

ROCA

Introduction

[ROCA](#) is a crypto challenge where we're given a secret that was encrypted with a vulnerable RSA key generation technique, this weakness allows the private key of a key pair to be recovered from the public key^[1]. Upon recovering the private key we can use it along with the public key to decrypt the secret flag.

⚠ Make sure you create a Python virtual environment for this challenge and move the `reto.py` and `secret.enc` files inside of it.

Understanding the RSA algorithm

Before getting into the challenge's details we need to understand how the RSA algorithm works. The algorithm has been broken down without adding too many mathematical quirks to make it more understandable for a beginner level audience in crypto CTF challenges.

📄 We will use small integers for learning purposes in the following example.

1. Choose two large prime numbers p and q
 - To make factoring infeasible, p and q must be chosen at random from a large space of possibilities, such as all prime numbers between 2^{1023} and 2^{1024} . Many different algorithms for prime selection are used in practice.

$$p = 61$$

$$q = 53$$

2. Compute $n = p * q$
 - n is used as the modulus for both the public and private keys. Its length, usually expressed in bits, is the key length.

$$n = p * q$$

$$n = 3233$$

3. Compute the Carmichael's totient function of the product as $\lambda(n) = lcm(p - 1, q - 1)$

$$\lambda(n) = lcm(60, 52)$$

$$\lambda(3233) = lcm(60, 52) = 780$$

- The *lcm* (least common multiple) may be calculated through the Euclidean algorithm.
 - $\lambda(n)$ is **kept secret**.
4. Choose any number $1 < e < 780$ that is coprime to 780. Choosing a prime number for e leaves us only to check that e is not a divisor of 780.

$$e = 17$$

- e having a short bit-length and small Hamming weight results in more efficient encryption – the most commonly chosen value for e is $2^{16} + 1 = 65537$
 - e is released as part of the **public key**.
5. Determine d as:

$$d = e^{-1}(\text{mod } \lambda(n)) = 413$$

$$d = 413$$

- d is the modular multiplicative inverse of e modulo $\lambda(n)$.
- This means: solve for d the equation $de \equiv 1(\text{mod } \lambda(n))$; d can be computed efficiently by using the extended Euclidean algorithm, since, thanks to e and $\lambda(n)$ being coprime, said equation is a form of Bézout's identity, where d is one of the coefficients^[2].

Important

- The **public key** consists of the modulus n and the public exponent e .
- The **private key** consists of the private exponent d , which must be kept **secret**.
- p , q , and $\lambda(n)$ must also be kept **secret** because they can be used to calculate d . In fact, they can all be discarded after d has been computed^[2-1].

Exploring the files we were given

We're given two files `reto.py` and `secret.enc`, next we're gonna reverse engineer these files to understand how they work and what they're doing.

`reto.py`

This file contains four lines of library imports:

```
from Crypto.Util.number import *
import gmpy2, binascii
from myrsa import myreal_p, myreal_q
from oximoron import flag
```

The first line uses the `pycryptodome` library and the second line `gmpy2`, these can be imported with `pip`, the `binascii` module is part of Python and doesn't need to be installed, the other two lines are custom libraries that were created specifically for the challenge, these are not provided, but we won't be needing them.

This file contains **two public keys**:

```
n = myreal_p * myreal_q
e = 65537
```

An implicit key `n` which is the product of the `myreal_p` and `myreal_q` variables (these two previous variables have to be prime numbers) and a explicit key `e` which has the value `65537`.

Next up we have the RSA encryption routine:

```
ciphertext = binascii.hexlify(long_to_bytes(pow(bytes_to_long(flag), e, n)))
```

Let's understand what this previous line is doing from the inside out as this is key to solving the challenge:

1. `bytes_to_long(flag)` converts the secret/flag into a large integer. We need an integer because the secret is either text or binary data and the RSA algorithm needs a number, the secret data will be represented as this number in the next step.
2. `pow()` returns the calculation of the secret/flag (remember that in the previous step the secret has been converted to a number) to the power of `e` modulus `n`, this is basically the core of the RSA algorithm:

$$secret^e \bmod n$$

3. `long_to_bytes()` converts the encrypted integer returned in the `pow()` function back to bytes, this is done because we need a format that can be stored and transmitted, the raw integer (encrypted data) can't be passed to the `hexlify()` function as it expects bytes as input.
4. `binascii.hexlify()` converts the encrypted message into a hex string, this is done because the output of the encryption algorithm contains non-printable characters that are difficult to display, store, or transmit.
5. `ciphertext` the encrypted message is stored in this variable.

Now let's take a look at the following line:

```
file = open('secret.enc', 'w')
```

What this does is it opens the `secret.enc` file in write (`w`) mode and stores the file object in the `file` variable.

