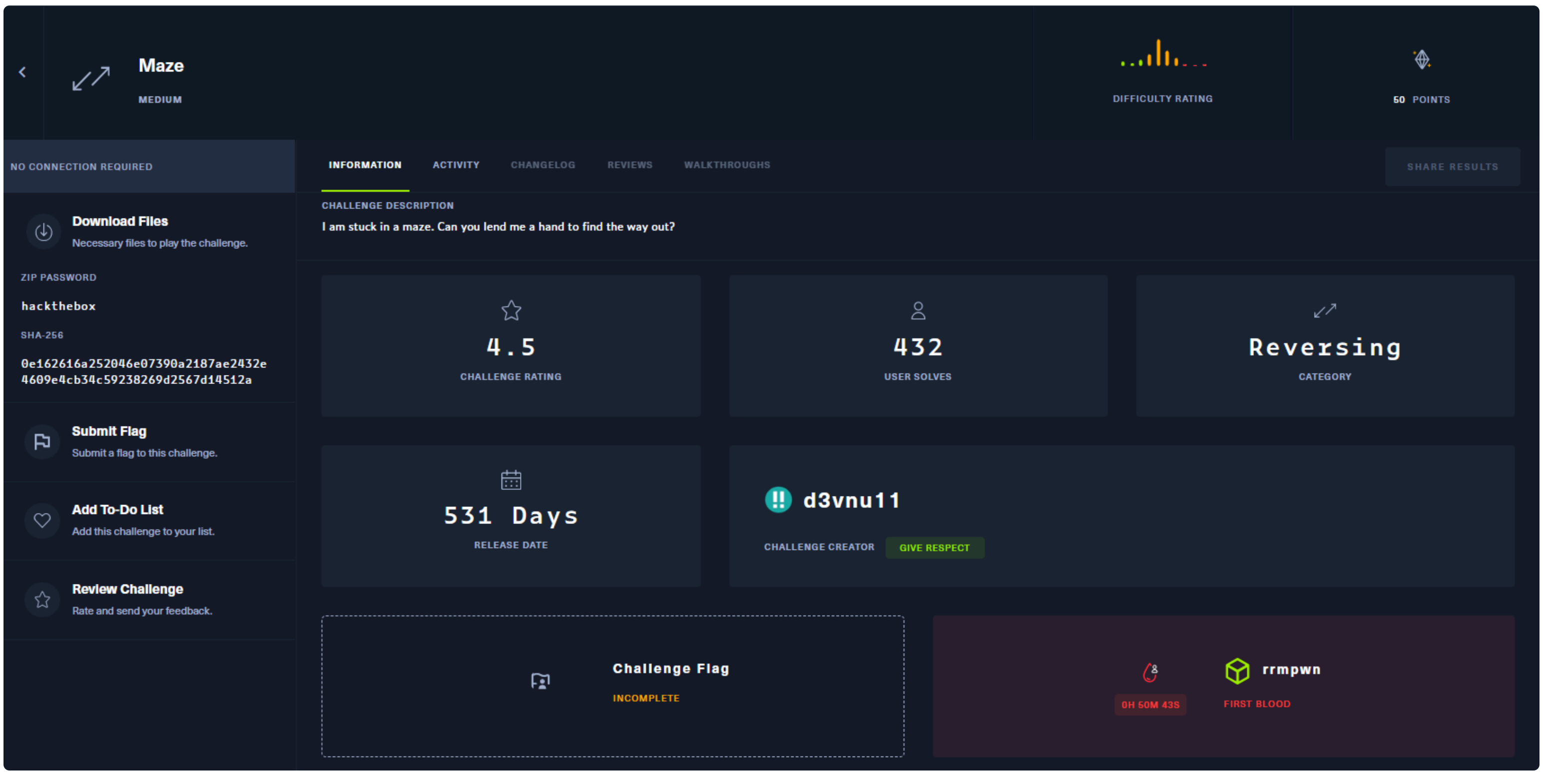
Maze



Attached files : `maze.exe` , `enc_maze.zip` , `maze.png`

Step-by-Step Solution

1. Extract Python Files from Executable

Why: The executable is a PyInstaller bundle containing Python bytecode we need to analyze.
How:

```
# Use pyinstxtractor to unpack the executable
python3.8 pyinstxtractor.py maze.exe

# Output: Creates 'maze.exe_extracted' directory
```

