CyberApocalypse 2026 — Crypto — False Witness

> *Caldrin Vowmark knows that not every seal deserves belief. Some marks still carry the weight of a living vow; others only imitate one well enough to pass a glance. She can question the realm's witnesses as many times as she likes, but certainty, it turns out, is much harder to earn than a convincing lie.*

- **Category:** Crypto
- **Files:** `server.py`
- **Flag format:** `HTB{...}`

---

1. The target

The whole challenge is a single ~60 line service:

```
P = 0xCD4A96D3B7FA7251A1BB765933FB676FCAE8C9026682E34F779122DFD66915BB
FLAG = open("flag.txt", "rb").read().strip()
N = sha256().digest_size * 8            # 256

KEY = secrets.token_bytes(32)
KEY_BITS = list(map(int, f"{int.from_bytes(KEY):0256b}"))

def H(msg):
    return pow(G, msg, P)

class Oracle:
    def __init__(self):
        self.appeared = {}
    def oracle(self, i):
        if i in self.appeared:
            return self.appeared[i]
        else:
            ret = secrets.randbelow(2**N) if KEY_BITS[i] == 0 else PK[i % len(PK)][secrets.randbelow(2)]
            self.appeared[i] = ret
        return ret

def keygen():
    sk = [(secrets.randbelow(2**N), secrets.randbelow(2**N)) for _ in range(N)]
    pk = [(H(s[0]), H(s[1])) for s in sk]
    return sk, pk
```

Flow:

1. The AES-ECB encryption of the flag under the random 32-byte `KEY` is printed **immediately**, before anything else.
2. We are asked for a generator `G`, accepted as long as `1 < G < P`.
3. `keygen()` builds a Lamport-style one-time-signature keypair using `H(m) = G^m mod P`.
4. We may call `oracle(i)` for `i ∈ [0, 255]` as many times as we like.

The structure is a nod to Lamport signatures / "witness" commitments, hence the flavour text — but the actual key material of the signature scheme is a red herring. The only secret that matters is `KEY`, and `KEY_BITS` is exactly the AES key expanded bit by bit.

2. What the oracle actually leaks

For each index `i`, `oracle(i)` returns one of two very different things:

`KEY_BITS[i]`	return value
`0`	`secrets.randbelow(2**256)` — a uniform 256-bit integer
`1`	`PK[i][0]` or `PK[i][1]` — i.e. `G^s mod P` for a secret random `s`

Note the memoisation in `self.appeared`: the answer for a given `i` is fixed on the first call, so re-querying gains nothing. We get exactly **one sample per key bit**, 256 samples total.

So recovering the AES key reduces to a distinguishing problem, repeated 256 times:

> *Given one sample, is it uniform in `[0, 2^256)`, or is it a uniformly random element of `⟨G⟩ ⊆ F_P^*`?*

With an honest generator this is hopeless-ish per sample: `P` is a 256-bit prime, so if `G` generates a large subgroup, `G^s mod P` looks essentially uniform in `[0, P)`. The *only* statistical edge would be the tiny gap between `P ≈ 0.802 · 2^256` and `2^256`: a sample in `[P, 2^256)` proves the bit is `0`, which happens with probability ≈ 19.8%. That leaks about 0.28 bits per index and gives no way to ever confirm a `1`. Nowhere near enough for a 256-bit key.

The trick is that **we choose `G`**.

3. The bug: attacker-controlled generator, no order check

The server only validates `1 < G < P`. It never checks that `G` generates a large subgroup — or any subgroup of a particular order at all.

`P` is prime:

```
$ python3 -c "from sympy import isprime; print(isprime(0xCD4A96D3B7FA7251A1BB765933FB676FCAE8C9026682E34F779122DFD66915BB
True
```

so `F_P^*` is cyclic of order `P−1`, which is even. Therefore it contains a unique element of order 2, namely `−1 mod P = P−1`.

Choose

```
G = P − 1
```

This passes the `1 < G < P` check, and now the image of `H` collapses to a two-element set:

```
H(m) = (P−1)^m mod P = (−1)^m mod P ∈ {1, P−1}
```

Consequently **every** entry of `PK` is either `1` or `P−1`. The distinguisher becomes trivial:

- If `oracle(i) ∈ {1, P−1}` → `KEY_BITS[i] = 1`.
- Otherwise → `KEY_BITS[i] = 0`.

