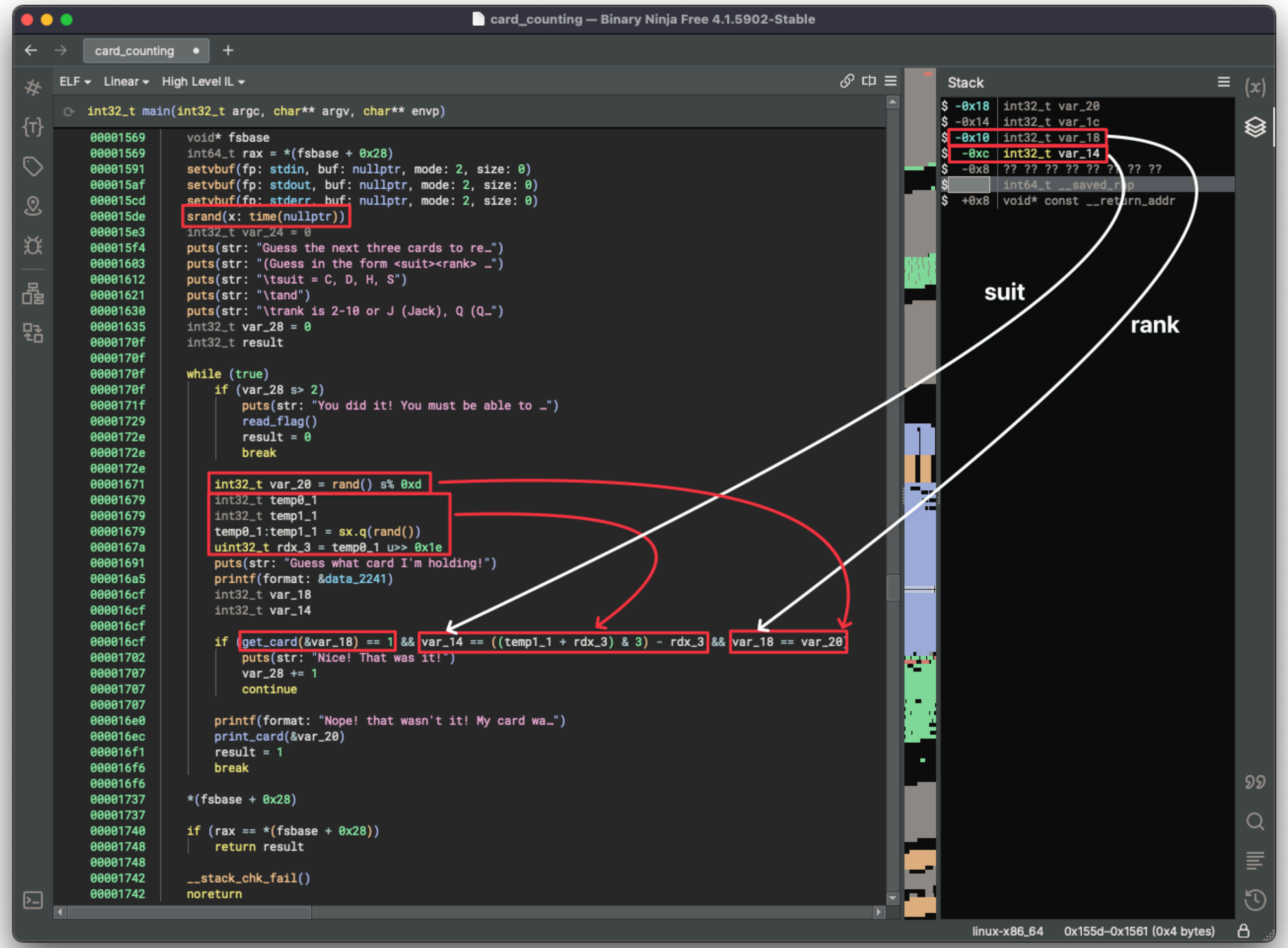
Week13-Crypto Write-ups

Name: **Xinsheng Zhu**
UnivID: **N10273832**
NetID: **xz4344**

!!! 700/300 pts solved !!!

