

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.

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

Approach

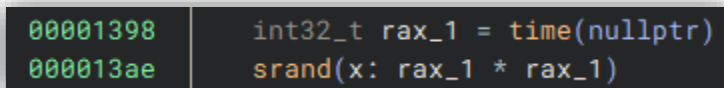
I analyzed the provided binary in Binary Ninja and identified the encryption logic. Here's how the encryption works:

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 < 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
[Icons]

1 from pwn import *
2 import ctypes, re, time
3
4 # Load libc for C's rand() and srand()
5 libc = ctypes.CDLL("libc.so.6")
6 libc.srand.argtypes, libc.rand.restype = [ctypes.c_uint], ctypes.c_int
7
8 # XOR decrypt with seed
9 def decrypt(ciphertext, seed):
10     libc.srand(seed)
11     return bytes(c ^ (libc.rand() & 0xFF) for c in ciphertext)
12
13 # Connect to server and extract ciphertext
14 p = remote('offsec-chalbroker.osiris.cyber.nyu.edu', 1517)
15 p.sendlineafter(b'abc123:', b'vc2499')
16 response = p.recvall(timeout=2).decode()
17 ciphertext = bytes.fromhex(re.search(r'[a-fA-F0-9]{64,}', response).group(0))
18 p.close()
19
20 # Brute-force squared seeds
21 current_time = int(time.time())
22 for t in range(current_time - 3600, current_time + 3600):
23     try:
24         plaintext = decrypt(ciphertext, t * t).decode('utf-8')
25         if "flag{" in plaintext:
26             print(f"[+] Got the Flag!\nSeed: {t * t}\nPlaintext: {plaintext}")
27             exit()
28     except UnicodeDecodeError:
29         continue
30
31 print("[~] Flag not found.")
32

```

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

flag{p3rh4p5_n07_50000_53cur3:(_8e3da4fc176ccfdb}

```
(kali㉿kali)-[~]  
$ python3 Super_Secret_Letter.py  
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1517: Done  
[+] Receiving all data: Done (196B)  
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1517  
[+] Got the Flag!  
Seed: 3008321586115001641  
Plaintext: 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.

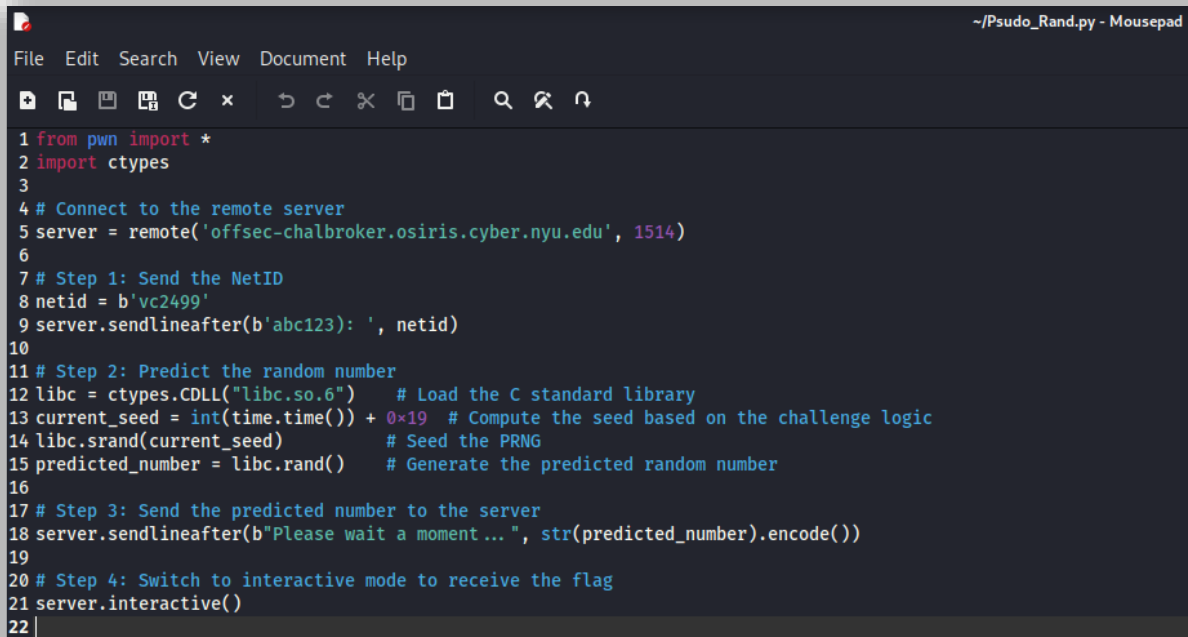
```
000012fa  e851feffff    call    time  
000012ff  83c019       add     eax, 0x19  
00001302  89c7        mov     edi, eax  
00001304  e817feffff    call    srand  
00001309  e882feffff    call    rand
```

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.

