Выполнил Мешечкин Д. Инфобез-2837 (Ранее 2345)

Задание 1: Файл 1.txt зашифрован с помощью AES-128 в режиме ECB на ключе YELLOW SUBMARINE и закодирован в base64.
Примечание: буквы ключа заглавные, длина ровно 16 символов (байт) - замечательный ключ для AES-128.
Дешифруйте файл. В конце концов у вас есть ключ. Проще всего использовать OpenSSL::Cipher в режиме AES-128-ECB, но это не наш путь.
Мы должны реализовать режим ECB сами, это пригодится нам в дальнейшем.
Хорошая новость в том, что для этого не нужно писать AES-128 с нуля. Мы сделаем AES-128 из подручных средств.
На Python функция дешифрования будут выглядеть примерно так:

```
def aes128_decrypt(block, key):
        if len(block) != 16:
        return None

cipher = AES.new(key, AES.MODE_ECB)
        return cipher.decrypt(block)
```

FAQ
В такой схеме вы можете восстановить содержимое неизвестной строки, сделав несколько запросов к функции-оракулу! Алгоритм выглядит примерно так:
Шаг 1. Узнайте размер блока (вы уже его знаете, но все равно выполните этот шаг). Для этого подавайте на вход строки из одинаковых байт, каждый раз добавляя по одному байте: “A”, “AA”, “AAA” и так далее. Подумайте о том, в какой момент вы сможете точно определить длину блока.
Шаг 2. Поймите, что функция использует ECB режим шифрования. Вам это уже известно, но все равно выполните этот шаг.
Шаг 3. Создайте блок данных, длина которого в точности на единицу меньше длины блока (например, если длина блока 8, то блок данных будет “AAAAAAA”). Задайтесь вопросом: что функция шифрования поставит на позицию последнего байта?
Шаг 4. Подавайте на вход функции-оракула все возможные значения последнего байта (“AAAAAAAA”, “AAAAAAAB”, “AAAAAAAC” и так далее). Запомните первый блок каждого получившегося шифротекста.
Шаг 5. Возьмите блок шифротекста из шага 3 и найдите его в списке из шага 4. Теперь вы знаете первый байт неизвестной строки.
Шаг 6. Повторите алгоритм для второго и последующих байт.

```
from base64 import b64decode
from Crypto import Random
from Crypto.Cipher import AES

UNKNOWN_STRING = b"""
Um9sbGluJyBpbiBteSA1LjAKV2l0aCBteSByYWctdG9wIGRvd24gc28gbXkg
aGFpciBjYW4gYmxvdwpUaGUgZ2lybGllcyBvbiBzdGFuZGJ5IHdhdmluZyBq
dXN0IHRvIHNheSBoaQpEaWQgeW91IHN0b3A/IE5vLCBJIGp1c3QgZHJvdmUg
YnkK"""

# b"Rollin' in my 5.0\nWith my rag-top down so my hair can blow\nThe girlies on standby waving just to say hi\nDid you stop? No, I just drove by\n"

KEY = Random.new().read(16)

def pad(your_string, msg):
        """prepends the `msg` with `your_string` and then, applies PKCS#7 padding

        Args:
                your_string (bytes): the byte-string to prepend to `msg`
                msg (bytes): a byte-string that is to be padded
```

```
Returns:
        bytes: the padded byte-string
"""
paddedMsg = your_string + msg

size = 16
length = len(paddedMsg)
if length % size == 0:
        return paddedMsg

# PKCS#7 padding if the plain-text after padding isn't a multiple of AES.BLOCK_SIZE
padding = size - (length % size)
padValue = bytes([padding])
paddedMsg += padValue * padding

return paddedMsg

def encryption_oracle(your_string):
"""encrypts `your_string` + msg` + `UNKNOWN_STRING` using AES-ECB-128

Args:
        your_string (bytes): byte-string used to prepend

Returns:
        [bytes]: the byte-string of encrypted text
"""
msg = bytes('The unknown string given to you was:\n', 'ascii')
# append the `UNKNOWN_STRING` given to us to the `msg`
plaintext = msg + b64decode(UNKNOWN_STRING)
# add `your_string` to prepend to `plaintext` and apply `PKCS#7` padding to correct size
paddedPlaintext= pad(your_string, plaintext)

cipher = AES.new(KEY, AES.MODE_ECB)
ciphertext = cipher.encrypt(paddedPlaintext)

return ciphertext

def detect_block_size():
        """detects the `block_size` used by the encryption_oracle()
        Returns:
        int: the `block_size` used by the encryption_oracle
        """
        feed = b"A"
        length = 0
        while True:
        cipher = encryption_oracle(feed)
        # on every iteration, add one more character
        feed += feed
        # if the length of the ciphertext increases by more than 1,
        # PKCS#7 padding must have been added to make the size of plaintext == block_size
        # increase in the size gives the value of block_size
        if not length == 0 and len(cipher) - length > 1:
        return len(cipher) - length
        length = len(cipher)

def detect_mode(cipher):
        """detects whether the cipher-text was encrypted in ECB or not
        Args:
        cipher (bytes): byte-string of cipher-text
```

```
Returns:
        str: "ECB" | "not ECB"
        """
        chunkSize = 16
        chunks = []
        for i in range(0, len(cipher), chunkSize):
        chunks.append(cipher[i:i+chunkSize])

        uniqueChunks = set(chunks)
        if len(chunks) > len(uniqueChunks):
        return "ECB"
        return "not ECB"

        def ecb_decrypt(block_size):
        """decrypts the plaintext (without key) using byte-at-a-time attack (simple)

                Args:
        block_size (int): the `block_size` used by the `encryption_oracle()` for encryption
        """
        # common = lower_cases + upper_cases + space + numbers
        # to optimize brute-force approach
        common = list(range(ord('a'), ord('z'))) + list(range(ord('A'), ord('Z'))) + [ord(' ')] +
list(range(ord('0'), ord('9')))
        rare = [i for i in range(256) if i not in common]
        possibilities = bytes(common + rare)

        plaintext = b'' # holds the entire plaintext = sum of `found_block`'s
        check_length = block_size

        while True:
        # as more characters in the block are found, the number of A's to prepend decreases
        prepend = b'A' * (block_size - 1 - (len(plaintext) % block_size))
        actual = encryption_oracle(prepend)[:check_length]

        found = False
        for byte in possibilities:
        value = bytes([byte])
        your_string = prepend + plaintext + value
        produced = encryption_oracle(your_string)[:check_length]
        if actual == produced:
        plaintext += value
        found = True
        break

        if not found:
        print(f'Possible end of plaintext: No matches found.')
        print(f'Plaintext: \n{ plaintext.decode('ascii') }')
        return

        if len(plaintext) % block_size == 0:
        check_length += block_size
```

```
def main():
        # detect block size
        block_size = detect_block_size()
        print(f"Block Size is { block_size }")

        # detect the mode (should be ECB)
        repeated_plaintext = b"A" * 50
        cipher = encryption_oracle(repeated_plaintext)
        mode = detect_mode(cipher)
        print(f"Mode of encryption is { mode }")

        # decry,pt the plaintext inside `encryption_oracle()`
        ecb_decrypt(block_size)

if __name__ == "__main__":
        main()
```