CS 406 : Cryptography and Network Security Spring 2024

Lab 3 — MAC And Hash

Instructor: Manoj Prabhakaran *Lab TA:* Nilabha Saha

Challenge 1: `openssl` Sorcery

In this challenge, you will use `openssl` to calculate various MAC digests. You have been provided a `message.txt` file on which you must perform all the MAC computations. If you ever need any key or IV, use the ones provided in `key.hex` and `iv.hex` respectively.
MAC1: Compute the HMAC of `message.txt` with the SHA256 digest.
MAC2: Compute the CMAC of `message.txt` with AES-128 cipher in CBC mode.
MAC3: Compute the GMAC of `message.txt` with AES-128 cipher in GCM mode.
Store the outputs of the various MAC computations above in hex format. The flag for this challenge is:

cs406{**MAC1_MAC2_MAC3**}

Note that for this (and only this) challenge, the flag accepted is case-insensitive.

Challenge 2: Extension of Deception

Recall CBC-MAC from the lectures and recall the fact that CBC-MAC is vulnerable to a length-extension attack. In this challenge, you will implement the same.
You are provided the MAC digest of `DATA` (refer to the server script) and you need to submit the MAC digest of a string that starts with `DATA` and contains `admin=true` as a substring.
Good luck on your mission!

Challenge 3: Tick-Tock on the HMAC

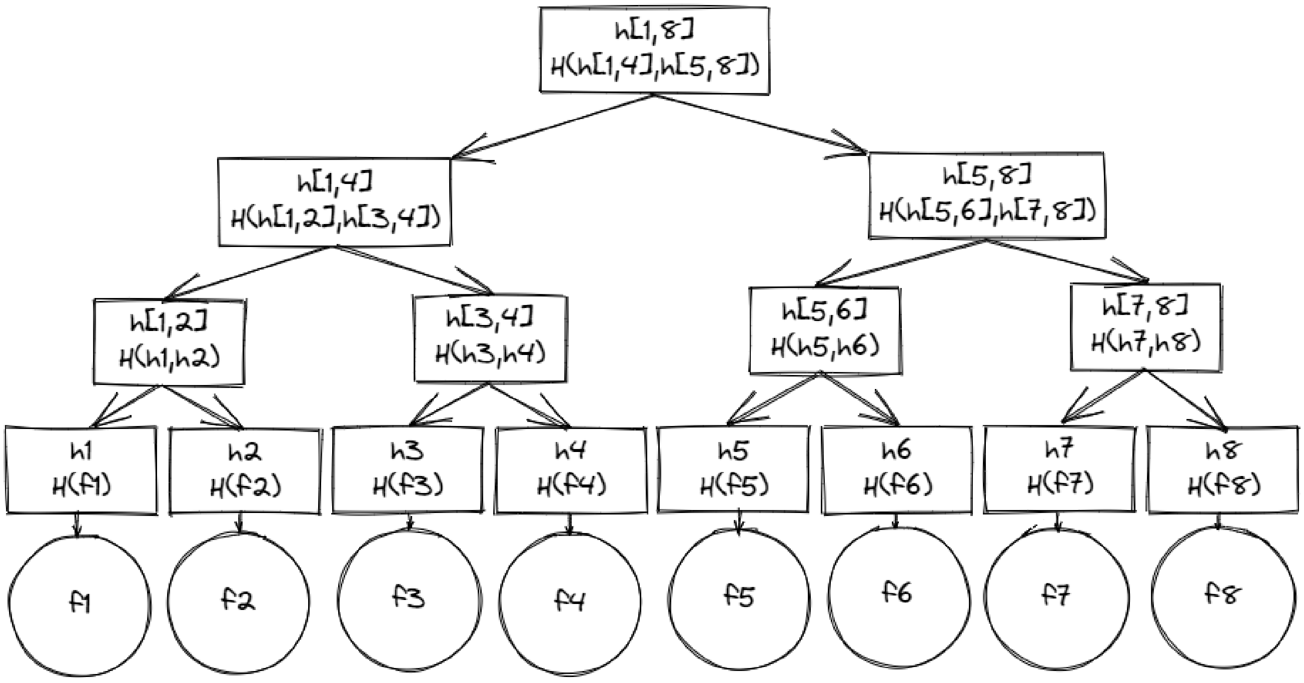
Even if your MAC-scheme is secure, it could be vulnerable to other side-channel attacks. In fact, you will implement a timing-based side channel attack on insecure digest comparisons.
Here's the crux: when you are comparing a user-submitted MAC digest to a computed MAC digest, you should **never** use the standard string comparison operator. This is because they usually perform an early exit when they find the first mismatch in the two strings. This leaks timing information in the following sense: if the string comparison takes longer, that means the user-submitted MAC has a longer matching prefix with the correct MAC. This timing leak can be used to recover the full MAC, as you will do in this challenge.
An artificial timing delay has been introduced in this challenge where each successful byte comparison consumes 1 second. Use this to your advantage to guess the first 10 characters of the MAC's hexdigest (refer to the server script for the exact details).

