Week13- Write-up

Vishnu Vardhan Ciripuram
N14912012
vc2499

Challenge: Super Secure Letter

Objective
The challenge presents an encrypted message (ciphertext) that we need to decrypt. The encrypted message is generated from a flag using an XOR-based encryption method combined with PRNG.

Obective is to recover the plaintext (flag) from the given ciphertext by reverse engineering the encryption logic used in the binary.

Approach

I analyzed the provided binary in Binary Ninja and identified the encryption logic. Here's how the encryption works:

1. The Seed for PRNG
   The binary seeds the pseudo-random number generator (PRNG) using the following formula:

   srand(time(nullptr) * time(nullptr));

   - time(nullptr) fetches the current Unix timestamp.
   - The seed for srand() is the square of the timestamp.

```
00001398    int32_t rax_1 = time(nullptr)
000013ae    srand(x: rax_1 * rax_1)
```

The XOR Encryption
After seeding, the binary loops over the flag bytes and XORs each byte with the least significant byte (LSB) of the PRNG output generated using rand():

for (int i = 0; i < strlen(flag); i++) {

printf("%02x", rand() & 0xFF ^ flag[i]);

}

- rand(): Produces the next random number.
- rand() & 0xFF: Extracts the least significant byte.
- flag[i]: The current byte of the flag.
- XOR (^) is applied between the PRNG output and the flag byte.

```
00001456    for (int32_t i = 0; i s< rax_4; i += 1)
0000143e        printf(format: "%02x", zx.q(zx.d(rand()) ^ sx.d(*(sx.q(i) + &var_a8))))
0000143e
```

Decryption Plan

The XOR operation is symmetric, so the encryption process can be reversed:

Original Byte = Ciphertext Byte ^ PRNG Output

To recover the flag:

- Brute-force potential seeds by squaring timestamps near the current time.
- Replicate the PRNG sequence using libc.srand(seed) and libc.rand().
- XOR the PRNG output with the ciphertext bytes.

Here is the Python code I used to solve the challenge:

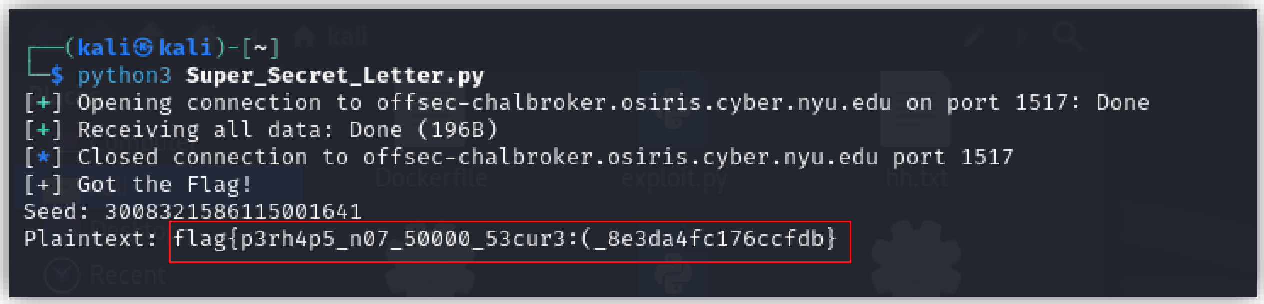
~/Super_Secret_Letter.py - Mousepad

File Edit Search View Document Help

```python
from pwn import *
import ctypes, re, time

# Load libc for C's rand() and srand()
libc = ctypes.CDLL("libc.so.6")
libc.srand.argtypes, libc.rand.restype = [ctypes.c_uint], ctypes.c_int

# XOR decrypt with seed
def decrypt(ciphertext, seed):
    libc.srand(seed)
    return bytes(c ^ (libc.rand() & 0xFF) for c in ciphertext)

# Connect to server and extract ciphertext
p = remote('offsec-chalbroker.osiris.cyber.nyu.edu', 1517)
p.sendlineafter(b'abc123): ', b'vc2499')
response = p.recvall(timeout=2).decode()
ciphertext = bytes.fromhex(re.search(r'[a-fA-F0-9]{64,}', response).group(0))
p.close()

# Brute-force squared seeds
current_time = int(time.time())
for t in range(current_time - 3600, current_time + 3600):
    try:
        plaintext = decrypt(ciphertext, t * t).decode('utf-8')
        if "flag{" in plaintext:
            print(f"[+] Got the Flag!\nSeed: {t * t}\nPlaintext: {plaintext}")
            exit()
    except UnicodeDecodeError:
        continue

print("[-] Flag not found.")