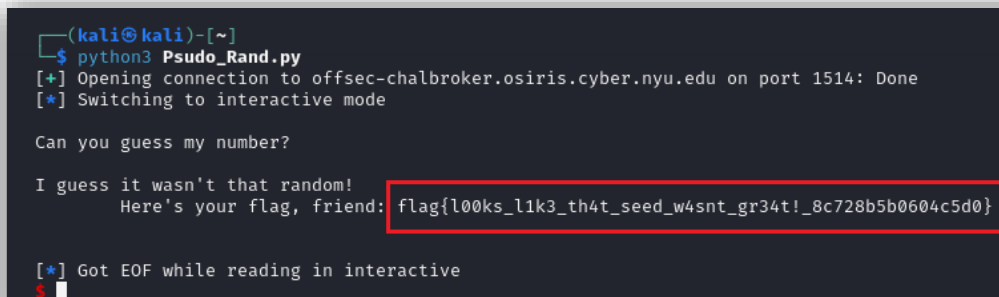
Script I wrote:



```
~/Pseudo_Rand.py - Mousepad
File Edit Search View Document Help
1 from pwn import *
2 import ctypes
3
4 # Connect to the remote server
5 server = remote('offsec-chalbroker.osiris.cyber.nyu.edu', 1514)
6
7 # Step 1: Send the NetID
8 netid = b'vc2499'
9 server.sendlineafter(b'abc123'): ', netid)
10
11 # Step 2: Predict the random number
12 libc = ctypes.CDLL("libc.so.6") # Load the C standard library
13 current_seed = int(time.time()) + 0x19 # Compute the seed based on the challenge logic
14 libc.srand(current_seed) # Seed the PRNG
15 predicted_number = libc.rand() # Generate the predicted random number
16
17 # Step 3: Send the predicted number to the server
18 server.sendlineafter(b'Please wait a moment ... ', str(predicted_number).encode())
19
20 # Step 4: Switch to interactive mode to receive the flag
21 server.interactive()
22 |
```

I ran the script and it retrieved the flag:

flag{l00ks_l1k3_th4t_seed_w4snt_gr34t!_8c728b5b0604c5d0}



```
(kali@kali)-[~]
$ python3 Pseudo_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 1

Please wait a moment while we generate your ciphertext ...

e = 5
n = 9236124135376191527559713985703603262247125684837692826894734457523373879711990264424395588393214001622789984239540692795946561137470540356751
17216264860952239292000565903570881324256217704320709904263008579844817852956588557453497298977100092235943385123701242193862885751157660301113250
21845261309412609853808500724731097175000242495469002434650602902190190090141123249814122730257317639621691885771308081852687764941715916429770261
94083985770028810587270209941588000662279739394442929449849683554935468158740390429323172034070725277972039474518553166858083757159422489983162059
58193022160384291106283725867029121665303951417427567349052471819314642455481206015955966108044215711832327661964875187495883309332045713659742504
64850587682003818251198843471252657978313983287440873931930321128092415581757679316273458429489577272770889720936723870891021180037990885505277930
76584036110444611577261924893982368451889222936309695366884687853681156254332250825055265849990366063256255858254327367407458145062398557873518778
43070891179150302586274590025579033267144951643334628448610160038356407877392193095600153217379376333970787070380334483094715925725741366149661070
239167344181803218824849336745864926013009520025906442147963135639421
c = 7324660324032034633748782201400591050916515209774800158862190801231052519913105485896352957396810155545657830442308991488611233518011542876362
56224501788431211373393979285367431147462247232800498601734297522121125039472891344547140605630693905456070823153952178218271149081883765493058259
73588252160297464551814194723852805260905010911677403395051271259081735472941935103917294814838916757883495726308107172345312059902575338915916400
3580156556784561222929789541618570427971801897293426725473228423540494549680581510111690590052669334468493

Can you get the plaintext?

(kali@kali)-[~]
$
```

RSA encryption is performed as:

$$c = m^e \bmod n$$

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 c using `gmpy2.iroot`.
3. Convert the result into a readable flag.