Card Counting (100 pts)

In this challenge, with nothing provided, we need to predict the future cards. We directly open the binary file `card_counting` with Binary Ninja to inspect the High-Level IL.


card_counting — Binary Ninja Free 4.1.5902-Stable
card_counting
ELF
Linear
High Level IL
Stack
suit
rank
linux-x86_64
0x155d-0x1561 (0x4 bytes)

```
int32_t main(int32_t argc, char** argv, char** envp)

00001569    void* fsbase
00001569    int64_t rax = *(fsbase + 0x28)
00001591    setvbuf(fp: stdin, buf: nullptr, mode: 2, size: 0)
000015af    setvbuf(fp: stdout, buf: nullptr, mode: 2, size: 0)
000015cd    setvbuf(fp: stderr, buf: nullptr, mode: 2, size: 0)
000015de    srand(x: time(nullptr))
000015e3    int32_t var_24 = 0
000015f4    puts(str: "Guess the next three cards to re…")
00001603    puts(str: "(Guess in the form <suit><rank> …")
00001612    puts(str: "\tsuit = C, D, H, S")
00001621    puts(str: "\tand")
00001630    puts(str: "\trank is 2-10 or J (Jack), Q (Q…")
00001635    int32_t var_28 = 0
0000170f    int32_t result
0000170f
0000170f    while (true)
0000170f        if (var_28 s> 2)
0000171f            puts(str: "You did it! You must be able to …")
00001729            read_flag()
0000172e            result = 0
0000172e            break
0000172e
00001671        int32_t var_20 = rand() s% 0xd
00001679        int32_t temp0_1
00001679        int32_t temp1_1
00001679        temp0_1:temp1_1 = sx.q(rand())
0000167a        uint32_t rdx_3 = temp0_1 u>> 0x1e
00001691        puts(str: "Guess what card I'm holding!")
000016a5        printf(format: &data_2241)
000016cf        int32_t var_18
000016cf        int32_t var_14
000016cf
000016cf        if (get_card(&var_18) == 1 && var_14 == ((temp1_1 + rdx_3) & 3) - rdx_3 && var_18 == var_20)
00001702            puts(str: "Nice! That was it!")
00001707            var_28 += 1
00001707            continue
00001707
000016e0        printf(format: "Nope! that wasn't it! My card wa…")
000016ec        print_card(&var_20)
000016f1        result = 1
000016f6        break
000016f6
00001737    *(fsbase + 0x28)
00001737
00001740    if (rax == *(fsbase + 0x28))
00001748        return result
00001748
00001742    __stack_chk_fail()
00001742    noreturn
```

```
$ -0x18 | int32_t var_20
$ -0x14 | int32_t var_1c
$ -0x10 | int32_t var_18
$  -0xc | int32_t var_14
$  -0x8 | ?? ?? ?? ?? ?? ?? ?? ??
$       | int64_t __saved_rbp
$  +0x8 | void* const __return_addr
```

The `get_card` function (screenshot omitted) simply maps different card input characters to specific numeric values. It reads up to 5 bytes (clearing any trailing newline character) and complies with the following parsing rules:

- For parsing the rank of the card:

Card Rank	2	3	4	5	6	7	8	9	10	J	Q	K	A
Internal Value	0	1	2	3	4	5	6	7	8	9	10	11	12

- For paring the suit of the card:

Card Suit	C(lubs)	D(iamonds)	H(earts)	S(pades)
Interval Value	0	1	2	3

If both the suit and rank are valid, it populates the argument array with the suit and rank, returning 1.

The `main` function randomly selects a card, then prompts the user to guess the card:

1. It seeds a random number generator `srand()` with the current time.
2. It specifies the game rules and that the format of the input for a guessed card should be `<suit><rank>`.
3. There are three rounds in total to guess three cards. For each round:
    i. It generates a random card, whose rank is firstly determined by `rand() % 13` (range 0~12) and whose suit is secondly determined by `rand() & 3` (range 0~3).
    ii. It calls `get_card` to parse and validate the user's input.
    iii. It compares the user's guess against the generated card. If the guess is correct, increment a counter; otherwise, the game ends with a failure message.
4. If the user correctly guesses three cards in a row, the program calls `read_flag` to display the flag.

