Cryptohack General

-

Challenge room walkthrough

Paweł Murdzek

22 października 2025

Spis treści

Introduction

This document is a walkthrough for the "General" category of challenges on Cryptohack.

1. Encoding

1.1. ASCII

— **Code:**

```python
def decrypt_ascii_array(ascii_array):
    """
    Decrypt an array of ASCII decimal values to readable text.

    Args:
        ascii_array (list): List of ASCII decimal values (0-127)

    Returns:
        str: Decrypted text
    """
    try:
        # Convert each ASCII value to its corresponding character
        decrypted_text = ''.join(chr(value) for value in ascii_array if 0 <= value <= 127)
        return decrypted_text
    except (ValueError, TypeError) as e:
        return f"Error: {e}"
```

— **Flag:** `crypto{ASCII_pr1nt4bl3}`

— **Explanation:** This code iterates through the list of provided integers. Each integer is treated as an ASCII decimal value and is converted to its corresponding character using the ‘chr()‘ function. The ‘join()‘ method then concatenates all these characters into the final flag string.

1.2. Hex

— **Code:**

```python
def decrypt_hex_string(hex_string):
    """
    Decrypt a hexadecimal string to readable ASCII text.

    Args:
        hex_string (str): Hexadecimal string (without 0x prefix)

    Returns:
        str: Decrypted ASCII text
    """
    try:
        # Convert hex pairs to ASCII characters
        decrypted_text = ''
        for i in range(0, len(hex_string), 2):
            hex_pair = hex_string[i:i+2]
            ascii_value = int(hex_pair, 16)

            # Check if it's a valid ASCII character (0-127)
            if 0 <= ascii_value <= 127:
                decrypted_text += chr(ascii_value)
            else:
                decrypted_text += f'[{ascii_value}]'  # Non-ASCII values in brackets

        return decrypted_text

    except ValueError as e:
        return f"Error: Invalid hexadecimal string - {e}"
    except Exception as e:
        return f"Error: {e}"
```

— **Flag:** `crypto{You_will_be_working_with_hex_strings_a_lot}`

— **Explanation:** The code reads the input hex string two characters at a time. Each two-character pair represents one byte. The `int(hex_pair, 16)` function converts this hex pair into its decimal integer equivalent. This integer is then converted to its ASCII character using ‘chr()‘. All characters are concatenated to form the flag.

1.3. Base64

— **Code:**

```python
def hex_to_base64(hex_string):
    """
    Convert a hexadecimal string to Base64.

    Args:
        hex_string (str): Hexadecimal string to convert.

    Returns:
        str: Base64 encoded string.
    """
    try:
        # Convert hex string to bytes
        byte_data = bytes.fromhex(hex_string)
        # Encode bytes to Base64
        base64_data = base64.b64encode(byte_data).decode("utf-8")
        return base64_data
    except ValueError as e:
        return f"Error: Invalid hexadecimal string. {e}"
```

— **Flag:** `crypto/Base+64+Encoding+is+Web+Safe/`
— **Explanation:** The challenge requires converting hex to Base64. The code first decodes the hex string into raw bytes using `bytes.fromhex()`. Then, it re-encodes these raw bytes into a Base64 string using the `base64.b64encode()` function. The final `.decode("utf-8")` ensures the output is a string, not a bytes object.

1.4. Bytes and Big Integers

— **Code:**

```python
from Crypto.Util.number import long_to_bytes

def integer_to_message(integer_value):
    """
    Convert a large integer into a readable message.

    Args:
        integer_value (int): The integer to convert.

    Returns:
        str: The decoded message.
    """
    try:
        # Convert the integer to bytes
        message_bytes = long_to_bytes(integer_value)
        # Decode the bytes to a string
        return message_bytes.decode('utf-8')
    except Exception as e:
        return f"Error: {e}"
```

— **Flag:** `crypto{3nc0d1n6_4ll_7h3_w4y_d0wn}`
— **Explanation:** This problem is the reverse of converting a string to a big integer. The `pycryptodome` library's `long_to_bytes()` function is used. It takes the large integer and converts it into its equivalent byte representation. These bytes are then decoded using UTF-8 to reveal the underlying text message.

