Message Authentication Code

Chang Liu[1], Liyan Jiang[1], and Yiwen Yang[1]

[1]Group 4, VE475, UM-JI

June 29, 2022

Abstract

Message Authentication Code (MAC), is a short piece of information used for authenticating a message [1]. This project aims at performing some personal study on MAC as well as discussing the topic using cryptography analysis methods covered in the course, so as to acquire more knowledge in the cryptography field. We first briefly introduce the structure and process of MAC, then two implementations of MAC algorithm are explained in detail: Hash-based Message Authentication Code (HMAC) and Block Cipher-based Message Authentication Code (CMAC). Next, the security of both implementations is analyzed and attack methods are discussed. Finally we present the application of MAC in Transport Layer Security (TLS) Protocol.

Contents

1 Introduction

1.1 Background

Digital data security has long been a popular issue with the revolution of network connections and data communications. During the data transfer process, digital information can easily be added, deleted and modified by everyone. Therefore, when a data receiver gets a piece of information from the network, the information is not guaranteed to be the same message as the sender sends. To verify one message, one may want to check its integrity, authenticity and non-repudiation.

- **Integrity:** Completeness, consistency and accuracy of data [2]. Integrity can be ensured if content or time of data has not been accidentally modified.
- **Authenticity:** Originality or genuineness of data. Whether the message originates from the sender.
- **Non-repudiation:** If the message is sent to a third party, whether the third party can also verify integrity and authenticity of data [3].

Message authentication code (MAC) is one of the major methods to check both integrity and authenticity, but not the non-repudiation of message. MAC is appended to the message by the sender, and receiver is able to use MAC to tell whether the message has accidentally changed or is sent by a third party who does not own the key. MAC is now widely used in Internet security and unreliable data storage.

1.2 Methods

MAC achieves the authentication by creating a short piece of information given the original message and one symmetric key. The process flow diagram is shown in Figure 1.

1. **Requirements**

   MAC works based on a symmetric key. The key must be shared before MAC between sender and receiver in person.

2. **Sender**

   The sender first prepared the original message. Then with the symmetric key, the sender combines the message and the key using a MAC algorithm.

   The MAC algorithm is implemented in two ways: 1) Hash-based Message Authentication Code (HMAC), which uses a hash function such as *MD5*, *SHA-1* and *SHA-2*, to generate a key-hashed checksum. 2) Block Cipher-based Message Authentication Code (CMAC), which uses block cipher algorithms such as *AES* and *DES* to get cipher text of fixed length.

   Finally, the sender appends MAC to the message and sends them in a bundle to the receiver.

3. **Receiver**

   The receiver gets the message and MAC from the network. First, the same symmetric key shared before is applied on the MAC algorithm and the message to generate

a MAC from the receiver side. The MAC algorithm must be the same one in the sending process, which can be agreed on when sharing the key.

Next, the receiver simply compares the two MAC. If two MAC are the same, then the receiver can confirm the integrity and authenticity of the message; Otherwise he cannot, which means that data content has been either accidentally modified or tampered by someone else.

Since MAC is based on the symmetric key, the integrity can be ensured if the key is not leaked to someone other than the sender and the receiver, and the security level should be same as that of the MAC algorithm. Similarly, authenticity can also be verified from the receiver side. However, MAC does not guarantee non-repudiation. If a third party receives the message, he cannot tell whether the message originates from the sender or the receiver. Therefore, if the receiver manipulates a message and claims it to be sent from the sender, the sender cannot deny, since they have the same key. To provide non-repudiation, one need to use Digital Signature [4] instead of MAC.

