UofTCTF 2026: MAT347

1 Overview

Category: ECDSA + RNG

The challenge uses the NIST P-256 elliptic curve over a prime field $\mathbb{F}_p$. A secret scalar x is chosen, and the public point

$$Q = xG$$

is printed, where G is the standard generator of a prime-order subgroup of size n.

The server gives two options for up to 670 queries:

- `sign`: returns a signature (r, s) on a chosen message m.
- `exchange`: returns an AES-CBC encryption of the flag using a key derived from $(kQ).x$.

The elliptic curve is not the weak part. P-256 is designed to resist brute force. The weak part is the custom RNG that produces the nonce k.

2 Relevant challenge code (snippets)

The core bug is that the RNG state update is *linear* modulo 2^{256}. Also, the RNG output depends only on the old counter value.

Listing 1: RNG, sign, exchange (key parts)

```
def h(m):
    return bytes_to_long(sha256(m.encode()).digest())

class RNG:
    def __init__(self):
        self.cnt = bytes_to_long(os.urandom(32))
        self.mod = 2**256
    def next(self, m):
        res = h(str(self.cnt)) # output depends only on old cnt
        a = 0 if m is None else h(m) # chosen by attacker for sign()
        self.cnt = (self.cnt+1+a) % self.mod
        return res

def sign(m, rng):
    z = h(m)
    k = rng.next(m)
    r = int((k*G).x())
    s = (pow(k, -1, E.order()) * z + (x*r)) % E.order()
```

```
    return r, s

def exchange(rng):
    k = rng.next(None)
    S = int((k*Q).x())
    key = long_to_bytes(S % (2**128), 16)
    iv = long_to_bytes(S >> 128, 16)
    return AES.new(key, AES.MODE_CBC, iv=iv).encrypt(pad(flag, 16))
```

3 Key observation: the RNG is predictable in its state update

The RNG keeps an internal counter `cnt` $\in \{0, \dots, 2^{256} - 1\}$. On each call:

$$k = \text{SHA256}(\texttt{str(cnt)}),$$

then it updates:

$$\texttt{cnt} \leftarrow \texttt{cnt} + 1 + a \pmod{2^{256}},$$

where $a = 0$ for `exchange` and $a = \text{SHA256}(m)$ for `sign`(m).

Let $H(m)$ be SHA-256 of message m interpreted as a 256-bit integer. Define:

$$\Delta(m) = 1 + H(m) \pmod{2^{256}}.$$

Then:

- after `exchange`: `cnt` $\leftarrow$ `cnt` $+ 1$,
- after `sign`(m): `cnt` $\leftarrow$ `cnt` $+ \Delta(m)$.

So we cannot predict the *starting* counter, but we *can* control how it moves.

4 Why repeating `cnt` repeats the nonce k

Because $k = \text{SHA256}(\texttt{str(cnt)})$, if we can return `cnt` to a previous value, the next nonce k will repeat exactly. This is the main goal: *force nonce reuse.*

5 Nonce reuse breaks the signature (recovering k)

The signing code is:

$$r = x\text{-coordinate}(kG), \qquad s \equiv k^{-1}z + xr \pmod{n},$$

where $z = H(m) \bmod n$ and n is the (prime) order of $\langle G \rangle$.

If we get two signatures on different messages m_1, m_2 using the *same* nonce k, then r is also the same (because it only depends on k).

Write $z_i = H(m_i) \bmod n$. Then:

$$s_1 \equiv k^{-1}z_1 + xr \pmod{n}, \qquad s_2 \equiv k^{-1}z_2 + xr \pmod{n}.$$

Subtract:

$$s_1 - s_2 \equiv k^{-1}(z_1 - z_2) \pmod n.$$

Multiply both sides by k:

$$k(s_1 - s_2) \equiv z_1 - z_2 \pmod n.$$

If $s_1 \not\equiv s_2 \pmod n$ (true with overwhelming probability), we can invert:

$$k \equiv (z_1 - z_2) \cdot (s_1 - s_2)^{-1} \pmod n.$$

Important: We do not need the private key x to decrypt the flag. We only need the nonce k used inside `exchange`.

6 Decrypting the `exchange` ciphertext once k is known

The `exchange` function computes:

$$S = (kQ).x \in \{0, \ldots, p-1\}.$$

Then it sets:

$$\text{key} = S \bmod 2^{128}, \qquad \text{iv} = \left\lfloor \frac{S}{2^{128}} \right\rfloor,$$

and encrypts the padded flag with AES-CBC.

So once we recover k, we compute kQ (we know public Q), take its x-coordinate, build key/IV, and decrypt.

7 How we force nonce reuse (the modular knapsack step)

Each chosen message moves the counter by $\Delta(m) = 1 + H(m) \pmod{2^{256}}$. If we find messages $m_1, \ldots, m_t$ such that

$$\sum_{i=1}^{t} \Delta(m_i) \equiv 0 \pmod{2^{256}},$$

then signing all of them returns `cnt` to the same value.

