“L1c3nC3cH3cK” Official Writeup

27th January 2026

Prepared By: {CYNX}`Team

Title: L1c3nC3cH3cK

Description: My very own, very original, super amazing, license checker!

Buy the license key from me for a cheap cheap price of $500 to gain access to a super duper premium cat video!

P.S. Wrap the license key in the flag format

Flag: flag{ESMM-T62U-9704-E97E}

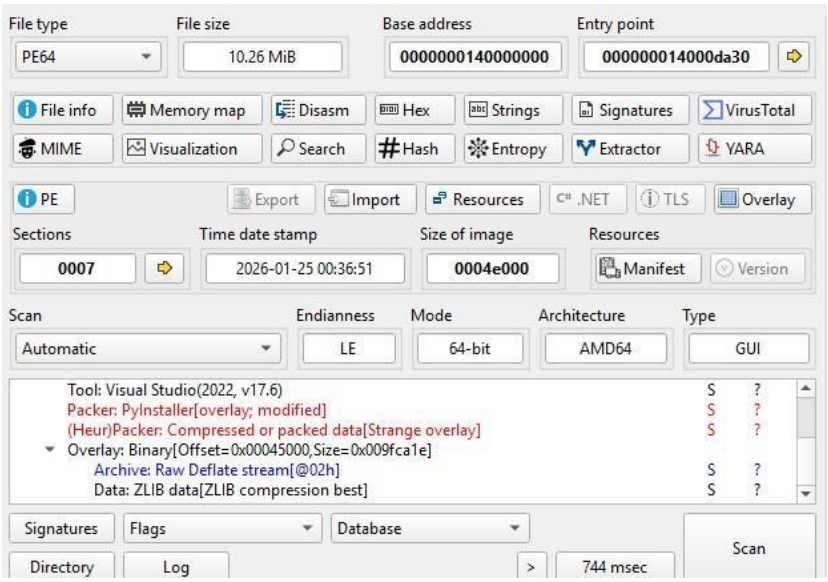
Difficulty: Medium/Hard

Writeup classification: Official

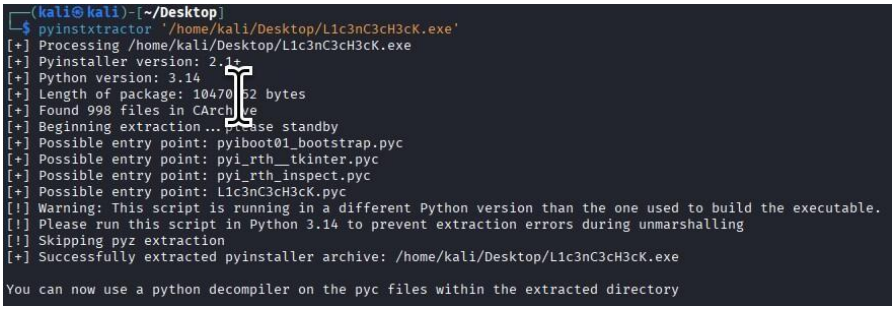
Solve

Running the given executable, we are greeted with a license checker. Inputting a dummy input gives us an error popup:

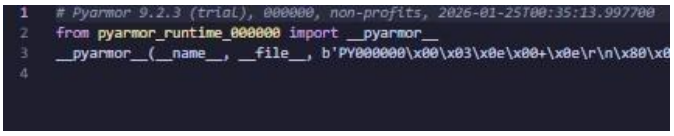
Analyzing this executable in Detect-It-Easy (DiE), we can see that it's been compiled with pyinstaller:



We can simply reverse this with the ` pyinstxtractor ` tool:



Dump the .pyc into pylingual.io and we find that the source code has been obfuscated with pyarmor:



Annoying. Regardless, let's just try our best to extract the source code from the pyarmor obfuscation. To do so, we can use `Process Hacker` alongside `Pyinjector` to spawn a python shell. We'll need to do some manual enumeration to find the main frame of the program:

` import dis;dis.dis(list(sys._current_frames().values())[1].f_code)`

```
pyshell >>> import dis;dis.dis(list(sys._current_frames().values())[1].f_code)
1611           RESUME                   0

1613           LOAD_FAST_BORROW         0 (self)
               LOAD_ATTR                0 (tk)
               LOAD_ATTR                3 (mainloop + NULL|self)
               LOAD_FAST_BORROW         1 (n)
               CALL                     1
               POP_TOP
               LOAD_CONST               1 (None)
               RETURN_VALUE
```

Here, we find a Tkinter GUI frame loop. But heading back one frame, we're able to see the actual main frame.

