Homework 4

Posted: Wednesday, November 1, 2023 – 11:59pm
Due: Wednesday, November 8, 2023 – 11:59pm

Task 1 – Encrypt & MAC (15 points)

We consider an instance of Encrypt-and-MAC (which we refer to as E&M), where the plaintext is encrypted using counter-mode encryption with AES, and then, to get the final ciphertext, we append to the counter-mode ciphertext a MAC of the *plaintext* using some MAC algorithm that produces 256 bit tags. (The specific scheme used here is HMAC with SHA-256; however, the concrete scheme is not important for solving this question.) Decryption would recover the plaintext first from the counter-mode ciphertext, but only output it if the MAC tag is correct; otherwise, it outputs $\perp$ if the MAC tag is incorrect.

a) **[3 points]** Imagine that you intercept the following two ciphertexts C_1 and C_2, using the same secret key for both (which is however unknown to you):

C_1 =
```
43 37 13 80 52 47 53 18 8f d5 71 d8 62 2a e6 1f
64 cc b5 51 d9 b3 48 11 9a db c4 10 cb de f7 7c
f1 80 f7 52 9d 0d a0 c6 f0 a4 fd b2 8a 3d 56 e2
10 5c
7e b4 b1 3b 8c 4c d9 00 15 23 ba 1e 55 dc 2c d5
60 8e 84 c0 93 cd 21 d1 12 6d da c1 e7 b2 a5 e9
```

C_2 =
```
85 3b 56 f7 98 57 35 9c ad 58 2e b6 e6 cb 1a 23
b9 a0 8d 1c 32 e8 63 8d a8 06 71 44 b9 d7 81 79
5a 0a 79 49 6c a1 5f fe 88 65 40 8a a8 31 94 df
66 d8
7e b4 b1 3b 8c 4c d9 00 15 23 ba 1e 55 dc 2c d5
60 8e 84 c0 93 cd 21 d1 12 6d da c1 e7 b2 a5 e9
```

What can you infer about the plaintexts encrypted by the two ciphertexts above? Justify your answer!

b) **[3 points]** Is E&M a good scheme in terms of IND-CPA security? Explain your answer!

c) **[9 points]** Show that E&M does not satisfy ciphertext integrity. In particular, describe a concrete attack strategy against INT-CTXT security.

Hint: There is an attacker that after getting two correctly generated message ciphertext examples (M_1, C_1), (M_2, C_2) is able to generate a new ciphertext C_3 different from C_1, C_2 for which decryption does not output error.

Task 2 – Authenticated Encryption (15 points)

We consider a variant of AES-based counter-mode encryption (with equal block and key length, both 16 bytes = 128 bits), with one additional block used to guarantee integrity, and using PKCS#7 padding for simplicity to guarantee equal-length encrypted blocks.

Concretely, to encrypt a message M with a secret key K, the encryption algorithm proceeds as follows:

1. It pads the message M into blocks $M[1], \ldots, M[\ell]$ for some $\ell \geq 1$ using PKCS#7 padding. Each block is now 16-byte long.
2. It creates a new block $M[\ell+1] = M[1] \oplus \cdots \oplus M[\ell]$, where $\oplus$ denotes bit-wise XOR.
3. It selects a random 16-byte initialization value IV.
4. It computes $C[0] = \text{IV}$, and

$$C[i] = \mathsf{AES}_K(\text{IV} + i) \oplus M[i]$$

for all $i = 1, \ldots, \ell+1$. (The meaning of $+$ on strings is the same as explained in class, and $\mathsf{AES}_K(X)$ is the output of AES on key K and block X.)

The final ciphertext is $C = C[0] \| C[1] \| \cdots \| C[\ell] \| C[\ell+1]$, where $\|$ denotes concatenation of strings.

The decryption algorithm first recovers $M[1], \ldots, M[\ell], M[\ell+1]$ from C as in counter mode. Then, if $M[1] \oplus \cdots \oplus M[\ell] = M[\ell+1]$ *and* $M[1] \oplus \cdots \oplus M[\ell]$ has valid padding, it outputs M (obtained by removing the padding from $M[1], \ldots, M[\ell]$). Otherwise, it outputs $\perp$.

a) [6 points] Does this scheme guarantee ciphertext integrity? Explain!

b) [9 points] Imagine we modify the above scheme so that the last ciphertext block is now

$$C[\ell+1] = \mathsf{AES}_K(\text{IV} \oplus M[1] \oplus \cdots \oplus M[\ell]) \; .$$

Decryption would now check that this equality is satisfied, instead of $M[1] \oplus \cdots \oplus M[\ell] = M[\ell+1]$.

Does the resulting scheme satisfy ciphertext integrity?

