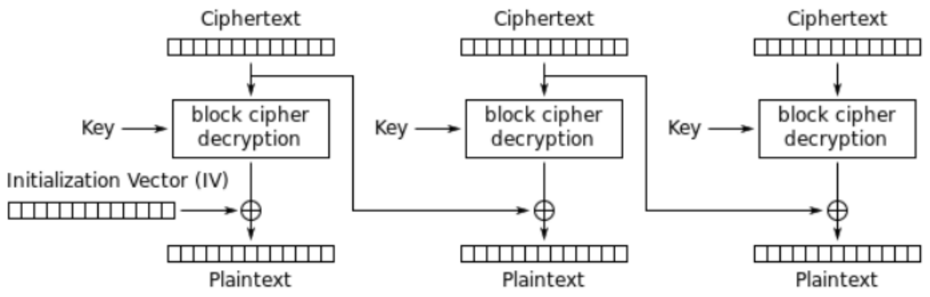
CS528 Lab2 Report By Nidhi Hegde

What is the padding oracle attack?

The padding oracle attack is a type of cryptographic attack that targets systems that use encryption with padding, such as the widely used CBC mode of operation. The attack takes advantage of the fact that many systems do not properly authenticate the padding bytes at the end of an encrypted message.

Cipher-Block Chaining, or CBC, is a block cipher mode of encrypting variable length inputs. When encrypting with CBC mode, each plaintext block is XORed with the previous ciphertext block before being encrypted. This also means that when decrypting, each block of plaintext is generated by being XORed with the previous block of ciphertext, as seen below.



Cipher Block Chaining (CBC) mode decryption

In order for the message to be a multiple of the block length, CBC mode uses a padding scheme. PKCS7 defines a standard for padding the message to the necessary length, in which the final block is filled with B bytes with the value B. For example, if the block size is 16 bytes and your message only fills 12 bytes of the last block, the final 4 bytes will be padded with (04, 04, 04, 04). A Padding Oracle attack is possible when a system indicates whether the padding of a message is valid before checking the validity of the message. If an attacker knows the padding scheme, they can manipulate an intercepted ciphertext until a padding error does not occur, allowing them to determine, byte-by-byte, what the plaintext contains without knowing the key!

The attack works by repeatedly sending modified ciphertexts to the system and observing the system's response. By carefully manipulating the padding bytes at the end of the ciphertext, the attacker can determine whether the padding is valid or not. With enough queries, the attacker can extract the plaintext from the ciphertext without knowing the encryption key.

Here is a step-by-step description:

1. The attacker starts by dividing the cipher text into blocks.
2. The attacker changes the last byte of the penultimate block of the cipher text until the padding is correct. This is accomplished by sending the cipher text to an oracle that tells the attacker whether the padding is correct or not.
3. Once the padding is correct, the attacker changes the last byte of the previous block to decrypt the last byte of the penultimate block.
4. Repeat the previous step, moving the attacker's focus back one byte at a time until the entire penultimate block is decrypted.

5. Then the attacker moves onto the next to last block and repeats the process, moving back one byte at a time until the entire block is decrypted.
6. The attacker repeats this process until the entire message is decrypted.

Description of my implementation with PKCS7 padding

The key idea behind the attack is this: by making modifications to the IV, we can predictably modify the plaintext block. Flipping a bit in the IV will flip the corresponding bit in the plaintext. Setting the IV's final byte to any value will xor that value into the plaintext's final byte. If we iterate through every possible value for the final IV byte, eventually one of them will set the plaintext's final byte to 0x01 – and our padding oracle will tell us when this happens, because 0x01 is valid padding! The function padding_oracle_attack_exploit() takes the ciphertext and IV as input and uses the attack() function to exploit the padding oracle vulnerability. The attack() function takes a single block of ciphertext and tries to construct a valid plaintext by modifying the IV. Each function is explained in detail below.

Main()

Main starts with encrypting the given test plaintext using given encrypt function. The resulting ciphertext is then sent to the padding_oracle_attack_exploit() function along with IV.
Plaintext and IV can be provided as arguments while executing this code. As follows:

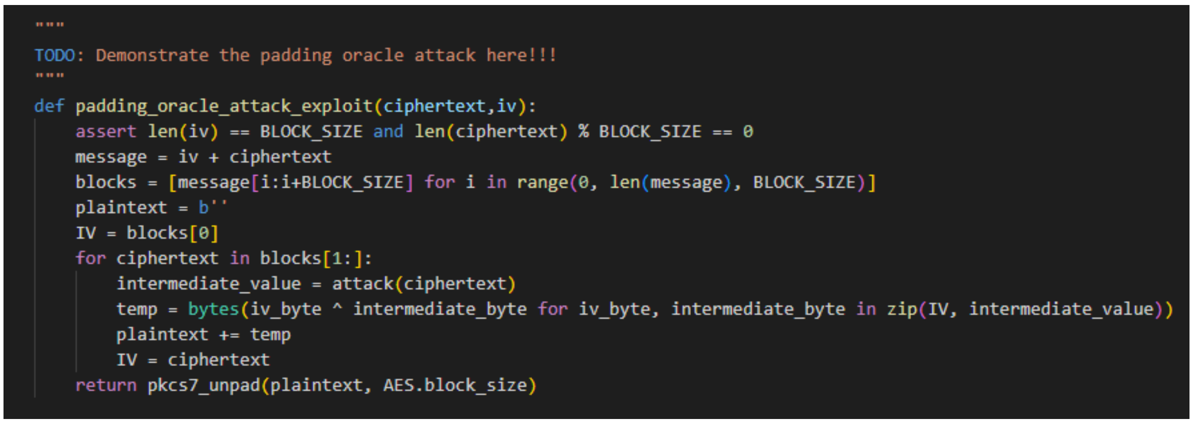
- **Python poa.py -p “Hello” -i “0000000000000001”**