1.5. Encoding Challenge

— **Code:**

```python
from pwn import * # pip install pwntools
import json
import base64
import codecs

r = remote('socket.cryptohack.org', 13377, level='debug')

def json_recv():
    line = r.recvline()
    return json.loads(line.decode())

def json_send(hsh):
    request = json.dumps(hsh).encode()
    r.sendline(request)

def decode_message(data_type, encoded_value):
    if data_type == "base64":
        # Decode Base64
        return base64.b64decode(encoded_value).decode('utf-8')
    elif data_type == "hex":
        # Decode Hex
        return bytes.fromhex(encoded_value).decode('utf-8')
    elif data_type == "rot13":
        # Decode ROT13
        return codecs.decode(encoded_value, 'rot_13')
    elif data_type == "bigint":
        # Decode Big Integer
        hex_value = encoded_value[2:]  # Remove the "0x" prefix
        return bytes.fromhex(hex_value).decode('utf-8')
    elif data_type == "utf-8":
        # Decode UTF-8 (just return the string as-is)
        return ''.join([chr(b) for b in encoded_value])
    else:
        raise ValueError(f"Unknown encoding type: {data_type}")

# Start receiving and decoding messages
while True:
    received = json_recv()
    print("Received:", received)

    data_type = received["type"]
    encoded_value = received["encoded"]

    try:
        decoded_value = decode_message(data_type, encoded_value)
        print("Decoded value:", decoded_value)

        to_send = {
            "decoded": decoded_value
        }
        json_send(to_send)
    except Exception as e:
        print("Error:", e)
        break
```

— **Flag:** `crypto{3nc0d3_d3c0d3_3nc0d3}`
— **Explanation:** This script automates solving 100 rounds of encoding challenges. It uses `pwntools` to connect to the server. In each round, it receives a JSON object, identifies the `type` of encoding, and calls a helper function. This function decodes the `encoded` value (using `base64.b64decode`, `bytes.fromhex`, `codecs.decode`,

`bytes.fromhex` for bigint, or `chr()` for utf-8) and sends the plaintext back to the server to receive the next challenge.

2. XOR

2.1. XOR Starter

— **Code:**

```python
def xor_with_key(input_string, key):
    """
    XOR each character in the input string with the given key.

    Args:
        input_string (str): The string to XOR.
        key (int): The integer key to XOR with.

    Returns:
        str: The resulting XORed string.
    """
    return ''.join(chr(ord(char) ^ key) for char in input_string)
```

— **Flag:** `crypto{aloha}`

— **Explanation:** The code performs a simple single-key XOR. It iterates over each character in the `input_string`, converts the character to its ASCII integer value using `ord()`, and then performs a bitwise XOR (`^`) with the provided integer `key`. The resulting integer is converted back to a character with `chr()`.

2.2. XOR Properties

— **Code:**

```python
from pwn import *

def xor_bytes(bytes1, bytes2):
    """
    XOR two byte arrays and return the result.

    Args:
        bytes1 (bytes): The first byte array.
        bytes2 (bytes): The second byte array.

    Returns:
        bytearray: The XORed result as a bytearray.
    """
    return bytearray(b1 ^ b2 for b1, b2 in zip(bytes1, bytes2))

# Given values
KEY1 = bytes.fromhex('a6c8b6733c9b22de7bc0253266a3867df55acde8635e19c73313')
MASK1 = bytes.fromhex('37dcb292030faa90d07eec17e3b1c6d8daf94c35d4c9191a5e1e')
MASK2 = bytes.fromhex('c1545756687e7573db23aa1c3452a098b71a7fbf0fdddde5fc1')
MASK3 = bytes.fromhex('04ee9855208a2cd59091d04767ae47963170d1660df7f56f5faf')

# Calculate keys
KEY2 = xor_bytes(KEY1, MASK1)
KEY3 = xor_bytes(KEY2, MASK2)

# Calculate FLAG
FLAG = xor_bytes(KEY1, xor_bytes(KEY2, xor_bytes(KEY3, MASK3)))

# Print results
print("KEY1:", KEY1.hex())
print("KEY2:", KEY2.hex())
print("KEY3:", KEY3.hex())
print("FLAG:", "".join(chr(b) for b in FLAG))
```