Secure Implementation Note: To prevent the timing leak, one must use a secure string compare function: a function that checks the two strings byte-by-byte for all the bytes even if it finds a mismatch. This allows the comparison to occur in constant time for a given string length, thus eliminating the timing leak. Programming languages often have functions defined to do this, such as the `hmac.compare_digest()` function in Python.

Note on Reality: In real life, we don't have such artificial timing delays, and the timings delays are usually much smaller. One way to deal with this is to fix a large enough number N, send the same input N times, and take the cumulative delay as the delay for that input. This allows you to magnify a minuscule timing delay δ in to an effective larger timing delay $N * \delta$. If N is chosen large enough, any real miniscule timing delay can be effectively transformed into an "artificially long" timing delay. This method, in effect, allows you to artificially increase the resolution of your time difference measurements.

Challenge 4: Commitment Issues (Merkle's Version)

One use case of hash functions is in the construction of Merkle Trees.
Merkle trees are built using an underlying cryptographic hash function. Let us assume that we have at our disposal a collision-resistant hash function H. Now, say that our vector consists of 8 elements denoted by $f_1, f_2, \ldots, f_8$. We can create a Merkle tree on them as follows:



Every node (rectangle in the figure) in the tree stores a hash:

- The i^{th} leaf of the tree stores the hash h_i of the vector element f_i.
- Each internal node of the tree stores the hash of its two children.
- The top hash stored in the root node is called the Merkle root hash.

It is often useful to think of computing the Merkle tree as computing a collision-resistant hash function, denoted by $\mathcal{MHT}$, which takes n inputs and outputs the Merkle root hash, $h_{\text{root}} = \mathcal{MHT}(f_1, \ldots, f_n)$. The Merkle root hash is sent to a verifier.
Now if a prover wants to open the component x_i and convince the verifier that the correct value of the component x_i is being sent, he sends the value of x_i, and then traverses up the Merkle tree till the immediate child of the root (inclusive) and sends the hash value stored at the sibling of each node it hits. The verifier can compute the hashes and reconstruct the path up to the root of the Merkle tree. If all the hashes check out, the authenticity of x_i stands verified. For instance, if the prover wants to reveal $f3$ in the figure, he sends f_3, $h4$, $h[1,2]$, and $h[5,8]$ to the verifier. The verifier then computes $h3 = H(f3)$, $h[3,4] = H(h3, h4)$, $h[1,4] = H(h[1,2], h[3,4])$, $h[1,8] = H(h[1,4], h[5,8])$, and finally verifies whether $h[1,8] \overset{?}{=} h_{\text{root}}$. The sequence of hashes sent by the prover constitutes the Merkle proof of x_i.

The Challenge

The server constructs a Merkle tree on top of the flag. Each character of the flag forms a lead node of the Merkle tree. The length of the flag is n, which is assured to be a perfect power of two in this challenge. You are allowed to make $n/4$ Merkle proof queries. Your mission, should you choose to accept it, is to recover the entire flag. Refer to the server script for the exact implementation details.

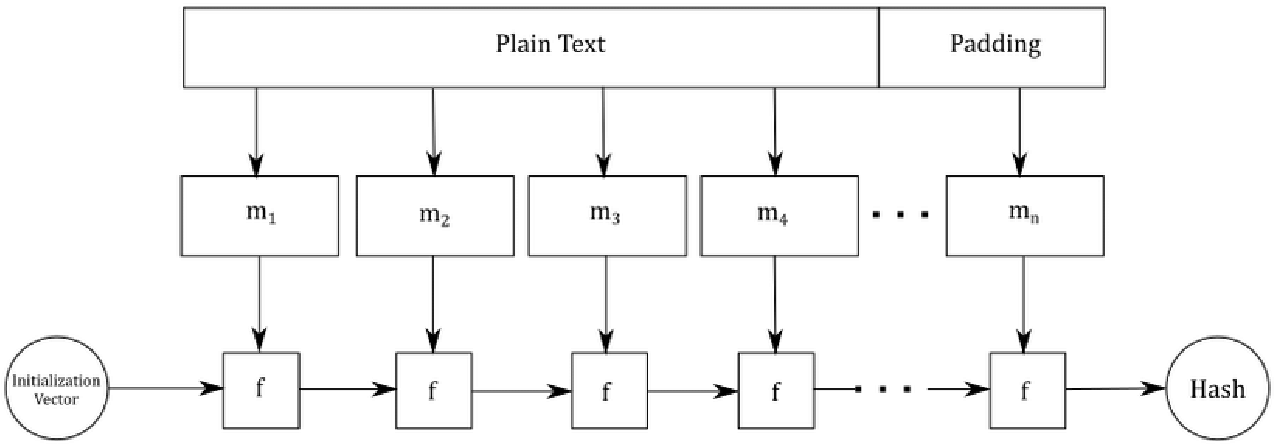
Think: Did we really violate the collision resistance of the hash function in solving this challenge?