```

I ran the script and the code successfully recovered the flag:

flag{p3rh4p5_n07_50000_53cur3:(_8e3da4fc176ccfdb}



Challenge: Pseudo Rand

Objective

The challenge Pseudo Rand requires me to guess a "random" number generated by the server. The number is generated using a weak PRNG (rand()) seeded with the current time plus a fixed offset.

My objective was to predict the "random" number generated by the server based on the weak seeding mechanism and retrieve the flag

Analyzing the Binary

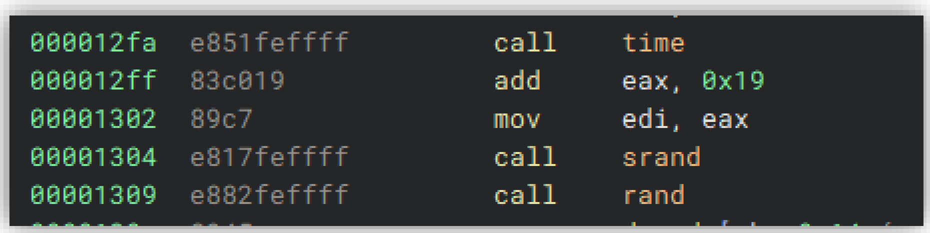
I started by opening the provided binary in Binary Ninja. I navigated to the main() function, where I noticed the following logic:

srand(time(nullptr) + 0x19);

int generated_number = rand();

This immediately stood out. Here's what I understood:

- The random number generator (rand()) is seeded using the current time (time(nullptr)) plus an offset (+0x19).
- Since rand() is deterministic, I could replicate this locally if I knew the seed value.



To predict the server's random number, I decided to use Python. I loaded the libc.so.6 library using ctypes because it contains the srand() and rand() functions, mirroring the C behavior.

Here’s the step-by-step logic I implemented:

1. Compute the seed as current time + 0x19.
2. Call srand() with this seed to initialize the PRNG.
3. Use rand() to generate the first random number, which would match the server's output.

Script I wrote:

~/Psudo_Rand.py - Mousepad

```python
from pwn import *
import ctypes

# Connect to the remote server
server = remote('offsec-chalbroker.osiris.cyber.nyu.edu', 1514)

# Step 1: Send the NetID
netid = b'vc2499'
server.sendlineafter(b'abc123): ', netid)

# Step 2: Predict the random number
libc = ctypes.CDLL("libc.so.6")    # Load the C standard library
current_seed = int(time.time()) + 0x19  # Compute the seed based on the challenge logic
libc.srand(current_seed)           # Seed the PRNG
predicted_number = libc.rand()     # Generate the predicted random number

# Step 3: Send the predicted number to the server
server.sendlineafter(b"Please wait a moment...", str(predicted_number).encode())

# Step 4: Switch to interactive mode to receive the flag
server.interactive()
```

I ran the script and it retrieved the flag:

flag{l00ks_l1k3_th4t_seed_w4snt_gr34t!_8c728b5b0604c5d0}

```
┌──(kali㉿kali)-[~]
└─$ python3 Psudo_Rand.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1514: Done
[*] Switching to interactive mode

Can you guess my number?

I guess it wasn't that random!
        Here's your flag, friend: flag{l00ks_l1k3_th4t_seed_w4snt_gr34t!_8c728b5b0604c5d0}


[*] Got EOF while reading in interactive
$ 
```

