Date

Challenge 1:

We were asked to compute 3 types of MAC's - HMAC, CMAC & GMAC on message.text using the provided key (key.hex) & IV (iv.hex) & combine them into the flag csu09m{MAC1_MAC2_MAC3}

$k = (Get-Content key.hex -Raw).Trim()
$IV = (Get-Content iv.hex -Raw).Trim()

This loads the key & IV in powershell

MAC1:

openssl mac -in message.txt -digest sha256 -macopt key:$k

-digest sha256: specifies the hash fn for HMAC

-macopt key:$k: Provides secret key

MAC2:

openssl mac -in message.txt -cipher aes-129-cbc -macopt key

-cipher aes-129-cbc: selects AES 128 in CBC mode.

CMAC uses block cipher on the provided key to generate the MAC.

MAC3:

openssl mac -in message.txt -cipher aes-128-gcm -macopt:$K -macopt IV:$IV

aes-128-gcm: selects AES-128 in GCM mode

-macopt IV:$IV: gives initialization vector

GMAC provides authentication using AES-GCM without encrypting the message.

Finally combined all 3 MAC's (in hexadecimal)

csu09m{MAC1_MAC2_MAC3}

2] CBC-MAC Reuse vulnerability

CBC MAC for message $M = P_1 | P_2 \ldots P_n$ is

$$C_i = E(P_i \oplus C_{i-1})$$

Final MAC is $T = C_n$

If you know MAC $T_{old}$ for $M_{old}$ we can forge MAC for a new longer message $M_{new} = M_{old} | P_{new}$ without knowing key K

In this Q we know MAC digest for an unknown, padded, original message $P_{orig\text{-}padded}$ we need to produce a valid MAC $T_{forged}$ for a new message $M_{forged}$ that starts with $P_{orig\text{-}padded}$ & includes string 'admin=true'.

$$T_{forged} = E(P_{admin} \oplus T_{orig})$$

$$T_{Mobius} = E(P_{Mobius} \oplus IV_{orig})$$

We must find $P_{mobius}$ s.t

$$E(P_{admin} \oplus T_{orig}) = E(P_{mobius} \oplus IV_{orig})$$

$$\therefore P_{Mobius} = P_{admin} \oplus T_{orig} \oplus IV_{orig}$$

~~Now we find MAC of $P_{mobius}$~~

We construct final message by appending padded original data & ('admin=true - - - -

forged_mobius is set to $T_{mobius}$, which is what the server calculates a final MAC of new message

Challenge 3:

Here the server ~~compare~~ compare takes ~1s per correct leading char (hex char). Thus for each prefix length i the candidate hex character that produces the largest measured response time at position 'i' is almost certainly the correct character.
So we send the reqd message to the server which computes the target HMAC & expects us to guess the HMAC hext digest.

Now we iterate over i=0---9 (the 10 hex characters we need)
Let 'known' be discovered prefix of length 'i'
For each hex digit d ∈ {0,---9, a---f}
I constructed a full guess
g = known + d + padding
can be any fixed characters to meet servers expected length

Since the time leak depends only on prefix
Now we send 'g' & measure time elapsed from just before sending to immediately after recieving the server response.

Repeat the send measure-step N times for the same 'd' & record the median (or mean) elapsed time for 'd'.

Now we select the digit 'd*' whose median elapsed time is maximal - treat 'd*' as the correct character for position 'i'.

& finally append 'd*' to 'known'.

Code steps when 'send-guess' returns success

Challenge 4:

Here we have to recover all leaf bytes of the Merkle tree (flag character) using at most N/4 proofs.

Each proof ~~requires~~ reveals sibling hashes on the path: the lowest sibling is the hash of a adjacent single byte leaf, so its brute forceable ($\leq 256$ tries).

We ~~request~~ proofs for indices 0, 4, 8 ... (one per four leaf block). N/4 such queries cover every 4-leaf group. Using brute force: for every revealed single leaf sibling hash 'h' find byte $b \in [0, \dots 255]$ with $H[b] == h$ by hashing 256 candidates. That recovers the corresponding character

Then we combine the returned leaf value(s) from the query with the brute-forced neighbour bytes to fill the entire 4 leaf block

Then recomputing parent hashes for the block (& upward) & confirm they match sibling hashes & ultimately the provided root

This exploits high leaf entropy (1 byte) & public proof data - it doesnt break collision resistance for the hash fn