Lab 1: Breaking MACs

Message Authentication Codes (MACs) are very important in the construction of cryptosystems. They allow a receiving party to ensure that the data has not been tampered with while in transit. A common construction of a MAC is by using a hash function. In this lab, we will be exploring the internals of hash functions to see how they can be exploited to break weak MAC schemes built from hashes.

Background: Hash function construction

Hash functions take a message of any length to a message of fixed length. In practice, a hash function is often built out of a small function which takes only fixed length inputs, chained together so that it can accept any input length. This smaller function is called a *compression function*. Hash functions commonly use the Merkle-Damgård construction to turn collision-resistant compression functions into collision-resistant hash functions. For example, the compression function in SHA256 takes two inputs of sizes 512 and 256 bits, and creates one 256-bit output. In order to hash a large message, the message is first padded to a multiple of 512 bits, and an initial hash value (called IV) is set. Then, each 512-bit chunk of the message is fed to the compression function, along with the previous hash value, and the final hash value i s the total hash of the message.

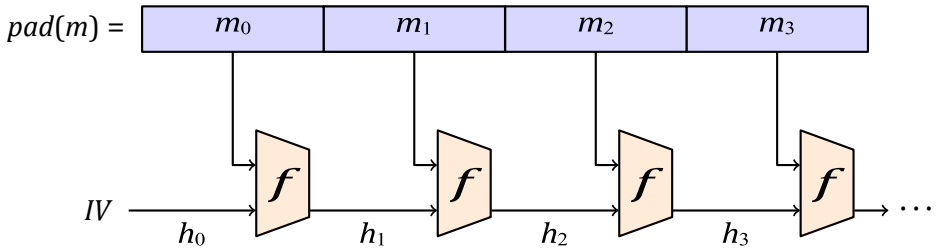

Figure 1: The Merkle-Damgård construction with compression function f.

Concretely, suppose the compression function $f : \{0,1\}^{512+256} \to \{0,1\}^{256}$ takes 768 bits of input and produces 256 bits of output. The message m is padded to a multiple of 512 bits (the *block size*), and chunked into 512-bit long blocks $m_0, m_1, \ldots, m_n$. The initial hash value h_0 is initialised to a special constant (this is known as an *initialisation vector*, or IV). Then, each subsequent hash value $h_{i+1} = f(m_i, h_i)$. At the end, h_{n+1} is the value of the hash: a 256-bit hash that depends on the contents of the entire message.

Background: MACs from Hash Functions

We explore four ways of constructing a MAC from a hash function, the first three of which are fairly natural, and the last being less natural, but the recommended IETF standard[1]. Throughout, m refers to the message sent, and k and k' are secrets that the sender and intended recipient share.

Secret prefix	$\text{MAC}_k(m) = h(k \parallel m)$	Arbitrary addition to end of message m
Secret suffix	$\text{MAC}_k(m) = h(m \parallel k)$	Vulnerable to birthday attack
Envelope	$\text{MAC}_{k,k'}(m) = h(k \parallel m \parallel k')$	More secure: envelope method
HMAC	$\text{HMAC}_k(m) = h(k \oplus opad \parallel h(k \oplus ipad \parallel m))$	IETF standard

For more details on these methods, refer to 9.5.2 of the *Handbook of Applied Cryptography*[2].

[1] See http://www.ietf.org/rfc/rfc2104.txt

[2] Chapter 9: http://cacr.uwaterloo.ca/hac/about/chap9.pdf

Exercises

If you have not had previous experience with Python, you should also review our brief introduction to Python 3. The examples during the semester will be in the Python 3 programming language and using the PyCrypto cryptographic library. You should install them on your home computer when you get a chance.

Secret Prefix Method

The secret prefix method works by prepending a secret key k to the message m you desire to send and using the resulting hash: $\text{MAC}_k(m) = h(k \parallel m)$. This method is totally insecure, as the message can be extended and the hash modified to match, without knowledge of the key.

For this question, assume that the sending and receiving party share a secret key k, and are transmitting a message m. The message and its MAC are transmitted in the clear, and the MAC of the message is $\text{MAC}_k(m) = h(k \parallel m)$ for some hash function h using the Merkle-Damgård construction. You are in a position where you can intercept the messages and modify them before forwarding them on to the receiving party.

1. Assuming the hash function h is secure, is it possible to recover the secret key k purely from the message and the MAC?
2. You now want to append a message m' onto the end of the sent message m. By drawing out the last iteration of the Merkle-Damgård construction, show how the MAC can be modified (through *hash extension*) to be consistent with this new message.
3. How difficult is this attack computationally?

