CHAPTER

6

Message Authentication Codes

A *message authentication code*, or MAC, is a construction that detects tampering with messages. Encryption prevents Eve from reading the messages but does not prevent her from manipulating the messages. This is where the MAC comes in. Like encryption, MACs use a secret key, K, known to both Alice and Bob but not to Eve. Alice sends not just the message m, but also a MAC value computed by a MAC function. Bob checks that the MAC value of the message received equals the MAC value received. If they do not match, he discards the message as unauthenticated. Eve cannot manipulate the message because without K she cannot find the correct MAC value to send with the manipulated message.

In this chapter we will only consider authentication. The mechanisms for combining encryption and authentication will be dealt with in Chapter 7.

6.1 What a MAC Does

A MAC is a function that takes two arguments, a fixed-size key K and an arbitrarily sized message m, and produces a fixed-size MAC value. We'll write the MAC function as $\textsc{mac}(K, m)$. To authenticate a message, Alice sends not only the message m but also the MAC code $\textsc{mac}(K, m)$, also called the tag. Suppose Bob, also with key K, receives a message m' and a tag T. Bob uses the MAC verification algorithm to verify that T is a valid MAC under key K for message m'.

We start with a look at the MAC function in isolation. Be warned that using a MAC function properly is more complicated than just applying it to the message. We'll get to those problems later on, in Section 6.7.

6.2 The Ideal MAC and MAC Security

There are various ways to define the security of a MAC. We describe here our preferred definition. This definition is based on the notion of an ideal MAC function, which is very similar to the notion of an ideal block cipher. The primary difference is that block ciphers are permutations, whereas MACs are not. This is our preferred definition because it encompasses a broad range of attacks, including weak key attacks, related-key attacks, and more.

The ideal MAC is a random mapping. Let n be the number of bits in the result of a MAC. Our definition of an ideal MAC is:

Definition 8 *An ideal MAC function is a random mapping from all possible inputs to n-bit outputs.*

Remember that, in this definition, the MAC takes two inputs, a key and a message. In practice, the key K is not known to the attacker or, more precisely, it is not fully known. There could be a weakness in the rest of the system that provides partial information about K to the attacker.

We define the security of a MAC as follows.

Definition 9 *An attack on a MAC is a non-generic method of distinguishing the MAC from an ideal MAC function.*

Cryptography is a broad field. There are more formal definitions that theoreticians use. When possible, we prefer the definition above because it is broader and more aligned with the full range of attacks one might consider. Our attack model includes some forms of attacks not captured by the conventional, formal definitions, such as related-key attacks and attacks that assume that the attacker has partial knowledge about the key. That is why we prefer our style of security definitions, which are robust even if the function is abused or used in an unusual environment.

The more restrictive standard definition is one in which an attacker selects n different messages of her choosing, and is given the MAC value for each of these messages. The attacker then has to come up with $n + 1$ messages, each with a valid MAC value.

6.3 CBC-MAC and CMAC

CBC-MAC is a classic method of turning a block cipher into a MAC. The key K is used as the block cipher key. The idea behind CBC-MAC is to encrypt the message m using CBC mode and then throw away all but the last block of ciphertext. For a message $P_1, \ldots, P_k$, the MAC is computed as:

$$\begin{aligned} H_0 &:= \mathrm{IV} \\ H_i &:= E_K(P_i \oplus H_{i-1}) \\ \mathrm{MAC} &:= H_k \end{aligned}$$

Sometimes the output of the CBC-MAC function is taken to be only part (e.g., half) of the last block. The most common definition of CBC-MAC requires the IV to be fixed at 0.

In general, one should never use the same key for both encryption and authentication. It is especially dangerous to use CBC encryption and CBC-MAC authentication with the same key. The MAC ends up being equal to the last ciphertext block. What's more, depending on when and how CBC encryption and CBC-MAC are applied, using the same key for both can lead to privacy compromises for CBC encryption and authenticity compromises for CBC-MAC.

