Lab3 Report

Rishi Vardhan Majji
24B0969

October 2025

1 Challenge 1: openssl Sorcery

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.

CODE:

```
# Read key and IV from files
KEY=$(cat key.hex)
IV=$(cat iv.hex)

# Compute HMAC-SHA256 of message.txt using the key
MAC1=$(openssl mac -digest SHA256 -macopt hexkey:$KEY -in message.txt HMAC \
| tr -d '\n' | tr '[:lower:]' '[:upper:]')

# Compute CMAC-AES-128-CBC of message.txt using the key
MAC2=$(openssl mac -cipher AES-128-CBC -macopt hexkey:$KEY -in message.txt CMAC \
| tr -d '\n' | tr '[:lower:]' '[:upper:]')

# Compute GMAC-AES-128-GCM of message.txt using key and IV
MAC3=$(openssl mac -cipher AES-128-GCM -macopt hexiv:$IV -macopt hexkey:$KEY -in\
message.txt GMAC | tr -d '\n' | tr '[:lower:]' '[:upper:]')

# Print final flag in cs409{MAC1_MAC2_MAC3} format
printf 'cs409{%s_%s_%s}\n' "$MAC1" "$MAC2" "$MAC3"
```

2 Challenge 2: Extension of Deception

2.1 Description

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

2.2 Approach

We have the MAC of the original message $\rightarrow t_1$.
Need MAC of `originalmsg||&admin=true` $\rightarrow$ it's just continuing the MAC over the second block.
If $m_1||m_2 \rightarrow t_1||t_2$, we need to make $t_2 = \text{func}(t_1 \oplus m_2)$. We can get that without using m_1 by asking the oracle to MAC a single block chosen as $m_2 \oplus t_1 \oplus IV$.

The code goes as follows.....

```
original_mac_bytes = bytes.fromhex(original_mac)
iv_bytes = bytes.fromhex(iv)
```

we define the data we want to append

```
append_data = b"&admin=true"
append_block = pad(append_data, AES.block_size)  # this is m2 (padded)
```

Now craft the message for the oracle (Möbius Hacker)
mobius_data_bytes = $m2 \oplus t1 \oplus iv$

```
mobius_data_bytes = strxor(strxor(append_block, original_mac_bytes), iv_bytes)
mobius_data = mobius_data_bytes.hex()
```

now the oracle will compute $func(m2 \oplus t1) = t2$

What we are doing above–

1. Convert the server strings `original_mac` and `iv` from hex to bytes: `original_mac_bytes`, `iv_bytes`.
2. Set `append_data = b"&admin=true"` and pad it to a full AES block: this gives `append_block` ($= m_2$).
3. Compute `mobius_data_bytes` $= m_2 \oplus t_1 \oplus IV$ using `strxor`. Hex-encode it to get `mobius_data`.
4. Send `mobius_data` to the oracle. The oracle will do AES_Enc(IV $\oplus$ mobius_data) = AES_Enc($t_1 \oplus m_2$) = the MAC we need for $m_1||m_2$.

Now, the second part

```
DATA = b"user=cs409learner&password=V3ry$3cur3p455"
DATA = pad(DATA, AES.block_size)

creds_bytes = DATA + append_block
creds = creds_bytes.hex()
```

now this mac would be the mac for $m1||m2$

```
forged_mac = mobius_mac
```

What we are doing above–

1. Pad the original credentials to full AES blocks: `DATA` becomes m_1 (padded).
2. Append `append_block` (padded `"admin=true"`) to get `creds_bytes` = $m_1||m_2$.
3. Hex-encode `creds_bytes` as `creds` and send to the server as your credentials.
4. The tag returned by the oracle (`mobius_mac`) equals the MAC for $m_1||m_2$, so set `forged_mac = mobius_mac` and send it to the server.

We used the oracle to compute $t_2 = \text{func}(t_1 \oplus m_2)$ by sending $m_2 \oplus t_1 \oplus IV$. The oracle returned the tag for $m_1||m_2$. We then sent the combined credentials and that forged tag to the server and obtained admin.

```
the flag -->cs409{53cur1ty_f0r_4ll_t1m3_4lw4y5}
```

3 Challenge 3: Tick-Tock on the HMAC

3.1 Description

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.
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 com parison consumes 1 second. Use this to your advantage to guess the first 10 characters of the MAC's hexdigest

3.2 Approach

The server compares your HMAC guess byte-by-byte and exits early when a mismatch occurs. Each correct byte comparison adds an artificial delay of **1 second**. So: longer response time means a longer matching prefix. We use timing (and the server's "omniscient" reply when we guess exactly) to recover the first 10 hex characters of the MAC.

