# CRYPTOHACK: SYMMETRIC CRYPTOGRAPHY SOLUTIONS

CRYPTOGRAPHY

daniel.juarez@iteso.mx, ID: 752809

2025-03-10

# Contents

# 1 CRYPTOHACK: SYMMETRIC CRYPTOGRAPHY SOLUTIONS

## 1.1 BRINGING IT ALL TOGETHER

**TASK:** We've provided the key expansion code, and ciphertext that's been properly encrypted by AES-128. Copy in all the building blocks you've coded so far, and complete the decrypt function that implements the steps shown in the diagram. The decrypted plaintext is the flag.

Where the piece of code where we will work is:

```
def decrypt(key, ciphertext):
    round_keys = expand_key(key) # Remember to start from the last round key and work
    ↪  backwards through them when decrypting

    # Convert ciphertext to state matrix

    # Initial add round key step

    for i in range(N_ROUNDS - 1, 0, -1):
        pass # Do round

    # Run final round (skips the InvMixColumns step)

    # Convert state matrix to plaintext

    return plaintext
```

To solve this challenge, we must create a code that combines all the solutions from the previous challenges. Notice that the AES decryption process is as follows:

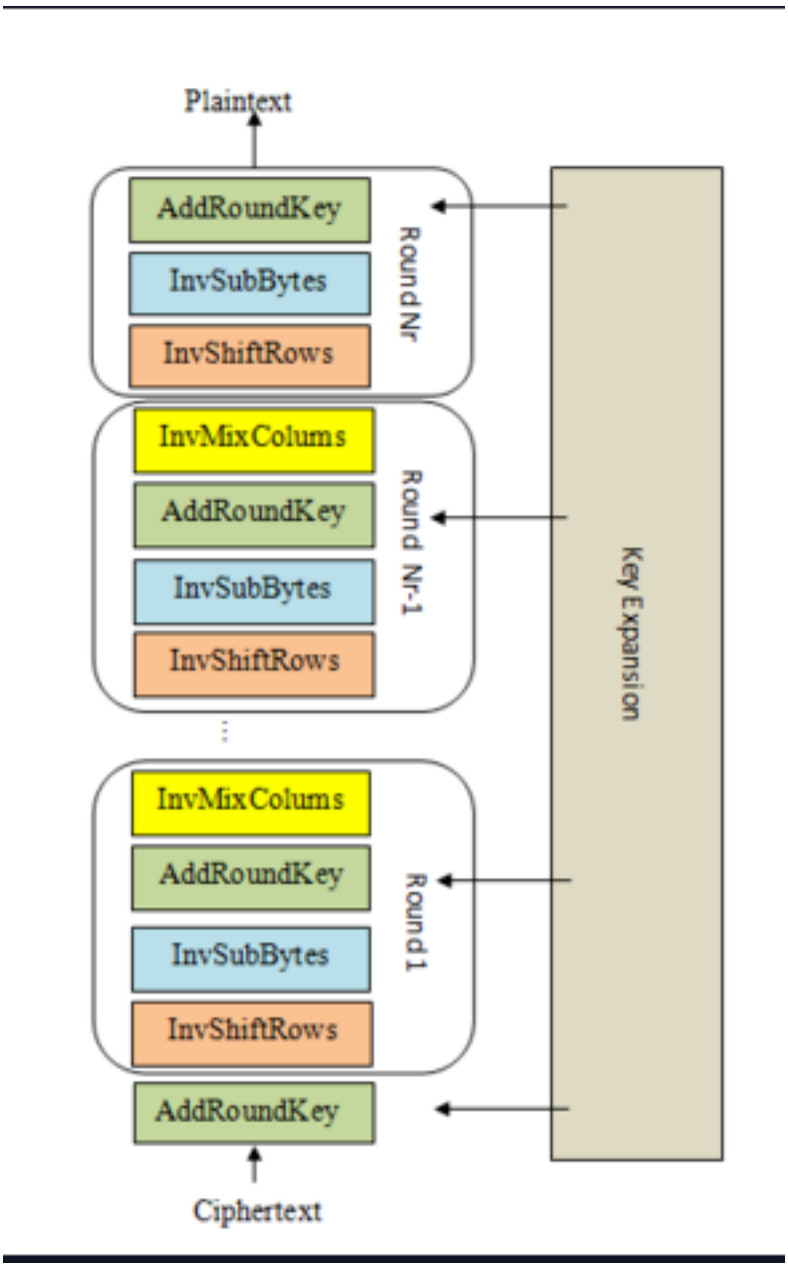


**Figure 1.1:** AES decryption process

At the beginning of the code, notice how the key and the ciphertext are in a *binary string* format, thus we need to consider ways to manipulate this as recently we have only been working with matrices:

```
key        = b'\xc3,\\\xa6\xb5\x80^\x0c\xdb\x8d\xa5z*\xb6\xfe\\'
ciphertext = b'\xd1O\x14j\xa4+O\xb6\xa1\xc4\x08B)\x8f\x12\xdd'
```