Task 3 – Set Commitments (10 points)

A storage server stores m files $F_1, \ldots, F_m$, and makes them available to users. To guarantee integrity of these files, the server initially (reliably) publishes a single "commitment" com and later users use com to verify a downloaded file is unaltered. A straw-man solution is to simply let $\mathsf{com} = \mathsf{com}_1 \| \mathsf{com}_2 \| \cdots \| \mathsf{com}_m$, where $\mathsf{com}_i = \mathsf{H}(S, F_i)$ for all $i = 1, \ldots, m$, for a collision-resistant hash function $\mathsf{H} : \{0,1\}^s \times \{0,1\}^* \to \{0,1\}^n$ and a public seed S. When the user downloads file F_i from the server, they accept the file if $\mathsf{H}(S, F_i) = \mathsf{com}_i$. The drawback is that the size of com grows *linearly* in m – i.e., $|\mathsf{com}| = n \cdot m$. We want solutions that improve upon this.

Assuming m is a power of two, a different proposal is to assign $\mathsf{H}(S, F_1), \ldots, \mathsf{H}(S, F_m)$ to the m leaves of a binary tree, and then the value of every node of the tree is $\mathsf{H}(S, h_l \| h_r)$, where h_l, h_r are the values assigned to the left and right child, respectively, of the node. Finally, the server publishes the value assigned to the root of tree as the hash, which we denote as com.

What does the server need to send along with the file F_i to allow a client retrieving that file to verify its integrity with respect to com? In your solution, the content sent by the server (excluding the verbatim copy of F_i) must be $O(n \log m)$ bits long. Provide an intuition of why this solution is secure, i.e., the adversary cannot change any file without being detected. No formal proof needed.

Task 4 – Number Theory & Groups (10 points)

a) [2 points] List all elements of $\mathbb{Z}_{35}^*$.

b) [3 points] Find a generator of $\mathbb{Z}_{37}^*$.

c) [5 points] Find $x \in \mathbb{Z}_{187}$ such that $125x \equiv 4 \pmod{187}$. Explain how you found x.

Bonus Task – Timing Attacks (15 points)

Note: We will set up a separate submission for this task, with a later deadline, and its score will be used to round up your overall homework score at the end of the quarter. Your score will not be added to the score of Homework 4.

The goal of this task is to practice with the notion of a *timing side channel*, and see how it can break an otherwise secure authenticated encryption scheme.

To this end, we consider Encrypt-then-MAC encryption, combining ChaCha20-based encryption[1] with (truncated) HMAC with SHA-256, i.e., the output of HMAC (which is 256 bits) is truncated further to 64 bits.[2] The encryption procedure in particular works as follows:

```
def EtMEncrypt(key_enc,key_mac, pt):
    backend = default_backend()
    iv = os.urandom(16)
    algorithm = algorithms.ChaCha20(key_enc, iv)
    cipher = Cipher(algorithm, mode=None, backend=default_backend())
    encryptor = cipher.encryptor()
    ct = encryptor.update(pt) + encryptor.finalize()
    h = hmac.HMAC(key_mac, hashes.SHA256(), backend=default_backend())
    h.update(iv+ct)
    tag = h.finalize()
    final = iv + ct + tag[:8]
    return final
```

[1] This works like CTR mode does – ChaCha20 takes as input an IV, and a secret key, and returns an appropriately long mask to be XORed with the plaintext.

[2] Such short tags are not advisable in practice, but this will keep the attack feasible.

Decryption will first check validity of the tag (i.e., the last 8 bytes) with the actual first 8 bytes of the HMAC output on the rest of the ciphertext, and only then proceeds to decrypt the plaintext using ChaCha20. The implementation of the decryption algorithm however checks correctness of the tag using the following (intentionally faulty) equality check:

```
def CheckEq(a, b):
    if len(a) != len(b):
        return False
    for i in range(len(a)):
        sleep(0.005)
        if a[i] != b[i]:
            return False
    sleep(0.005)
    return True
```

Here, `sleep`(0.005) simply pauses for (approximately) 0.005s.

Download the Python code on EdStem named `hw4-etm.py`. When executed, the Python script will initialize the variable `cipher` with a ciphertext encrypting the string "Hello world!". Your goal is to modify `cipher` into a valid encryption of the plaintext "Hello folks!" by only making calls to `EtmOracle` and timing measurements. The former returns a boolean value (True or False) depending on whether the last 8 bytes of the input ciphertext constitute a valid tag or not.

Place your attack code in the designated area in `hw4-etm.py` – in particular, assign the resulting ciphertext to `cipher_forged`, for which the plaintext is then printed to screen (if the decryption is valid), or an exception is raised (if the MAC is not correct).

Hint. You can measure timing of a sequence of operation in Python by using

```
start = timer()
# insert your code here to be timed
end = timer()
delta = end - start
```

Your submission must only modify `hw4-etm.py` in the designated area.