Message Authentication Codes

So far, we've been largely focused on the concept of **confidentiality**, or the property of protecting information from being read by eavesdropper. Now we will focus on **integrity** and **authenticity**, which entail ensuring the message wasn't edited mid-transmission and ensuring it was sent by the person we are expecting it from.

First, for integrity: we need to construct some function that lets us verify that the message wasn't tampered with. This function will output some tag which we add to the end of the message, and the receiver can then check for themselves whether it matches. Call this function $MAC(K,M)$. There will be some secret key K that all parties in the conversation share (like a normal symmetric key).

If Alice wanted to send a message M to Bob (not a secret message), she could send $(M, MAC(K,M))$. Mallory (an eavesdropper who can also modify messages) can read M, but she cannot change M. This is because Bob will then compute $MAC(K,M')$ and see that $MAC(K,M') \neq MAC(K,M)$, and reject the message. Even if Mallory tries to change $MAC(K,M)$, she cannot do so such that it matches M' without knowing K.

Formally, we expect the following properties:

1. **Deterministic**
   The MAC must be determinsitic given the same set of arguments, so that anyone can compute it and end up with the same result.

2. **Unforgeable**
   Without knowing the secret key K, it should be impractical to create a MAC on any message. Formally, given any number of past pairs of $(M, MAC(K,M))$, it is practically impossible to find $MAC(K,M')$ for some $M' \neq M$.

With this scheme, Alice and Bob can always ensure the integrity of their messages! But how is this accomplished?

Constructing MACs

It turns out there exists a convenient and secure method of constructing MACs using cryptographic hash functions, named HMAC. First, let's revisit some key concepts from hash functions: they are **one-way**, meaning we can't easily find m from $H(m)$, and computationally indistinguishable from random noise. The latter part means the output of a hash function looks very similar to random bits, so we can't tell how close two messages are by comparing their hashes.

First, let's try constructing something known as the NMAC. Say Alice and Bob share two symmetric, 128 bit keys K_1 and K_2. If we just did $NMAC(K_1, K_2, M) = H(K_1||M)$, then we could run into length-extension attacks as seen in the last week's note. We can fix this by adding a second layer of hash: $NMAC(K_1, K_2, M) = H(K_2||H(K_1||M))$, where $K_1 \neq K_2$. Why we need different keys is a outside of the scope of this course, and has to do with reducing the security of NMAC to its underlying hash function. You can read this paper for more information if you are interested.

If we tried to add some x' to the end of this new MAC, we could only find $H(K_2||H(K_1||M)||x')$, which will not evaluate to the intended MAC (which would need to be $H(K_2||H(K_1||M||x')))$

With that, we are equipped to define the HMAC:

$$\text{HMAC}(K, M) = H(K \oplus opad||H((K \oplus ipad)||M))$$

where

$$ipad = \text{0x363636......}$$
$$opad = \text{0x5C5C5C...}$$

That formula looks a bit complicated at first glance, but let's break it down piece by piece. In the NMAC we needed two different keys K_1 and K_2, but it turns out that they don't need to be completely unrelated, just different in at least one bit. By XOR-ing with these two pads, we can make $K_2 = K \oplus opad$ and $K_1 = K \oplus ipad$, and only have to use one key K.

Why these specific constants? They are chosen to provide very different bit patterns, but theoretically we only need 1 bit difference. Cryptographers tend to be a bit paranoid, however, and for good reason!

Digital Signatures

MACs can provide integrity when two users share a key, however, we still aren't able to tell when a message is sent by a specific person. This is where **authentication** comes in, and is provided by digital signatures.

We want to have a message M "signed" via $S_M = \text{Sign}(S_K, M)$ with some secret key S_K. Anybody will then be able to take this S_M and verify that S_M is signed by the owner of the secret key corresponding to P_K. They know P_K but don't know S_K, yet can still verify that it was signed by the owner of P_K!

To construct this, we need a one-way function F_s which is everyone can compute one-way ($F_S(X)$) but only the person knowing S can find the inverse of: $F_S^{-1}(X)$. With this, we are equipped to define our signature scheme using RSA:

$$\text{Sign}(S_k, M) = M^{S_k} \mod n$$

To verify a signature S, we can raise $\text{Sign}(S_k, M) = M^{S_k} \mod N$ to the power of P_k. Recall that in RSA, the secret key is d and the public key is e. We know that $x^{ed} \equiv x \mod N$, so it follows that

$$(M^{S_k})^{P_k} \equiv M \mod N.$$

This could technically work for a signature, but there is one key modification we need to do first: M should be replaced with its hash $H(M)$. Therefore, our final signature is:

$$\text{Sign}(S_k, M) = H(M)^{S_k} \mod n$$

The verification follows the same idea. So, why does this work? Thinking back to our idea of some one-way F_S, we can see that $F(X) = X^{P_k} \mod N$ and $F^{-1}(X) = X^{S_k} \mod N$. If you recall from Note 4, it is computationally hard to recover S given $X^S \mod N$ (the discrete logarithm problem), so attackers that don't know S_x cannot find $F^{-1}(X)$.

To sign a message M, Alice would generate an RSA keypair and send $F^{-1}(H(M))$. Anyone who wanted to verify that this was true could find $F(F^{-1}(H(M))) = H(M)$. Remember, signatures don't provide confidentiality, so everyone knows M.

Why do we need $H(M)$? This is to ensure Mallory cannot just choose produce a signature on some random message. Consider the flawed scheme of $\text{Sign}(S_k, M) = M^{S_k} \mod n$. Mallory can choose any $x \mod N$ and pretend that $x = M^{S_k}$. The corresponding message would then be $x^{P_k} \mod N$, as $x^{P_k} = M^{S_k P_k} = M$. When somebody went to verify this, they would find that $(M^{S_k})^{P_k} \equiv (x)^{P_k}$.

We cannot set x^{P_k} to whatever we want, but it is still a valid forgery, and therefore this scheme is broken. To fix it, we introduce the hash function from earlier. For the same attack to work, Mallory would need to find $H(M) = (M^{S_k})^{P_k}$, which is infeasible given it's one-way nature.

Contributors:
- Ryan Cottone