COMP2700 Lab 10 – Solutions

Exercise 1.

Attack A: This would yield passwords for all users.

Attack B: Finding second preimages works only when the attacker is already in possession of an existing password and hash value of a user. This would not yield the preimage of a different hash value, so it will not give the attackers the passwords of other users.

Attack C: A collision of two arbitrary hash values (and passwords) will not yield an existing password.

Exercise 2.

Use the birthday attack: for a search space of n, to have a 0.5 probability of collision one needs approximately $1.177\sqrt{n}$ random inputs. Using this formula, for hash of length 64, 128 and 160 bit, we need approximately $1.177 * 2^{32}$, $1.177 * 2^{64}$, $1.77 * 2^{80}$ random inputs, respectively.

Exercise 3

There are at least three ways of constructing a second preimage.

- If the input is longer than one block, we can construct a second preimage by exploiting the commutativity property of XOR: $x \oplus y = y \oplus x$. So reordering two adjacent blocks does not affect the value of the hash. For example, if the input x has two blocks, i.e., $x = x_1 x_2$ then $H(x) = e_k(x_1) \oplus e_k(x_2) = e_k(x_2) \oplus e_k(x_1) = H(x_2 x_1)$.

  So a second preimage of the hash of x is $x_2 x_1$ (i.e., swapping the order of the two blocks of x). If the input is longer than two blocks, we can also use the associativity property ($x \oplus (y \oplus z) = (x \oplus y) \oplus z$), in conjunction with commutativity, to rearrange the input blocks in any order, and obtain the same hash.

  Note that this attack works only when the input is longer than one block. The next attack gets around this restriction.

- Exploiting the inverse and the identity of XOR, that is, $x \oplus x = 0$ and $x \oplus 0 = x$. Given an input x, let x_1 be its first block, so x can be written as $x_1 x'$ for some x'. Then a second preimage of $H(x)$ can be constructed as follows: $y = x_1 x_1 x$. To see why this works, consider the following equality:

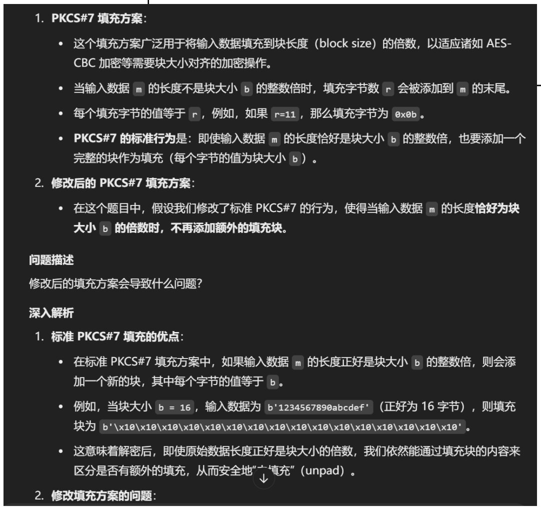
$$H(y) = H(x_1\, x_1\, x) = e_k(x_1) \oplus e_k(x_1) \oplus H(x) = H(x).$$

Since the message is longer than one block, we can use the first attack described in above. Essentially we just need to break the original input into two blocks of 16 bytes and swap the order to obtain the second preimage. Here's an example of the attack using python:

```
>>> input=b'0123456789ABCDEF0123456789abcdef'
>>> ecbhash.ecbhash(input)
'764527247f54aef5dff2d85e70c88371'
>>> snd_preimg=input[16:32] + input[0:16]
>>> snd_preimg
b'0123456789abcdef0123456789ABCDEF'
>>> ecbhash.ecbhash(snd_preimg)
'764527247f54aef5dff2d85e70c88371'
>>>
```

- We can exploit the fact the encryption is known, so we can decrypt the hash value back to a one-block message, which would be a second preimage (since the original input has two blocks, so is clearly distinct from our second preimage).

```
>>> import ecbhash

>>> input = b'0123456789ABCDEF0123456789abcdef'

>>> h = ecbhash.ecbhash(input)

>>> h_bytes = bytes.fromhex(h)

>>> from Crypto.Cipher import AES

>>> cipher = AES.new(ecbhash.key, AES.MODE_ECB)

>>> second_preimage = cipher.decrypt(h_bytes)

>>> h1 = ecbhash.ecbhash(second_preimage)

>>> h

'764527247f54aef5dff2d85e70c88371'

>>> h1

'764527247f54aef5dff2d85e70c88371'
```