Thus, we can use the Python ctypes utility to wrap C binaries to mimic what the C program does, generating the same random numbers to form cards to retrieve the flag.

```python
import ctypes
......
def get_card(suit, rank):
    card = ""
    if suit == 0:
        card += "C"
    elif suit == 1:
        card += "D"
    elif suit == 2:
        card += "H"
    elif suit == 3:
        card += "S"
    if rank == 9:
        card += "J"
    elif rank == 10:
        card += "Q"
    elif rank == 11:
        card += "K"
    elif rank == 12:
        card += "A"
    else:
        card += str(rank + 2)
    return card

print(p.recvuntil(b"!\n").decode())

libc = ctypes.CDLL("libc.so.6")
libc.time.argtypes = [ctypes.POINTER(ctypes.c_long)]
libc.srand.argtypes = [ctypes.c_uint]
current_time = ctypes.c_long()
libc.time(ctypes.byref(current_time))
libc.srand(ctypes.c_uint(current_time.value))
libc.rand.restype = ctypes.c_int

for _ in range(3):
    print(p.recvuntil(b"> ").decode())
    rank = libc.rand() % 13
    suit = libc.rand() & 3
    card = get_card(suit, rank)
    p.sendline(card.encode())
    log.info(f"Sending guessed card: {card}")

p.interactive()
```

The captured flag is `flag{U_mus7_H4V3_a_cryst41_B411!!_0f5ce88c4f714520}`.

It should be noted that due to the network latency when connecting to the server, in order to retrieve the flag, under rare cases, the script should be run multiple times, or the seed for the random number generator should be plus 1.

Cool Block Challenge (100 pts)

With IV and Ciphertext provided by the server and PKCS#7 padding applied to AES-CBC, this challenge is apparently a Padding Oracle Attack. If the sent payload passes the padding validation, the server responds with a ":)".

The core idea for brute-forcing each byte of each block of the plaintext from the ciphertext is `c_n ^ s ^ X_(n-1) = c_n ^ s ^ c_(n-1) ^ g_m_n ^ pad = m_n ^ g_m_n ^ pad = pad`. `X_(n-1)` is what we send to the server. The padding validation succeeds if the guess byte `g_m_n` is the same as the actual byte `m_n`. In this way, we can retrieve the whole plaintext block by block (byte by byte for each block).

The Python script is omitted here because the specific implementation is relatively complex. A more detailed explanation is in the code comments.

The captured flag is `flag{F1gh71ng_g14n75_w1th__p4dding_4774ck5!_1580a3e1f12316dc}`.

Linear (200 pts)

```
❯ nc offsec-chalbroker.osiris.cyber.nyu.edu 1513
Please input your NetID (something like abc123): xz4344
hello, xz4344. Please wait a moment...
Here's a happy little Galois field:
🌱🦌🐑🌳🌳💧🐖🐴💧🐈
🪨💧🟦🐖🟩🐕🦌💧🐕🦃
🌿🟦🐴🦌🐴🪨💧🪨🪨🐑
🌳🟦🐖🐈🦃🦌🐖🌳🌳🟩
🐖🌿🪨🐑🐕💧🐈🌳🪨🟫
🐈🦃🪨🐑🌳🌿💧🐕🟩🐴
🐖🌿🐖🟩🟩🐖🐑🐖🟩🌱
🌱🦃🟦🦌🟩🪨🐕🟩🐈🐑
🐕🪨🟫🟦🐖🌱🟫🐈💧🌿
🐕🌳🟫🟩🐖🟫🌱🐄🪨🐕
Here's the encrypted flag:
440a0108f88a2b316cc4d24a89cf40bba5cf601a6bba62f8a7a4443dce0cc3ddc585908ab5cc32c11751d2d906ec6bd329
```

This challenge is a simulation of LFSR. According to the provided script, we can see that:

1. The implemented Fibonacci Linear Feedback Shift Register (LFSR) maintains a 32-bit register with taps on bits 24, 26, 28, 29, 30, and 31, which are the indices of bits that are XORed together to produce the next bit in the sequence. Several LFSR Methods are also implemented, including `get_int`, `get_byte`, and so on. However, the initialized seed for the LSFR is unknown.
2. The server first generates a 10x10 emoji-filled "Galois field" matrix through the LSFR. With two dictionaries of emojis (`animals = ['🐄', '🐖', '🦌', '🐑', '🦃', '🐕', '🐈', '🐴']`, `fields = ['🌱', '🌳', '🌿', '🪨', '🟦', '🟫', '💧', '🟩']`), it's obvious that four bits represent an emoji: the first generated bit from the LSFR determines the type of the emoji (1 for animal and 0 for field); the next three bits from the LSFR determine the index of the emoji in the corresponding dictionary.
3. Then with a state of the LSFR after generating the 10x10 emoji matrix, the server encryptes the flag by XORing each byte of the flag with a pseudo-random byte generated by the LSFR.