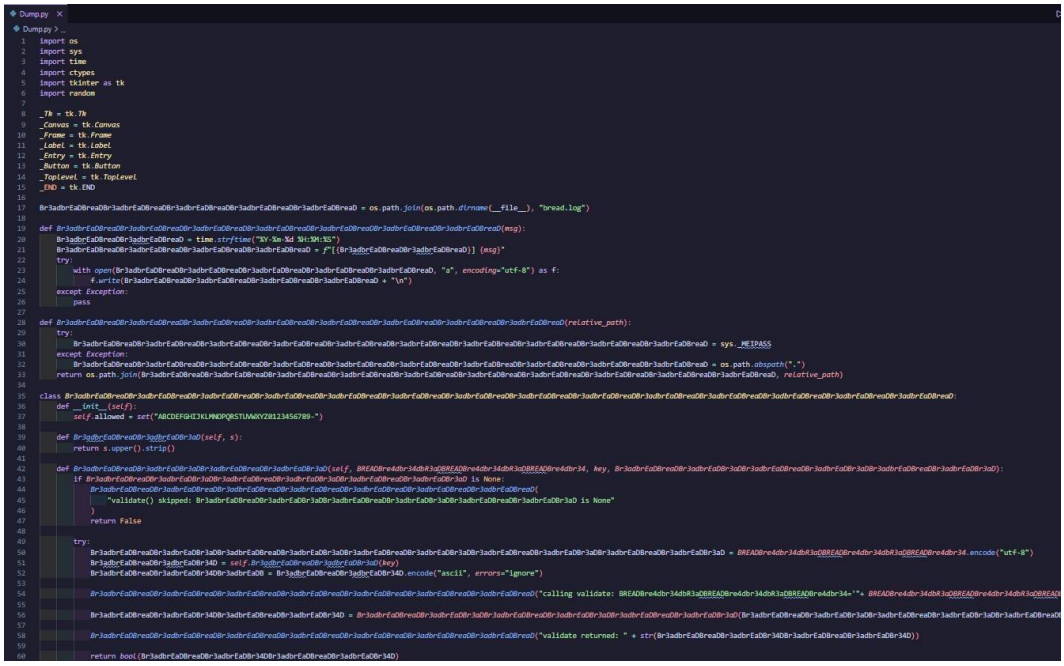
Looking at it, most of the 'main' parts of the code isn't encrypted and we can technically just analyze the instructions here:

```
pyshell >>> import dis;dis.dis(list(sys._current_frames().values())[1].f_back.f_code)
  0           NOP

  1           NOP
              LOAD_CONST               1 (<built-in function C_ENTER_CO_OBJECT_INDEX>)
              PUSH_NULL

  2           LOAD_CONST               2 (b'\x007,\x82\xcc\x01\x00\x00\x03\x00\x00\x1a\x06\x02\x00\x00\x06\x00\x00\x
)
              BUILD_TUPLE              1
              PUSH_NULL
              CALL_FUNCTION_EX
              POP_TOP
              NOP
              LOAD_CONST               0 (<built-in function C_ASSERT_ARMORED_INDEX>)
              STORE_NAME               0 (__assert_armored__)
              NOP

  1   L1:     LOAD_SMALL_INT           0
              LOAD_CONST               3 (None)
              IMPORT_NAME              1 (os)
              STORE_NAME               1 (os)

  2           LOAD_SMALL_INT           0
              LOAD_CONST               3 (None)
              IMPORT_NAME              2 (sys)
              STORE_NAME               2 (sys)

  3           LOAD_SMALL_INT           0
              LOAD_CONST               3 (None)
```

