sometimes called a bit flipping attack.

This would mean that decryption would no longer be successful for block ciphers, or for stream ciphers it would alter part of the plaintext message.

You can try this for yourself. Assume we are using a stream cipher which performs an XOR operation bit by bit with a random keystream which is as long as the Ciphertext. Using this website <https://xor.pw/> enter input 1 as send rose £500 and change the type of input to ASCII, for input 2 provide this pseudo random Hex string 1ec71OrTqrYSfSFg Next calculate the XOR value in hexadecimal base 16, which should be 73656e6420726f736520c2a3353030

This is shown in Figure 1.

XOR Calculator

Thanks for using the calculator. [View help page](#).

I. Input: [ASCII \(base 256\)](#)

send rose £500

II. Input: [hexadecimal \(base 16\)](#)

1ec71OrTqrYSfSFg

Calculate XOR

III. Output: [hexadecimal \(base 16\)](#)

73656e6420726f736520c2a3353030

Figure 1

Now take that output, enter it as input I and enter the same random hex string 1ec71OrTqrYSfSFg as input II, then calculate the output as ASCII. This is shown in Figure 2 where you can see the Ciphertext XOR'd with the keystream results in the plaintext.

XOR Calculator

Thanks for using the calculator. [View help page.](#)

I. Input: [hexadecimal \(base 16\)](#) ▼

73656e6420726f736520c2a3353030

II. Input: [hexadecimal \(base 16\)](#) ▼

1ec710rTqrYSfSFg

Calculate XOR

III. Output: [ASCII \(base 256\)](#) ▼

send rose £500

Figure 2

Now try changing the right most 3 in input I (the ciphertext) to 0 and recalculate the output. This is shown in Figure 3. Notice how the plaintext now becomes send rose £50.

XOR Calculator

Thanks for using the calculator. [View help page.](#)

I. Input: [hexadecimal \(base 16\)](#) ▼

73656e6420726f736520c2a3353000

II. Input: [hexadecimal \(base 16\)](#) ▼

1ec710rTqrYSfSFg

Calculate XOR

III. Output: [ASCII \(base 256\)](#) ▼

send rose £50

Figure 3

Clearly we do not wish for an attacker to be able to alter our message in such a way.

As you may have inferred, this is a problem around the integrity of the data. The result is that we should not use a stream or block cipher and mode of operation 'as is'. Instead, we must find a way of ensuring the initialisation vector (if using a block cipher) and ciphertext cannot be modified without resulting in a warning or error of some kind.

We can achieve this through use of a cryptographic hash function in combination with a secret key. This is called a Message Authentication Code (MAC), and this is made use of extensively over the internet where most messages either have a MAC included as part of a cipher or appended at the end of a message. They provide the recipient with assurance that the message has not been altered.

MAC function takes a message and secret key and produces what is called an authentication tag. The output behaves similarly to a hash function, in that it is not reversible and any change to the input results in a different authentication or MAC code. In fact, many MAC functions are based on cryptographic hash functions, such as HMAC-SHA-256.

The authentication tag can only be calculated by those who know the secret key, so if an attacker were to intercept a message with the MAC appended they should not be able to change the message and update the hash value as it relies on knowledge of the secret key.

In this way, a MAC can be considered like a private hash function, only those with the secret key will be able to produce the output because you have access to the secret key. MACs are deterministic, that is the same hash function, message and key will always generate the same authentication tag.

When sending a message, a MAC function can be applied using a pre-shared secret key and the message can be sent along with the authentication tag to the recipient, who can then complete the MAC calculation on the message using the secret key to ensure the message has not been altered (integrity) and it has come from who you expect (due to the required secret key).

However, even this is not as secure as it could be. With some hash functions there are potential attacks which can append data and provide an updated hash value. This is beyond the scope for our purposes, but it is worth highlighting. This type of attack can be used against SHA-2, and are called length extension attacks if you wish to find out more.

One of the most commonly used MACs is HMAC, which extends the general structure of a MAC function. HMAC stands for hash-based message authentication code. In this approach, we derive two sub-keys from the secret key. We will call these k_1 and k_2 . The message is concatenated with k_1 and a hash value is calculated (we will call this h_1). h_1 is then concatenated with k_2 and hashed once more. This output is the final authentication tag.

HMAC has been mathematically proven to be resistant to forgery of authentication tags so long as the hash function has specific properties, of which all cryptographically secure hash functions should.

However, there are still some things to be aware of when making use of MACs. Authentication tags and corresponding messages may be replayed at another time if not appropriately dealt with, authentication tags need to be sufficiently long to be secure (generally 128 bits or longer, with HMACs the hash length defines the authentication tag length) and verification of authentication tags can be prone to errors. Full detail on this is beyond the scope of the module, but a particularly good resource on practical cryptography is Chapter 4 of [Real World Cryptography](#) by David Wong.

Last modified: Wednesday, 28 September 2022, 1:32 PM

◀ [2.9 AES, Blowfish and modes of operation \(23-24\) \(13:56\)](#)

Jump to...

[2.11: Lab: OpenSSL for AES ▶](#)

© University of Strathclyde

You are logged in as **Neil Hutton (Log out)**

CS808