Thus, in order to decrypt the flag, we need to:

1. Receive the emoji matrix from the server and parse it reversely to derive the seed for the custom LSFR.
2. Synchronize the custom LSFR with the server.
3. Receive the encrypted flag from the server and decrypt the flag via XOR.

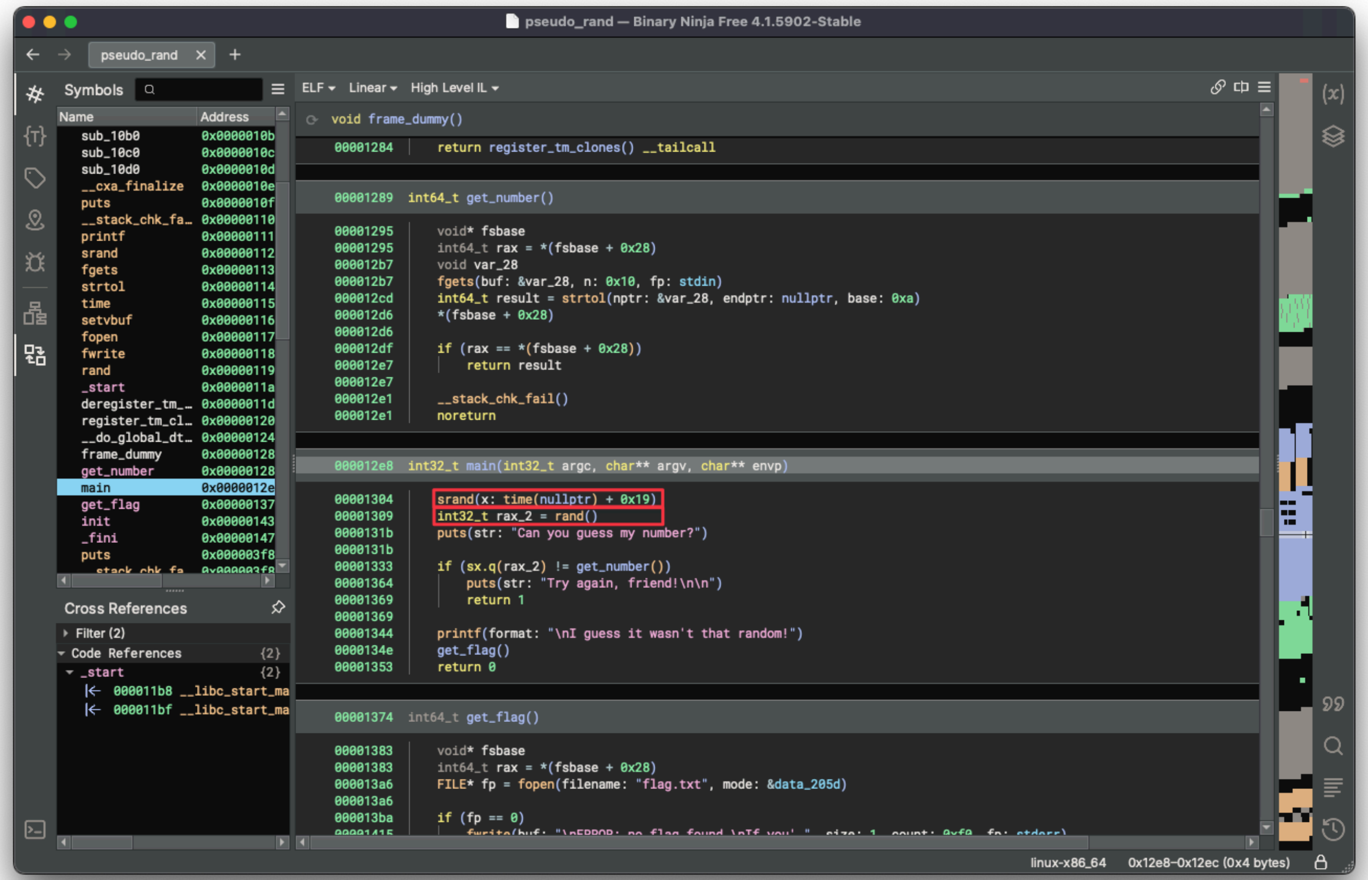
Part of the Python script to solve the challenge is shown below.

```python
from functools import reduce
from operator import xor
......
# Receive the Galois field
print(p.recvuntil(b"Here's a happy little Galois field:\n").decode())
galois = ""
for _ in range(10):
    line = p.recvline().decode().strip()
    for i in range(0, 10):
        galois += line[i]
log.info(f"Receiving Galios field: {galois}")

# Derive the seed
seed = ""
for i in range(8):
    if galois[i] in animals:
        seed += '1'
        seed += format(animals.index(galois[i]), '03b')
    elif galois[i] in fields:
        seed += '0'
        seed += format(fields.index(galois[i]), '03b')
seed = int(seed[::-1], 2)
log.info(f"Calculating seed: {seed}")

# Synchronize the LFSR
lfsr = LFSR(seed)
for _ in range(100):
    lfsr.get_int(1)
    lfsr.get_int(3)

# Receive the encrypted flag
print(p.recvuntil(b"Here's the encrypted flag:\n").decode())
encrypted_flag = p.recvline().decode().strip()
log.info(f"Receiving encrypted flag: {encrypted_flag}")

# Decrypt the flag
encrypted_flag_chars = [int(encrypted_flag[i:i+2], 16) for i in range(0, len(encrypted_flag), 2)]
decrypted_flag = ""
for encrypted_flag_char in encrypted_flag_chars:
    decrypted_flag += chr(encrypted_flag_char ^ lfsr.get_byte())
log.info(f"Decrypting flag: {decrypted_flag}")

p.interactive()
```