If no arguments are passed, default values for IV and plaintext are:

- iv = '0000000000000000'
- plaintext = 'This is cs528 padding oracle attack lab with hello world~~~!!'

padding_oracle_attack_exploit()

In the padding_oracle_attack_exploit() function, we first extract the IV from the ciphertext and then divide the ciphertext into blocks. Then we iterate over all blocks except the first block. For each block, we use the attack() function to construct the plaintext for that block. We can then XOR the plaintext with the previous block to get the intermediate plaintext. Finally, we remove the padding from the intermediate plaintext and returns the resulting plaintext.



```
"""
TODO: Demonstrate the padding oracle attack here!!!
"""
def padding_oracle_attack_exploit(ciphertext,iv):
    assert len(iv) == BLOCK_SIZE and len(ciphertext) % BLOCK_SIZE == 0
    message = iv + ciphertext
    blocks = [message[i:i+BLOCK_SIZE] for i in range(0, len(message), BLOCK_SIZE)]
    plaintext = b''
    IV = blocks[0]
    for ciphertext in blocks[1:]:
        intermediate_value = attack(ciphertext)
        temp = bytes(iv_byte ^ intermediate_byte for iv_byte, intermediate_byte in zip(IV, intermediate_value))
        plaintext += temp
        IV = ciphertext
    return pkcs7_unpad(plaintext, AES.block_size)
```

attack()

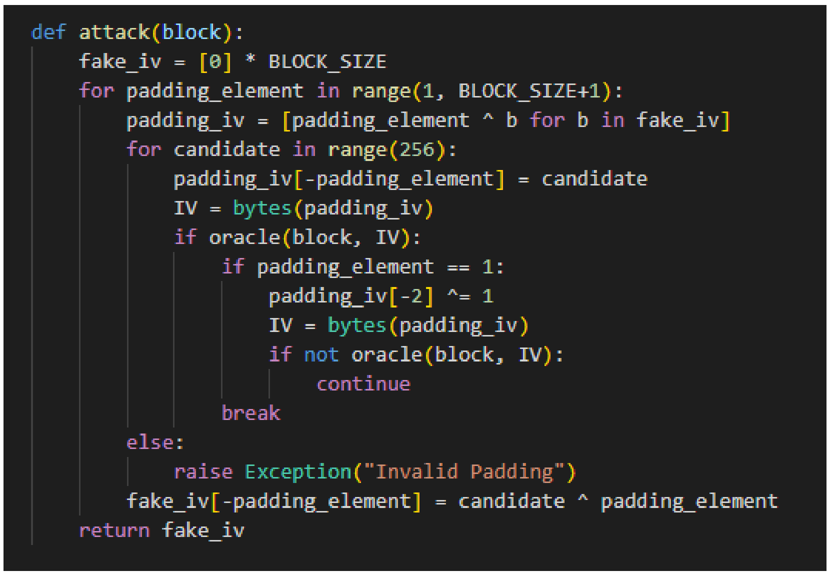
In the attack() function, we first create a fake IV of all zeros. This is an IV which will set some (eventually all) of the plaintext's bytes to zero. Why zero? zero gives us options. If we want to set a plaintext byte to any value other than zero, we can just xor that value into the fake IV. In other words, the fake IV gives us a way of manipulating the plaintext however we like.

How do we build a fake IV? Well, as soon as we set the plaintext's final byte to 0x01, we can take the corresponding IV byte and xor that against 0x01. This modified IV byte will set the plaintext's final byte to 0x00 – and so it will work as the final byte of our fake IV. Once we have that, we can derive a new IV which is guaranteed to set the plaintext's final byte to 0x02, and we can start trying to set the plaintext's penultimate byte to 0x02 as well.

We then iterate over all possible padding values from 1 to the block size. For each padding value, we create a padding IV by XORing the padding value with the fake IV. Later iterate over all possible values of the last byte of the padding IV to find the correct value that produces a valid padding. Once the correct value is found, update the fake IV and continues to the next padding value. If a valid padding is not found for any padding value, an exception is raised.

Second reason why a zeroing IV is useful: One of the basic properties of xor is this:

if IV XOR BLOCK = 0, then IV = BLOCK. In other words, we can recover the output of decryption – it is equal to our zeroing or fake IV!



```python
def attack(block):
    fake_iv = [0] * BLOCK_SIZE
    for padding_element in range(1, BLOCK_SIZE+1):
        padding_iv = [padding_element ^ b for b in fake_iv]
        for candidate in range(256):
            padding_iv[-padding_element] = candidate
            IV = bytes(padding_iv)
            if oracle(block, IV):
                if padding_element == 1:
                    padding_iv[-2] ^= 1
                    IV = bytes(padding_iv)
                    if not oracle(block, IV):
                        continue
                break
        else:
            raise Exception("Invalid Padding")
        fake_iv[-padding_element] = candidate ^ padding_element
    return fake_iv
```

Defenses:

This attack is a chosen-ciphertext attack. It depends on the attacker being able to submit arbitrary ciphertexts to the oracle. As such, we can prevent the attack by authenticating the ciphertexts, by switching from CBC mode to an authenticated encryption mode like GCM or OCB; alternately, along with CBC using something like HMAC.

References:

[1] Cryptopals: Exploiting CBC Padding Oracles
https://research.nccgroup.com/2021/02/17/cryptopalsexploiting-cbc-padding-oracles/