— **Flag:** `crypto{x0r_i5_ass0c1at1v3}`

— **Explanation:** This solution uses the associative property of XOR. The challenge defines a chain of keys: `KEY2 = KEY1 ^ MASK1` and `KEY3 = KEY2 ^ MASK2`. The flag is `FLAG = KEY1 ^ KEY2 ^ KEY3 ^ MASK3`. The code simply implements these operations sequentially, XORing the byte arrays together to calculate `KEY2`, then `KEY3`, and finally the `FLAG`.

2.3. Favourite byte

— **Code:**

```python
def single_byte_xor_brute_force(ciphertext_bytes):
    """
    Tries every possible single-byte key (0-255) to decrypt the ciphertext.

    Args:
        ciphertext_bytes (bytes): The raw bytes of the encrypted message.

    Returns:
        A list of tuples (score, key, decrypted_text) sorted by score.
    """
    results = []
    for key_val in range(256):
        key_byte = bytes([key_val])
        decrypted = repeating_key_xor(ciphertext_bytes, key_byte)
        score = score_text(decrypted)

        try:
            # Attempt to decode for display purposes
            decrypted_str = decrypted.decode('utf-8', errors='ignore')
```

```
            results.append((score, key_val, decrypted_str))
        except UnicodeDecodeError:
            continue

    # Sort results by score in descending order
    results.sort(key=lambda x: x[0], reverse=True)
    return results

def brute_force_single_byte_xor_from_input(hex_string=None):
    """
    Brute-force single-byte XOR decryption for a user-provided hex string.

    Args:
        hex_string (str, optional): The hex string to decrypt. If None, prompts user for input.
    """
    if hex_string is None:
        hex_string = input("Enter the encrypted message (in hex format): ").strip()

    try:
        ciphertext = bytes.fromhex(hex_string)
    except ValueError:
        print("Invalid hex input. Please provide a valid hex string.")
        return

    print("\n--- Brute-forcing single-byte XOR keys ---")
    brute_force_results = single_byte_xor_brute_force(ciphertext)

    print("Top 100 possible decryptions:")
    for i, (score, key, text) in enumerate(brute_force_results):#(brute_force_results[:100]):
        print(f"#{i+1}: Score={score}, Key=0x{key:02x} ('{chr(key) if 32<=key<=126 else '.'}'), Text='{text}'")
```

— **Flag:** `crypto{0x10_15_my_f4v0ur173_by7e}`

— **Explanation:** The code performs a brute-force attack to find a single-byte XOR key. It iterates through all 256 possible byte values (0-255). For each value, it XORs the entire ciphertext with that byte. A (missing) scoring function (`score_text`) is used to evaluate how "English-like" the resulting plaintext is, likely by checking character frequencies. The script then prints the decryptions with the highest scores, one of which is the flag.