The captured flag is `flag{v3ry_ps3ud0_not_so_r4nd0m!_1b853fd035c1f885}` .

Pseudo Rand (50 pts)

In this challenge, with nothing provided, we need to guess a number. We directly open the binary file `pseudo_rand` with Binary Ninja to inspect the High-Level IL.



Name	Address
sub_10b0	0x000010b
sub_10c0	0x000010c
sub_10d0	0x000010d
__cxa_finalize	0x000010e
puts	0x000010f
__stack_chk_fa…	0x0000110
printf	0x0000111
srand	0x0000112
fgets	0x0000113
strtol	0x0000114
time	0x0000115
setvbuf	0x0000116
fopen	0x0000117
fwrite	0x0000118
rand	0x0000119
_start	0x000011a
deregister_tm_…	0x000011d
register_tm_cl…	0x0000120
__do_global_dt…	0x0000124
frame_dummy	0x0000128
get_number	0x0000128
main	0x000012e
get_flag	0x0000137
init	0x0000143
_fini	0x0000147
puts	0x00003f8

```
Code References                 {2}
  _start                        {2}
    |← 000011b8 __libc_start_ma
    |← 000011bf __libc_start_ma
```

```
void frame_dummy()
00001284      return register_tm_clones() __tailcall

00001289  int64_t get_number()

00001295      void* fsbase
00001295      int64_t rax = *(fsbase + 0x28)
000012b7      void var_28
000012b7      fgets(buf: &var_28, n: 0x10, fp: stdin)
000012cd      int64_t result = strtol(nptr: &var_28, endptr: nullptr, base: 0xa)
000012d6      *(fsbase + 0x28)
000012d6
000012df      if (rax == *(fsbase + 0x28))
000012e7          return result
000012e7
000012e1      __stack_chk_fail()
000012e1      noreturn

000012e8  int32_t main(int32_t argc, char** argv, char** envp)

00001304      srand(x: time(nullptr) + 0x19)
00001309      int32_t rax_2 = rand()
0000131b      puts(str: "Can you guess my number?")
0000131b
00001333      if (sx.q(rax_2) != get_number())
00001364          puts(str: "Try again, friend!\n\n")
00001369          return 1
00001369
00001344      printf(format: "\nI guess it wasn't that random!")
0000134e      get_flag()
00001353      return 0

00001374  int64_t get_flag()

00001383      void* fsbase
00001383      int64_t rax = *(fsbase + 0x28)
000013a6      FILE* fp = fopen(filename: "flag.txt", mode: &data_205d)
000013a6
000013ba      if (fp == 0)
00001415          fwrite(buf: "\nERROR: no flag found.\nIf you' ", size: 1, count: 0xf0, fp: stderr)
```