Bonus Challenge: Nostradamus' Resurrection

We will now look at an interesting attack on certain hash functions based on the Merkle–Damgård construction. For the purposes of this challenge, we will simplify the construction in the following ways: 1) we assume there is no finalisation, and 2) we assume that the padding scheme does not incorporate the length of the message and is independent of said length.

Note on Assumptions' Removal: Assumption 1) can be easily dropped. In fact, as you read through the attack description, think of how you can implement the attack in the absence of Assumption 1). Furthermore, Assumption 2) can also be dropped using a concept called *expandable messages*, however we do not delve into the details of that in this challenge.

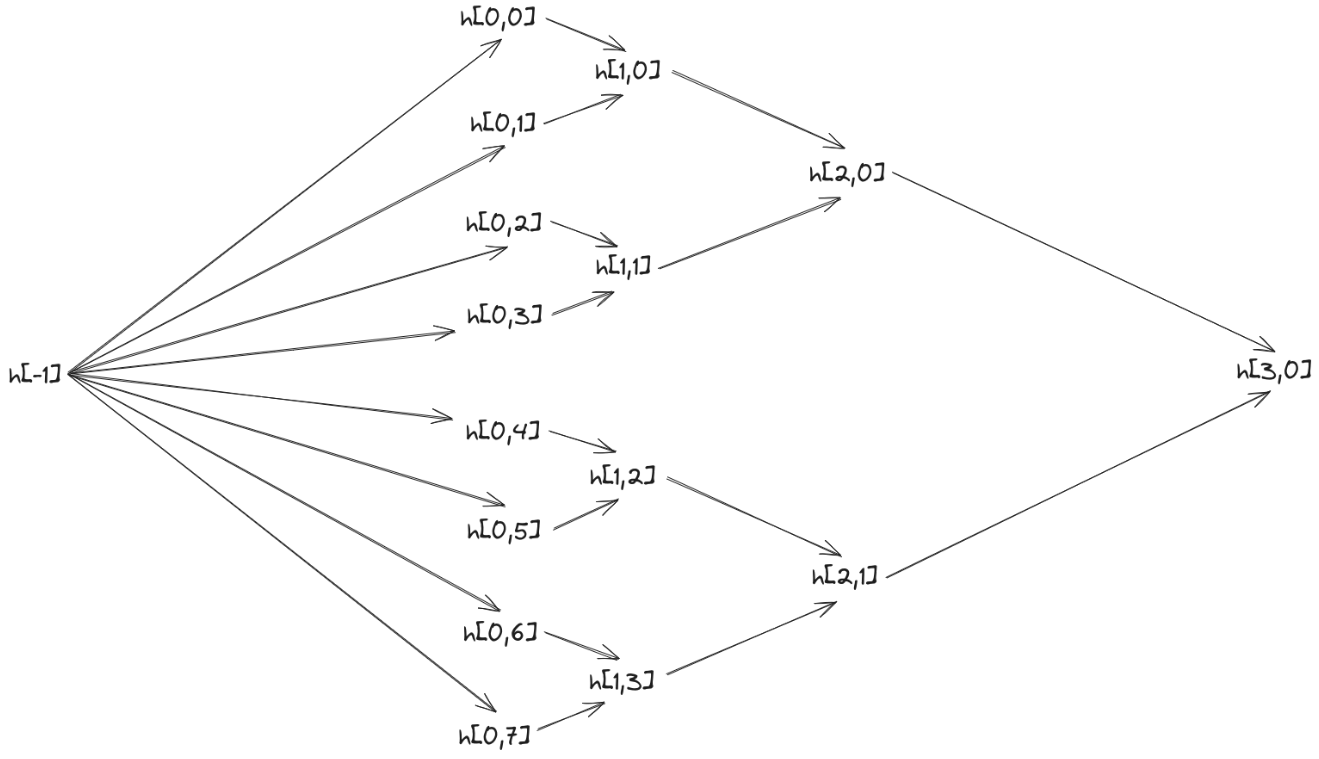
Here's a pictorial recap of how the MD-construction of our challenge would look like; here f is the compression function:



We want to achieve something one can call "Chosen Target Forced Prefix (CTFP)" collision. Here, we are allowed to submit a hash value, which we shall call the target. After this, we would be given a prefix, and we'd need to come up with a message which starts with the said (forced) prefix and hashes to the (chosen) target value we had submitted earlier. Enter the Nostradamus attack.

Say each message block is of length n. We also choose a fixed value of a natural number k (the larger the better, you can see why in a bit). Pick an arbitrary n-bit block $h[-1]$. We now compress 2^k distinct arbitrary message blocks chaining from $h[-1]$; call the outputs $h[0,0]$, $h[0,1]$, $h[0,2]$, $\dots$, $h[0,2^k-1]$, $h[0,2^k]$. Now for each even i, we find message blocks M_1, M_2 such that $f(h[0,i],M_1) = f(h[0,i+1],M_2)$; this gives us 2^{k-1} more compression outputs, call them $h[1,0]$, $h[1,1]$, $h[1,2]$, $\dots$, $h[1,2^{k-1}-1]$, $h[1,2^{k-1}]$. We continue this procedure, till we get to the final root block $h[k,0]$.

Here's a picture for visualisation:



This is called a "diamond structure". Here's the cool part: finding message block M_1, M_2 which make the two compression outputs collide can be done "easily". How so? Say the two compression outputs are C_1, C_2. Choose $2^{n/2}$ random message blocks and compute the compression output with each of those message block and C_1. Store the output in a HashSet (checks membership in amortised $\mathcal{O}(1)$ time). Now generate random message blocks and compute the compression output with each of the message blocks with C_2 until you get an output already present in the HashSet mentioned before. We've now found a collision and the corresponding message blocks we found for C_1 and C_2 become M_1 and M_2 respectively. Within $2^{n/2}$ blocks generated for C_2, you should achieve a collision with very high probability (why?). This takes $\mathcal{O}(2^{n/2})$ time as opposed to $\mathcal{O}(2^n)$ time a naïve collision-finding algorithm would take — a quadratic speedup (which makes a lot of practical difference, try it out yourself; if a commercial laptop counts from 1 to 2^{16} in a second, it takes around 18 hours to count from 1 to 2^{32}, and around 9 million years to count from 1 to 2^{64})!
Now if you have a sequence of message blocks whose compression chain output is one of the $h[i, j]$, then you can follow the edges from that compression output, appending the corresponding message blocks, and end up at $h[k, 0]$. Since there are $2^{k+1} - 1$ such compression outputs, we are highly likely to run into such a compression output "easily" (take the forced prefix, add a new block and keep modifying that block until the full compression output lies within the diamond structure). Once we are done with that, we take the last compression output of the diamond structure, h[k,0], add a padded message block to it, and pass it through the final compression function, and that would be our full hash output.
Thus, for any forced prefix, we can find a suffix of message blocks which makes the whole string hash to the chosen target.

Note on Precomputation: For a given target hash, you can precompute the diamond structure and store it. You can now receive a sequence of forced prefixes, and you can find a corresponding sequence of messages which all hash to the same target hash on-the-fly without having to recompute the diamond structure everytime, i.e., for each target hash, you only need to compute the diamond structure once, irrespective of the number of forced prefixes. For the purposes of the challenge, since the diamond structure computation can be expensive, you can store the diamond structure in a file for quick retrieval to prevent recomputation in case of an error in the latter part of your solution script.

Note on Parallelisability: One of the powerful features about the diamond structure is that its construction can largely be parallelised and spread across multiple threads to speed up computation. If you're feeling a little ambitious, try exploiting this property in your solution script! You might want to look into the `threading` library in Python for the same.

In this challenge, you will be asked to submit a target hash, and asked to send corresponding messages for a sequence of 100 forced prefixes supplied by the server. Implement the Nostradamus attack to achieve the same.

For far greater details and extensions of this idea, including why it is called the Nostradamus attack, you can refer to the original paper at https://eprint.iacr.org/2005/281.pdf.