Lab 1 :: Breaking HMACs

Luke Anderson *luke@lukeanderson.com.au*

March 8, 2016

1 Introduction

In the lectures we discussed the implementation of Hash-based Message Authentication Codes (HMAC), specifically on making a HMAC from a non-keyed hash function h and a shared secret k.

HMACs commonly use the Merkle-Damgrd construction to turn collision resistant hash functions into collision resistant compression functions. Most hash functions only expect to operate on blocks of a certain size, usually with the input and output being between 128 and 512 bits in size. Compression functions need to work on any sized message, and must generate the same sized hash each time. This is important as the size of the hash is independent of the size of the message to be hashed.

For an illustration of how this works, suppose we have a hash/compression function f that takes $(n + r)$-bits input and outputs n-bits output, with $n = 128$ and $r = 256$[1]. To encode a message m of bit length 1024, we break the message down into four messages, m_1, m_2, m_3, m_4. We start off with a random bit string of length n called an Initialisation Vector (IV) as h_0. To produce the first block, we calculate $h_1 = f(h_0|m_1)$, with $(x|y)$ meaning to concatenate or stick x and y together. To produce each successive block, we calculate $h_i = f(h_{i-1}|m_i)$. At the end, h_n is an n-bit hash that depends on the contents of the entire message.

Four methods for including the secret key in the hash function were discussed:

[1]There is a graphical example in the HMAC slide on *cryptologic.org*
[2]See *http://www.ietf.org/rfc/rfc2104.txt*

$MAC_k(m) = h(k\|m)$	Arbitrary addition to end of message m
$MAC_k(m) = h(m\|k)$	Vulnerable to birthday attack
$MAC_{k,k'}(m) = h(k\|m\|k')$	More secure: envelope method
$MAC_k(m) = h(k \oplus opad\|h(k \oplus ipad\|m))$	IETF standard[2]

For more details on the methods, refer to 9.5.2 of the *Handbook of Applied Cryptography*[3].

In this lab, we'll be exploring these vulnerabilities in more detail.

2 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 $h(k|m)$. This method is entirely insecure as the message can extended arbitrarily by adding m' to the end. All that is required to generate a valid hash that contains the key is to take $h(k|m)$ and update it by appending m' (i.e. $h(k|m|m')$). Note that this attack requires near zero computation other than running the hash function once!

3 Secret Suffix Method

The secret suffix method modifies the above scheme by appending the secret key k as a suffix, $h(m|k)$. This prevents the above attack (as generating a valid HMAC requires ending with the secret key k) but is vulnerable to a birthday attack on the message itself.

The reason for this is due to how hashing works, in particular in reference to the Merkle-Damgrd method. To prevent $h(x_1|x_2)$ from being equivalent to $h(x_2|x_1)$ the initial block x_1 must impact the hashing of the subsequent block x_2. In the Merkle-Damgrd method 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 $h_k(m) = h_k(m')$. This is as the hashing of the message blocks is not dependent on the secret key k.

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

Finding m and m' only require $2^{n/2}$ attempts. This is substantially lower than brute forcing the message space which requires 2^{n-1}.

4 Envelope Method

The envelope method prevents both of these attacks. A secret key is placed at the end of the message to prevent appending to the initial message as in the prefix method vulnerability. In addition, a secret key is placed at the start, impacting the result of hashing the message and preventing a birthday attack as seen in the suffix method.

5 Before you begin

By the end of today's lab, you should have registered for Wargames. If you haven't yet done this, speak to your tutor.

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.

6 Task

Lab files *lab1a.py* and *lab1b.py* give you an introduction to hashing and HMACs using the PyCrypto library.

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.

6.1 Extension Task

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'.

Note: A similar question is likely to appear in Wargames!