The `main` function relies on the predictability of `rand()` when seeded with a known time:

1. It seeds the random number generator `srand()` with the current time plus a constant value `0x19`.
2. It generates a random number with `rand()`.
3. It prompts the user to guess the number. If the user's input matches the generated random number, the program prints the flag.

Thus, we can use the Python ctypes utility to wrap C binaries to mimic what the C program does, generating the same random number to retrieve the flag.

```python
import ctypes
......
libc = ctypes.CDLL("libc.so.6")
libc.time.argtypes = [ctypes.POINTER(ctypes.c_long)]
libc.srand.argtypes = [ctypes.c_uint]
current_time = ctypes.c_long()
libc.time(ctypes.byref(current_time))
libc.srand(ctypes.c_uint(current_time.value + 0x19 + 1))
libc.rand.restype = ctypes.c_int
random_number = libc.rand()

p.sendline(str(random_number).encode())
log.info(f"Sending random number: {random_number}")

p.interactive()
```

The captured flag is `flag{l00ks_l1k3_th4t_seed_w4snt_gr34t!_42aae8e6c9c89860}`.

It should be noted that due to the network latency when connecting to the server, in order to retrieve the flag, under rare cases, the script should be run multiple times, or the seed for the random number generator should be plus 1.

RSA_1 (100 pts)

In this challenge, given the phi-co-prime number `e`, the public modulus `n`, and the ciphertext `c` in an RSA handshake, we are required to get the plaintext `m`.

Basically, we have `c = m ^ e % n`. When the plaintext `m` is short and the phi-co-prime number `e` is small, there exists `m ^ e < n`. In this case, `c = m ^ e % n = m ^ e`.

Thus, we should perform an RSA small exponent attack, computing the plaintext `m` by `m = c ^ -e`, which means finding the `e` th root of `c` to decrypt the encryption!

Part of the script using Python's gmpy2 module to solve the challenge is shown below.

```python
import gmpy2
......
print(p.recvuntil(b"?\n").decode())
result = gmpy2.iroot(c, e)
if result[1]:
    m = result[0]
    plaintext = bytes.fromhex(hex(m)[2:]).decode()
    log.info(f"Decrypting plaintext: {plaintext}")
```

The captured flag is `flag{n0_f4ct0r1ng_r3qu1r3d!_11d8753e3aa8aef6}`.

RSA_2 (100 pts)

In this challenge, given two phi-co-prime numbers `e1` and `e2`, two same public moduli `n1` and `n2`, and two ciphertexts `c1` and `c2` in two different RSA handshake with the same plaintext `m`, we are required to get this plaintext `m`.

With `n1 = n2`, it's obvious that we should perform an RSA common modules attack. According to `GCD(e1, e2) = 1`, we can calculate `a` and `b` by `a * e1 + b * e2 = 1`. According to `c1 ^ a * c2 ^ b = m ^ (a * e1 + b * e2) mod n = m mod n`, we can calculate `m` by `m = ((c1 ^ a (mod n)) * (c2 ^ b (mod n))) mod n`.