Secret Suffix Method

The secret suffix method modifies the above scheme by appending the secret key k as a suffix, $h(m \parallel k)$. This prevents the previous attack (as generating a valid MAC requires ending with the secret key k) but is vulnerable to messages with colliding hashes. In the Merkle-Damgård diagram, the hash output from previous iteration is shown being fed into the next iteration. In the secret suffix method, there is nothing before the message. If an attacker can find m and m' such that $h(m) = h(m')$ then $\text{MAC}_k(m) = h(m \parallel k) = h(m' \parallel k) = \text{MAC}_k(m')$, and hence could get the sender to send an "innocent" message m, only to swap it out for another message m'.

1. Assuming the hash function h is collision resistant, how long (computationally) does this attack take in terms of n, where n is the length of the message m? (Finding any two strings $m \neq m'$ such that $h(m) = h(m')$ can be done using a *birthday attack*.)
2. Can the computation required for this sort of attack be performed offline, or do you need to be constantly eavesdropping on a conversation?
3. Which common hash algorithm which showed up in lab1a.py is not collision resistant? How long would it take to carry out a collision attack on that hash?

Envelope Method

In the envelope method, the MAC of a message m using the secret keys k, k' is $\text{MAC}_{k,k'}(m) = h(k \parallel m \parallel k')$. Placing the keys at both ends of the message prevents both of the above attacks, yet this method is still vulnerable to much more sophisticated attacks, and therefore is not recommended.

Using HMAC in PyCrypto

Recall that the HMAC construction for a message m and key k is $\text{HMAC}_k(m) = h(k \oplus opad \parallel h(k \oplus ipad \parallel m))$, where *opad* is the repeating constant 0x5c, *ipad* is the repeating constant 0x36, and $\oplus$ denotes XOR. Many different hash functions h may be "plugged into" the HMAC construction, and PyCrypto allows this.

The file lab1b.py has a sample communication between Batman and Commissioner Gordon. Open and read the file, and ensure it runs and that Batman correctly verifies Gordon's communication.

1. Try changing either Gordon's or Batman's secret keys, and ensure that the message is no longer verified.
2. Try swapping the SHA256 algorithm out for a larger hash, SHA512. Ensure that Batman can still verify Gordon's message.
3. Try changing the message in transit (below the comment "INTERCEPTIONS HERE"), and ensuring that Batman realises the message has been tampered with.
4. You are the Joker, and you know that the Batman and Gordon are using HMAC SHA256 to authenticate their messages. Since you're a crypto expert, you know the message is pretty much unable to be tampered with. Assuming you can get a copy of this message, how could you use that to your advantage? (Perhaps at a later point in time...)

Finding Collisions

In lab1c.py, we are given a set message m and perform brute force to find a message m' such that $h_k(m) = h_k(m')$. This is done by generating a large set of modified messages M' and checking if any $h(m')$ in M' is equal to $h(m)$. Whilst this works, it is expected to take 2^{n-1} attempts!

Observe how the speed decreases as the size of the hash gets larger. Also note that the bit lengths that the computer is slowing on (24-32 bits) is far smaller than the hash lengths used in the real world (minimum of 128 bits) and that difficulty increases exponentially according to the bit length.

Extension Task — Writing a Birthday Attack

Whilst the brute force method works, it is expected to take 2^{n-1} attempts! We want to take advantage of the birthday attack to speed up our attack to $2^{n/2}$.

Imagine if we could control the original message m to some degree and receive the hash $h_k(m)$ back for it. For example, an automated system may send an email with $h_k(m)$ and m = "Your friend Mallory changed her phone number to #" (where # is an arbitrarily small or large number, 0-9) any time Mallory changes her phone number. If Mallory can intercept those emails, she can generate as large a collection M of $< h(m), m >$ pairs as she'd like. Her end goal is to send Bob an email with a valid $h_k(m)$ and a mean message m = "Alice sent a message to your wall – 'I hate you' (post id: #)" (where # is again an arbitrary number, 0-9).

Help Mallory achieve this by performing a modified birthday attack. Generate a large number of variations of the initial message $M = m$ (for example, by appending a receipt number to it) and check for collisions against any of M'.

Modify your lab1c.py to now execute a birthday attack!

Tip: To do this in python, you should use a dictionary or set. You should loop continuously, on each iteration changing m_1 and m_2. The results can then be saved in a dictionary/set like where the key is $h_k(m)$. Once you find $h_k(m_1)$ in the set that holds m_2 hashes, you have found a collision.

Here's some pseudo-code example of what to do:

```
M1, M2 = {}, {}
while True:
        m1 = " Message 1 (% s)" % (rand_int ,)
        m2 = " Message 2 (% s)" % (rand_int ,)
        h1 = hash(m1)
        h2 = hash(m2)
        if h1 in M2 or h2 in M1:
                #Found a collision!
        else :
                M1[h1] = m1
                M2[h2] = m2
```