Where the solution turned out to be:

```
def decrypt(key, ciphertext):
    round_keys = expand_key(key) # Remember to start from the last round key and work
↪   backwards through them when decrypting

    # Convert ciphertext to state matrix
    state = bytes2matrix(ciphertext)

    # Initial add round key step
    addInitialKey(state, round_keys[-1])

    for i in range(N_ROUNDS - 1, 0, -1):
        inv_shift_rows(state)
        state = sub_bytes(state, sbox=inv_s_box)
        addInitialKey(state, round_keys[i])
        state = inv_mix_columns(state)


    # Run final round (skips the InvMixColumns step)
    inv_shift_rows(state)
    state = sub_bytes(state, sbox=inv_s_box)
    addInitialKey(state, round_keys[0])

    # Convert state matrix to plaintext
    matrix2bytes(state)

    return state

plaintext = decrypt(key, ciphertext)

for i in range(4):
    for j in range(4):
        print(plaintext[i][j], end='')
```

Where the flag for this challenge is:

```
[0xncat@archlinux cryptography]$ python3 aes_decrypt_f491744105801ec03d6a6f7a0e7f8101.py
crypto{MYAES128}
```

## 1.2 MODES OF OPERATION STARTER

The next challenges including this one do not have **tasks**, but rather a python code from two APIs. The task will be to analyze the scripts and try to find ways to exploit them. In this challenge, we are provided with the following code:

```
from Crypto.Cipher import AES
```

```
KEY = ?
FLAG = ?


@chal.route('/block_cipher_starter/decrypt/<ciphertext>/')
def decrypt(ciphertext):
    ciphertext = bytes.fromhex(ciphertext)

    cipher = AES.new(KEY, AES.MODE_ECB)
    try:
        decrypted = cipher.decrypt(ciphertext)
    except ValueError as e:
        return {"error": str(e)}

    return {"plaintext": decrypted.hex()}


@chal.route('/block_cipher_starter/encrypt_flag/')
def encrypt_flag():
    cipher = AES.new(KEY, AES.MODE_ECB)
    encrypted = cipher.encrypt(FLAG.encode())

    return {"ciphertext": encrypted.hex()}
```

When calling the **encrypt_flag** API, it will encrypt the flag with AES in **ECB** mode, it then returns the ciphertext in **hexadecimal**. When calling the **decrypt** API, it will try to decrypt the received ciphertext and then return it in **hexadecimal** format. We can script a simple solution for this challenge that sends a GET request to **encrypt_flag**, parses the flag to **decrypt** and then convert the returned plaintext into **ASCII**:

```
#!/usr/bin/env python3
import requests
import json


def extract_text(text, type):
    text = json.loads(text)
    text = text[f"{type}"]
    return text

def get_text(text, action):
    text = requests.get(f"http://aes.cryptohack.org/block_cipher_starter/{action}/{text if
↪  action == "decrypt" else ""}").text
    text = extract_text(text, "ciphertext" if action == "encrypt_flag" else "plaintext")
    return text


ciphertext = get_text(None, "encrypt_flag")
print(bytearray.fromhex(get_text(ciphertext, "decrypt")).decode())
```

For some challenges, routes might be the same, the only thing that changes is if we are calling the

**decrypt** or **encrypt** APIs and if we want to extract the ciphertext or the plaintext from the returned json. The solution for this challenge is:

```
[0xncat@archlinux cryptography]$ python3 modes_operation_starter.py
crypto{bl0ck_c1ph3r5_4r3_f457_!}
```

## 1.3 PASSWORDS AS KEYS

The APIs code is:

```
from Crypto.Cipher import AES
import hashlib
import random


# /usr/share/dict/words from
#
↪  https://gist.githubusercontent.com/wchargin/8927565/raw/d9783627c731268fb2935a731a618aa8e95cf465/words
with open("/usr/share/dict/words") as f:
    words = [w.strip() for w in f.readlines()]
keyword = random.choice(words)

KEY = hashlib.md5(keyword.encode()).digest()
FLAG = ?


@chal.route('/passwords_as_keys/decrypt/<ciphertext>/<password_hash>/')
def decrypt(ciphertext, password_hash):
    ciphertext = bytes.fromhex(ciphertext)
    key = bytes.fromhex(password_hash)

    cipher = AES.new(key, AES.MODE_ECB)
    try:
        decrypted = cipher.decrypt(ciphertext)
    except ValueError as e:
        return {"error": str(e)}

    return {"plaintext": decrypted.hex()}


@chal.route('/passwords_as_keys/encrypt_flag/')
def encrypt_flag():
    cipher = AES.new(KEY, AES.MODE_ECB)
    encrypted = cipher.encrypt(FLAG.encode())

    return {"ciphertext": encrypted.hex()}
```

The above code encrypts the flag using AES in ECB using as a key a random word from a given wordlist.