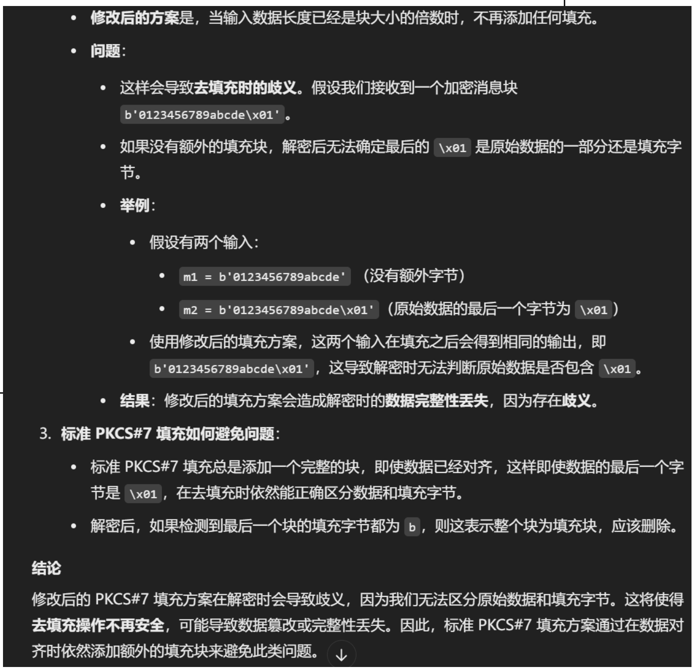
1. **PKCS#7 填充方案**:
   - 这个填充方案广泛用于将输入数据填充到块长度（block size）的倍数，以适应诸如 AES-CBC 加密等需要块大小对齐的加密操作。
   - 当输入数据 `m` 的长度不是块大小 `b` 的整数倍时，填充字节数 `r` 会被添加到 `m` 的末尾。
   - 每个填充字节的值等于 `r`，例如，如果 `r=11`，那么填充字节为 `0x0b`。
   - **PKCS#7 的标准行为**是：即使输入数据 `m` 的长度恰好是块大小 `b` 的整数倍，也要添加一个完整的块作为填充（每个字节的值为块大小 `b`）。
2. **修改后的 PKCS#7 填充方案**:
   - 在这个题目中，假设我们修改了标准 PKCS#7 的行为，使得当输入数据 `m` 的长度**恰好为块大小 `b` 的倍数时，不再添加额外的填充块。**

问题描述

修改后的填充方案会导致什么问题?

深入解析

1. **标准 PKCS#7 填充的优点**:
   - 在标准 PKCS#7 填充方案中，如果输入数据 `m` 的长度正好是块大小 `b` 的整数倍，则会添加一个新的块，其中每个字节的值等于 `b`。
   - 例如，当块大小 `b = 16`，输入数据为 `b'1234567890abcdef'`（正好为 16 字节），则填充块为 `b'\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10\x10'`。
   - 这意味着解密后，即使原始数据长度正好是块大小的倍数，我们依然能通过填充块的内容来区分是否有额外的填充，从而安全地“去填充”（unpad）。
2. **修改填充方案的问题**:



- **修改后的方案**是，当输入数据长度已经是块大小的倍数时，不再添加任何填充。
- **问题**:
  - 这样会导致**去填充时的歧义**。假设我们接收到一个加密消息块 `b'0123456789abcde\x01'`。
  - 如果没有额外的填充块，解密后无法确定最后的 `\x01` 是原始数据的一部分还是填充字节。
  - **举例**:
    - 假设有两个输入:
      - `m1 = b'0123456789abcde'` （没有额外字节）
      - `m2 = b'0123456789abcde\x01'` （原始数据的最后一个字节为 `\x01`）
    - 使用修改后的填充方案，这两个输入在填充之后会得到相同的输出，即 `b'0123456789abcde\x01'`，这导致解密时无法判断原始数据是否包含 `\x01`。
  - **结果**：修改后的填充方案会造成解密时的**数据完整性丢失**，因为存在**歧义。**

3. **标准 PKCS#7 填充如何避免问题**:
   - 标准 PKCS#7 填充总是添加一个完整的块，即使数据已经对齐，这样即使数据的最后一个字节是 `\x01`，在去填充时依然能正确区分数据和填充字节。
   - 解密后，如果检测到最后一个块的填充字节都为 `b`，则这表示整个块为填充块，应该删除。

结论

修改后的 PKCS#7 填充方案在解密时会导致歧义，因为我们无法区分原始数据和填充字节。这将使得**去填充操作不再安全**，可能导致数据篡改或完整性丢失。因此，标准 PKCS#7 填充方案通过在数据对齐时依然添加额外的填充块来避免此类问题。

Exercise 4.

