Задание выполнил студент группы 2345 Романенко Кирилл

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.

Содержание файла 1.txt

```
CRIwqt4+szDbqkNY+I0qbDe3LQz0wiw0SuxBQtAM5TDdMbjCMD/venUDW9BL
PEXODbk6a48oMbAY6DDZsuLbc0uR9cp9hQ0QQGATyyCESq2NSsvhx5zKlLtz
dsnfK5ED5srKjK7Fz4Q38/ttd+stL/9WnDzlJvAo7WBsjI5YJc2gmAYayNfm
CW2lhZE/ZLG0CBD2aPw0W417QYb4cAIOW92jYRiJ4PTsBBHDe8o4JwqaUac6
rqdi833kbyAOV/Y2RMbN0oDb9Rq8uRHvbrqQJaJieaswEtMkgUt3P5Ttgeh7
J+hE6TR0uHot8WzHyAKNbUWHoi/5zcRCUipvVOYLoBZXlNu4qnwoCZRSBgvC
wTdz3Cbsp/P2wXB8tiz6l9rL2bLhBt13Qxyhhu0H0+JKj6soSeX5ZD1Rpilp
9ncR1tHW8+uurQKyXN4xKeGjaKLOejr2xDIw+aWF7GszU4qJhXBnXTIUUNUf
RlwEpS6FZcsMzemQF30ezSJHfpW7DVHzwiLyeiTJRKoVUwo43PXupnJXDmUy
sCa2nQz/iEwyor6kPekLv1csm1Pa2LZmbA9Ujzz8zb/gFXtQqBAN4zA8/wt0
VfoOsEZwcsaLOWUPtF/Ry3VhlKwXE7gGH/bbShAIKQqMqqUkEucZ3HPHAVp7
ZCn3Ox6+c5QJ3Uv8V7L7SprofPFN6F+kfDM4zAc59do5twgDoClCbxxG0L19
TBGHiYP3CygeY1HLMrX6KqypJfFJW5O9wNIF0qfOC2lWFgwayOwq41xdFSCW
0/EBSc7cJw3N06WThrW5LimAOt5L9c7Ik4YIxu0K9JZwAxfcU4ShYu6euYmW
LP98+qvRnIrXkePugS9TSOJOHzKUoOcb1/KYd9NZFHEcp58Df6rXFiz9DSq8
0rR5Kfs+M+Vuq5Z6zY98/SP0A6URIr9NFu+Cs9/gf+q4TRwsOzRMjMQzJL8f
7TXPEHH2+qEcpDKz/5pE0cvrgHr63XKu4XbzLCOBz0DoFAw3vkuxGwJq4Cpx
kt+eCtxSKUzNtXMn/mbPqPl4NZNJ8yzMqTFSODS4bYTBaN/uQYcOAF3NBYFd
5x9TzIAoW6ai13a8h/s9i5FlVRJDe2cetQhArrIVBquF0L0mUXMWNPFKkaQE
BsxpMCYh7pp7YlyCNode12k5jY1/lc8jQLQJ+EJHdCdM5t3emRzkPgND4a7O
NhoIkUUS2R1oEV1toDj9iDzGVFwOvWyt4GzA9XdxT333JU/n8m+N6hs23MBc
Z086kp9rJGVxZ5f80jRz3ZcjU6zWjR9ucRyjbsuVn1t4EJEm6A7KaHm13m0v
wN/O4KYTiiY3aO3siayjNrrNBpn1OeLv9UUneLSCdxcUqjRvOrdA5NYv25Hb
4wkFCIhC/Y2ze/kNyis6FrXtStcjKC1w9Kg8O25VXB1Fmpu+4nzpbNdJ9LXa
hF7wjOPXN6dixVKpzwTYjEFDSMaMhaTOTCaqJig97624wv79URbCgsyzwaC7
YXRtbTstbFuEFBee3uW7B3xXw72mymM2BS2uPQ5NIwmacbhta8aCRQEGqIZ0
78YrrOlZIjar3lbTCo5o6nbbDq9bvilirWG/SgWINuc3pWl5CscRcgQQNp7o
LBgrSkQkv9AjZYcvisnr89TxjoxBO0Y93jgp4T14LnVwWQVx3l3d6S1wlsci
dVeaM24E/JtS8k9XAvgSoKCjyiqsawBMzScXCIRCk6nqX8ZaJU3rZ0LeOMTU
w6MC4dC+aY9SrCvNQub19mBdtJUwOBOqGdfd5IoqQkaL6DfOkmpnsCs5PuLb
GZBVhah5L87IY7r6TB1V7KboXH8PZIYc1zlemMZGU0o7+etxZWHgpdeX6JbJ
Is3ilAzYqw/Hz65no7eUxcDg1aOaxemuPqnYRGhW6PvjZbwAtfQPlofhB0jT
Ht5bRlzF17rn9q/6wzlc1ssp2xmeFzXoxffpELABV6+yj3gfQ/bxIB9NWjdZ
K08RX9rjm9CcBlRQeTZrD67SYQWqRpT5t7zcVDnx1s7ZffLBWm/vXLfPzMaQ
YEJ4EfoduSutjshXvR+VQRPs2TWcF7OsaE4csedKUGFuo9DYfFIHFDNg+1Py