2.4. You either know, XOR you don't

— **Code:**

```
#!/usr/bin/env python3
"""
Find XOR key using known plaintext attack.
We know the plaintext contains "crypto{" and can use this to find the key.
"""

hex_cipher = '0e0b213f26041e480b26217f27342e175d0e070a3c5b103e2526217f27342e175d0e077e263451150104'
known_plain = 'crypto{'

# Convert hex to bytes
cipher_bytes = bytes.fromhex(hex_cipher)

print("Known-Plaintext Attack")
print("=" * 80)
print(f"Ciphertext (hex): {hex_cipher}")
print(f"Known plaintext: '{known_plain}'")
print(f"Ciphertext length: {len(cipher_bytes)} bytes")
print("=" * 80)
print()

def repeating_key_xor(ciphertext_bytes, key_bytes):
    if not key_bytes:
        return ciphertext_bytes
    decrypted_bytes = bytearray()
    for i in range(len(ciphertext_bytes)):
        decrypted_bytes.append(ciphertext_bytes[i] ^ key_bytes[i % len(key_bytes)])
    return bytes(decrypted_bytes)

# First, try single-byte XOR
print("1. Testing Single-Byte XOR")
print("-" * 80)
for offset in range(len(cipher_bytes) - len(known_plain) + 1):
    potential_key_bytes = []
    for i in range(len(known_plain)):
        potential_key_bytes.append(cipher_bytes[offset + i] ^ ord(known_plain[i]))

    # Check if all bytes are the same (single-byte key)
    if len(set(potential_key_bytes)) == 1:
        key = potential_key_bytes[0]
        print(f"! Found single-byte XOR key at offset {offset}!")
        print(f"  Key: {key} (0x{key:02x}, ASCII: '{chr(key) if 32<=key<=126 else '?'}')")

        decrypted = bytes([b ^ key for b in cipher_bytes])
        try:
            decrypted_str = decrypted.decode('utf-8')
            print(f"  Decryption: {decrypted_str}")
            if 'crypto{' in decrypted_str:
                print(f"  !!! SUCCESS! !!!")
        except:
            pass
        print()

# Now try repeating key patterns
print("\n2. Analyzing Repeating Key Patterns")
print("-" * 80)

# Extract potential key by XORing known plaintext with ciphertext at position 0
potential_key = bytes([cipher_bytes[i] ^ ord(known_plain[i]) for i in range(len(known_plain))])
print(f"Key bytes from 'crypto{{' at position 0:")
print(f"  Hex: {potential_key.hex()}")
print(f"  As string: '{potential_key.decode('utf-8', errors='ignore')}'")
print(f"  As bytes: {list(potential_key)}")
print()

# Try different key lengths
print("3. Testing Different Key Lengths")
```

```
print("-" * 80)

for key_len in range(1, 21):
    # Use the first key_len bytes as the repeating key
    test_key = potential_key[:key_len]
    decrypted = repeating_key_xor(cipher_bytes, test_key)

    try:
        decrypted_str = decrypted.decode('utf-8', errors='strict')

        # Check if it contains crypto{ and looks like valid text
        if 'crypto{' in decrypted_str:
            print(f"! Key length {key_len} - PROMISING!")
            print(f"  Key (hex): {test_key.hex()}")
            print(f"  Key (str): '{test_key.decode('utf-8', errors='ignore')}'")
            print(f"  Decrypted: {decrypted_str}")

            # Check if decryption is complete (ends with })
            if decrypted_str.strip().endswith('}'):
                print(f"  !!! COMPLETE MESSAGE! !!!")
            print()
    except:
        pass

# Advanced: Try to find key by looking for patterns in the ciphertext
print("\n4. Pattern Analysis - Finding Repeating Blocks")
print("-" * 80)

# Look for repeating sequences in ciphertext (might indicate repeating key)
for block_size in range(2, 11):
    blocks = {}
    for i in range(0, len(cipher_bytes) - block_size + 1):
        block = cipher_bytes[i:i+block_size]
        if block in blocks:
            blocks[block].append(i)
        else:
            blocks[block] = [i]

    repeating = {k: v for k, v in blocks.items() if len(v) > 1}
    if repeating:
        print(f"Block size {block_size}: Found {len(repeating)} repeating sequences")
        for block, positions in list(repeating.items())[:3]:  # Show first 3
            distances = [positions[i+1] - positions[i] for i in range(len(positions)-1)]
            print(f"  {block.hex()} at positions {positions}, distances: {distances}")

print("\n" + "=" * 80)
print("Analysis complete!")
```

— **Flag:** `crypto{1f_y0u_Kn0w_En0uGH_y0u_Kn0w_1t_4ll}`

— **Explanation:** This is a known-plaintext attack. Since the flag format is `crypto{...}`, we can XOR the known plaintext "crypto{" with the start of the ciphertext. This reveals the first 7 bytes of the repeating XOR key. The script then tests different possible key lengths (from 1 to 20) using slices of this recovered key material. The correct key length (7) successfully decrypts the entire message.