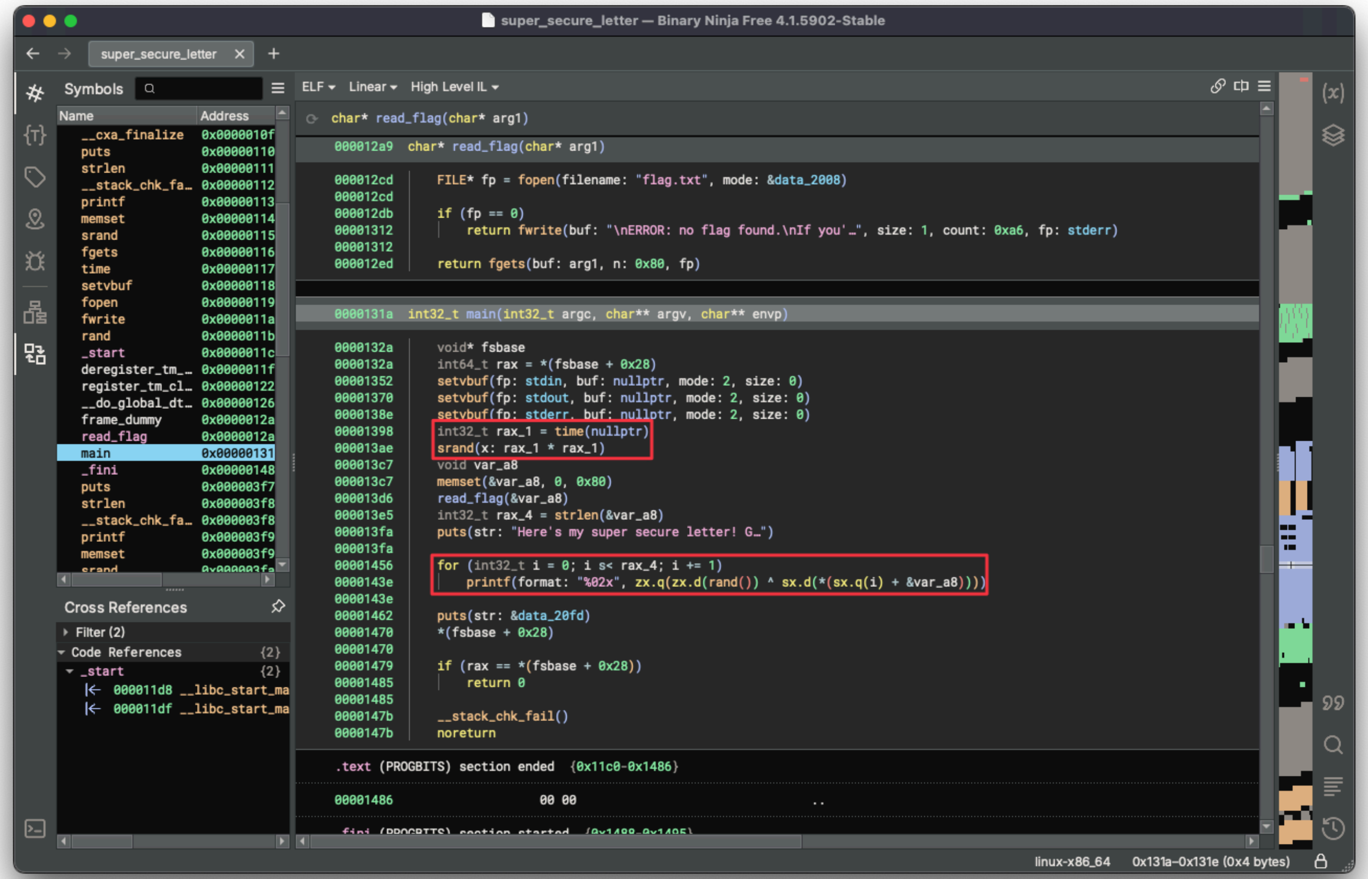
Part of the script using Python's binascii and gmpy2 modules to solve the challenge is shown below.

```python
import gmpy2
......
print(p.recvuntil(b"?\n").decode())
if n1 == n2:
    result = gmpy2.invert(e1, e2)
    a = result
    b = (1 - a * e1) // e2
    m = (gmpy2.powmod(c1, a, n1) * gmpy2.powmod(c2, b, n2)) % n1
    plaintext = binascii.unhexlify(hex(m)[2:]).decode()
    log.info(f"Decrypting plaintext: {plaintext}")
```

The captured flag is `flag{n1c3_j0b_br34k1ng_T3xtB00k_RSA!_e31a072c43777142}`.