Using CBC-MAC is a bit tricky, but it is generally considered secure when used correctly and when the underlying cipher is secure. Studying the strengths and weaknesses of CBC-MAC can be very educational. There are a number of different collision attacks on CBC-MAC that effectively limit the security to half the length of the block size [20]. Here is a simple collision attack: let M be a CBC-MAC function. If we know that $M(a) = M(b)$ then we also know that $M(a \parallel c) = M(b \parallel c)$. This is due to the structure of CBC-MAC. Let's illustrate this with a simple case: c consists of a single block. We have

$$\begin{aligned} M(a \parallel c) &= E_K(c \oplus M(a)) \\ M(b \parallel c) &= E_K(c \oplus M(b)) \end{aligned}$$

and these two must be equal, because $M(a) = M(b)$.

The attack proceeds in two stages. In the first stage, the attacker collects the MAC values of a large number of messages until a collision occurs. This takes 2^{64} steps for a 128-bit block cipher because of the birthday paradox. This provides the a and b for which $M(a) = M(b)$. If the attacker can now get the sender to authenticate $a \parallel c$, he can replace the message with $b \parallel c$ without changing the MAC value. The receiver will check the MAC and accept the bogus message $b \parallel c$. (Remember, we work in the paranoia model. It is quite

acceptable for the attacker to create a message and get it authenticated by the sender. There are many situations in which this is possible.) There are many extensions to this attack that work even with the addition of length fields and padding rules [20].

This is not a generic attack, as it does not work on an ideal MAC function. Finding the collision is not the problem. That can be done for an ideal MAC function in exactly the same way. But once you have two messages a and b, for which $M(a) = M(b)$, you cannot use them to forge a MAC on a new message, whereas you can do that with CBC-MAC.

As another example attack, suppose c is one block long and $M(a \parallel c) = M(b \parallel c)$. Then $M(a \parallel d) = M(b \parallel d)$ for any block d. The actual attack is similar to the one above. First the attacker collects the MAC values of a large number of messages that end in c until a collision occurs. This provides the values of a and b. The attacker then gets the sender to authenticate $a \parallel d$. Now he can replace the message with $b \parallel d$ without changing the MAC value.

There are some nice theoretical results which argue that, in the particular proof model used, CBC-MAC provides 64 bits of security when the block size is 128 bits [6] and when the MAC is only ever applied to messages that are the same length. Unfortunately, this is short of our desired design strength, though in practice it's not immediately clear how to achieve our desired design strength with 128-bit block ciphers. CBC-MAC would be fine if we could use a block cipher with a 256-bit block size.

There are other reasons why you have to be careful how you use CBC-MAC. You cannot just CBC-MAC the message itself if you wish to authenticate messages with different lengths, as that leads to simple attacks. For example, suppose a and b are both one block long, and suppose the sender MACs a, b, and $a \parallel b$. An attacker who intercepts the MAC tags for these messages can now forge the MAC for the message $b \parallel (M(b) \oplus M(a) \oplus b)$, which the sender never sent. The forged tag for this message is equal to $M(a \parallel b)$, the tag for $a \parallel b$. You can figure out why this is true as an exercise, but the problem arises from the fact that the sender MACs messages that are different lengths.

If you wish to use CBC-MAC, you should instead do the following:

1. Construct a string s from the concatenation of l and m, where l is the length of m encoded in a fixed-length format.
2. Pad s until the length is a multiple of the block size. (See Section 4.1 for details.)
3. Apply CBC-MAC to the padded string s.
4. Output the last ciphertext block, or part of that block. Do not output any of the intermediate values.

The advantage of CBC-MAC is that it uses the same type of computations as the block cipher encryption modes. In many systems, encryption and MAC

are the only two functions that are ever applied to the bulk data, so these are the two speed-critical areas. Having them use the same primitive functions makes efficient implementations easier, especially in hardware.