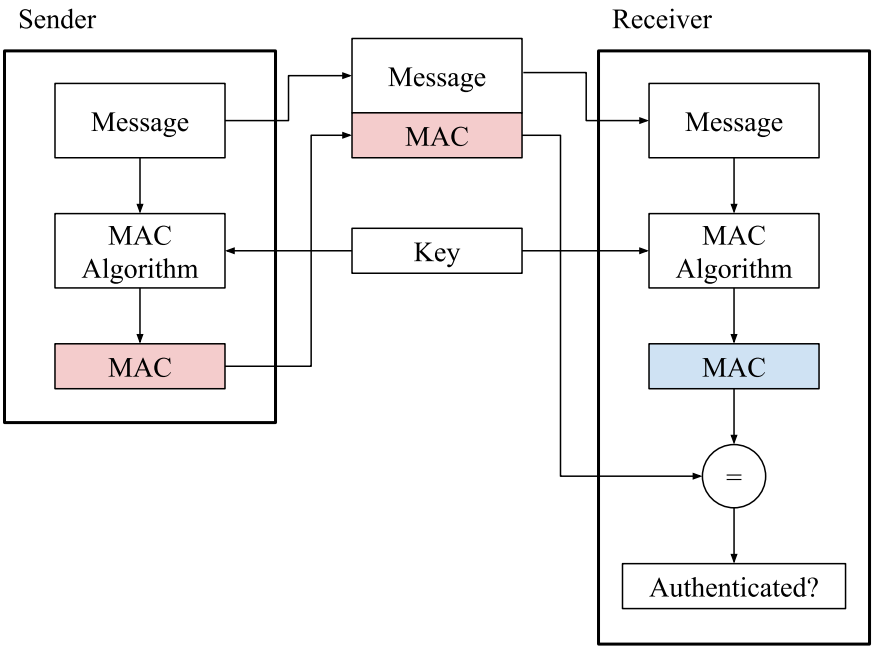

Figure 1: Message Authentication Code

2 Implementation

To suit various applications of MAC with different requirements, many different implementations of MAC are proposed. Among them, HMAC is a group of MAC generation algorithms based on hash functions like *MD5* and *SHA family*. At the same time, *Poly1305-AES* is another typical algorithm based on *AES*, which has advantage in speed.

2.1 HMAC

In [5], *HMAC* method is introduced, which fully takes the advantage of hash functions that it can map input with any length into a fix-length output. To add more randomness to output and improve the security, some extra steps are taken in *HMAC* algorithm. The overall process of *HMAC* can be represented by the formula

$$\text{HMAC}_k(x) = F(\overline{k} \oplus \text{opad} | F(\overline{k} \oplus \text{ipad} | x))$$

where F is a key-less hash function, $\oplus$ denotes bit-wise XOR operation, $|$ denotes concatenation and $\overline{k}$ is completed key. Here, *HMAC* based on *SHA-1* will be taken as the example.

To calculate this equation, first they key k should be processed because its length may not satisfy requirements. In *HMAC*, the key length is expected to be 512 bits. If k is exactly 512-bit long, nothing needs to be done and $\overline{k} = k$. If k is shorter than 512 bits, 0's are appended until the length reaches 512 bits. When it is longer than 512 bits, hash function F is applied to reduce length and then 0's are appended to form 512-bit $\overline{k}$.

opad and *ipad* are also easy to obtained, because they are simply repetition of fixed bytes. *opad* is a repetition of 0x36 while *ipad* is obtained by repeating 0x5C. To match the length of $\overline{k}$, the bytes are repeated by $512/8 = 64$ times.

With preparation done, *MAC* can be finally calculated. Input message x is first concatenated to the right of $\overline{k} \oplus$ ipad, and then fed to hash function *SHA-1*. *SHA-1* function will always return with a 160-bit long output, which will be concatenated to the right of $\overline{k} \oplus$ opad and inputted into *SHA-1* again. This time, the output is exactly the final output of *HMAC* algorithm.

2.1.1 SHA-1

To make the whole process clear, *SHA-1* algorithm described by [6] is also introduced here. The overall structure of *SHA-1* is that input are divided into blocks, on which the same process will be applied. At the same time, theses processes are chained together so that former blocks have influence on latter blocks.

First, input is divided in to 512-bit long blocks. To assure the completion of the last block, padding method is applied. The input is first appended with a 1, followed by several 0's. Finally, the length of original input is attached as a 64-bit integer. The number of 0's is determined by the final goal that, including the 64-bit length, total length is a multiple of 512.

