CS 406: Cryptography and Network Security
Hands On - 3

Chaitanya Garg, 210050039

March 21, 2024

Challenge - 1

This challenge was pretty straight forward, we just use the 'openssl mac' command with appropriate flags to get the desired flag.

The script used is:

```
a=$(openssl mac -digest sha256 -macopt hexkey:"$(cat key.hex)" -in message.txt HMAC)
b=$(openssl mac -cipher AES-128-CBC -macopt hexkey:"$(cat key.hex)" -in message.txt
    CMAC)
c=$(openssl mac -cipher AES-128-GCM -macopt hexkey:"$(cat key.hex)" -macopt hexiv:"$(
    cat iv.hex)" -in message.txt GMAC)
echo CS406{$a\_$b\_$c}
```

Challenge - 2

Here the idea is that lets say we know the CBC-MAC (MAC1) of a , lets say, k block message (m0). Lets say we have another block b0.
Let $m1 = m0 || b0$ (concatenate b0 to m0).
Let $m2 = b0 \oplus MAC1 \oplus iv$ (b0 xor MAC1 xor initialisation vector).
Then $MAC(m1) = MAC(m2)$.

This is because $MAC(m1)$ can be recursively written as:

$$MAC(m1) = Enc_k(MAC(m0) \oplus b0)$$
$$\implies MAC(m1) = Enc_k(MAC1 \oplus b0)$$

and also:

$$MAC(m2) = Enc_k(m2 \oplus iv)$$
$$\implies MAC(m2) = Enc_k((b0 \oplus MAC1 \oplus iv) \oplus iv)$$
$$\implies MAC(m2) = Enc_k(b0 \oplus MAC1)$$

Hence $MAC(m1) = MAC(m2)$

Now in our case $m0 = DATA$, then we know MAC of $m0$. We also know iv.
Let $b0 =$ "$\&admin = true$" with appropriate padding. Then we want the MAC of $m1$ as defined above to get the flag, but mobius does not allow this. So we can compute $m2$ as defined above and get its MAC from mobius, and it will allow it. And we already proved that $m1$ also has this mac. So we can get the flag.

The script used is:

```
DATA = b"user=cs406learner&password=V3ry$3cur3p455"
DATA = pad(DATA, AES.block_size)

s = "&admin=true".encode()
s = pad(s, AES.block_size)
d = strxor(s, bytes.fromhex(original_mac))
d = strxor(d, bytes.fromhex(iv))
mobius_data = d.hex()
# ===== YOUR CODE ABOVE =====

# ===== YOUR CODE BELOW =====
creds = (DATA.hex() + s.hex())
forged_mac = mobius_mac
```

Challenge - 3

Here we just send a random message, then we start guessing from the 0 - string. We then iterate from the first char to the last char in this guess. In each char, we iterate over 0-f hex value and compute the time for the send_guess function to return response. Based on if this time is greater than the number of characters till now (greater because small process communication and computation time) we can deduce the current correct char of MAC. After iterating over all chars in string, we would have got the correct MAC, and hence the flag.

The script used is:

```
msg = "41" * msg_len

res = -1
guess = "0"*10
for idx in range(10):
    for g in range(16):
        h = hex(g)[-1]

        guess = guess[:idx] + h + guess[idx+1:]
        start = time.time()
        res = send_guess(guess)
        end = time.time()

        if res == 1:
            break

        time_taken = end - start

        if time_taken > idx + 1:
            break

    if res == 1:
        break
```

Challenge - 4

Here the idea is, lets say I send index 0, then the oracle will give me the value at index 0 ($f0$), $h1$ and $h[2-3]$, other values can be ignored.
Then I directly know $f0$. For $f1$, I can iterate over the 256 possible chars and compare $H(f1) == h1$ to get the correct value of $f1$. Similarly for $f3$ and $f4$, there will be 256 * 256 = 65536 possible pairs of chars, we can compute what $h[2-3]$ would have been for each pair and compare will the value given to find the correct pair.
Hence with index 0, we can find $f0$ to $f3$ very efficiently.
So now its pretty simple, we can send index 0 to get $f0 - f3$, then send index 4 to get $f4 - f8$ and so on. Hence in $n/4$ iterations we will get the complete flag.