Alternatively, we can decrypt the pyarmor modules and dumping the contents into a .pyc file:

```python
import dis
import marshal
import struct
import sys
import types
import time
import importlib.util

def find_enter_func(consts):
    for k in range(len(consts)):
        try:
            if consts[k].__name__ == "C_ENTER_CO_OBJECT_INDEX":
                return k
        except:
            continue
    return -1

def recurs_dec(target):
    for i in range(len(target.co_consts)):
        tmp = target.co_consts[i]
        try:
            if isinstance(tmp, types.CodeType) and "<frozen L1c3nC3cH3cK>" in tmp.co_filename:
                index_func = find_enter_func(tmp.co_consts)
                if index_func != -1:
                    tmp.co_consts[index_func](tmp.co_consts[index_func + 1])
                    recurs_dec(tmp)
        except:
            continue

target_frame = list(sys._current_frames().values())[1].f_back

recurs_dec(target_frame.f_code)

root_code = target_frame.f_code

all_codes = {}
children_map = {}
queue = [root_code]
```

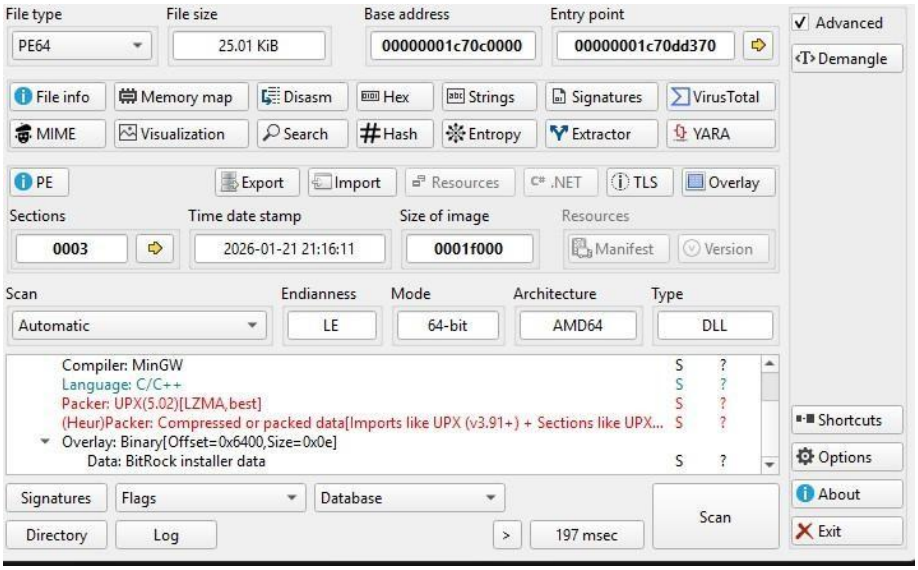
`exec(open(r"Solve.py").read())`



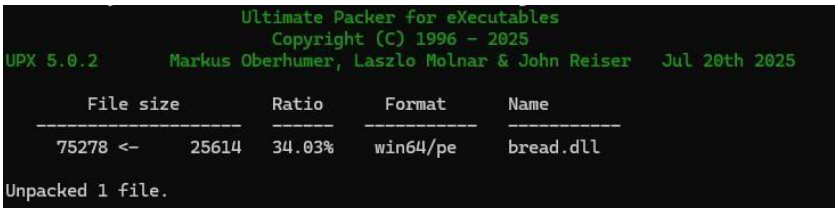
It's just a tad bit obfuscated but, we can still analyze the logic of the code to find that it’s pretty much useless. The main serial logic lies behind the `bread.dll`:

```
50, text='X DLL NOT FOUND X', font=('Courier'
50, text='Missing: bread.dll', font=('Courier'
50, text='Please place bread.dll in the same f
50, text='[ OK ]', command=self.WheatBread.de
```

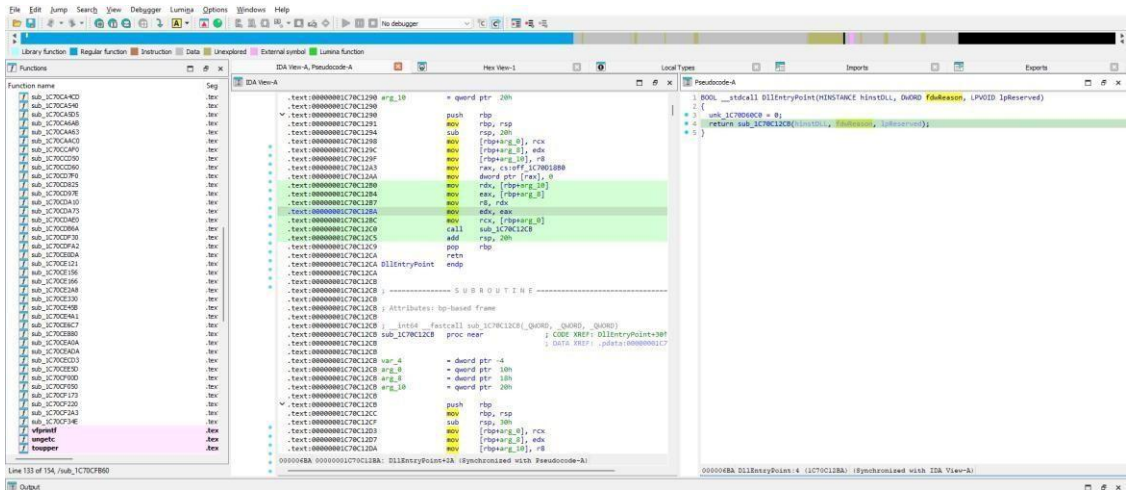
Let’s drop this in DiE to see if we can find any additional info before analyzing in IDA:



Seems like it’s packed with UPX. We can simply unpack this with the automated UPX command:



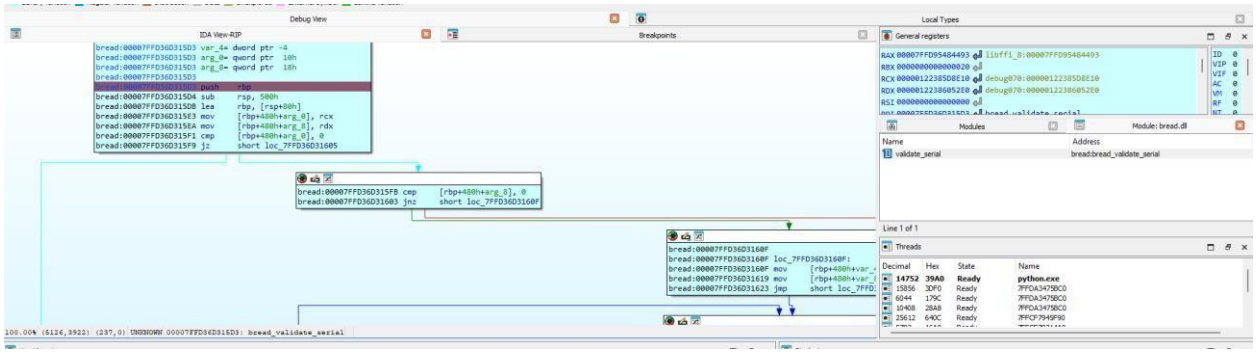
Now let’s utilize IDA to decompile and debug the DLL file:



From here, we can simply set a breakpoint on the `validate_serial` function, then set the process options as such:

Application: <path\to\python.exe>

Parameters: Dump.py



After setting the breakpoint, we load the DLL then continue until the program runs. Then after entering any input, the breakpoint will hit and we can start our analysis.

This program essentially validates a license based on the format of `xxxx-xxxx-xxxx-xxxx`. The validation computes a hash from the username, requires Block 3 to be the XOR'd hash as hex, linking Blocks 1 and 4 to the hash, derives Block 2 from cross-block modular equations:

Serial Validation Algorithm

Username:

- Hash username using rotating XOR with golden ratio multiplier (0x9E3779B1)
- Rotation amount: ((index % 5) * 7) % 32
- Extract 16-bit hash: user_hash = (acc >> 16) & 0xFFFF
- Count consonants (non-vowel letters)

Block Conversions:

- Each block character converted to base-36 value (0-35)
- Block value: val = digit[0]*46656 + digit[1]*1296 + digit[2]*36 + digit[3]

Validation Constraints:

1. **Block 3 Check:**
   a. B3 (as hex) == (user_hash ^ 0x3C3C) 2.

Main Constraint (20-bit):

a. (val_b1 + val_b4) & 0xFFFFF == (7 * user_hash) & 0xFFFFF 3.

Cross-Block Dependencies:

a. b2[0] == (b3[1] + b1[2]) % 36
b. b2[3] == (b4[0] ^ username_length) % 36
c. (b4[1] + b4[2]) % 64 == (b1[0] + b2[2]) % 64
d. (b4[3] * b2[1] + b1[3]) % 43 == (b3[0] + b3[1] + b3[2] + b3[3]) % 43 4.