After blocks are divided, chained calculation is conducted on them. First, global variables $H_j, 0 \leq j \leq 4$ are introduced, each of which has a length of 32 bits. They are initialized before the first block is processed,

$$H_0 = 67452301, H_1 = \text{EFCDAB89}, H_2 = \text{98BADCFE}, H_3 = 10325476, H_4 = \text{C3D2E1F0}$$

Then comes to the operation on each block, which takes 80 rounds. To support operation of 80 rounds for each block, an array W_i of 32-bit numbers is used. The first 16 element, W_0 to W_{15} are the input in this block, which is exactly 512 bits. As for latter elements, they are generated by the equation

$$W_i = W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}$$

Before starting 80 rounds of calculation, temperate variables A, B, C, D, E are assigned by the values of H_0 to H_4 respectively. Then in ith round, the follow process is done (here numbers are always 32-bit and overflow part will be ignored)

$$\begin{cases} \text{TEMP} = S^5(A) + f_i(B,C,D) + E + W_i + K_i \\ E = D \\ D = C \\ C = S^{30}(B) \\ B = A \\ A = \text{TEMP} \end{cases}$$

In the equations, S^k denotes cyclic left shifting for k bits, and f_i, K_i are defined as follows

$$\begin{cases} f_i(B,C,D) = (B \wedge C) \vee (\ B \wedge D), (0 \leq i \leq 19) \\ f_i(B,C,D) = B \oplus C \oplus D, (20 \leq i \leq 39) \\ f_i(B,C,D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D), (40 \leq i \leq 59) \\ f_i(B,C,D) = B \oplus C \oplus D, (60 \leq i \leq 79) \end{cases}$$

$$\begin{cases} K_i = \text{5A827999}, (0 \leq i \leq 19) \\ K_i = \text{6ED9EBA1}, (20 \leq i \leq 39) \\ K_i = \text{8F1BBCDC}, (40 \leq i \leq 59) \\ K_i = \text{CA62C1D6}, (60 \leq i \leq 79) \end{cases}$$

Both of them depends on the round index.

After 80 round, results are chained by $H_0 = H_0 + A, H_1 = H_1 + B, H_2 = H_2 + C, H_3 = H_3 + D, H_4 = H_4 + E$. It easy to sea that results accumulate in H_i after processing of all the blocks are completed. The concatenation of these 5 variables give the final output, which is $32 \times 5 = 160$ bits.

2.2 Poly1305-AES

As is indicated in the name, this method proposed in [7] based on both *AES* algorithm and polynomial calculation. To run *Poly1305-AES*, an 128-bit nonce n and another 256-bit secrete key should be given along with the message m.

First, the secret key should meet some satisfactions. The secret key will be regarded as a concatenation of two 128-bit parts, (k, r). k is the key used for *AES* processing, while r is regarded a combination of 16 little-endian unsigned numbers. Denoting them by $r[0], r[1], \cdots, r[15]$, the upper 4 bits of $r[3], r[7], r[11], r[15]$ have to be 0 and the lower 2 bits of $r[4], r[8], [12]$ should be 0 too.

Second, the message should be padded to suit this algorithm. Message m is firstly divided into chucks of 128 bits, and then appended with 0x01. After this, all chunks except the last one are 136-bits long, so 0's may be attached to the last chunk such that all chunks have the same length. After padding, m is also regarded as a array $c_1, c_2, \cdots, c_q, q = \lceil \text{length of } m/16 \rceil$ each of which is a 136-bit unsigned number.

Being prepare, the final output of this algorithm is given by

$$(((c_1 r^q + c_2 r^{q-1} + \cdots + c_q r^1) \mod 2^{130}) + \text{AES}_k(n))) \mod 2^{128}$$

The *AES* here takes 128-bit key and 128-bit input and returns an 128-bit output, which is done in one block.

Due to modular powering and *AES*, final result looks random while carrying authentication information of the message.

3 Security

3.1 Security Guarantee

3.1.1 Poly1305-AES

As is described above, Poly1305-AES guarantees that for any attacker to find a pair of (n, m, a) such that

$$a = \text{Poly1305}_r(m, \text{AES}_k(n)),$$

the only way to break this is to break AES, aside from receiving authenticated messages (n, m, a) sent by the sender. In other words, if the attacker failed to break AES, then any pair of (n, m, a) that does not satisfy the above equation is discarded, and only authenticated messages are received.