Super Secure Letter (50 pts)

In this challenge, with nothing provided, we need to recover a message using the ciphertext letter. We directly open the binary file `super_secure_letter` with Binary Ninja to inspect the High-Level IL.



super_secure_letter — Binary Ninja Free 4.1.5902-Stable

```
Name                Address
__cxa_finalize      0x0000010f
puts                0x00000110
strlen              0x00000111
__stack_chk_fa…     0x00000112
printf              0x00000113
memset              0x00000114
srand               0x00000115
fgets               0x00000116
time                0x00000117
setvbuf             0x00000118
fopen               0x00000119
fwrite              0x0000011a
rand                0x0000011b
_start              0x0000011c
deregister_tm_…     0x0000011f
register_tm_cl…     0x00000122
__do_global_dt…     0x00000126
frame_dummy         0x0000012a
read_flag           0x0000012a
main                0x00000131
_fini               0x00000148
puts                0x000003f7
strlen              0x000003f8
__stack_chk_fa…     0x000003f8
printf              0x000003f9
memset              0x000003f9
```

```
Cross References
Filter (2)
Code References                       {2}
  _start                              {2}
    000011d8 __libc_start_ma
    000011df __libc_start_ma
```

```
char* read_flag(char* arg1)

000012a9  char* read_flag(char* arg1)

000012cd      FILE* fp = fopen(filename: "flag.txt", mode: &data_2008)
000012cd
000012db      if (fp == 0)
00001312          return fwrite(buf: "\nERROR: no flag found.\nIf you'…", size: 1, count: 0xa6, fp: stderr)
00001312
000012ed      return fgets(buf: arg1, n: 0x80, fp)

0000131a  int32_t main(int32_t argc, char** argv, char** envp)

0000132a      void* fsbase
0000132a      int64_t rax = *(fsbase + 0x28)
00001352      setvbuf(fp: stdin, buf: nullptr, mode: 2, size: 0)
00001370      setvbuf(fp: stdout, buf: nullptr, mode: 2, size: 0)
0000138e      setvbuf(fp: stderr, buf: nullptr, mode: 2, size: 0)
00001398      int32_t rax_1 = time(nullptr)
000013ae      srand(x: rax_1 * rax_1)
000013c7      void var_a8
000013c7      memset(&var_a8, 0, 0x80)
000013d6      read_flag(&var_a8)
000013e5      int32_t rax_4 = strlen(&var_a8)
000013fa      puts(str: "Here's my super secure letter! G…")
000013fa
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
00001462      puts(str: &data_20fd)
00001470      *(fsbase + 0x28)
00001470
00001479      if (rax == *(fsbase + 0x28))
00001485          return 0
00001485
0000147b      __stack_chk_fail()
0000147b      noreturn

    .text (PROGBITS) section ended  {0x11c0-0x1486}

00001486                 00 00                      ..
```

linux-x86_64   0x131a–0x131e (0x4 bytes)

The `main` function demonstrates a simple XOR-based encryption scheme combined with pseudo-randomness to generate a "secure letter":

1. It seeds the random number generator `srand()` with the square of the current time.
2. It reads the flag from a file and stores it in a memory buffer.
3. Each character of the flag is XORed with a random number generated by `rand()`.
4. The result is formatted as a two-digit hexadecimal number and printed.

Thus, we can use the Python ctypes utility to wrap C binaries to mimic what the C program does, generating the same random numbers to decrypt the encrypted flag.

Part of the script using Python's ctypes utility to solve the challenge is shown below.

```python
import ctypes
......
print(p.recvuntil(b"!\n").decode())
secured_letter = p.recvline().decode().strip()
log.info(f"Receiving secured letter: {secured_letter}")

libc = ctypes.CDLL("libc.so.6")
libc.time.argtypes = [ctypes.POINTER(ctypes.c_long)]
libc.srand.argtypes = [ctypes.c_uint]
current_time = ctypes.c_long()
libc.time(ctypes.byref(current_time))
libc.srand(ctypes.c_uint(current_time.value ** 2))
libc.rand.restype = ctypes.c_int
```

```
encrypted_flag = [int(secured_letter[i:i+2], 16) for i in range(0, len(secured_letter), 2)]
decrypted_flag = ""
for i in range(len(encrypted_flag)):
    decrypted_flag += chr(encrypted_flag[i] ^ libc.rand() & 0xff)
log.info(f"Decrypting flag: {decrypted_flag}")
```

The captured flag is `flag{p3rh4p5_n07_50000_53cur3:(_7fa5a2182ae33a78}`.

It should be noted that due to the network latency when connecting to the server, in order to retrieve the correct flag, under rare cases, the script should be run multiple times, or the seed for the random number generator should be plus 1.