Challenge: RSA_1

Objective

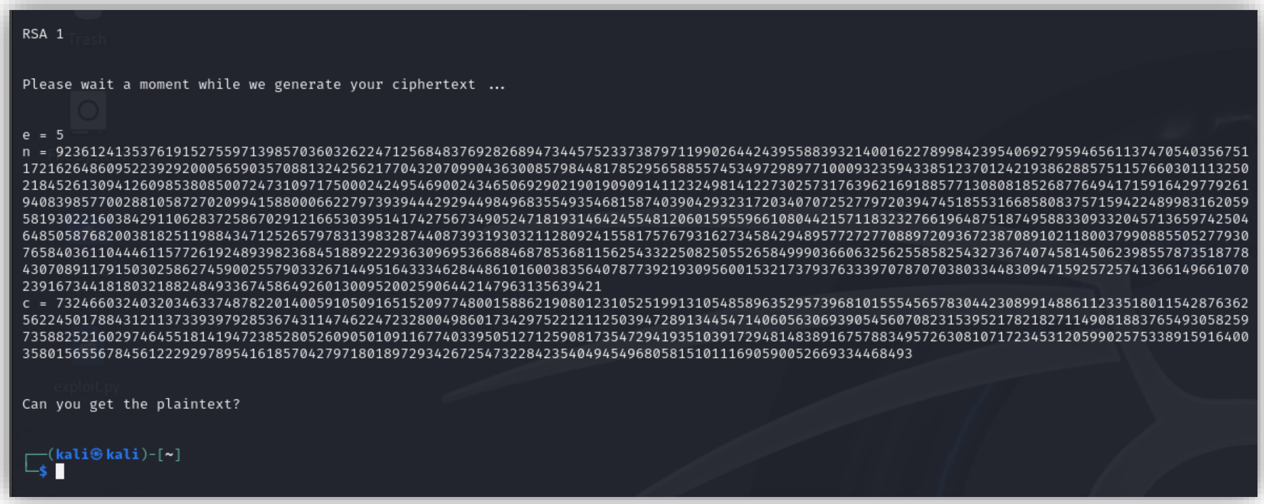
The goal of this challenge was to decrypt a ciphertext that was encrypted using RSA with a small public exponent e=5. By exploiting the weakness caused by the small exponent, I was able to recover the plaintext directly.

Solution Steps

Connecting to the Server

I connected to the server using pwntools and sent my NetID to retrieve the challenge parameters:

- e: Public exponent (fixed at 5)
- n: Modulus
- c: Ciphertext



RSA encryption is performed as:

c=m^e mod n

When e is small (in this case e=5) and m^e < n, the ciphertext c becomes simply m^e without wrapping due to the modulus n.
Thus, to recover the plaintext m, I computed the **5th root** of c.

I wrote a script using pwntools to:

1. Connect to the server and retrieve e,n,c.
2. Compute the 5th root of ccc using gmpy2.iroot.
3. Convert the result into a readable flag.

Script Used:

```
~/RSA_1.py - Mousepad
File  Edit  Search  View  Document  Help
```

```python
from pwn import remote
from gmpy2 import iroot
import binascii

# Connect to the server
host, port = 'offsec-chalbroker.osiris.cyber.nyu.edu', 1515
connection = remote(host, port)

# Send NetID
connection.sendlineafter(b'abc123): ', b'vc2499')

# Function to safely parse 'key = value' lines
def get_value():
    while True:
        line = connection.recvline().strip()
        if b'=' in line:
            return int(line.split(b'=')[1])

# Retrieve the challenge parameters
public_exponent = get_value()  # e
modulus = get_value()          # n
ciphertext = get_value()       # c

# Step 1: Compute the 5th root of the ciphertext
plaintext, is_exact = iroot(ciphertext, public_exponent)

# Step 2: Convert the result to a readable flag
if is_exact:
    flag = binascii.unhexlify(hex(plaintext)[2:]).decode()
    print("Flag:", flag)
else:
    print("Failed to compute the exact root.")

# Close the connection
connection.close()
```

I ran the script and it retrieved the flag:

flag{n0_f4ct0r1ng_r3qu1r3d!_0c53cc2c0ebd78a6}

```
┌──(kali㉿kali)-[~]
└─$ python3 RSA_1.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1515: Done
Flag: flag{n0_f4ct0r1ng_r3qu1r3d!_0c53cc2c0ebd78a6}
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1515

┌──(kali㉿kali)-[~]
└─$ 
```

Challenge: RSA_2

Objective

The goal of this challenge was to decrypt a message that was encrypted using RSA with the **same modulus n** but two different public exponents e1 and e2. Using the vulnerability in this setup, I recovered the original plaintext message.