There are **no false negatives**: when the bit is `1` the answer is guaranteed to be in `{1, P−1}`. The only error mode is a false positive — a uniform 256-bit draw landing exactly on `1` or `P−1` — with probability `2 / 2^256 ≈ 2^−255` per index. Over 256 indices the failure probability is about `2^−247`, i.e. it never happens.

This is precisely the "false witness" of the title: `P−1` is a mark that *looks* like a generator to the server's check, but carries no weight at all.

Why order 2 and not something else

Any small-order `G` works — the smaller the subgroup, the cheaper the test. Order 2 is optimal and, conveniently, `P−1` is a closed-form element of order 2 for any odd prime `P`, so no factoring of `P−1` is needed. (`G = 1` would be even better — a one-element image — but it is exactly what the `1 < G` check excludes, which shows the author *thought* about degenerate generators and only blocked the most obvious one.)

4. Putting it together

1. Read the hex ciphertext printed at connect time.
2. Send `G = P - 1`.
3. Query `oracle(i)` for `i = 0 … 255`, mapping each answer to a bit.
4. `KEY = int("".join(bits), 2).to_bytes(32, "big")` — note `KEY_BITS` was built with `f"{int.from_bytes(KEY):0256b}"`, i.e. big-endian MSB-first, so the reassembly is a straight big-endian conversion.
5. Decrypt the blob with AES-ECB and strip PKCS#7 padding.

5. Exploit

```python
#!/usr/bin/env python3
import sys
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad

P = 0xCD4A96D3B7FA7251A1BB765933FB676FCAE8C9026682E34F779122DFD66915BB
G = P - 1  # element of order 2  ->  H(m) in {1, P-1}
NBITS = 256


def solve(io):
    io.recvuntil(b"Here is something for you:\n")
    ct = bytes.fromhex(io.recvline().strip().decode())

    io.recvuntil(b"generator: ")
    io.sendline(str(G).encode())

    bits = []
    for i in range(NBITS):
        io.recvuntil(b"> ")
        io.sendline(b"1")
        io.recvuntil(b"Enter offset: ")
        io.sendline(str(i).encode())
        io.recvuntil(b"Oracle result: ")
        val = int(io.recvline().strip())
        bits.append(1 if val in (1, P - 1) else 0)

    key = int("".join(map(str, bits)), 2).to_bytes(32, "big")
    print("[+] key:", key.hex())
    pt = AES.new(key, AES.MODE_ECB).decrypt(ct)
    try:
        pt = unpad(pt, 16)
    except ValueError:
        pass
    print("[+] flag:", pt.decode(errors="replace"))
    return pt


if __name__ == "__main__":
    from pwn import process, remote, context
    context.log_level = "error"
    if len(sys.argv) >= 3:
        io = remote(sys.argv[1], int(sys.argv[2]))
    else:
        io = process([sys.executable, "server.py"])
    solve(io)
    io.close()
```

Usage:

```
$ python3 solve.py                # local server.py
$ python3 solve.py <host> <port>  # remote
```

Run against the live service:

```
$ python3 solve.py 154.57.164.81 32148
[+] key: b9107529356f38fafe6907c1fc568fed75e8c236234be3bf9bc6ae2bec29b54b
[+] flag: HTB{___l34k1ng_b1ts_0n3_by_0n3___}
```

Deterministic and instant: 256 queries, one round trip each, no guessing, no search. The flag name — "leaking bits one by one" — confirms the intended per-bit key-recovery attack.

6. Flag

```
HTB{___l34k1ng_b1ts_0n3_by_0n3___}
```

7. Lessons

- **Never accept a group element from an untrusted party without validating its order.** The check `1 < G < P` says nothing about the subgroup `⟨G⟩`. The correct check is `G != 1` **and** `G^q mod P == 1` for the intended prime order `q` (or equivalently `G^((P−1)/q) != 1` for a designated generator), using a group with known structure such as a safe-prime or a standardised DH/ECC group. This is the same family of bug as small-subgroup confinement attacks on Diffie–Hellman.
- **Indistinguishability arguments break when the adversary controls the parameters.** "A random group element looks uniform" is only true for a group the adversary cannot shrink.
- **Don't mix uniform-random and structured values as a covert channel for secret bits.** Any distinguisher against the structured distribution becomes a full key-recovery oracle. Here one bit of distinguishing advantage per index maps one-to-one onto one bit of the AES key.
- Memoising oracle answers (`self.appeared`) limits the adversary to one sample per bit — a sensible hardening that would have defeated a purely statistical attack, but it is irrelevant once the distinguisher is perfect.