2. Decompile Python Bytecode

Why: We need to recover the original Python logic.
How:

```
# Decompile main script
uncompyle6 maze.exe_extracted/maze.pyc > maze.py

# Decompile obfuscated module
uncompyle6 maze.exe_extracted/PYZ-00.pyz_extracted/obf_path.pyc > obf_path.py
```

Step	How	What made me try this
Identify the packer	`strings maze.exe	head` shows “PYZ” *and* “pyi” magic: dead giveaway it's PyInstaller	
Unpack	`python pyinstxtractor.py maze.exe` → creates `maze.exe_extracted/` with tons of `.pyc` files	`pyinstxtractor` is my go-to because it keeps time-stamps and directory layout intact.
Decompile	`uncompyle6 maze.pyc > maze.py` and the same for `obf_path.pyc`	Reading source is faster than reading byte-code; plus I'm after hard-coded secrets.

Key Findings in `maze.py`:

- Password for ZIP: `Y0u_Ar3_W4lkiNG_t0_Y0uR_D34TH`
- Decryption logic modifies every 10th byte:
  1. Add 80 (mod 256)
  2. XOR with key (initially zeros)

3. Unpack Obfuscated Code

1. **Dump the blob** – I saved it as `BLOB` in `obf_crack.py`.
2. **Look for compression headers** – `BLOB.find(b'\xfd7zXZ')` locates an LZMA stream .
3. **LZMA → zlib → marshal** – The three-stage decompress recipe in `obf_crack.py` yields readable Python code .

Why: `obf_path.py` contains a marshaled blob we need to unpack.
How: Use `obf_crack.py`:

```python
import lzma, zlib

BLOB = b'\xe3\x00...'  # Full blob from obf_path.py

# Find LZMA header and decompress
start = BLOB.find(b'\xfd7zXZ')
lzma_block = BLOB[start:]
lzma_decompressed = lzma.decompress(lzma_block)

# ZLIB decompress
final_payload = zlib.decompress(lzma_decompressed)

with open("dec_obf_path.py", "wb") as f:
    f.write(final_payload)
```

Output: Heavily obfuscated Python with another marshaled blob.

4. Analyze Second Obfuscation Layer

Why: The new script contains critical seed logic.
How: Use `obf_crack2.py` to disassemble (use python3.8 again):

```
import marshal, dis

STAGE5 = b'\xe3\x00...'  # Blob from dec_obf_path.py
code = marshal.loads(STAGE5)

def try_dis(obj, level=0):
    if isinstance(obj, types.CodeType):
        dis.dis(obj)  # Disassemble bytecode
        # Recursively process constants
        for const in obj.co_consts:
            try_dis(const, level+1)

try_dis(code)
```

Key Findings:

1. Reads `maze.png`
2. Computes seed from bytes at offsets:
   - 4817
   - 2624
   - 2640
   - 2720
3. Seed = sum of these bytes

5. Compute the Seed

Why: The seed generates the decryption key.
How:

```
with open('maze.png', 'rb') as f:
    data = f.read()
    seed = data[4817] + data[2624] + data[2640] + data[2720]
print(seed)  # Output: 493
```

6. Decrypt the Maze Binary

Why: The extracted `maze` file is encrypted.
How:

```
import random, pyzipper

# 1. Extract maze file from ZIP
with pyzipper.AESZipFile('enc_maze.zip') as zf:
    zf.pwd = b'Y0u_Ar3_W4lkiNG_t0_Y0uR_D34TH'
    zf.extract('maze')

# 2. Generate key using seed
random.seed(493)
key = [random.randint(32, 125) for _ in range(300)]  # 300-byte key

# 3. Decrypt the binary
with open('maze', 'rb') as f:
    data = bytearray(f.read())

for i in range(0, len(data), 10):
    data[i] = (data[i] + 80) % 256  # First operation
    data[i] = (data[i] ^ key[i % 300]) % 256  # Second operation

with open('dec_maze.elf', 'wb') as f:
    f.write(data)  # Output: Valid ELF binary
```