Script Used:

```
~/RSA_1.py - Mousepad
File Edit Search View Document Help
1 from pwn import remote
2 from gmpy2 import iroot
3 import binascii
4
5 # Connect to the server
6 host, port = 'offsec-chalbroker.osiris.cyber.nyu.edu', 1515
7 connection = remote(host, port)
8
9 # Send NetID
10 connection.sendlineafter(b'abc123): ', b'vc2499')
11
12 # Function to safely parse 'key = value' lines
13 def get_value():
14     while True:
15         line = connection.recvline().strip()
16         if b'=' in line:
17             return int(line.split(b'=')[1])
18
19 # Retrieve the challenge parameters
20 public_exponent = get_value() # e
21 modulus = get_value() # n
22 ciphertext = get_value() # c
23
24 # Step 1: Compute the 5th root of the ciphertext
25 plaintext, is_exact = iroot(ciphertext, public_exponent)
26
27 # Step 2: Convert the result to a readable flag
28 if is_exact:
29     flag = binascii.unhexlify(hex(plaintext)[2:]).decode()
30     print("Flag:", flag)
31 else:
32     print("Failed to compute the exact root.")
33
34 # Close the connection
35 connection.close()
36
```

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 e_1 and e_2 . 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 e_1 and e_2 .
- A shared modulus n .
- Two ciphertexts c_1 and c_2 , which were encrypted using the same modulus n but with different exponents.

```

RSA 2
Please wait a moment while we generate your ciphertexts ...

e1 = 257
n1 = 32369843653904802644564238660846104715809572012699808973770426695404981480078154383886694236292800680479171472853434289556188029411702004010989111029344545133943483793024396833282947953932193829498748877720
712795287392433429075675088818977113491271210764044709523267828968781761816577891037308944664868425466441728415284238144099811401273476011880136293934137647312844248042208862009520381064171521220290689794172
4080625148849187312655183703862229474126816676252881038798866118327983058473212490633662207895268557251661371706738105316467755854499620495805544897924423229263980404525068899642981470364485840485916013274574
198561159780894868513875658149117651964576294398381090879193935986252831583269511973601537187835366185393867816870594834228431941959332297219726195944080285986858366164598274789111524391417922586786623797
128244109771525727898851140074592154028097415462081795571012343419643227676088782391261793671528653569136166982718161871743221298001341143207728035536474606786118131497152580709168392919215171589174440865776
937821503801498328178118098283824967966121815235670676460264203114698762738470387027629728462666104199245353715265142679108340284206492777656527570637239665640483613610290971987
c1 = 2092825347991890132746354091874400986524981024320256701299103219359011853033649754028871940401810491685907659288495421988088325857086625221113862911464863756759499546842372309465260415322303445
8547723381167469305195511868580242644039131292626638258108507663178021406277103988014201501449446072166238799842322870761512939302042686732819268556923348526799038101564755773102390230158679243418159586133898946
904122258171168022421198510953283608356181482116208819299866068367365831781628588959348362205990670546319251405531290571085149621639145587532973457763971932732486369349795743841835548302664202957459255929461
88118671998890747868888830837475705815390838865170905230244840774075384062611335884129572447126991245089762795814513690472955208688678673694952565557299557456320481947743936696288798017340603625827492631860
869153784289227212121616902231457586975145749221022581171430604209935680968064073189220304702892221587686011640447811326165944346459261204586996872422543667619793916562362027444129844246253308
733391615202835398859745442255650113851267873875687745603912614169400281698285308949704457207383804262828619433261112762441490961321756719567448796883482502738220290817626978

e2 = 65537
n2 = 32369843653904802644564238660846104715809572012699808973770426695404981480078154383886694236292800680479171472853434289556188029411702004010989111029344545133943483793024396833282947953932193829498748877720
712795287392433429075675088818977113491271210764044709523267828968781761816577891037308944664868425466441728415284238144099811401273476011880136293934137647312844248042208862009520381064171521220290689794172
4080625148849187312655183703862229474126816676252881038798866118327983058473212490633662207895268557251661371706738105316467755854499620495805544897924423229263988404525068899642981470364485840485916013274574
198561159780894868513875658149117651964576294398381090879193935986252831583269511973601537187835366185393867816870594834228431941959332297219726195944080285986858366164598274789111524391417922586786623797
128244109771525727898851140074592154028097415462081795571012343419643227676088782391261793671528653569136166982718161871743221298001341143207728035536474606786118131497152580709168392919215171589174440865776
937821503801498328178118098283824967966121815235670676460264203114698762738470387027629728462666104199245353715265142679108340284206492777656527570637239665640483613610290971987
c2 = 231728599657289816583673402559711783718243853588323294759482219758800022923381689108152783345776574344325173029265863720687168029718088252934347018545209609972247103422335219032176794137175384195118094
5845558192949916674435022820268526633864360297139820855102220357666722039132422065498018140657719132598385865913188649961959002320450776774823224192676383012139133008781460294048930293180755738
083960201164597964740076204369222960149155732326127014964900219487797630142330774789086592709025000863497889062688224616711341409613050683357456370469067486052150965990208037665182896512886259734798120062
8590894395114376211034185997659284677225439064928844801913762776878162895327283091677857986304582138907700454198169597710918315272794700091120262768783924412349283207304859869430271137816093762802412519863279
34011499562973374382083067005615698876442011838080334197324202689659347001095929659120228484292548232580118361958451567671139016013544050234790835654538830866571131877547716614075555275982418082424678410447
242437171110284295431441872921183802259849087294986815400242630824582187135869787874218171293781609799045205212694134199299894769979720024666749319282195366821512603356166759

Can you still decrypt the messages?
kali@kali:~$
```

