Crypto challenges write up

1. PSAgames

```python
from Crypto.Util.number import bytes_to_long, getPrime, GCD
from Crypto.Util.Padding import pad
from secret import FLAG
WELCOME = '''Welcome to my custom PSA cryptosystem!
In this cryptosystem, the message is PKCS#7 padded and then encrypted with RSA.
They say padding makes encryption more secure, right? ;)'''
MENU = '''
[1] Encrypt the flag
[2] Exit
'''
class PSA:
    def __init__(self):
        self.bit_size = 512
        self.e = 11
    def gen_modulus(self):
        while True:
            p = getPrime(self.bit_size // 2)
            q = getPrime(self.bit_size // 2)
            if GCD(self.e, (p - 1) * (q - 1)) == 1:
                break
        return p * q
    def encrypt(self, msg):
        m = bytes_to_long(pad(msg, 16))
        n = self.gen_modulus()
        c = pow(m, self.e, n)
        return c, n
def main():
    psa = PSA()
    print(WELCOME)
    while True:
        try:
            print(MENU)
            opt = input('> ')
            if opt == '1':
                enc, modulus = psa.encrypt(FLAG)
                print(f"\n{hex(enc)}\n{hex(modulus)}")
            elif opt == '2':
                print('Bye.')
                exit(1)
            else:
                print('\nInvalid option!')
        except:
            print('\n\nSomething went wrong.')
            exit(1)
if __name__ == '__main__':
    main()
```

Figure 1: server code

```python
    def encrypt(self, msg):
        m = bytes_to_long(pad(msg, 16))
        n = self.gen_modulus()
        c = pow(m, self.e, n)
        return c, n
```

Figure 2: encrypt function

Noticed e=11, PKCS#7 was used as the padding and the message was the same flag each time which means each request returns c = m^11 mod n with n different each time. But this padding is applied **before RSA**, and more importantly,there is **no randomized padding** (like OAEP) So the padded message is still m = pad(FLAG) which is identical each time.

The main vulnerability was the same message was encrypted 11 times and the padding is not randomized so if we collect 11 ciphertext we can use the **Chinese Remainder Theorem (CRT).**

```
fixit@BQAA-5065:~/HTB$ cat solve.py
from pwn import *
from Crypto.Util.number import long_to_bytes
from Crypto.Util.Padding import unpad
import gmpy2
from functools import reduce
HOST = '154.57.164.69'
PORT = 31087
E = 11
def crt(remainders, moduli):
    total_modulus = reduce(lambda x, y: x * y, moduli)
    result = 0
    for c_i, n_i in zip(remainders, moduli):
        M_i = total_modulus // n_i
        y_i = gmpy2.invert(M_i, n_i)
        result += c_i * M_i * y_i
    return result % total_modulus
def solve():
    c_list = []
    n_list = []
    io = remote(HOST, PORT)
    print(f"[*] Collecting {E} samples...")
    for i in range(E):
        io.sendlineafter(b'> ', b'1')
        def get_hex():
            while True:
                line = io.recvline().strip().decode()
                if line.startswith('0x'):
                    return int(line, 16)
        c = get_hex()
        n = get_hex()
        c_list.append(c)
        n_list.append(n)
        print(f"[+] Sample {i+1}/{E} acquired.")
    io.close()
    # 1. Apply CRT
    print("[*] Calculating CRT...")
    result_crt = crt(c_list, n_list)
    # 2. Extract 11th root
    print("[*] Extracting 11th root...")
    m, exact = gmpy2.iroot(result_crt, E)
    if exact:
        # 3. Unpad and print
        full_msg = long_to_bytes(int(m))
        try:
            flag = unpad(full_msg, 16)
            print(f"\nFLAG FOUND: {flag.decode()}")
        except:
            print(f"\nRaw recovered bytes: {full_msg}")
    else:
        print("\nError: Could not find an exact integer root. Ensure all 11 samples are unique.")
if __name__ == "__main__":
    solve()
```

It stores these values and applies the Chinese Remainder Theorem to combine them into a single integer representing m^11.

Using `gmpy2.iroot`, it computes the exact 11th root to recover the padded plaintext.

The result is converted from an integer to bytes.

Finally, PKCS#7 padding is removed to reveal the flag.

2.

```python
import os

flag = open("flag.txt", "rb").read()

def genkeys(n):
    keys = [os.urandom(5) for _ in range(n)]
    return keys

def encrypt(keys, flag):
    for key in keys:
        flag = bytes([a ^ b for a, b in zip(flag, key * (len(flag)+5//5))])
    return flag

keys = genkeys(1955) # with this many keys, this is totally secure
print(encrypt(keys, flag).hex())
```

Figure 4: script code

XOR is linear, so applying it many times does not add security — all 1955 XOR operations collapse into a single XOR with one combined key.
Because each key is only 5 bytes and repeats across the flag, the scheme becomes equivalent to encrypting with one repeating 5-byte keystream.
Knowing the flag starts with **HTB{** lets us recover 4 of the 5 keystream bytes using XOR reversal.
Only one byte remains unknown.
We brute-force the final byte to reconstruct the keystream and decrypt the entire flag.

```python
from Crypto.Util.strxor import strxor

ciphertext = bytes.fromhex("cf7bfbb7e9b770ceffd2f51c8da0e1fe70d7ffbee370cda4bcf470d4f8e3fe70daf8ffd8448ab5b8d84989bed2b370cafde1eb56e6b8bcea4ae6a1b9e44788a2beb810c4")

# We know the first 4 bytes are HTB{
known_prefix = b"HTB{"
# Recover first 4 bytes of the 5-byte key
key_part = [ciphertext[i] ^ known_prefix[i] for i in range(4)]

print("[*] Brute-forcing the 5th byte of the key...")

for b5 in range(256):
    # Construct the full 5-byte key
    test_key = bytes(key_part + [b5])

    # Repeat key to match ciphertext length
    full_key_stream = (test_key * (len(ciphertext) // 5 + 1))[:len(ciphertext)]
    decrypted = strxor(ciphertext, full_key_stream)

    # Check if it's a valid ASCII flag and has the closing brace
    if b"}" in decrypted:
        try:
            res = decrypted.decode()
            # If it contains common English/Leetspeak words, it's the one
            if "HTB{" in res:
                print(f"\n SUCCESS!")
                print(f" Key (hex): {test_key.hex()}")
                print(f" Flag: {res}")
                break
        except UnicodeDecodeError:
            continue
```