Why those two passes?
The original script added 80 before XORing with an all-zero key—a deliberate sabotage so the players thinks the algorithm is pointless. Re-inserting the proper key flips the switch and the output turns into a bona-fide ELF header (7F 45 4C 46).

7. Extract Encrypted Flag from ELF

Why: The flag is hidden in the binary's data section.
How:

```
with open('dec_maze.elf', 'rb') as f:
    elf = f.read()

# Find flag array using known header values
pattern = b'\xde\x00\x00\x00\x11\x01\x00\x00'  # 222, 273 in little-endian
pos = elf.find(pattern)

# Extract 4-byte integers until invalid value
encrypted_flag = []
for i in range(0, 100*4, 4):
    val = int.from_bytes(elf[pos+i:pos+i+4], 'little')
    if val < 96 or val > 378: break  # Valid range: 32*3 to 126*3
    encrypted_flag.append(val)
```

8. Reconstruct the Flag

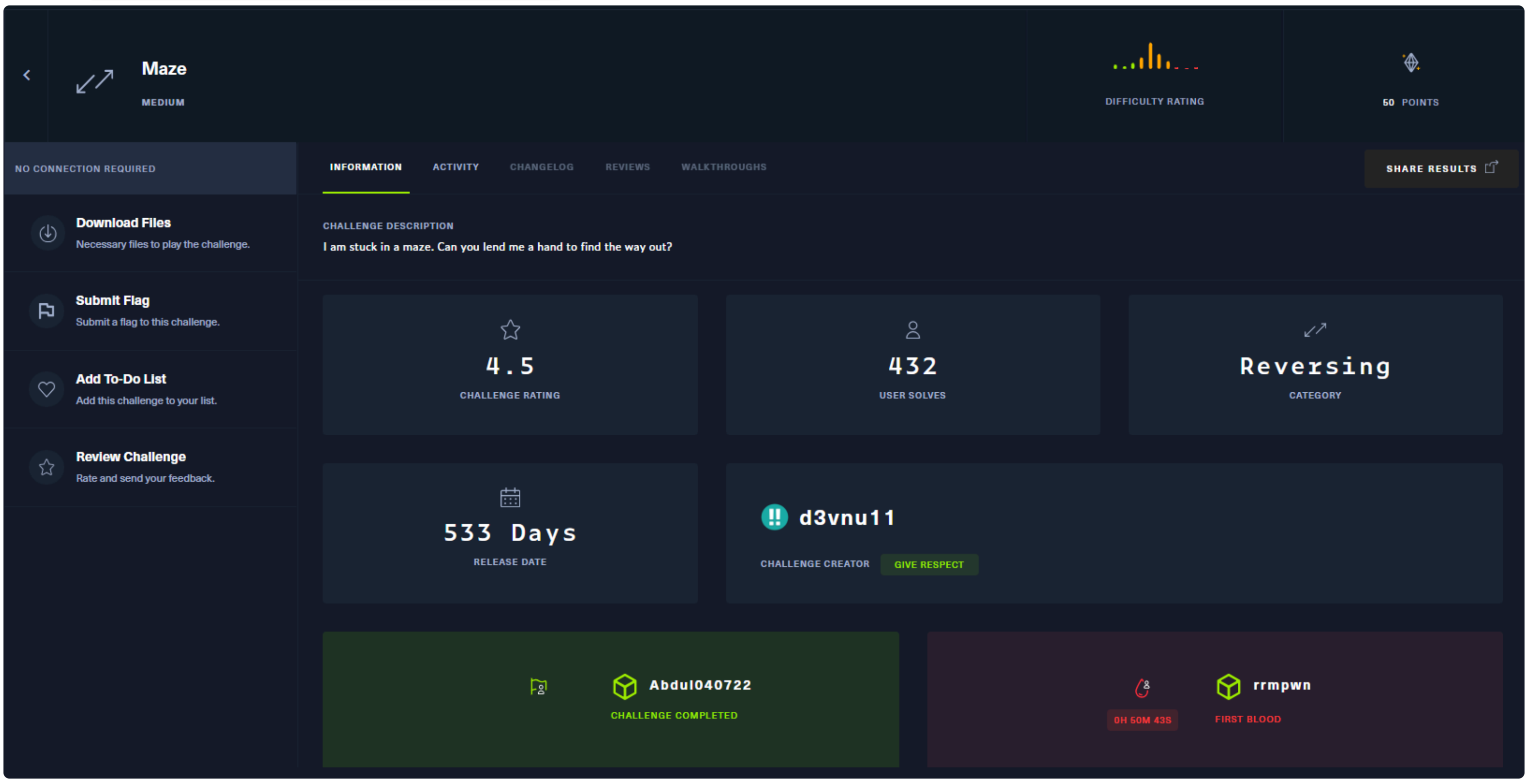
Why: The flag is derived from triple-character sums.
How:

```
flag = [72, 84, 66]  # 'H','T','B'

for i in range(1, len(encrypted_flag)):
    # Calculate next char: S[i] = flag[i] + flag[i+1] + flag[i+2]
    next_char = encrypted_flag[i] - flag[i] - flag[i+1]
    flag.append(next_char)

print(''.join(chr(c) for c in flag))  # Full flag
```