After storing the file in write mode, some data is written to it:

```
file.write("secret: {}\nN: {}\nE: {} ".format(ciphertext, str(n), str(e)))
```

The written data to the `secret.enc` file consists of 3 lines:

1. the encrypted data in hex format.
2. the product of the `myreal_p` and `myreal_q` variables which is stored in the `n` variable.
3. the value of the `e` variable.


The final `secret.enc` file looks like this:

```
secret:
b'55d4e09b61c53557c2a265141206ba394a92648e290c0377ca58aec1b6811254125590ea3393
563c485bad44cd5c80b85c219927a8bea340a4aa39dd7310afca'
N:
873285103090110331554602410752741242346005412079158264532729607203014952059559
8830189737190136429774375847088648891282895380791041462467946364644597106651
E: 65537
```

Decrypting the secret

Now that we understand what the files are doing and what they're for, we can start the decryption process of the secret.

At first I tried to create several Python scripts that harnessed the power of my PC's dedicated GPU to compute the factors of `n`, which in our case is the value stored in the `N` variable from the `secrets.enc` file, but the GPU wasn't powerful enough to compute such large numbers.

 In this writeup we'll use `n` and `N` interchangeably as they have the same value and this won't affect the challenge's solution.

Later, after doing some research I found a site called [factordb](#) that contains a database of factorized numbers, on this site I was able to find the factors of the `N` number we were given.

After having the factors of `n` which are:

`p = 89434994450522445031250973494975469910185057723838915257965779692637908486619`

`q = 97644675717314693028299287382143581611499104251746026552841672042065803811329`

we can create a python script to do the inverse of the one were given, that is, decrypting the secret.

Creating the Python script

First off we start by creating a python file called `roca.py` but you can name it whatever you want, or you can opt to use the file located in the repo that was created for this challenge's writeup over at

Now make sure you install the required libraries by running the following command inside the virtual environment:

```
pip install -r requirements.txt
```

We need to import the `pycryptodome`, `gmpy2` and `binascii` libraries:

```
from Crypto.Util.number import *
import gmpy2, binascii
```

`main()` function

Then a `main()` function will be created that'll work as the program's execution entry point. To tell the Python interpreter to treat the `main()` function as the entry point add the following two lines to the end of the file outside the main function:

```
if __name__ == "__main__":
    main()
```

In the `main()` function first we add the public keys `n` and `e`, and then the factorized primes from `n` are also included:

```
n =
873285103090110331554602410752741242346005412079158264532729607203014952059559
8830189737190136429774375847088648891282895380791041462467946364644597106651
e = 65537

p =
89434994450522445031250973494975469910185057723838915257965779692637908486619
q =
97644675717314693028299287382143581611499104251746026552841672042065803811329
```

After the public keys and primes have been added, the `secret` variable to be decrypted is added as well as some output fancy text to show before computing the private key:

```

secret =
b'55d4e09b61c53557c2a265141206ba394a92648e290c0377ca58aec1b6811254125590ea3393
563c485bad44cd5c80b85c219927a8bea340a4aa39dd7310afca'

print("\n" + "="*50)
print("COMPUTING PRIVATE KEY:")
print("="*50)

```

compute_private_key() function

Next up we have a call to a function that'll compute the private key `d` from the primes `p` and `q`, store the returned key in the `d` variable and some text that will output the generated key:

```

d = compute_private_key(e, p, q)
print(f"Private Key is: \n{d}\n")

```

The `compute_private_key()` function must be placed in the `roca.py` file **just before** the `main()` function:

```

def compute_private_key(e, p, q) -> int:
    phi_n = (p - 1) * (q - 1)
    d = pow(e, -1, phi_n)

    return d

```

The `compute_private_key()` function takes as arguments the public key `e`, and the primes `p` and `q`.

The function contains:

1. A `phi_n` variable which is the product of `p` minus 1 and `q` minus 1.
2. The `d` private key variable which is computed from finding the multiplicative inverse of `e` modulo `phi_n` ^[3] (aka the Extended Euclidean Algorithm ^[4]).
3. The private key `d` is returned to the function call.

Now back to the `main()` function, after the function call to compute the private key we have another three lines of fancy text about the decryption process and the call to the function that performs the decryption routine:

```

print("\n" + "="*50)
print("DECRYPTION PROCESS")
print("="*50)

```

```
result = decrypt_secret(secret, d, n)
```

decrypt_secret() function

Just like the previous one, the `decrypt_secret()` function must also be placed in the `roca.py` file **just before** the `main()` function:

```
def decrypt_secret(secret, d, n):
```

The `decrypt_secret()` function takes three arguments, the `secret` to be decrypted, the private key `d` and the public key `n`.