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

In RSA, if two ciphertexts c_1 and c_2 are encrypted with the same modulus n but different public exponents e_1 and e_2 , we can use **Bezout's Identity** to recover the plaintext m .

Bezout's Identity states that for integers e_1 and e_2 , there exist coefficients a and b such that:

$$a \cdot e_1 + b \cdot e_2 = 1$$

This property allows us to combine c_1 and c_2 to recover m as follows:

$$m = (c_1^a \cdot c_2^b) \bmod n$$

If b is negative, we use the **modular inverse** of c_2

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

- which connected to the server and retrieved the values for e_1, n, c_1, e_2, n, c_2 .
- Using **gmpy2.gcdext**, I calculated the coefficients a and b .
- If b was negative, I computed the modular inverse of c_2 using **gmpy2.invert**
- I combined the ciphertexts c_1 and c_2 using the formula: $m = (c_1^a \cdot c_2^b) \bmod n$
- Finally, I converted the decrypted integer m into a hexadecimal string and decoded it into ASCII to extract the flag.

Script used:

```
File Edit Search View Document Help
*~/RSA_2.py - Mousepad

1 from pwn import remote
2 from gmpy2 import gcdext, powmod, invert
3 import binascii
4
5 # Connect to the challenge server
6 host, port = 'offsec-chalbroker.osiris.cyber.nyu.edu', 1516
7 connection = remote(host, port)
8
9 # Send your NetID
10 connection.sendlineafter(b'abc123'): ', b'vc2499')
11
12 # Function to parse 'key = value' lines safely
13 def get_value():
14     while True:
15         line = connection.recvline().strip()
16         if b'=' in line:
17             return int(line.split(b'=')[1])
18
19 # Receive and parse the public keys and ciphertexts
20 public_exponent1 = get_value() # e1
21 modulus1 = get_value() # n1
22 ciphertext1 = get_value() # c1
23
24 public_exponent2 = get_value() # e2
25 modulus2 = get_value() # n2
26 ciphertext2 = get_value() # c2
27
28 # Solve for coefficients a and b in the equation: a*e1 + b*e2 = 1
29 _, coefficient_a, coefficient_b = gcdext(public_exponent1, public_exponent2)
30
31 # Handle negative coefficient_b by finding modular inverse of ciphertext2
32 if coefficient_b < 0:
33     ciphertext2 = invert(ciphertext2, modulus1)
34     coefficient_b = -coefficient_b
35
36 # Combine results to recover the original message
37 message_integer = (powmod(ciphertext1, coefficient_a, modulus1) *
38                    powmod(ciphertext2, coefficient_b, modulus1)) % modulus1
39
40 # Convert the decrypted integer to a readable flag
41 message_hex = hex(message_integer)[2:] # Strip '0x' prefix
42 flag = binascii.unhexlify(message_hex).decode()
43 print("Flag:", flag)
44
45 # Close the connection
46 connection.close()
47
```

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)-[~]
$
```