Still, we don't advocate the use of CBC-MAC directly, because it is difficult to use correctly. One alternate that we recommend is CMAC [42]. CMAC is based on CBC-MAC and was recently standardized by NIST. CMAC works almost exactly like CBC-MAC, except it treats the last block differently. Specifically, CMAC XORs one of two special values into the last block prior to the last block cipher encryption. These special values are derived from the CMAC key, and the specific one used by CMAC depends on whether the length of the message is a multiple of the block cipher's block length or not. The XORing of these values into the MAC disrupts the attacks that compromise CBC-MAC when used for messages of multiple lengths.

6.4 HMAC

Given that the ideal MAC is a random mapping with keys and messages as input and that we already have hash functions that (try to) behave like random mappings with messages as input, it is an obvious idea to use a hash function to build a MAC. This is exactly what HMAC does [5, 81]. The designers of HMAC were of course aware of the problems with hash functions, which we discussed in Chapter 5. For this reason, they did not define HMAC to be something simple like $\text{MAC}(K, m)$ as $h(K \parallel m)$, $h(m \parallel K)$, or even $h(K \parallel m \parallel K)$, which can create problems if you use one of the standard iterative hash functions [103].

Instead, HMAC computes $h(K \oplus a \parallel h(K \oplus b \parallel m))$, where a and b are specified constants. The message itself is only hashed once, and the output is hashed again with the key. For details, see the specifications in [5, 81]. HMAC works with any of the iterative hash functions we discussed in Chapter 5. What's more, because of HMAC's design, it's not subject to the same collision attacks that have recently undermined the security of SHA-1 [4]. This is because, in the case of HMAC, the beginning of the message to hash is based on a secret key and is not known to the attacker. This means that HMAC with SHA-1 is not as bad as straight SHA-1. But given that attacks often get better over time, we now view HMAC with SHA-1 as too risky and do not recommend its use.

The HMAC designers carefully crafted HMAC to resist attacks, and proved security bounds on the resulting construction. HMAC avoids key recovery attacks that reveal K to the attacker, and avoids attacks that can be done by the attacker without interaction with the system. However, HMAC—like CMAC—is still limited to $n/2$ bits of security, as there are generic birthday attacks against the function that make use of the internal collisions of the

iterated hash function. The HMAC construction ensures that these require $2^{n/2}$ interactions with the system under attack, which is more difficult to do than performing $2^{n/2}$ computations on your own computer.

The HMAC paper [5] presents several good examples of the problems that arise when the primitives (in this case, the hash function) have unexpected properties. This is why we are so compulsive about providing simple behavioral specifications for our cryptographic primitives.

We like the HMAC construction. It is neat, efficient, and easy to implement. It is widely used with the SHA-1 hash function, and by now you will find it in a lot of libraries. Still, to achieve our 128-bit security level, we would only use it with a 256-bit hash function such as SHA-256.

6.5 GMAC

NIST recently standardized a new MAC, called GMAC [43], that is very efficient in hardware and software. GMAC was designed for 128-bit block ciphers.

GMAC is fundamentally different from CBC-MAC, CMAC, and HMAC. The GMAC authentication function takes three values as input—the key, the message to authenticate, and a nonce. Recall that a nonce is a value that is only ever used once. CBC-MAC, CMAC, and HMAC do not take a nonce as input. If a user MACs a message with a key and a nonce, the nonce will also need to be known by the recipient. The user could explicitly send the nonce to the recipient, or the nonce might be implicit, such as a packet counter that both the sender and the recipient maintain.

Given its different interface, GMAC doesn't meet our preferred definition of MAC in Section 6.2, which involves being unable to distinguish it from an ideal MAC function. Instead, we have to use the unforgeability definition mentioned at the end of that section. Namely, we consider a model in which an attacker selects n different messages of his choosing, and is given the MAC value for each of these messages. The attacker then has to come up with $n + 1$ messages, each with a valid MAC value. If an attacker can't do this, then the MAC is unforgeable.

Under the hood, GMAC uses something called an universal hash function [125]. This is very different from the types of hash functions we discussed in Chapter 5. The details of how universal hash functions work are outside our scope, but you can think of GMAC as computing a simple mathematical function of the input message. This function is much simpler than anything like SHA-1 or SHA-256. GMAC then encrypts the output of that function with a block cipher in CTR mode to get the tag. GMAC uses a function of its nonce as the IV for CTR mode.