The code goes as follows.....

```
# The server told us the required message length in msg_len
msg = b"We are what we repeatedly do. Excellence, then, is not an act, but a habit, said A
msg = msg[:msg_len]
msg = msg.hex()
```

What we are doing above–

1. Prepare the plaintext message we want to send as bytes.
2. Truncate it to the exact length requested by the server (`msg_len`).
3. Hex-encode the truncated bytes and send that as the message.

second part–

```
HEX = "0123456789abcdef"
decoded = ""

# keep guessing until we have 10 hex chars
while len(decoded) < 10:
    found = False
    for x in HEX:
        candidate = decoded + x
        guess = candidate + "0" * (10 - len(candidate))
# build candidate: decoded + guess + padding to length 10
# do a few trials to reduce noise, take average time
        trials = 3
        times = []
        result_ = -1
        for i in range(trials):
            start = time.time()
            result = send_guess(guess)
            elapsed = time.time() - start
            times.append(elapsed)
            if result == 1:               # if server says omniscient -> done
                decoded = candidate
                result_ = 1
                break
        if result_ == 1:
            found = True
            break

        avg_t = sum(times) / len(times)
        # if avg time exceeds current matched-prefix length by ~0.5s(for latency),
        # we take it as matched
        if avg_t > (len(decoded)+0.5):
            decoded += x
            found = True
            break

    if not found:
        continue
```

if we reached full 10 chars, stop

```
    if len(decoded) == 10:
        break
```

What we are doing above–

1. Try each hex nibble (0–f) for the next position of the 10-char prefix.
2. For a candidate nibble, form a 10-char guess by padding the rest with zeros (server only checks the prefix).
3. Send the guess multiple times (here 3 trials) and record the elapsed time for each trial to reduce noise.
4. If the server replies with the special success message ("omniscient"), we got the exact prefix — set `decoded` and stop.
5. Otherwise compare the average elapsed time: if it's noticeably larger than the current matched-prefix length (we use a margin like +0.5s to tolerate latency), treat the guess("x") as correct and append it to `decoded`.
6. Repeat until we recover all 10 hex characters.

We exploit the byte-by-byte comparison timing leak: each matched byte adds 1s. By measuring response time (and confirming with the server's exact-match reply), we recover the first 10 hex chars of the MAC.

```
the flag --> cs409{k3$h4_0r_t4yl0r_5w1ft?}
```

4 Challenge 4: Commitment Issues (Merkle's Version)

4.1 Description

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.

4.2 Approach

We are allowed $n/4$ Merkle-proof queries. So we query one index every 4 bytes (indices 0,4,8,...). For each query we get the leaf value and a proof (list of hex hashes). Use the proof to recover nearby bytes: the last proof element is the immediate sibling leaf hash (one character), the second-last (if present) is a hash of a 2-byte subtree. Brute-force printable characters / printable pairs to match those hashes and fill the flag. Unknown bytes become '¿ and are sent as-is.

The code goes as follows.....

Main recovery loop (one query per 4-byte block)

```
recovered = [None] * DATA_LEN

# one query per 4-byte block (allowed = DATA_LEN/4)
for idx in range(0, DATA_LEN, 4):
    val, proof = get_proof(idx)
    recovered[idx] = val if isinstance(val, int) else val[0]

    proof_bytes = [bytes.fromhex(p) for p in proof]
# immediate sibling leaf (last proof element)
# last proof hash = sibling leaf
    if proof_bytes:
        sib_hash = proof_bytes[-1]
        sib_idx = idx ^ 1
# brute-force single printable char whose hash matches
        if recovered[sib_idx] is None:
            for ch in string.printable:
                if sha256(ch.encode()).digest() == sib_hash:
                    recovered[sib_idx] = ord(ch)
                    break

# sibling subtree of size 2 (second-last proof element) -> brute-force printable pairs
# second-last proof hash = sibling subtree (2-byte pair)
    if len(proof_bytes) >= 2:
        pair_hash = proof_bytes[-2]
        base = (idx // 4) * 4
        a_pos, b_pos = base + 2, base + 3
# brute-force printable pairs to match subtree hash
        if (0 <= a_pos < DATA_LEN) and (0 <= b_pos < DATA_LEN) and (recovered[a_pos] is No
            found = False
            for a in string.printable:
                ha = sha256(a.encode()).digest()
                for b in string.printable:
                    if sha256(ha + sha256(b.encode()).digest()).digest() == pair_hash:
                        recovered[a_pos], recovered[b_pos] = ord(a), ord(b)
                        found = True
                        break
                if found:
                    break

# fill unknowns with '?' and produce final bytes
data = bytes([c if c is not None else ord('?') for c in recovered])
```

What we are doing above–

1. Create a list `recovered` of length `DATA_LEN` to store byte values (or `None`).
2. For every index `idx` in steps of 4: call `get_proof(idx)`. The server returns the value at `idx` and the Merkle proof.
3. Convert each proof element from hex to bytes (`proof_bytes`).

4. The last proof element is the sibling leaf hash. Brute-force all printable characters and match sha256(ch) to that hash. If matched, we recovered the sibling character (set `recovered[sib_idx]`).

5. If the proof has at least 2 elements, the second-last element is the hash of a 2-byte sibling subtree. We brute-force printable pairs (a, b). For each pair we compute sha256(sha256(a) $\|$ sha256(b)) and compare with that proof hash. If matched, we recovered two bytes at positions `a_pos` and `b_pos`.

6. After looping all blocks, replace any `None` with '¿ and form the final `data` bytes to send (hex-encoded).

Query one index every 4 bytes, use the proof hashes to brute-force sibling leaves and 2-byte sibling subtrees, fill recovered array, replace unknowns with "?" and submit the hex.

```
the flag --> cs409{maybe_you_don't_know_what's_lost_'til_you_find_it_merkle!}
```