The hash is then used as a password to protect the ciphertext and avoid decrypting it. For this we can attack the ciphertext by hashing all possible options from the wordlist as follows:

```
#!/usr/bin/env python3

from Crypto.Cipher import AES
import requests
import hashlib
import json


def get_flag():
    encrypted_flag = requests.get("http://aes.cryptohack.org/passwords_as_keys/encrypt_flag/")
    encrypted_flag = json.loads(encrypted_flag.text)
    encrypted_flag = encrypted_flag["ciphertext"]
    return encrypted_flag

def decrypt(ciphertext, password_hash):
    ciphertext = bytes.fromhex(ciphertext)
    key = bytes.fromhex(password_hash)

    cipher = AES.new(key, AES.MODE_ECB)
    try:
        decrypted = cipher.decrypt(ciphertext)
    except ValueError as e:
        return {"error": str(e)}

    try:
        return {"plaintext": bytearray.fromhex(decrypted.hex()).decode()}
    except Exception as e:
        return "NONE"

def bruteforce_hash(encrypted_flag):
    with open('/opt/wordlists/words') as f:
        words = [w.strip() for w in f.readlines()]
    print(f"bruteforcing...")

    for word in words:
        key = hashlib.md5(word.encode()).hexdigest()
        flag = decrypt(encrypted_flag, key)

        if flag != "NONE": break

    return flag


encrypted_flag = get_flag()
flag = bruteforce_hash(encrypted_flag)

print(flag)
```

Where the solution for this challenge is:

```
bruteforcing...
{'plaintext': 'crypto{k3y5__r__n07__p455w0rdz?}'}
```

## 1.4 ECB ORACLE

The APIs code is:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad


KEY = ?
FLAG = ?


@chal.route('/ecb_oracle/encrypt/<plaintext>/')
def encrypt(plaintext):
    plaintext = bytes.fromhex(plaintext)

    padded = pad(plaintext + FLAG.encode(), 16)
    cipher = AES.new(KEY, AES.MODE_ECB)
    try:
        encrypted = cipher.encrypt(padded)
    except ValueError as e:
        return {"error": str(e)}

    return {"ciphertext": encrypted.hex()}
```

The above encrypts the ciphertext using AES in ECB mode. Notice how the ciphertext is composed by *plaintext* + *FLAG*. That means, for a 15 bytes plaintext such as:

*plaintext* = *aaaaaaaaaaaaaaa*

And the flag:

*flag* = *crypto{FLAG}*

That will allow the first 16 byte ciphertext block to be:

*ciphertext(16)* = aaaaaaaaaaaaaaac*

Since we are encrypting the ciphertext in ECB mode, if we encrypt aaaaaaaaaaaaaaac and compare the first 16 byte block of the ciphertext with the first 16 byte block from the text we sent in order to check if what we send corresponds to a valid part of the ciphertext. From previous flags it is known that flags allways start with *crypto{*, so we can send the following plaintext as a starting for the bruteforcing. Therefore, the following code solves the challenge:

```python
#!/usr/bin/env python3
import requests
import json
import string
from os import system


def extract_text(text):
    text = json.loads(text)
    text = text["ciphertext"]
    return text

def get_text(text):
    text = requests.get(f"http://aes.cryptohack.org/ecb_oracle/encrypt/{text.encode("utf-
↪  8").hex()}/").text
    text = extract_text(text)
    return text

def generate_plaintext(characters_quantity, letter):
    return letter * characters_quantity

def bruteforce(plaintext, wordlist, flag):
    ciphertext = get_text(plaintext)
    flag_block = ciphertext[:64]

    for word in wordlist:
        payload = plaintext + flag + word
        ciphered_payload = get_text(payload)
        ciphered_payload_block = ciphered_payload[:64]

        if flag_block == ciphered_payload_block:
            return word


flag = "crypto{"
wordlist = string.printable

while True:
    plaintext = generate_plaintext(31 - len(flag), 'a')
    flag += bruteforce(plaintext, wordlist, flag)
    system("clear")
    print(flag)

    if '}' in flag: break

print("[!] DONE")
```

The solution creates a wordlist to bruteforce the ciphertext containing all printable characters, then as said above, it will compare the first 16 bytes from both ciphertexts to check if they are similar, indicating a valid character from the flag has been found:

```
crypto{p3n6u1n5_h473_3cb}
[!] DONE
```

## 1.5 ECB CBC WTF

The APIs code is:

```
from Crypto.Cipher import AES


KEY = ?
FLAG = ?