Note : We can optimize our algorithm using the fact that out flag may only have a limited number of characters (human readable, ascii code 33 to 126, so 94 characters), hence reducing the iterations for $f1$ from 256 to 94 and for $f2$ and $f3$ from 65536 to 8836.
Also we don't need to iterate over all pairs or characters to find $f(4n+1)$ to $f(4n+3)$ for each index. We can have a initial calculation to create a reverse map, from the hash to the char or pair of chars and then while iterating just lookup in said map.
But for this question, the simple unoptimized solution was fast enough, so I didn't bother optimizing.

The script used is:

```
from pwn import *
import time
from hashlib import sha256

HOST = "0.cloud.chals.io"
PORT = 19258

# Uncomment the 'process' line below when you want to test locally, uncomment the 'remote' line below when you want to execute your exploit on the server
# target = process(["python3", "./server.py"])
target = remote(HOST, PORT)

def recvuntil(msg):
```

```
    resp = target.recvuntil(msg.encode()).decode()
    print(resp, end='')
    return resp

def sendline(msg):
    print(msg)
    target.sendline(msg.encode())

def recvline():
    resp = target.recvline().decode()
    print(resp, end='')
    return resp

def recvall():
    resp = target.recvall().decode()
    print(resp, end='')
    return resp


recvuntil("Length: ")
flag_len = int(recvuntil("\n")[:-1])

flag = []
for idx in range(0, flag_len, 4):
    recvuntil(f"{flag_len - 1}: ")
    sendline(str(idx))
    recvuntil("Value: ")
    val = int(recvuntil("\n")[:-1])
    recvuntil("Proof: ")
    proof = eval(recvuntil("]\n")[:-1])
    flag.append(chr(val))

    nxt = proof[-1]
    for f in range(256):
        hash = sha256(chr(f).encode()).digest().hex()

        if hash == nxt:
            flag.append(chr(f))
            break

    nxt = proof[-2]
    for f1 in range(256):
        done = False
        for f2 in range(256):
            hash1 = sha256(chr(f1).encode()).digest()
            hash2 = sha256(chr(f2).encode()).digest()
            hash = sha256(hash1 + hash2).digest().hex()

            if hash == nxt:
                flag.append(chr(f1))
                flag.append(chr(f2))
                done = True
                break

        if done:
            break

print("\n\nFlag : ", ''.join(flag), "\n")

target.close()
```

Bonus

The whole algorithm of the attack is already given and very well explained, so we just had to write code for it. I wrote the generator.py script that would generate the h_matrix and h_map (map from a hash to the position of the hash in the matrix) for a given k, and store it into a file.
Then in solution_template.py firstly I wrote the code to load h_matrix and h_map from the file and then send target hash as the compression of the final hash in matrix, with a padding block (because it is added while checking in server).

Then the get_cpft_message function is straight forward. With the prefix given, we first pad it for

message length consistency, then we try to find its digest in the map. If not found we add a random block to padded prefix and check again, until we find it in map.
Then we know to position of this hash in h_matrix, hence we can append appropriate blocks stored in the matrix in the path from this node to final node to get a message whose hash would be the same as final hash.
We can then send this message as it would give correct target hash.