Besides securing that the messages are "meaningful" (in the context that the pair (n, m, a) matches the equation), even if the attacker can see the authenticated messages from the sender, or the receiver accepts it or not, the security of Poly1305-AES is guaranteed if the attacker fails to fails to break AES.

By the description mentioned in [7], there exists a quantitative representation of such guarantee of security. Let's make the following assumptions:

1. The attacker sees at most C authenticated messages from the sender.
2. The attacker attempts at most D forgeries.
3. The attacker has a probability of at most δ to distinguish AES_k, assuming uniform random permutation.
4. All messages have length at most L.

Then the attacker's chance of success is at most

$$\text{chance}_{MAX} = \delta + 1.949 \times 8D \times \frac{L}{16} \times \frac{1}{2^{106}}$$

With a small value of δ, the chance of Poly1305-AES to be broken is small too. If AES is broken, then another function should be applied to guarantee at least the same level of security.

3.1.2 HMAC

HMAC is proved to be a pseudorandom function under the assumption that the compression function of the underlying hash function is a pseudorandom function. However, if HMAC is implemented with MD5 or SHA-1, this does not guarantee the security of HMAC. Usually, the cryptographic strength of HMAC depends on the size of secret key used. And by the common attack methods on HMAC, a successful attack depends on one of the following conditions:

1. The attacker could coumpute the output of the compression function, even with initial vector unknown to the attacker (usually in brute force).

2. The attacker happens to find a collision in the hash functions.

3.2 Attack Methods

3.2.1 Multiple forgery attack

If we apply MAC with a common suffix but to two different messages, say, A and B, we have:

$$A = P_1||P_2||...||P_i||S_{i+1}||...||S_n$$
$$B = Q_1||Q_2||...||Q_i||S_{i+1}||...||S_n$$

So, the attack attempts to find a pair of distinct prefixes P and Q that yields a common internal state, and by this definition, $P||S$ and $Q||S$ are called a colliding pair, and such pair have the same authentication tag. After that, we generate a forgery by requesting a tag for message $P||S^{'}$ (also the tag for $Q||S^{'}$), and the attack is implemented.

3.2.2 Side Channel Attacks

In SCAs, the attackers gather information in side channel, such as power consumption, while the victim device is running cryptographic applications.

Usually, SCAs include the following two types of attacks:

1. **simple power analysis (SPA)**
   In this method, the attacker directly analyzes the power consumption by parsing it into a series of basic operations, like XOR, to recover the information behind it.

2. **differential power analysis (DPA)**
   In this method, instead of directly operating the power consumption, the attacker usually uses statistical tools to analyze the mean, the average, the peak of power consumption for more detailed confirmation of the assumptions the attackers made previously.

4 Application

4.1 AEAD

Authenticated Encryption (AE) is an algorithm form that operates the encryption and the authentication at the same time. Typical AE contains

- **Encrypt-then-MAC (EtM)** First, plain text P is encrypted using key K_1, yielding cipher text C. Then C is hashed using another key K_2, giving a MAC M. Finally, $\{C, M\}$ is sent.

- **MAC-then-Encrypt (MtE)** First, plain text P is hashed using key K, giving a MAC M. Then $\{P, M\}$ is encrypted using the same key K, yielding cipher text C. Finally, C is sent.

MtE is proved to be vulnerable, and EtM is suggested to use in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) [8].

Authenticated encryption with associated data (AEAD) is a variant of AE that allows a recipient to check the integrity of both the encrypted and unencrypted information in a message [9]. AEAD adds associated data (AD) that is not encrypted to the cipher and to the context where it is supposed to appear. If the cipher appears in a different context, the decryption can fail even with the correct key. So with AEAD, it protects against the operation of copying and pasting a cipher text into another context. AEAD is very similar to EtM, except that AD is not hashed or encrypted and should be known as input, MAC is a hashed value and can be sent together with cipher.

