HackRush CTF Writeup

Praveen Venkatesh

Team humanHackers (flyingMango)

Ancient:

Question: I found some weird text, Can you find out what this means? 𑀩𑁆𑀭𑀸𑀳𑁆𑀫𑀻 𑀮𑀺𑀧𑀺 𑀪𑀸𑀭𑀢 𑀓𑀻 𑀧𑁆𑀭𑀸𑀘𑀻𑀦𑀢𑀫 𑀮𑀺𑀧𑀺𑀬𑁄𑀁 𑀫𑁂𑀁 𑀲𑁂 𑀏𑀓 𑀳𑁃 𑀬𑀳 𑀳𑁃 𑀆𑀧𑀓𑀸 𑀚𑀯𑀸𑀩 : HackRushCTF{𑀅𑀰𑁄𑀓}

Approach:

- Search individual characters on google.
- All of the characters were part of the Unicode - Brahmi script, an older version of Devanagari script
- Upon converting from Brahmi to Devanagari, we get the following output:
    - ब्राह्मी लिपि भारत की प्राचीनतम लिपियों में से एक है यह है आपका जवाब : HackRushCTF{अशोक}
- The flag is clearly **HackRushCTF{ashok}**

Prime Magic 1:

Question :

```python
prime_magic_1.py > ...
from binascii import hexlify, unhexlify
import math
big_number = 2599234786109921906106922184321451886075632748617331902711875909179594182693067 7
exponent = 0x10001
exponent = 65537

flag = b"__redact__" # Who knows what was here?
flag = int(hexlify(flag), 16)
magic = pow(flag, exponent, big_number)
print("Something magical: {}".format(magic))

# Something magical: 2302696361255313845399424134185854566916195449801892315821048752094293732889946 3
```

Approach:

- Here, we are given a simple implementation of the RSA encryption algorithm.
- The RSA algorithm is implemented as follows
    - Choose two prime numbers (p, q)
    - We evaluate n = p*q. This forms part of the public key
    - A number e is chosen, that is an integer, not a factor of n, and is between 1 and phi(n)
        - phi(n) is the Euler totient function

    - e and n are publicly available. Here they're given as:
        - e = 65537
        - n = 25992347861099219061069221843214518860756327486173319027118759091795941826930677
    - The private key has not been revealed. It is formed by the totient function generated as phi(n) = (p-1)(q-1)
    - Encryption is carried out as
        - Encrypted = (hex(data))^e % n
- In order to decrypt the data, we need to find the factors of n. This is a tedious process and cannot be computed by a normal desktop pc in reasonable time. So we use factordb.com to get the factors which have already been precomputed. The factors are:
    - P = 6918082374901313855125397665325977135579
    - Q = 3757160792909754673945392226295475594863
- Once we have p and q, we evaluate the totient function as phi(n) = p-1 * q-1
- For encryption, we need to reverse the modulus operation. So we compute the inverse modulo of (e, phi(n)) to get a value d.
- Now, we can simply decrypt the data as (encrypted)^d % n to get the original data.

```
soln.py > ...
c = 23026963612553138453994241341858545669161954498018923158210487520942937328899463
n = 25992347861099219061069221843214518860756327486173319027118759091795941826930677
e = 65537
p = 3757160792909754673945392226295475594863
q = 6918082374901313855125397665325977135579
phi = (p-1)*(q-1)
from Crypto.Util.number import inverse

d=inverse(e,phi)
m=pow(c,d,n)
print(hex(m))
```

Prime Magic 2:

Question:

```python
prime_magic_2.py > ...
from binascii import hexlify

big_number = 13269353506569762322866448443179444023604712744966341096534397703952746262066379915270
exponent = 0x10001

flag = b"---REDACTED---" # Who knows what was here?
flag = int(hexlify(flag), 16)

magic = pow(flag, exponent, big_number)
print("Something magical: {}".format(magic))


# Something magical: 11901809647332451373849722974618021132106337910274926950679037190778251444311765762999
```

Approach:

- In this problem, we see a repeat of the previous RSA algorithm. However, we notice that upon factoring in the value of n, the decomposition is not 2 prime factors of n, but rather 9 prime factors.
- In order to solve this problem, we evaluate a new totient function as
  - phi(n) = (p-1)(q-1)(r-1)...(z-1), where p, q, r, z are the prime factors of n
- We then repeat the previous procedure to decrypt the flag.