The script used is:
generator.py

```
from hashlib import md5
from Crypto.Random import get_random_bytes

n = 4
k = 5
iv = b"\x42"*n

def compress(chain: bytes, block: bytes) -> bytes:
    return md5(chain + block).digest()[:n]

def digest(data) -> bytes:
    chain_value = iv
    for idx in range(0, len(data), n):
        block = data[idx:idx+n]
        chain_value = compress(chain_value, block)
    return chain_value

h_origin = b"aaaa"
h_origin_hash = compress(iv, h_origin)

h_matrix = [[]]
h_map = {}
hset = set()
for i in range(2**k):
    rand_block = get_random_bytes(n)
    while rand_block in hset:
        rand_block = get_random_bytes(n)
    hset.add(rand_block)
    h_matrix[0].append([compress(h_origin_hash, rand_block), rand_block])

    h_map[h_matrix[0][i][0]] = [0, i]


for p in range(k - 1, -1, -1):
    h_matrix.append([])
    for i in range(2 ** p):
        idx = 2 * i
        hmap = {}
        for _ in range(2 ** (4 * n)):
            m1 = get_random_bytes(n)
            c1_ = compress(h_matrix[k - 1 - p][idx][0], m1)
            while c1_ in hmap:
                m1 = get_random_bytes(n)
                c1_ = compress(h_matrix[k - 1 - p][idx][0], m1)

            hmap[c1_] = m1

        m2 = get_random_bytes(n)
        c2_ = compress(h_matrix[k - 1 - p][idx + 1][0], m2)
        while c2_ not in hmap:
            m2 = get_random_bytes(n)
            c2_ = compress(h_matrix[k - 1 - p][idx + 1][0], m2)


        h_matrix[k - p].append([c2_, hmap[c2_], m2])
        if h_matrix[k - p][i][0] != compress(h_matrix[k - 1 - p][2 * i][0], h_matrix[
    k - p][i][1]):
            raise Exception("Fault 1")
        elif h_matrix[k - p][i][0] != compress(h_matrix[k - 1 - p][2 * i + 1][0],
    h_matrix[k - p][i][2]):
            raise Exception("Fault 2")

        h_map[c2_] = [k - p, i]


```

```
with open ("save.txt", "w") as file:
    file.write(f"{n} {k} {iv.decode()}\n")

    for i in range(2**k):
        file.write(f"{h_matrix[0][i][0].hex()} {h_matrix[0][i][1].hex()}\n")

    for level in range(1, k + 1):
        for i in range(2 ** (k - level)):
            file.write(f"{h_matrix[level][i][0].hex()} {h_matrix[level][i][1].hex()} {h_matrix[level][i][2].hex()}\n")

    file.write(f"{len(h_map)}\n")

    for key in h_map:
        file.write(f"{key.hex()} {h_map[key][0]} {h_map[key][1]}\n")
```

solution_template.py

```

def compress(chain: bytes, block: bytes) -> bytes:
    return md5(chain + block).digest()[:n]

def digest(data) -> bytes:
    chain_value = iv
    for idx in range(0, len(data), n):
        block = data[idx:idx+n]
        chain_value = compress(chain_value, block)
    return chain_value

with open("save.txt", "r") as file:
    line = file.readline()

    l0 = line.split()
    n = int(l0[0])
    k = int(l0[1])
    iv = l0[2].encode()

    h_matrix = [[]]

    for i in range(2**k):
        line = file.readline()
        l0 = line.split()

        h_matrix[0].append([bytes.fromhex(l0[0]), bytes.fromhex(l0[1])])

    for level in range(1, k + 1):
        h_matrix.append([])
        for i in range(2 ** (k - level)):
            line = file.readline()
            l0 = line.split()

            h_matrix[-1].append([bytes.fromhex(l0[0]), bytes.fromhex(l0[1]), bytes.fromhex(l0[2])])

    num = int(file.readline())
    h_map = {}

    for _ in range(num):
        line = file.readline()
        l0 = line.split()

        h_map[bytes.fromhex(l0[0])] = [int(l0[1]), int(l0[2])]

padding_block = pad(b'', n)

target_hash = compress(h_matrix[k][0][0], padding_block).hex()
# ===== YOUR CODE ABOVE =====


def get_cpft_message(prefix: bytes) -> bytes:
    level = None
    index = None

    prefix = pad(prefix, n)
    message = prefix

```

```
    while True:
        dgst = digest(message)

        if dgst in h_map:
            level = h_map[dgst][0]
            index = h_map[dgst][1]
            break

        new_block = get_random_bytes(n)
        message = prefix + new_block

    while level < k:
        nextIdx = index // 2
        if index % 2 == 0:
            message = message + h_matrix[level + 1][nextIdx][1]
        else:
            message = message + h_matrix[level + 1][nextIdx][2]
        level += 1
        index = nextIdx

    return message
```