Solution Steps

Analyzing the Problem

I connected to the server using nc (netcat) and observed the following:

- Two public exponents e1 and e2.
- A shared modulus n.
- Two ciphertexts c1 and c2, which were encrypted using the same modulus n but with different exponents.

The challenge description suggested that this setup is vulnerable to the **Common Modulus Attack** in RSA.

In RSA, if two ciphertexts c1 and c2 are encrypted with the same modulus n but different public exponents e1 and e2, we can use **Bezout's Identity** to recover the plaintext m.

Bezout's Identity states that for integers e1 and e2, there exist coefficients a and b such that:

$a \cdot e1 + b \cdot e2 = 1$

This property allows us to combine c1and c2 to recover m as follows:

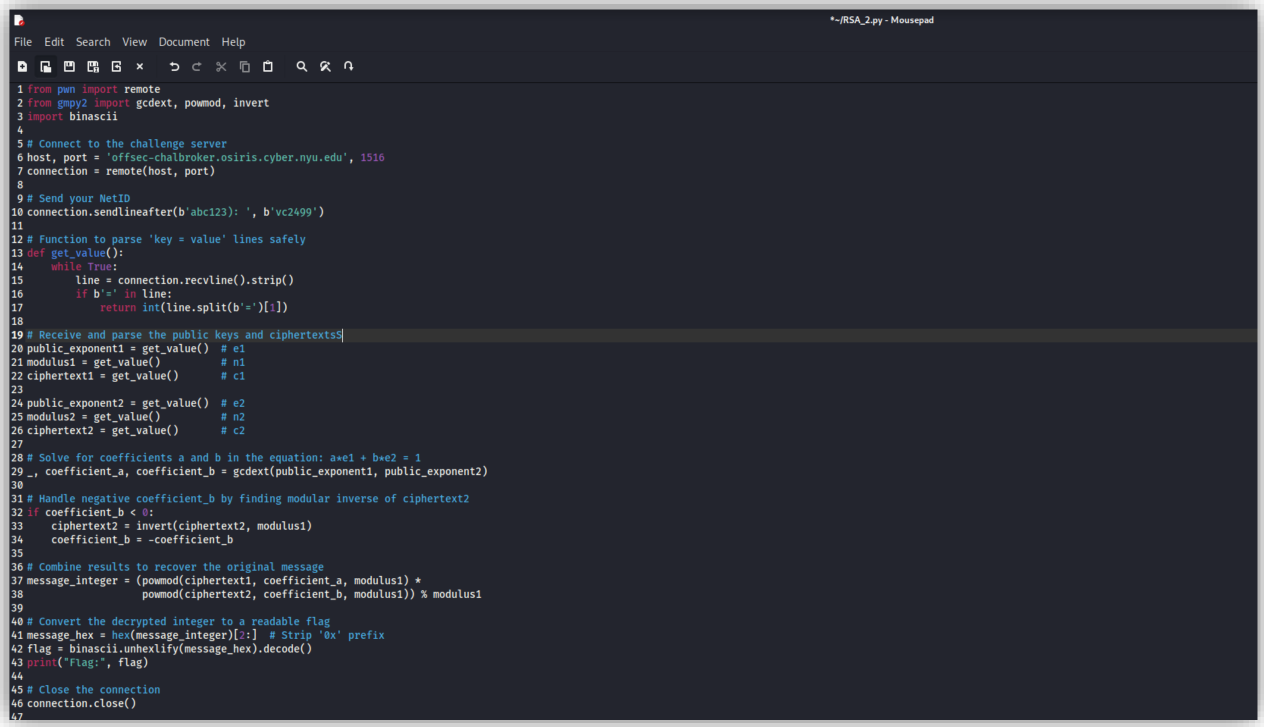
$m = (c1^a \cdot c2^b) \bmod n$

If b is negative, we use the **modular inverse** of c2

To solve the challenge, I wrote the following Python script using pwntools and gmpy2:

- which connected to the server and retrieved the values for e1,n,c1,e2,n,c2.
- Using **gmpy2.gcdext**, I calculated the coefficients a and b.
- If b was negative, I computed the modular inverse of c2 using **gmpy2.invert**
- I combined the ciphertexts c1 and c2 using the formula: m=(c1^a·c2^b) mod n
- Finally, I converted the decrypted integer m into a hexadecimal string and decoded it into ASCII to extract the flag.

Script used:

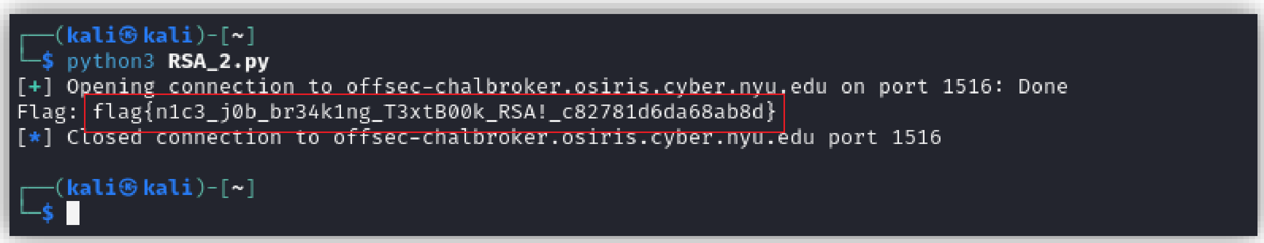


*~/RSA_2.py - Mousepad

File Edit Search View Document Help

```python
from pwn import remote
from gmpy2 import gcdext, powmod, invert
import binascii

# Connect to the challenge server
host, port = 'offsec-chalbroker.osiris.cyber.nyu.edu', 1516
connection = remote(host, port)

# Send your NetID
connection.sendlineafter(b'abc123): ', b'vc2499')

# Function to parse 'key = value' lines safely
def get_value():
    while True:
        line = connection.recvline().strip()
        if b'=' in line:
            return int(line.split(b'=')[1])

# Receive and parse the public keys and ciphertextsS
public_exponent1 = get_value()  # e1
modulus1 = get_value()          # n1
ciphertext1 = get_value()       # c1

public_exponent2 = get_value()  # e2
modulus2 = get_value()          # n2
ciphertext2 = get_value()       # c2

# Solve for coefficients a and b in the equation: a*e1 + b*e2 = 1
_, coefficient_a, coefficient_b = gcdext(public_exponent1, public_exponent2)

# Handle negative coefficient_b by finding modular inverse of ciphertext2
if coefficient_b < 0:
    ciphertext2 = invert(ciphertext2, modulus1)
    coefficient_b = -coefficient_b

# Combine results to recover the original message
message_integer = (powmod(ciphertext1, coefficient_a, modulus1) *
                   powmod(ciphertext2, coefficient_b, modulus1)) % modulus1

# Convert the decrypted integer to a readable flag
message_hex = hex(message_integer)[2:]  # Strip '0x' prefix
flag = binascii.unhexlify(message_hex).decode()
print("Flag:", flag)

# Close the connection
connection.close()
```

I ran the script and received the flag:
flag{n1c3_j0b_br34k1ng_T3xtB00k_RSA!_c82781d6da68ab8d}



```
┌──(kali㉿kali)-[~]
└─$ python3 RSA_2.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1516: Done
Flag: flag{n1c3_j0b_br34k1ng_T3xtB00k_RSA!_c82781d6da68ab8d}
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1516

┌──(kali㉿kali)-[~]
└─$
```