2.5. Lemur XOR

— **Code:**

```
import os
from PIL import Image
import numpy as np

# Set your image filenames here
image1_path = 'flag.png'
image2_path = 'lemur.png'
output_path = 'xor_revealed.png'

def xor_images(img1_path, img2_path, output_path):
    # Load images
    img1 = Image.open(img1_path).convert('RGB')
    img2 = Image.open(img2_path).convert('RGB')

    # Ensure images are the same size
    if img1.size != img2.size:
        raise ValueError('Images must be the same size!')

    arr1 = np.array(img1)
    arr2 = np.array(img2)

    # XOR the RGB values
    xor_arr = np.bitwise_xor(arr1, arr2)
    xor_img = Image.fromarray(xor_arr.astype('uint8'), 'RGB')
    xor_img.save(output_path)
    print(f'Revealed image saved to: {output_path}')

if __name__ == '__main__':
    # Place image1.png and image2.png in this folder before running
    xor_images(image1_path, image2_path, output_path)
```

— **Flag:** `crypto{XORly_n0t!}`

— **Explanation:** This solution leverages the property that XORing an image with a key (another image) and then XORing the result with the same key again restores the original. The code uses PIL to open both images and NumPy to convert them into arrays. `np.bitwise_xor` performs a pixel-by-pixel XOR on the two image arrays, revealing the hidden flag image, which is then saved.

3. Mathematics

3.1. Greatest Common Divisor

— **Code:**

```python
def gcd(a: int, b: int) -> int:
    """Compute the greatest common divisor of a and b using Euclid's algorithm."""
    while b:
        a, b = b, a % b
    return abs(a)

print(gcd(12,8))
print(gcd(66528,52920))
```

— **Flag:** `Your Flag Here`

— **Explanation:** This code implements the Euclidean algorithm to find the greatest common divisor (GCD). It repeatedly applies the logic: $\gcd(a, b) = \gcd(b, a \pmod b)$. The 'while' loop continues until 'b' becomes 0, at which point 'a' contains the GCD of the original two numbers.

3.2. Extended GCD

— **Code:**

```python
# p*u+q*v=gcd(p,q)
# p*u = gcd(p,q) - q*v
# u = (gcd(p,q) - q*v)/p
# v = (gcd(p,q) - p*u)/q
def extended_gcd(a: int, b: int) -> tuple[int, int, int]:
    """Compute the extended greatest common divisor of a and b.

    Returns a tuple (g, x, y) such that g = gcd(a, b) and g = a*x + b*y.
    """
    if a == 0:
        return b, 0, 1
    else:
        g, x1, y1 = extended_gcd(b % a, a)
        x = y1 - (b // a) * x1
        y = x1
        return g, x, y

result = extended_gcd(26513, 32321)
print(result)
```

— **Flag:** `-8404`

— **Explanation:** This code uses the Extended Euclidean Algorithm (EEA). Beyond finding the GCD g, it also finds two integers x and y that satisfy Bézout's identity: $a \cdot x + b \cdot y = g$. The function is implemented recursively, using the results from the smaller problem $(b \pmod a, a)$ to calculate the coefficients for (a, b).

3.3. Modular Arithmetic 1

— **Code:**

```python
print(11 % 6)
print(8146798528947 % 17)
```

— **Flag:** `4`

— **Explanation:** The code uses Python's modulo operator ("%") to solve the problems. $a \pmod n$ calculates the remainder when a is divided by n. This directly computes the answer for $11 \pmod 6$ and $8146798528947 \pmod{17}$.

