MAGIC KEYS

International Cyber Security Challenge

EUROPEAN CYBER SECURITY CHALLENGE

SEPT 2021

1. DESCRIPTION

There is a service listening on target host that can decrypt your messages. You can connect with *socat* or similar tool to it and use the text-based menu for operating it. The service is a custom piece of code, written in Python. Unfortunately implementing your own cryptography often ends up with buggy code that is easy to exploit, and result is complete loss of secrecy.

Steal the keys! Source code of the service is available for your inspection over HTTP on the target machine.

2. CHALLENGE SPECIFICATIONS

- Categroty: Crypto
- Difficulty: Medium
- Estimated time: 30 min

3. QUESTIONS AND ANSWERS

3.1 SECRET KEYS

1. randomstring1234
2. 5678endofstring

4. SETUP INSTRUCTIONS

Dockerfile and *docker-compose.yml* are provided to run the task in a container. KEY1, KEY2, PORT, and HTTPPORT are passed from docker-compose environment, see **.env**.

```
docker-compose build
```

```
docker-compose up
```

HTTP port is used to serve source of the script to contesters. PORT is where the service itself is listening. **KEY1 and KEY2 must be exactly 16 bytes (128 bits).**

5. ARTIFACTS PROVIDED

File	SHA-256
magic-keys.tar.gz	0f0f0d3d7d82701c3a4464ce5a2eb6bb68ee877cd52cede3d4fe6ce73d4ca878

6. TOOLS NEEDED

- socat, nc, or similar
- scripting language

7. WALKTHROUGH

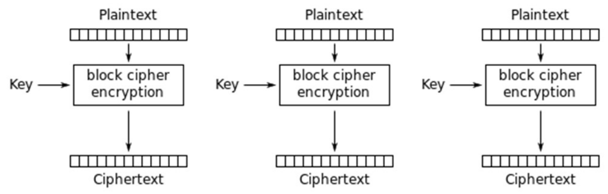
Start by inspecting the provided Pyhton script, and identify encryption properties:

```
def decrypt_message(key, IV):
        ...
    cipher = AES.new(key, AES.MODE_CBC, IV)
```

Down in the main function, call to the decrypt_message should catch eye: **key is used as iv.**

```
if choice == "1":
    decrypt_message(key, key)
```

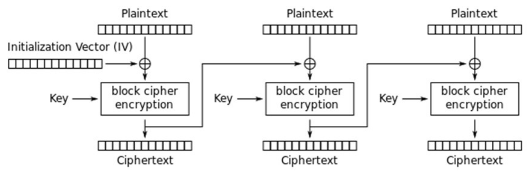
To understand, whether exploitation is possible, and how to do it, a little background theory is necessary. Block ciphers are used to encrypt or decrypt one fixed-length block of data using the supplied secret key. For AES, the block size is 128 bits (16 bytes). The method for handling messages longer than one block, is called **mode of operation**. The simplest mode of operation is ECB, where the input is split into blocks, every block is encrypted independently, and the results are concatenated together:



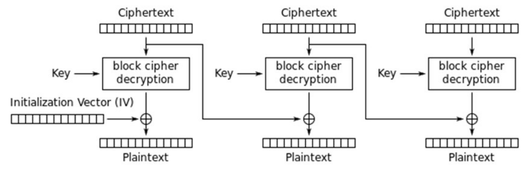
Electronic Codebook (ECB) mode encryption

This approach has a problem, that since the ciphertext block depends only on the plaintext block and key, repeated plaintext blocks result in repeated ciphertexts, and this could reveal a great deal of information. To overcome this, several other modes of operation have been invented. One of the oldest and most widely used is **CBC**. Explanation of it from Wikipedia:

In CBC mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted. This way, each ciphertext block depends on all plaintext blocks processed up to that point. To make each message unique, an initialization vector must be used in the first block.



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

Although relatively secure, this mode of operation has certain **weakness against chosen-ciphertext attack**. The CBC mode decryption expressed in pseudocode would be:

```
PlainBlock[1] = AES_Decrypt_Block (CipherBlock[1], Key) XOR IV
PlainBlock[2] = AES_Decrypt_Block (CipherBlock[2], Key) XOR CipherBlock[1]
…
PlainBlock[n] = AES_Decrypt_Block (CipherBlock[n], Key) XOR CipherBlock[n-1]
```

Knowing, that output of AES_Decrypt_Block depends only on input and key, we could modify the code a bit for cases, when two ciphertext blocks are identical:

```
If CipherBlock[1] == CipherBlock[2] then
        Let Block = Aes_Decrypt_Block(CipherBlock[1], Key)
        PlainBlock[1] = Block XOR IV
        PlainBlock[2] = Block XOR CipherBlock[1]
```

The XOR operation has a property, that a value XOR-ed to zero equals the value itself. This can be leveraged to better choose the ciphertext – namely two blocks of null bytes. That would mean:

```
PlainBlock[2]   = Block XOR 0000000000000000
                = Block
```

Another propery of XOR operation is that a value XOR-ed to itself is zero. This can lead to an interesting result, when PlainBlock[1] and PlainBlock[2] are XOR-ed together:

```
PlainBlock[2] XOR PlainBlock[1]  = Block XOR Block XOR IV =
                                 = 0000000000000000 XOR IV =
                                 = IV
```

Thus, by supplying two blocks of null bytes to CBC-mode block cipher decryption, we can retrieve the value of IV. This would not be a problem, if the IV is chosen randomly and never reused.

However, this is not the case. Even worse, the IV is set equal to the key, that should be kept secret.

Let's try it in action against the server on port 1343:

```
$ nc localhost 1343
You have 2 keys to choose from
=> 1
1) Decrypt content
2) Choose a different key
3) Exit
=> 1
Hex data for decryption
=> 0000000000000000000000000000000000000000000000000000000000000000
98ef83848f49b919f1ceadbe3ae53f54ea8eede0e024ca6d83a7c3d90bd70c60
```

The XOR-ing of plaintext halves can be done using a Python two-liner:

```
$ python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
c=bytes.fromhex('98ef83848f49b919f1ceadbe3ae53f54ea8eede0e024ca6d83a7c3d90bd70c60')
>>> "".join(chr(c[i]^c[i+16]) for i in range(16))
'randomstring1234'
```

We can verify it also in Python:

```
>>> from Crypto.Cipher import AES
>>> AES.new('randomstring1234',AES.MODE_CBC,'randomstring1234').encrypt(c)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

Done! Ciphertext is all zeros, as expected.

Second key can be recovered by repeating the steps.

ENISA
European Union Agency for Cybersecurity

Athens Office
1 Vasilissis Sofias Str.
151 24 Marousi, Attiki, Greece

Heraklion Office
95 Nikolaou Plastira
700 13 Vassilika Vouton, Heraklion, Greece

enisa.europa.eu

ISBN xxx-xx-xxxx-xxx-x
doi:xx.xxxx/xxxxxx
TP-xx-xx-xxx-EN-C