rlWJ0J/X0PduAuCZ+uQSsM/ex/vfXp6Z39ngq4exUXoPtAIqafrDMd8SuAty
EZhyY9V9Lp2qNQDbl6JI39bDz+6pDmjJ2jlnpMCezRK89cG11IqiUWvIPxHj
oiT1guH1uk4sQ2Pc1J4zjJNsZgoJDcPBbfss4kAqUJvQyFbzWshhtVeAv3dm
gwUENIhNK/erjpgw2BIRayzYw001jAIF5c7rYg38o6x3YdAtU3d3QpuwG5xD
fODxzfL3yEKQr48C/KqxI87uGwyg6H5gc2AcLU9JYt5QoDFoC7PFxcE3RVqc
```

```
7/Um9Js9X9UyriEjftWt86/tEyG7F9tWGxGNEZo3MOydwX/7jtwoxQE5ybFj
WndqLp8DV3naLQsh/Fz8JnTYHvOR72vuiw/x5D5PFuXV0aSVvmw5Wnb09q/B
owS14WzoHH6ekaWbh78xlypn/L/M+nIIEX1Ol3TaVOqIxvXZ2sjm86xRz0Ed
oHFfupSekdBULCqptxpFpBshZFvauUH8Ez7wA7wjL65GVlZ0f74U7MJVu9Sw
sZdgsLmnsQvr5n2ojNNBEv+qKG2wpUYTmWRaRc5EClUNfhzh8iDdHIsl6edO
ewORRrNiBay1NCzlfz1cj6VlYYQUM9bDEyqrwO400XQNpoFOxo4fxUdd+AHm
CBhHbyCR81/C6LQTG2JQBvjykG4pmoqnYPxDyeiCEG+JFHmP1IL+jggdjWhL
WQatslrWxuESEl3PEsrAkMF7gt0dBLgnWsc1cmzntG1rlXVi/Hs2TAU3RxEm
MSWDFubSivLWSqZj/XfGWwVpP6fsnsfxpY3d3h/fTxDu7U8GddaFRQhJ+0ZO
dx6nRJUW3u6xnhH3mYVRk88EMtpEpKrSIWfXphgDUPZ0f4agRzehkn9vtzCm
NjFnQb0/shnqTh4Mo/8oommbsBTUKPYS7/1oQCi12QABjJDt+LyUan+4iwvC
i0k0IUIHvk21381vC0ixYDZxzY64+xx/RNID+iplgzq9PDZgjc8L7jMg+2+m
rxPS56e71m5E2zufZ4d+nFjIg+dHD/ShNPzVpXizRVUERztLuak8Asah3/yv
wOrH1mKEMMGC1/6qfvZUgFLJH5V0Ep0n2K/Fbs0VljENIN8cjkCKdG8aBnef
EhITdV7CVjXcivQ6efkbOQCfkfcwWpaBFC8tD/zebXFE+JshW16D4EWXMnSm
/9HcGwHvtlAj04rwrZ5tRvAgf1IR83kqqiTvqfENcj7ddCFwtNZrQK7EJhgB
5Tr1tBFcb9InPRtS3KYteYHl3HWR9t8E2YGE8IGrS1sQibxaK/C0kKbqIrKp
npwtoOLsZPNbPw6K2jpko9NeZAx7PYFmamR4D50KtzgELQcaEsi5aCztMg7f
p1mK6ijyMKIRKwNKIYHagRRVLNgQLg/WTKzGVbWwq6kQaQyArwQCUXo4uRty
zGMaKbTG4dns1OFB1g7NCiPb6s1lv0/lHFAF6HwoYV/FPSL/pirxyDSBb/FR
RA3PIfmvGfMUGFVWlyS7+O73l5oIJHxuaJrR4EenzAu4Avpa5d+VuiYbM10a
LaVegVPvFn4pCP4U/Nbbw4OTCFX2HKmWEiVBB0O3J9xwXWpxN1Vr5CDi75Fq
NhxYCjgSJzWOUD34Y1dAfcj57VINmQVEWyc8Tch8vg9MnHGCOfOjRqp0VGyA
S15AVD2QS1V6fhRimJSVyT6QuGb8tKRsl2N+a2Xze36vgMhw7XK7zh//jC2H
```

Дешифруйте файл. В конце концов у вас есть ключ.

> Внимание! Не декодируйте это значение. Суть задания в том, что вы не знаете что внутри base64.
> В итоге ваша функция будет возвращать значение:
> AES-128-ECB(ваша-строка || неизвестная-строка, случайный-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-ораклу! 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 }")

    # decrypt the plaintext inside `encryption_oracle()`
    ecb_decrypt(block_size)


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