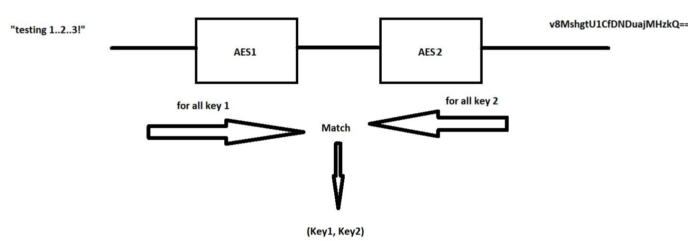
```python

c = 11901809647332451373849722974618021132106337910274926950679037190778251444311765762999
n = 13269353506569762322866448443179444023604712744966341096534397703952746262066379915270
e = 65537
p = 6918082374901313855125397665325977135579
q = 1918068156388358858595862185446103245933510130
# print()
print((p-1) * (q-1))
phi = 1* 2 * 4 * 6 * 10 * 12 * 16 * 3757160792909754673945392226295475594862 * 6918082374901313855125397665325977135578
# phi = 1326935350656976232286644844317944402360279466989187036277448798664190249349446926956 2
print(phi)
from Crypto.Util.number import inverse

d=inverse(e,phi)
m=pow(c,d,n)
print(hex(m))
```

Double the Trouble:

Question:

```python
import hashlib
import base64
from Crypto.Cipher import AES
import random
import string

def random_key():
    possible = list(string.printable)
    n = len(possible)
    key = b"".join([possible[random.randint(0, n - 1)].encode() for i in range(32)])
    key = b'0'*29 + key[-3:]
    print(key)
    return key

key1 = random_key()
key2 = random_key()

plaintext = b"testing 1..2..3!"


iv = hashlib.md5(b"Goodluck!").digest()

aes1 = AES.new(key1, AES.MODE_CBC, iv = iv)
single_pass = aes1.encrypt(plaintext)

aes2 = AES.new(key2, AES.MODE_CBC, iv = iv)
encrypted = aes2.encrypt(single_pass)

print(base64.b64encode(encrypted))

encrypted = b'v8MshgtU1CfDNDuajMHzkQ==' # Result of testing

# encrypted_flag = b"aUyXnj4SxFmYht39qIppFKIVDjQ/tTBbPwpSLo2IoHo="
```

Approach:



- In this problem, we have a message that has been encrypted twice using 2 random keys. This seems hard to crack on the outset, however, we also have a message, and its encrypted value as:
    - Message: testing 1..2..3!
    - Encrypted : v8MshgtU1CfDNDuajMHzkQ==
- In this problem, we also see that the keygen for generating the AES keys is limited to only 3 characters of the 32 byte key. Hence, there are only 100^3 combinations of keys for each of the encryptions.
- By utilizing this fact, we can decode the original keys using the following method:
    - For all possible keys, generate the encryption of the message "testing 1..2..3!". This will give the stage 1 encryption of the algorithm.
    - For all possible keys, decrypt the encrypted message : "v8MshgtU1CfDNDuajMHzkQ==". This again gives the stage 1 encryption of the algorithm.
    - So, now, if the encryption of the message and the decryption of the encrypted message are same, we know that we find the correct key pair used for the encryption.

- By following the above algorithm, due to the limited number of possible keys, we decode the flag quickly!

```python
def solve(plaintext,ciphertext,KeyGen):
    encrypted = {}
    for key in KeyGen():
        AEScipher = newAES(key)
        encrypted[AEScipher.encrypt(plaintext)] = key

    for key in KeyGen():
        AEScipher = newAES(key)
        decrypted = AEScipher.decrypt(ciphertext)
        if(decrypted in encrypted):
            Key1 = encrypted[decrypted]
            Key2 = key
            return (Key1,Key2)
```

```python
def sample_KeyGen():
    possible = list(string.printable)
    t = len(possible)**3 - 1
    l = len(possible) - 1
    a = 0
    b = 0
    c = 0
    for i in range(t):
        if c == l :
            b += 1
            c = 0
        if b == l :
            a += 1
            c = 0
            b = 0
        if a == l:
            print(a, b, c)
            break
        key = b'0'*29 + bytes(possible[a], encoding='utf8') \
            + bytes(possible[b], encoding='utf8') \
            + bytes(possible[c], encoding='utf8')
        c += 1
        yield key
```