**Hack The HMAC v1.0:**

1. All print statements were not compatible with Python v3: Added parentheses.
2. For all keys/passwords in which all Nn%0 characters have matching Nn%0 characters & all Nn%1 characters have matching Nn%1 characters, those keys will hash to the same value of 0.

Ex. “haha & tata & abcbaccc” will all hash to 0

This is due to the fact that cueh\_hash\_1(inp) separates Nn%0 character bit range from the Nn%1 bit range through 8-bit padding, all Nn%1 bits are XOR’d with Nn%1 bits & all Nn%1 bits are XOR’d with Nn%1 bits resulting in 0 every time.

Ex. For key ‘AB^AB’:

Binary for A: 01000001

Binary for B: 0100001000000000

XOR Process: AB^AB

0100001001000001

XOR 0100001001000001

-----------------------------------------

= 0000000000000000

1. Finding a collision key using a known message & digest that could possibly be obtained from mitm attack:

1. Convert digest to binary: 23900 = 0101110101011100 (Note: Only a lousy 16-bits)
2. Reverse the original message.
3. Iteratively convert every character in the message to binary & XOR each binary from the digest’s binary number. Store the resulting number so it can be XOR’d with every remaining character in the message.

Note1: Every 2nd character must shift it’s binary value to the right by 8 bits or effectively multiply it’s binary by 255.

Note2: If there were an odd number of characters in the message, the final binary value to be XOR’d must be that of the space character ‘ ‘.

1. Once all characters of the message have been processed, we can conclude that the remaining 16-bit binary number is a hash formed from a certain set of characters of a key/password.
2. For the collision key, we just need to convert each 8-bit half of the 16-bit binary into a character & voila!
3. Due to the nature of this algorithm producing a 16-bit binary from any length of key there is huge potential for a collision.

This is known as the ‘Pigeonhole Principle’: The larger the number of pigeons over the number of pigeon holes, the more likely there will be more than one pigeon nesting in a particular hole. Same concept, the more characters that are used to hash into a 16-bit number, the more likely multiple different characters will hash to the same value.

Resistance=number of possible inputs/number of possible hashes. (n-1+n)/65535

We can only have 65535 different hashes using 16-bit but an unlimited number of characters. This makes for a VERY weak collision resistance.

1. For the challenge of finding a key for the messages:
   1. 25193|power up gigamatrix server
   2. 21084|install toaster updates
   3. 25136|realign singularity polishing buffers
   4. 14382|enhance undulation
   5. 23900|detatch porpoise

The key was found to be: 'N ' (Note: the 2nd character between the quotes is hidden/invisible’)

With the binary values of: 01001110 & 00001001

I have also created the script which will automatically return a valid key after inputting the message digest value and the message itself. To download, please find at:

<https://github.com/EH404/CU-Semester2/blob/master/Week1/HMAC-CU%20Key%20Generator.py>

The verification of the key can be done by using the original code provided to us.

I’ve have amended it to display a range of print statements so I could grasp the concept of what the code is actually doing during runtime.

<https://github.com/EH404/CU-Semester2/blob/master/Week1/cueh_hmac_1%28edited%29.py>

There is a presentation describing the HMAC v1.0 algorithm and the key gen used to find the collision key.

<https://github.com/EH404/CU-Semester2/blob/master/Week1/HMAC_1%20Presentation1.pptx>

1. I’ve also discovered that HMACs can be vulnerable to ‘timing attacks’:

The subject over my head do attempt my own implementation but the brief description is this…

The key can be guessed based on the time a server takes to respond.

Each character of the key is guessed by sending the server several hundred guesses.

Should one of the characters be wrong, the server responds a fraction of a second quicker as it does not compare the rest of the characters once it knows a single character is already wrong.

If the character compared is right, the server will take a fraction of a second longer to respond.

Collecting the results of the test together will show every character that took the longest and therefore the key is discovered.

So for an 8-byte key, instead of having 256^8 possibilities to process via brute-force, the attacker only has just 256\*8 possibilities.

Therefore it is recommended that the code implements a response time that is constant.

**Sebastien Delhove - Group: “404 Not Found”**