3.4. Modular Arithmetic 2

— **Code:**

```python
print(3**17 % 17)
print(5**17 % 17)
print(7**17 % 17)
print(7**16 % 17)
print(3**16 % 17)
print(5**16 % 17)
```

— **Flag:** `1`

— **Explanation:** This code demonstrates Fermat's Little Theorem, which states that for a prime p and an integer a, $a^p \equiv a \pmod p$. Also, if a is not divisible by p, then $a^{p-1} \equiv 1 \pmod p$. Since 17 is prime, $a^{17} \pmod{17}$ will always equal $a \pmod{17}$, and $a^{16} \pmod{17}$ will equal 1 (for $a \neq 0$).

3.5. Modular Inverting

— **Code:**

```python
d=1
while 3*d % 13 != 1:
    print("for d = " + str(d) + ": 3*d % 13 = " + str(3*d % 13))
    d+=1

print("answer is: " + str(d) + " because 3*" + str(d) + " % 13 = " + str(3*d % 13))
```

— **Flag:** 9
— **Explanation:** The code finds the modular multiplicative inverse of 3 modulo 13 by brute force. It's looking for an integer d such that $3 \cdot d \equiv 1 \pmod{13}$. It starts d at 1 and increments it, checking the result of $(3 \cdot d) \pmod{13}$ at each step. The loop stops when it finds a d (which is 9) that makes the expression equal to 1.

4. Data Formats

4.1. PRIVACY-ENHANCED MAIL?

— **Code:**
```python
from cryptography.hazmat.primitives import serialization

# Load the PEM file
with open('privacy_enhanced_mail.pem', 'rb') as f:
    pem_data = f.read()

# Load the private key
private_key = serialization.load_pem_private_key(pem_data, password=None)

# Extract the private exponent d
d = private_key.private_numbers().d

# Print d as a decimal integer
print(d)
```
— **Flag:** 156827002880563313647871710458199736549911499491979599298608612281800217073168519244562055436653
— **Explanation:** This script uses the `cryptography` library to parse a PEM file, which is a standard format for storing cryptographic keys. It loads the file as an RSA private key using `load_pem_private_key()`. It then accesses the key's numerical components via `.private_numbers()` and extracts the private exponent, d.

4.2. CERTainly not

— **Code:**
```python
from cryptography import x509

with open('2048b-rsa-example-cert.der', 'rb') as f:
    der_data = f.read()

# Load the DER-encoded certificate
cert = x509.load_der_x509_certificate(der_data)

# Extract the public key (assuming RSA)
public_key = cert.public_key()

# Get the modulus n as a decimal integer
n = public_key.public_numbers().n

# Print n
print(n)
```
— **Flag:** 22825373692019530804306212864609512775374171823993708516509897631547513634635856375624003737068
— **Explanation:** The code parses a X.509 certificate stored in the binary DER format. It uses `x509.load_der_x509_certifi` to load the file. From the certificate object, it extracts the public key (`.public_key()`). Finally, it accesses the public key's numerical components and prints the RSA modulus, n.

4.3. SSH Keys

— **Code:**
```python
from cryptography.hazmat.primitives import serialization

# Load the SSH public key from file (assuming it's named 'bruce_rsa.pub')
with open('bruce_rsa.pub', 'r') as f:
    ssh_key_data = f.read().strip()

# Load the public key
public_key = serialization.load_ssh_public_key(ssh_key_data.encode())

# Extract the modulus n as a decimal integer
n = public_key.public_numbers().n

# Print n
print(n)
```
— **Flag:** 39314062729225234484361945998200930162414726581518015528450945185795078159906004596692596036452
— **Explanation:** This script parses an SSH public key file (which has its own format, different from PEM or DER). It uses `serialization.load_ssh_public_key()` to read the key data. Just like in the previous example, it then accesses the public key's numerical components to extract and print the RSA modulus, n.

4.4. Transparency

— **Flag:** `crypto{thx_redpwn_for_inspiration}`
— **Explanation:** This challenge was solved manually using a Certificate Transparency (CT) log search tool. By searching the domain associated with the challenge on a site like `sslmate.com`, one can inspect the public certificate logs. The flag is hidden on website.