GMAC is standardized and is a reasonable choice in many circumstances. But we also want to offer some words of warning. Like HMAC and CMAC, GMAC only provides at most 64 bits of security. Some applications may wish to use tags that are shorter than 128 bits. However, unlike HMAC and CMAC, GMAC offers diminished security for these short tag values. Suppose an application uses GMAC but truncates the tags so that they are 32 bits long. One might expect the resulting system to offer 32 bits of security, but in fact it is possible to forge the MAC after 2^{16} tries [48]. Our recommendation is to not use GMAC when you need to produce short MAC values.

Finally, requiring the system to provide a nonce can be risky because security can be undone if the system provides the same value for the nonce more than once. As we discussed in Section 4.7, real systems fail time and time again for not correctly handling nonce generation. We therefore recommend avoiding modes that expose nonces to application developers.

6.6 Which MAC to Choose?

As you may have gathered from the previous discussion, we would choose HMAC-SHA-256: the HMAC construction using SHA-256 as a hash function. We really want to use the full 256 bits of the result. Most systems use 64- or 96-bit MAC values, and even that might seem like a lot of overhead. As far as we know, there is no collision attack on the MAC value if it is used in the traditional manner, so truncating the results from HMAC-SHA-256 to 128 bits should be safe, given current knowledge in the field.

We are not particularly happy with this situation, as we believe that it should be possible to create faster MAC functions. But until suitable functions are published and analyzed, and become broadly accepted, there is not a whole lot we can do about it. GMAC is fast, but provides only at most 64 bits of security and isn't suitable when used to produce short tags. It also requires a nonce, which is a common source of security problem.

Some of the submissions for NIST's SHA-3 competition have special modes that allow them to be used to create faster MACs. But that competition is still ongoing and it is too early to say with much confidence which submissions will be deemed secure.

6.7 Using a MAC

Using a MAC properly is much more complicated than it might initially seem. We'll discuss the major problems here.

When Bob receives the value $\textsc{mac}(K, m)$, he knows that somebody who knew the key K approved the message m. When using a MAC, you have to be very

careful that this statement is sufficient for all the security properties that you need. For example, Eve could record a message from Alice to Bob, and then send a copy to Bob at a later time. Without some kind of special protection against these sorts of attacks, Bob would accept it as a valid message from Alice. Similar problems arise if Alice and Bob use the same key K for traffic in two directions. Eve could send the message back to Alice, who would believe that it came from Bob.

In many situations, Alice and Bob want to authenticate not only the message m, but also additional data d. This additional data includes things like the message number used to prevent replay attacks, the source and destination of the message, and so on. Quite frequently these fields are part of the header of the authenticated (and often encrypted) message. The MAC has to authenticate d as well as m. The general solution is to apply the MAC to $d \parallel m$ instead of just to m. (Here we're assuming that the mapping from d and m to $d \parallel m$ is one-to-one; otherwise, we'd need to use a better encoding.)

The next issue is best captured in the following design rule:

The Horton Principle: Authenticate what is meant, not what is said.

A MAC only authenticates a string of bytes, whereas Alice and Bob want to authenticate a message with a specific meaning. The gap between what is said (i.e., the bytes sent) and what is meant (i.e., the interpretation of the message) is important.

Suppose Alice uses the MAC to authenticate $m := a \parallel b \parallel c$, where a, b, and c are some data fields. Bob receives m, and splits it into a, b, and c. But how does Bob split m into fields? Bob must have some rules, and if those rules are not compatible with the way Alice constructed the message, Bob will get the wrong field values. This would be bad, as Bob would have received authenticated bogus data. Therefore, it is vital that Bob split m into the fields that Alice put in.