Next up we have the decryption logic that goes inside the `decrypt_secret()` function, we'll divide this into five steps to make explanation easier:

Step 1: Convert hex string back to bytes

What the first step does is the opposite of the last part from the encryption process, that is, converting the secret from hex format back to bytes while doing some type checking, conversion logic and error checks.

```
print("Step 1: Converting hex to bytes...")

try:
    # Remove the b' prefix and ' suffix if present, then convert
    if isinstance(secret, bytes):
        ciphertext_hex = secret.decode('utf-8')
    else:
        ciphertext_hex = secret

    ciphertext_bytes = binascii.unhexlify(ciphertext_hex)
    print(f"Ciphertext bytes length: {len(ciphertext_bytes)}\n")

except Exception as e:
    print(f"Error converting hex: {e}\n")
    return
```

Step 2: Convert bytes to large integer

This step involves converting the bytes that were outputted in the first step to a large integer.

```
print("Step 2: Converting bytes to integer...")
ciphertext_int = bytes_to_long(ciphertext_bytes)
print(f"Ciphertext as integer: {ciphertext_int}\n")
```

Step 3: RSA decryption

The third step performs the core RSA decryption routine, taking the `secret` raising it to the power of the private key `d` modulo `n`:

$$secret^d \bmod n$$

```
print("Step 3: Performing RSA decryption...")
decrypted_int = pow(ciphertext_int, d, n)
print(f"Decrypted integer: {decrypted_int}\n")
```

Step 4: Convert integer back to bytes

In the fourth step the decrypted integer is converted to bytes while checking for errors in the type conversion.

```
print("Step 4: Converting integer to bytes...")

try:
    decrypted_bytes = long_to_bytes(decrypted_int)
    print(f"Decrypted bytes: {decrypted_bytes}\n")

except Exception as e:
    print(f"Error converting to bytes: {e}\n")
    return
```

Step 5: Convert bytes to readable text

This last step involves converting the bytes from the previous one into readable text in the UTF-8 character encoding, if the secret can't be decoded into UTF-8 format it is shown as raw bytes as well as in hex format. Here we return the secret/flag to the function call in the `main()` function.

```
print("Step 5: Converting bytes to text...")

try:
    flag = decrypted_bytes.decode('utf-8')
    print(f"DECRYPTED FLAG: {flag}\n")
```



```

        return flag

    except UnicodeDecodeError:
        print("Could not decode as UTF-8, trying as raw bytes:")
        print(f"Raw bytes: {decrypted_bytes}")
        print(f"Hex representation: {decrypted_bytes.hex()}")

    return decrypted_bytes

```

Going back to the `main()` function, this last `if else` conditional checks whether the result of the decryption process returns a non-empty strings, non-zero numbers, populated collections, or any object that exists hence being `True` and printing the `success` message. If the routine returns `None`, an empty string `""`, an empty list `[]`, the number `0`, or explicitly `False` it will evaluate to `False` and prints the `error` message.

```

if result:
    print(f"\nSUCCESS! The decrypted message is: \n{result}")
else:
    print("\nERROR: Decryption failed or returned an empty result")

```

Results

Upon running the program we get the decrypted flag:

```
(venv) $ python roca.py
```

```

=====
COMPUTING PRIVATE KEY:
=====

Private Key is:
621481258039317421665725566283288089827390518628743174966911956298710916948548
154545116063634044871221939056383449529403124981022151995735949983128412161

=====
DECRYPTION PROCESS
=====

Step 1: Converting hex to bytes...
Ciphertext bytes length: 64

Step 2: Converting bytes to integer...

```

Ciphertext as integer:

```
449536315090551532263066375725085002372961138187388428648241729700911538571893
6978514830901766921373674371205369233485171530686850820730628631583773470666
```

Step 3: Performing RSA decryption...

Decrypted integer:

```
130400044828254138876075838730985967322046357423533150469364991879203532707721
54164171729789
```

Step 4: Converting integer to bytes...

Decrypted bytes: b'flag{coppersmith_weak_rsa_roca_attack}'

Step 5: Converting bytes to text...

DECRYPTED FLAG: flag{coppersmith_weak_rsa_roca_attack}

SUCCESS!: The decrypted message is:

flag{coppersmith_weak_rsa_roca_attack}

Conclusion

In this challenge we learned how mathematical failures can lead to vulnerabilities in cryptographic algorithms, hence being able to compromise the confidentiality of data.

References

[Python pow\(\) function](#)

[Python Simple RSA Algorithm Explanation Repo](#)

[StackOverflow RSA Algorithm Known n, how to Get p & q](#)

-
1. [Roca Vulnerability](#) ↩
 2. [RSA Cryptosystem](#) ↩ ↩
 3. [RSA Algorithm Cryptography](#) ↩
 4. [Extended Euclidean Algorithm](#) ↩