neverending_randomness

The server seeds Python's `random.Random` (Mersenne Twister) either with a large-entropy value read from `/opt/app/random` or with a predictable seed `int(time.time()) ^ pid`. The server then uses that RNG as an MT-based stream cipher to XOR the flag, and *then* leaks three subsequent 32-bit RNG outputs. By forcing the fallback to the time-based seed (or by brute-forcing that seed if already used), we can recover the MT state (really: test candidate seeds) quickly and decrypt the ciphertext.

Exploit strategy (step-by-step)

```
Connect and parse the server response: a Python-dict string with
ciphertext_hex, leak32 (list of 3 uint32 values), and pid.

Let n = len(ciphertext). For a candidate seed s = t ^ pid where t is an
integer        second timestamp candidate:
        Create rng = random.Random(s).
        Skip n 32-bit draws (one per keystream byte).
        Read the next three 32-bit draws. If they equal leak32, the seed is
correct.

If the fallback branch is not yet active (i.e., /opt/app/random still
supplies         large entropy), the brute will fail. In that case reconnect
repeatedly until        the fallback triggers (one connection consumes bytes
from the shared FD). On        the connection where the server used
fallback, the brute will find the              correct seed.

Once the correct t is found, re-seed and generate the keystream bytes (using
getrandbits(8) for each keystream byte), XOR them with ciphertext to recover
the flag.
```

solve.py

```
#!/usr/bin/env python3
import socket
import time
import ast
import random
from typing import Optional, Tuple

HOST = "ctf.ac.upt.ro"
PORT = 9923

# How many seconds around now to try for the seed (both directions)
RADIUS = 1800
```

```
# How many reconnect attempts before giving up
MAX_RECONNECTS = 5000

# Small sleep between reconnections to avoid busy-looping
SLEEP_BETWEEN = 0.02


def recv_all(conn: socket.socket, bufsize: int = 1 << 15, timeout: float = 3.0) -> bytes:
    """Receive until the server closes or we reach bufsize. Returns raw bytes."""
    conn.settimeout(timeout)
    data = bytearray()
    try:
        while True:
            chunk = conn.recv(4096)
            if not chunk:
                break
            data.extend(chunk)
            if len(data) >= bufsize:
                break
    except socket.timeout:
        # timeout is fine; we probably got the full dict
        pass
    return bytes(data)


def fetch_once(host: str = HOST, port: int = PORT, timeout: float = 3.0) -> Tuple[bytes, Tuple[int, int, int], int]:
    """Connect, read the server dict, and return (ciphertext_bytes, leak32_tuple, pid)."""
    with socket.create_connection((host, port), timeout=timeout) as s:
        raw = recv_all(s)
    if not raw:
        raise ConnectionError("No data received from server")
    # Server sends a Python dict string; use ast.literal_eval to parse safely
    parsed = ast.literal_eval(raw.decode().strip())
    ct = bytes.fromhex(parsed["ciphertext_hex"])
    leak = tuple(int(x) for x in parsed["leak32"])
    pid = int(parsed["pid"])
    return ct, leak, pid


def ring_timestamps(center: int, radius: int):
    """Yield timestamps in a ring around center: center, center+1, center-1, center+2, center-2, ..."""
    yield center
    for d in range(1, radius + 1):
        yield center + d
        yield center - d


def try_brute_time_seed(ct: bytes, leak: Tuple[int, int, int], pid: int,
radius: int = RADIUS) -> Optional[bytes]:
    """
    Try seeds of form (t ^ pid) for t in ring around current time.
    If a matching seed is found, return the decrypted plaintext bytes,
otherwise None.
    """
    skip = len(ct)  # number of 32-bit draws consumed by keystream bytes
    now = int(time.time())

    for t in ring_timestamps(now, radius):
        seed = t ^ pid
        rng = random.Random(seed)
        # Skip 'skip' 32-bit outputs (one per keystream byte)
        for _ in range(skip):
            rng.getrandbits(32)
        # Compare next three 32-bit outputs to leaked values
        got = (rng.getrandbits(32), rng.getrandbits(32),
rng.getrandbits(32))
        if got == leak:
            # Recreate keystream and decrypt
            rng = random.Random(seed)
            ks = bytes(rng.getrandbits(8) for _ in range(skip))
            pt = bytes(c ^ k for c, k in zip(ct, ks))
            return pt
    return None


def solve():
    print(f"[+] Target: {HOST}:{PORT}")
    tries = 0
    while tries < MAX_RECONNECTS:
        tries += 1
        try:
            ct, leak, pid = fetch_once()
        except Exception as e:
            # connection or parse error; keep trying
            print(f"[-] fetch error (attempt {tries}): {e}")
            time.sleep(SLEEP_BETWEEN)
            continue

        # Try brute-forcing time^pid seed on this connection's data
        pt = try_brute_time_seed(ct, leak, pid)
        if pt:
            # Found it
            print(f"[+] Success on attempt {tries}! PID={pid}")
            try:
                print(pt.decode("utf-8"))
            except Exception:
                # binary-safe fallback
                print(pt)
            return

        # Not found: likely the server used /opt/app/random (not fallback)
on this connection.
        # Keep reconnecting until the fallback branch happens.
        if tries % 50 == 0:
            print(f"[i] Attempt {tries}: not found yet; still trying to
force fallback...")
        time.sleep(SLEEP_BETWEEN)

    print("[-] Reached max reconnect attempts without success.")


if __name__ == "__main__":
    solve()
```