Tight Constraints:

a. If B4[0] is digit: use value directly, else: (value ^ 7) % 31
b. Must equal: (username_length + sum(b1_values)) % 31
c. (b4[1] + b4[2]) < 64 AND == ((b1[0] ^ b2[3]) & 0x3F) 5. **Digit Balance:**

a. (digit_count_in_serial % 5) == (consonant_count % 5)

With this info, we can craft a script using a constraint solver like Z3 to simultaneously satisfy all the modular equations:

```
import string
import random

ALPHABET = string.digits + string.ascii_uppercase
VAL_MAP = {ch: idx for idx, ch in enumerate(ALPHABET)}
```

```
def hash_v2(username):
    acc = 0     for idx, char in
enumerate(username):
        byte = ord(char)         rot = (idx % 5) *
7         acc ^= (byte << rot) | (byte >> (32 -
rot))         acc = (acc * 0x9E3779B1) & 0xFFFFFFFF
return (acc >> 16) & 0xFFFF

def count_consonants(username):
    vowels = set('AEIOU')
    return sum(1 for c in username.upper() if c.isalpha() and c not in vowels)

def val_to_base36(val):
    result = []
for _ in range(4):
        result.append(ALPHABET[val % 36])         val
//= 36
    return ''.join(reversed(result))

def verify_serial(username, serial):
try:
        parts = serial.upper().split('-')         if
len(parts) != 4:             return
False


        B1, B2, B3, B4 = parts         if not
all(len(p) == 4 for p in parts):
return False


        b1_vals = [VAL_MAP[c] for c in B1]
b2_vals = [VAL_MAP[c] for c in B2]
b3_vals = [VAL_MAP[c] for c in B3]
b4_vals = [VAL_MAP[c] for c in B4]


        val_b1 = sum(b1_vals[i] * (36 ** (3-i)) for i in range(4))
val_b4 = sum(b4_vals[i] * (36 ** (3-i)) for i in range(4))
```

```
        user_hash = hash_v2(username)        user_len
= len(username)        consonant_count =
count_consonants(username)
```

```
                expected_b3 =
user_hash ^
0x3C3C         actual_b3 = int(B3,
16)         if actual_b3 !=
expected_b3:             return False


        target = (7 * user_hash) &
0xFFFFF         actual = (val_b1 + val_b4)
& 0xFFFFF          if actual != target:
return False


        expected = (b3_vals[1] + b1_vals[2]) %
36          if b2_vals[0] != expected:
return False


        expected = (b4_vals[0] ^ user_len) %
36          if b2_vals[3] != expected:
return False
                lhs = (b4_vals[1] +
b4_vals[2]) % 64         rhs = (b1_vals[0] +
b2_vals[2]) %
64          if lhs != rhs:             return False
                lhs = (b4_vals[3] * b2_vals[1] +
b1_vals[3]) %
43          rhs = sum(b3_vals) % 43          if lhs != rhs:
return False
        sum_b1 = sum(b1_vals)          w0 = b4_vals[0]
left = w0 if B4[0].isdigit() else ((w0 ^ 7) % 31)
right = (user_len + sum_b1) % 31          if left !=
right:             return False


        sum_inner = b4_vals[1] + b4_vals[2]          if
sum_inner >= 64:             return False          if
sum_inner != (b1_vals[0] ^ b2_vals[3]) & 0x3F:
            return False
```

```
        digit_count = sum(c.isdigit() for c in
serial)         if digit_count % 5 != consonant_count
% 5:              return False


        return True
except:
return False

def generate_valid_key(username, max_attempts=1000000):
    user_hash      =      hash_v2(username)
user_len = len(username)     consonant_count =
count_consonants(username)          expected_b3
= user_hash ^ 0x3C3C     B3 =
f"{expected_b3:04X}"     b3_vals = [VAL_MAP[c]
for c in B3]          target = (7 * user_hash) &
0xFFFFF


    print(f"Username:    {username}")     print(f"User
hash: 0x{user_hash:04X}")     print(f"Block 3 (fixed):
{B3}")
    print(f"Target sum: {target} (0x{target:05X})")


    attempts = 0     while
attempts < max_attempts:
attempts += 1


        # Random B1          val_b1 =
random.randint(0, 36**4 - 1)
B1 = val_to_base36(val_b1)         b1_vals =
[VAL_MAP[c] for c in B1]


        val_b4_needed = (target - val_b1) & 0xFFFFF
if val_b4_needed >= 36**4:
            continue


        B4 = val_to_base36(val_b4_needed)
b4_vals = [VAL_MAP[c] for c in B4]
```