@chal.route('/ecbcbcwtf/decrypt/<ciphertext>/')
def decrypt(ciphertext):
    ciphertext = bytes.fromhex(ciphertext)

    cipher = AES.new(KEY, AES.MODE_ECB)
    try:
        decrypted = cipher.decrypt(ciphertext)
    except ValueError as e:
        return {"error": str(e)}

    return {"plaintext": decrypted.hex()}


@chal.route('/ecbcbcwtf/encrypt_flag/')
def encrypt_flag():
    iv = os.urandom(16)

    cipher = AES.new(KEY, AES.MODE_CBC, iv)
    encrypted = cipher.encrypt(FLAG.encode())
    ciphertext = iv.hex() + encrypted.hex()

    return {"ciphertext": ciphertext}
```

This challenge can be solved by understanding how **AES** in CBC mode works. To encrypt the flag, an initialization vector is used to introduce randomness in the ciphertext. The ciphertext in this case will be decrypted using AES in ECB mode. As seen in the API, the initialization vector is appended at the beginning of the ciphertex. The first 32 bytes of the text will be composed by the initialization vector since it is in hexadecimal notation (1 byte correspond to two characters). We can try to analize the AES CBC decryption process to get the answer:

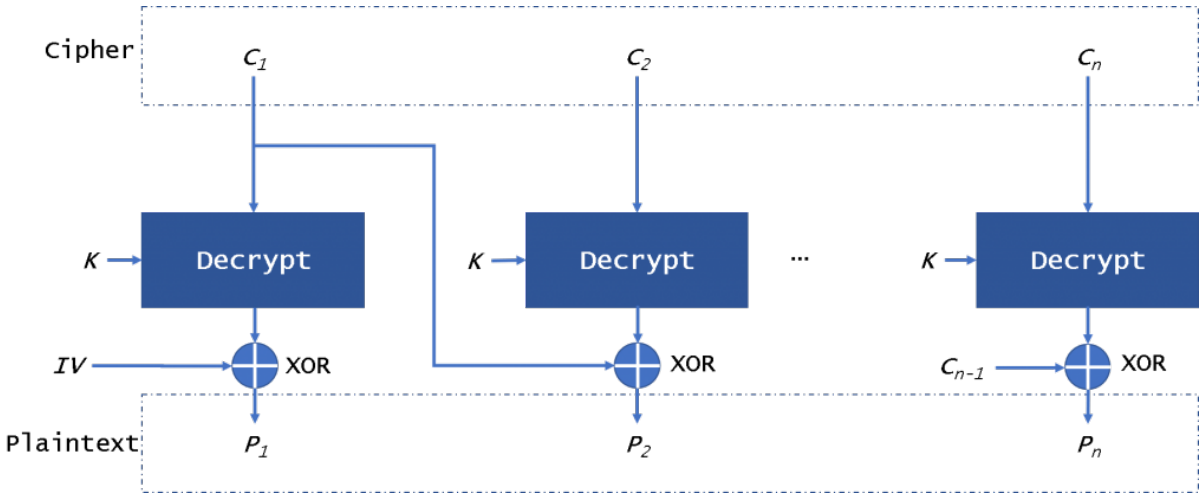


**Figure 1.2:** AES CBC decryption process

As seen in the image, The ciphertext will be decrypted first using the key, and then the initialization vector will be applied, following that logic, we can use the following script to get the answer:

```
#!/usr/bin/env python3
import requests
import json


def extract_text(text, type):
    text = json.loads(text)
    text = text[f"{type}"]
    return text

def get_text(text, action):
    text = requests.get(f"http://aes.cryptohack.org/ecbcbcwtf/{action}/{text if action ==
↪   "decrypt" else ""}").text
    text = extract_text(text, "ciphertext" if action == "encrypt_flag" else "plaintext")
    return text

def cbc_decryption(ciphertext):
    flag = ''

    for i in range(32, 96, 32):
        plaintext = get_text(ciphertext[i:i + 32], "decrypt")
        initial_xor = int(plaintext, 16) ^ int(ciphertext[i - 32:i], 16)
        flag += hex(initial_xor)[2:]

    flag = bytearray.fromhex(flag[:32]).decode() + bytearray.fromhex(flag[32:64]).decode()
    return flag


ciphertext = get_text(None, "encrypt_flag")
print(cbc_decryption(ciphertext))
```

Where the flag for this challenge is:

```
[0xncat@archlinux cryptography]$ python3 cbc.py
crypto{3cb_5uck5_4v01d_17_!!!!!}
```