4.2 TLS 1.2

Message Authentication Code (MAC) is adopted in Transport Layer Security (TLS) Protocol. TLS is a widely used protocol that allows client/server applications to communicate over the Internet in a way that is to prevent eavesdropping, tampering, and message forgery [10]. A basic TLS 1.2 handshake [10] goes as

1. $\rightarrow$ Client Hello
2. $\leftarrow$ Server Hello
3. $\leftarrow$ Server Certificate
4. $\leftarrow$ Server Key Exchange
5. $\leftarrow$ Server Hello Done
6. $\rightarrow$ Client Certificate
7. $\rightarrow$ Client Key Exchange
8. $\rightarrow$ Client Change Cipher Spec
9. $\rightarrow$ Finished
10. $\leftarrow$ Server Change Cipher Spec
11. $\leftarrow$ Finished
12. $\leftrightarrow$ Application Data

In TLS 1.2, Message Authentication Code (MAC) is checked in the finished stage. All previous exchanged data are calculated with the master secret to get a key-hashed MAC and checked on both client and server side. Supported MAC algorithms include *HMAC-MD5*, *HMAC-SHA1*, *HMAC-SHA2* and *AEAD* [10].

4.3 TLS 1.3

TLS 1.3 has changed greatly in MAC usage since TLS 1.2. A basic TLS 1.3 handshake [11] goes as

1. $\rightarrow$ Client Hello
2. $\rightarrow$ Client Key Exchange
3. $\leftarrow$ Server Hello
4. $\leftarrow$ Server Key Exchange

5. ← Server Parameters
6. ← Server Certificate
7. ← Server Certificate Verify
8. ← Finished
9. → Client Certificate
10. → Client Certificate Verify
11. → Finished
12. ↔ Application Data

In TLS 1.3, the only supported MAC algorithm for the symmetric encryption is *AEAD* [11], and all *HMAC* hash functions are removed. However, *HMAC* is still used in finished stage to check the certificate and certificate verify information. The key is instead generated from the master secret with a key derivation function [11].

5 Conclusion

In this project, we studied the mechanism and structure of MAC. We learnt that MAC plays a critical role in the authentication field, and it is widely used in Internet connection protocols. The general process of MAC is clearly explained, and we explored two implementations of MAC algorithms: HMAC and CMAC. The security of CMAC based on Poly1305-AES is proved to be as secure as AES, while HMAC is figured out to perform as a pseudorandom function. Beyond this, brute force and side channel attacks are also introduced. Through this work, we are able to further expand our knowledge in the vast cryptography area and improve major researching skills.

References

[1] Behrouz A. Forouzan. *Cryptography and network security*. McGraw-Hill Higher Education, 2008.

[2] *Data integrity and compliance with cgmp guidance for industry*. URL: https://www.fda.gov/media/97005/download.

[3] Adrian McCullagh and William Caelli. "Non-repudiation in the digital environment". In: 27.6 (2022). URL: https://firstmonday.org/ojs/index.php/fm/article/download/778/687?inline=1.

[4] Ravneet Kaur and Amandeep Kaur. "Digital Signature". In: *2012 International Conference on Computing Sciences*. 2012, pp. 295–301. DOI: 10.1109/ICCS.2012.25.

[5] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. "Keying Hash Functions for Message Authentication". In: This version of our paper has been truncated due to page limits. The full version is [3]. Berlin, Heidelberg: Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–15. ISBN: 0302-9743. DOI: 10.1007/3-540-68697-5_1.

[6] "Federal information processing standards publication. : NO180 1993". In: (1993). 102215427. ISSN: 0038-1816.

[7] D. J. Bernstein, H. Gilbert, and H. Handschuh. "The Poly1305-AES message-authentication code". In: Berlin, Heidelberg: Berlin, Heidelberg: Springer, 2005, pp. 32–49. ISBN: 0302-9743. DOI: 10.1007/11502760_3.

[8] Peter Gutmann. *Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)*. RFC 7366. Sept. 2014. DOI: 10.17487/RFC7366. URL: https://www.rfc-editor.org/info/rfc7366.

[9] *NIST issues first call for 'lightweight cryptography' to protect small electronics*. 2018. URL: https://www.nist.gov/news-events/news/2018/04/nist-issues-first-call-lightweight-cryptography-protect-small-electronics.

[10] Eric Rescorla and Tim Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Aug. 2008. DOI: 10.17487/RFC5246. URL: https://www.rfc-editor.org/info/rfc5246.

[11] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446.