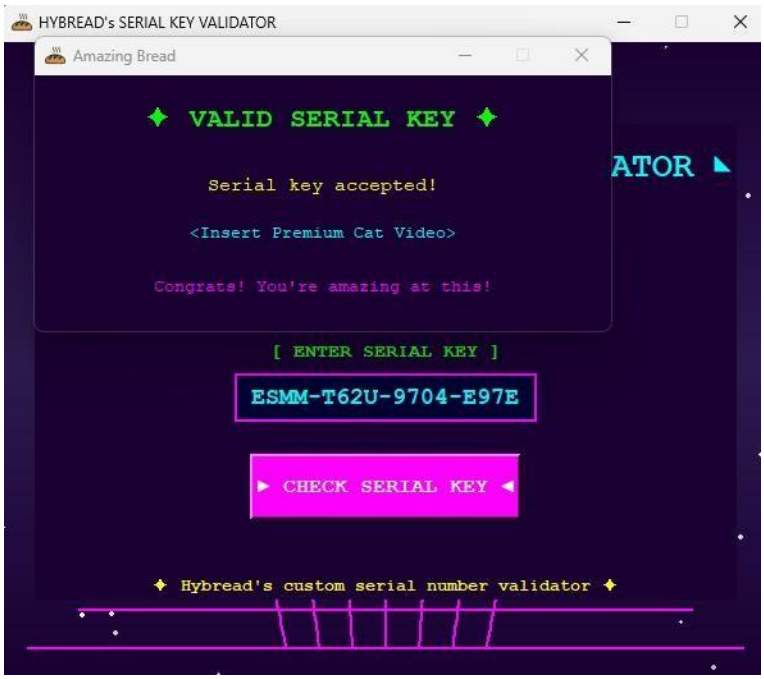
```
        sum_b1 = sum(b1_vals)         w0 =
b4_vals[0]
```

```
        left = w0 if B4[0].isdigit() else ((w0 ^ 7) % 31)         right =
(user_len + sum_b1) % 31        if left != right:
            continue


        sum_inner = b4_vals[1] + b4_vals[2]        if sum_inner >=
64:
            continue

        b2_0 = (b3_vals[1] + b1_vals[2]) % 36
b2_3 = (b4_vals[0] ^ user_len) % 36

        target_sum = (b4_vals[1] + b4_vals[2]) % 64        needed_sum =
(target_sum - b1_vals[0]) % 64                 if needed_sum >=
36:             continue        b2_2 = needed_sum
if sum_inner != (b1_vals[0] ^ b2_3) & 0x3F:
            continue
                target_mod43 = sum(b3_vals) % 43
rhs =
(target_mod43 - b1_vals[3]) % 43


        b2_1 = None        for candidate in range(36):
if (b4_vals[3] * candidate) % 43 == rhs:
                b2_1 = candidate
break                   if b2_1 is None:
            continue


        B2 = ''.join([ALPHABET[b2_0], ALPHABET[b2_1], ALPHABET[b2_2],
ALPHABET[b2_3]])
                serial = f"{B1}-{B2}-{B3}-{B4}"


        digit_count = sum(c.isdigit() for c in serial)        if digit_count
% 5 != consonant_count % 5:
```

```
            continue
                if verify_serial(username,
serial):
            print(f"Found valid key after {attempts:,}
attempts!")             return serial                 if attempts
% 100000 == 0:
            print(f"Attempts: {attempts:,}...")

    print(f"Failed to find valid key after {max_attempts:,} attempts")    return
None

if __name__ == "__main__":
    username = "FlagFetcher2026!"
valid_key = generate_valid_key(username)
if valid_key:         print(f"VALID SERIAL KEY:
{valid_key}")

        print(f"\nVerification: {verify_serial(username, valid_key)}")
```

```
[Running] python -u "c:\Users\hybri\Docume
Username: FlagFetcher2026!
User hash: 0xAB38
Block 3 (fixed): 9704
Target sum: 306824 (0x4AE88)
Found valid key after 17,191 attempts!
VALID SERIAL KEY: ESMM-T62U-9704-E97E

Verification: True
```

To double check, we can enter this serial we got into the application GUI:



Example valid flag = ESMM-T62U-9704-E97E

Flag = flag{valid_input}