This is easy to do in simple systems. Fields have a fixed size. But soon you will find a situation in which some fields need to be variable in length, or a newer version of the software will use larger fields. Of course, a new version will need a backward compatibility mode to talk to the old software. And here is the problem. Once the field length is no longer constant, Bob is deriving it from some context, and that context could be manipulated by the attacker. For example, Alice uses the old software and the old, short field sizes. Bob uses the new software. Eve, the attacker, manipulates the communications between Alice and Bob to make Bob believe that the new protocol is in use. (Details of how this works are not important; the MAC system shouldn't depend on other parts of the system being secure.) Bob happily splits the message using the larger field sizes, and gets bogus data.

This is where the Horton Principle [122] comes in.[1] You should authenticate the meaning, not the message. This means that the MAC should authenticate not only m, but also all the information that Bob uses in parsing m into its meaning. This would typically include data like protocol identifier, protocol version number, protocol message identifier, sizes for various fields, etc. One partial solution is to not just concatenate the fields but use a data structure like XML that can be parsed without further information.

The Horton Principle is one of the reasons why authentication at lower protocol levels does not provide adequate authentication for higher-level protocols. An authentication system at the IP packet level cannot know how the e-mail program is going to interpret the data. This precludes it from checking that the context in which the message is interpreted is the same as the context in which the message was sent. The only solution is to have the e-mail program provide its own authentication of the data exchanged—in addition to the authentication on the lower levels, of course.

To recap: whenever you do authentication, always think carefully about what other information should be included in the authentication. Be sure that you code all of this information, including the message, into a string of bytes in a way that can be parsed back into the fields in a unique manner. Do not forget to apply this to the concatenation of the additional data and the message we discussed at the start of this section. If you authenticate $d \parallel m$, you had better have a fixed rule on how to split the concatenation back into d and m.

6.8 Exercises

Exercise 6.1 Describe a realistic system that uses CBC-MAC for message authentication and that is vulnerable to a length extension attack against CBC-MAC.

Exercise 6.2 Suppose c is one block long, a and b are strings that are a multiple of the block length, and $M(a \parallel c) = M(b \parallel c)$. Here M is CBC-MAC. Then $M(a \parallel d) = M(b \parallel d)$ for any block d. Explain why this claim is true.

Exercise 6.3 Suppose a and b are both one block long, and suppose the sender MACs a, b, and $a \parallel b$ with CBC-MAC. An attacker who intercepts the MAC tags for these messages can now forge the MAC for the message $b \parallel (M(b) \oplus M(a) \oplus b)$, which the sender never sent. The forged tag for this

[1]For readers who did not grow up in the U.S.: this is named after one of the characters of Dr. Seuss, who was a writer of children's books [116].

message is equal to $M(a \parallel b)$, the tag for $a \parallel b$. Justify mathematically why this is true.

Exercise 6.4 Suppose message a is one block long. Suppose that an attacker has received the MAC t for a using CBC-MAC under some random key unknown to the attacker. Explain how to forge the MAC for a two-block message of your choice. What is the two-block message that you chose? What is the tag that you chose? Why is your chosen tag a valid tag for your two-block message?

Exercise 6.5 Using an existing cryptography library, compute the MAC of the message

```
4D 41 43 73 20 61 72 65 20 76 65 72 79 20 75 73
65 66 75 6C 20 69 6E 20 63 72 79 70 74 6F 67 72
61 70 68 79 21 20 20 20 20 20 20 20 20 20 20 20
```

using CBC-MAC with AES and the 256-bit key

```
80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01.
```

Exercise 6.6 Using an existing cryptography library, compute the MAC of the message

```
4D 41 43 73 20 61 72 65 20 76 65 72 79 20 75 73
65 66 75 6C 20 69 6E 20 63 72 79 70 74 6F 67 72
61 70 68 79 21
```

using HMAC with SHA-256 and the key

```
0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b
0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b 0b.
```

Exercise 6.7 Using an existing cryptography library, compute the MAC of the message

```
4D 41 43 73 20 61 72 65 20 76 65 72 79 20 75 73
65 66 75 6C 20 69 6E 20 63 72 79 70 74 6F 67 72
61 70 68 79 21
```

using GMAC with AES and the 256-bit key

```
80 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01
```

and the nonce

```
00 00 00 00 00 00 00 00 00 00 00 01.
```