Final Solution Script

```
import pyzipper
import random
import struct

# Step 1: Extract maze file
with pyzipper.AESZipFile('enc_maze.zip') as zf:
    zf.pwd = b'Y0u_Ar3_W4lkiNG_t0_Y0uR_D34TH'
    zf.extract('maze')

# Step 2: Compute seed from maze.png
with open('maze.png', 'rb') as f:
    data = f.read()
seed = data[4817] + data[2624] + data[2640] + data[2720]  # = 493

# Step 3: Generate key
random.seed(seed)
key = [random.randint(32, 125) for _ in range(300)]

# Step 4: Decrypt binary
with open('maze', 'rb') as f:
    data = bytearray(f.read())

for i in range(0, len(data), 10):
    data[i] = (data[i] + 80) % 256
    data[i] = (data[i] ^ key[i % len(key)]) % 256

with open('dec_maze.elf', 'wb') as f:
    f.write(data)

# Step 5: Extract flag array from ELF
with open('dec_maze.elf', 'rb') as f:
    elf = f.read()

pattern = b'\xde\x00\x00\x00\x11\x01\x00\x00'  # 222, 273
pos = elf.find(pattern)
encrypted_flag = []

for i in range(0, 100*4, 4):
    if pos+i+4 > len(elf): break
    val = struct.unpack('<I', elf[pos+i:pos+i+4])[0]
    if val < 96 or val > 378: break
    encrypted_flag.append(val)

# Step 6: Reconstruct flag
flag = [72, 84, 66]  # "HTB"

for i in range(1, len(encrypted_flag)):
    next_char = encrypted_flag[i] - flag[i] - flag[i+1]
    if not 32 <= next_char <= 126: break  # Printable ASCII check
    flag.append(next_char)

print('FLAG:', ''.join(chr(c) for c in flag))
```

Output: `HTB{w0W_Y0u_C0uld_E5c4p3_Th1s_M4Z33!!}`



Key Insights

1. **Seed Obfuscation**: Critical values (like the seed 493) are often hidden in unrelated files (e.g., image bytes)
2. **Progressive Decryption**:
   - First pass: Arithmetic operation (add 80)
   - Second pass: Cryptographic operation (XOR with PRNG key)
3. **Flag Reconstruction**:
   - Look for known header ("HTB")
   - Derive subsequent characters using adjacent values