HW2: Padding Oracle Attacks

The CBC mode of operation for a block cipher can encrypt messages with length a multiple of the underlying block cipher's block length.. What will happen if the length of a message is not multiple of the block length? To extend the functionality of block ciphers to arbitrary length of messages, we use padding. In this exercise we will see how padding gives rise to so-called padding oracle attacks that allow attackers to (very often) recover messages without access to the corresponding secret key.

There are several standard ways to pad a message. For the purpose of this assignment we shall use the padding standard as described in PKCS#7 (https://tools.ietf.org/html/rfc2315#section-10.3). PKCS#7 padding works as follows. If the message requires k bytes to be appended in order for the resulting string to to be a multiple of the block length, then pad the message with k bytes each of value k. If $k = 0$ (that is the message is already a multiple of the block length), then pad it with a new block, each byte of which is set to the block length. Under these rules, note that one never pads by more than the block length in bytes, which for AES is 16 (hex value 0x10). As one example assuming block length of 16 bytes:

msg = 0x0546f702d736563726574206d657373616765212121210
padded_msg =
0x0546f702d736563726574206d6573736167652121212100**0a0a0a0a0a0a0a0a0a0a**
(padded with 10 bytes, each with value 10=0x0a).

Given a message, first, it is padded using the method mentioned above, and then encrypted using CBC mode with some block cipher. Most often these days one uses AES, which has a block length of 16 bytes. At the time of decryption, the ciphertext is first decrypted and then padding is removed to output the plaintext. To remove padding, CBC decryption is performed, and then the last byte of the resulting string is inspected. It is interpreted as the number of bytes of padding, and the decryption routine scans from right to left in the string to remove these values. If at any point the padding value is not the expected one (e.g., the final padding byte was 02, but the second to last most byte is not 02) then a padding error is raised and decryption is aborted. Often padding errors are observable, either directly because a padding error message is reported or indirectly because of timing or other issues.

Serge Vaudenay in 2002 (https://www.iacr.org/archive/eurocrypt2002/23320530/cbc02_e02d.pdf) showed how, given access to a public decryption oracle[1] and a ciphertext, it is possible to recover the whole plaintext of some challenge encryption. This is without ever having access to the secret key

[1] Oracle in cryptographic context is a black box that implements some API, and anybody can call those APIs to get a result. In our context the padding oracle is a decryption routine.

What is given?
You are given an implementation of a padding oracle simulartor via the `PaddingOracle` class, which implements the AES-CBC encryption scheme with PKCS#7 padding. This class provides 3 APIs:

```
>> from paddingoracle import PaddingOracle
>> po = PaddingOracle(msg_len=13)
>> ctx = po.setup()   # You have to call this before you can query the
                     # oracle
>> po.test("A random message")
   False  # Obviously because this is not the correct plaintext
>> po.decrypt(ctx)
   True   # This means the padding is correct.
```

What you have to do?
You have to write a function that takes a random oracle instance and an arbitrary length ciphertext, and decrypts the ciphertext by implementing a padding oracle attack.

Part 1 (50 points): First assume the ciphertext is of length 2 blocks: The first block is C_0 = iv, and the second block C_1 = F_k(msg) ⊕ C_0. Given such a ciphertext write a function that recovers the plaintext by querying the padding oracle instance.
Signature of your function should be:
po_attack_2blocks(po, ctx)

Part 2 (30 points): In the second part you have to extend your attack to arbitrary length messages/ciphertexts.
Hint: You can use the function in the part 1 as a subroutine to this.
po_attack(po, ctx)

Part 3 (20 points): Finally, we shall mount a padding oracle attack against a production server (https://paddingoracle.herokuapp.com/). Assume that you have intercepted an encrypted cookie sent from this server containing some sensitive information, and the server happens to use CBC mode encryption. You want to learn the secret information in the cookie.
API provided by the server:

- https://paddingoracle.herokuapp.com/ctx
  - returns: url-safe-base64_encoded string
- https://paddingoracle.herokuapp.com/decrypt/<url-safe-base64-encoded-ctx>
  - returns 0, if decryption succeeds, 1, if there is padding error, 2, if the base64 encoding is wrong.
- https://paddingoracle.herokuapp.com/test/<url-safe-base64-encoded-msg>
  - returns 0, if decryption succeeds, 1, if there is padding error, 2, if the base64 encoding is wrong.

Note there is a PaddingOracleServer class in the paddingoracle.py file. This class provides an interface to communicate with a remote padding oracle server. The API is very similar to the one provided by PaddingOracle, so that you don't have to make any changes in the attack code. However, as this time the request goes over HTTP, the overall run time will be high (nearly half an hour). You might want to put some print statements in your code to check the status of the attack. Finally write the hex encoded plaintext that you recovered as a comment at the end of your poattack.py flile.

Turn into CMS your Python source files (.py) with your modifications. Please do not modify `paddingoracle.py`.