The modified PKCS#7 padding can in some cases, map two different input byte sequences to the same output. For example, m1=b'0123456789abcde' and m2=b'0123456789abcde\x01' will both be mapped to m2. This creates an ambiguity in a scenario where the padding needs to be reversed to obtain the original message. In other words, in this modified PKCS#7, the unpad function will not be well defined.

Exercise 5.

The issue here is very similar to Exercise 4, in the sense that the selective application of padding creates ambiguity. In the case of SHA-1 (or any hash function for that matter), this can lead to an easy collision attack. Here is an example. Suppose X is a message of length exactly 447 bits, then the padded message will look like the following bit string:

Y = X100….000110111111

Where the highlighted part is the representation of the number '447' (the length of X) in the binary notation. If we feed this padded input (Y) into the hash function, and if the function does not apply padding to Y, then its hash value will be identical to X.

In SHA-1, the input Y in this case will still be padded, with an extra block:

Z= Y10000….0001000000000

The new block (highlighted) starts with bit '1' followed by 447 bits of 0s, and 64 bit representation of the number '512' (the length of Y). Since the padded Y now contains an extra block, it will very likely hash to a different value than the hash of X.

Exercise 6.

Recall from the lecture that CBC-MAC is implemented by using a block cipher in CBC mode: the input message is encrypted using CBC mode, and the MAC value is the last block in the ciphertext. As explained in the lecture, given a one-block message x and its CBC-MAC y, we construct another, two-block, message $x' = x \,||\, (y \oplus IV \oplus x)$ where || denotes bitstring concatenation. The MAC of x' is the last block of the ciphertext of x', i.e.,

$$e_k(y \oplus (y \oplus IV \oplus x)) = e_k(IV \oplus x) = y.$$

So the MAC for x' is the same as the MAC for x.

Here's an attack executed in python, continuing the interactive session we started in this exercise:

```
>>> from Crypto.Util.strxor import *
>>> IV=b'fedcba9876543210'
>>> m=bytes.fromhex(mac)
>>> fake_input = input + strxor(m, strxor(IV, input))
>>> fake_input
b'TESTING CBC-MAC!8@N1\xed\x8ei"2!BVo\xd7\xf9J'
>>> cbcmac.verify_mac(fake_input, key, mac)
True
>>>
```

Exercise 7.

Attacks against hash functions based on the birthday paradox are not applicable here, because the attacker can not search for an x_i such that

$$MAC_{k(x_i)} = MAC_{k(x)}$$

because the attacker does not know k. These collision-search attacks only work against hash functions, because the functions are known and do not have keys.

Exercise 8.

a) Assume the length of x is n bit and the output of the hash function h() is m bit. That is: $x = x_1 x_2 \cdots x_n$ and $h(x) = y_1 y_2 \cdots y_m$.

   Then the encryption of $x||h(x)$ has length n+m bit:

$$c_1 c_2 \cdots c_n d_1 d_2 \cdots d_m$$

   where each $c_i = z_i \oplus x_i$ and $d_j = z_{j+n} \oplus y_j$, and where the first (n+m) bit of the key stream is $z_1 z_2 \cdots z_n z_{n+1} \cdots z_{n+m}$.

   Since the attacker knows x, and the hash function is not secret, the attacker knows each x_i and can compute $h(x)$, so he/she also knows d_j . These allow the attacker to compute the key stream:
   $z_i = c_i \oplus x_i, \quad 1 \leq i \leq n$
   $z_{j+n} = d_j \oplus y_j, \quad 1 \leq j \leq n$

   Given any message x', the attacker can compute $h(x')$, and since the attacker knows the key stream $z_1, \ldots, z_{n+m}$, the attacker can encrypt $(x' \,||\, h(x'))$ using this key stream.

b) No. Although the attacker can still recover the first n-bit of the key stream $z_1 \cdots z_n$ they cannot recover the subsequent bits $z_{n+1} \cdots z_{n+m}$ used to encrypt $MAC_{k2}(x)$ since the attacker cannot compute $MAC_{(k2)}(x)$ without knowing the key k_2.