In our solve we need two targeted "rewinds":

Rewind 1 (undo the `exchange` increment)

If we call `exchange` first, the counter increases by $+1$. To return to the previous counter we need:

$$\sum \Delta(m_i) \equiv -1 \pmod{2^{256}}.$$

Rewind 2 (undo the increment of a chosen signature message)

After signing one chosen message m_A, the counter increases by $\Delta(m_A)$. To return again, we need:

$$\sum \Delta(m_i) \equiv -\Delta(m_A) \pmod{2^{256}}.$$

Finding subsets with lattices (LLL)

This is a subset-sum problem modulo 2^{256}. A common CTF method is:

- generate many random messages m_i,
- compute the values $\Delta(m_i)$,
- use an LLL-based lattice construction to find a subset that hits the target mod 2^{256}.

With a few hundred candidates, it usually succeeds quickly. The query limit 670 is enough: for example, $t_1 \approx 300$ for rewind 1 and $t_2 \approx 300$ for rewind 2 plus a few extra calls.

8 Full attack steps

1. Call `exchange` once, save ciphertext C. This uses some nonce k_0 from some counter value $\texttt{cnt}_0$, then moves the counter to $\texttt{cnt}_0 + 1$.
2. Find a subset of messages with $\sum \Delta(m_i) \equiv -1 \pmod{2^{256}}$. Send `sign` for each of them. Now the counter is back to $\texttt{cnt}_0$.
3. Choose message m_A. Call `sign`(m$_A$)$andrecord(r_1, s_1)$. This uses nonce k_0 again.
4. Find a subset with $\sum \Delta(m_i) \equiv -\Delta(m_A) \pmod{2^{256}}$. Sign them. Now the counter is back to $\texttt{cnt}_0$.
5. Choose a different message m_B. Call `sign`(m$_B$)$andrecord(r_2, s_2)$. Now the nonce repeats again, so $r_2 = r_1$.
6. Recover k_0 from:

$$k_0 \equiv (z_1 - z_2) \cdot (s_1 - s_2)^{-1} \pmod{n}, \quad z_i = H(m_i) \bmod n.$$

7. Compute $S = (k_0 Q).x$, derive AES key/IV, and decrypt C to get the flag.

9 Why there is no easy ECC shortcut

Because G generates a prime-order subgroup of size $n \approx 2^{256}$, recovering x from $Q = xG$ is ECDLP. Generic attacks need about 2^{128} work, which is not feasible. So the intended solution is the RNG/nonce bug.

10 Appendix: solver code snippets

Below are the key parts used in a local solve. The full solver combines (1) a local process driver, (2) the LLL knapsack, and (3) the math for recovering k and decrypting.

Computing $\Delta(m)$ and recovering k

Listing 2: Recovering nonce k from two signatures with same r

```
MOD = 2**256
n = 0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551

```

```
def Hs(s):
    return bytes_to_long(sha256(s.encode()).digest())

def Delta(m):
    return (1 + Hs(m)) % MOD

# given (r, s1) on m1 and (r, s2) on m2 with same r:
z1 = Hs(m1) % n
z2 = Hs(m2) % n
k = ((z1 - z2) * inverse_mod((s1 - s2) % n, n)) % n
```

LLL knapsack (mod 2^{256}) to build a rewind set

Listing 3: LLL-based subset-sum mod 2^{256} (Sage code)

```
from sage.all import Matrix, ZZ

def find_subset_for_target(target, N=320, scale_bits=300, tries=80):
    target %= MOD
    M = 2**scale_bits

    for t in range(tries):
        msgs = [f"rw_{t}_{i}_{os.urandom(4).hex()}" for i in range(N)]
        vals = [Delta(m) for m in msgs]

        B = Matrix(ZZ, N+2, N+2)
        for i, v in enumerate(vals):
            B[i, i] = M
            B[i, N+1] = v
        B[N, N+1] = MOD
        for i in range(N):
            B[N+1, i] = M//2
        B[N+1, N+1] = target

        L = B.LLL()
        for row in L.rows():
            if row[N+1] != 0:
                continue
            bits = [1 if row[i] > 0 else 0 for i in range(N)]
            s = sum(vals[i] for i,b in enumerate(bits) if b) % MOD
            if s == target:
                return [msgs[i] for i,b in enumerate(bits) if b]
    raise RuntimeError("LLL failed; increase params")
```

Decrypting the exchange ciphertext

Listing 4: Decrypt exchange once k is known

```
Sx = int((k * Q).x())
key = long_to_bytes(Sx % (2**128), 16)
iv = long_to_bytes(Sx >> 128, 16)

pt = AES.new(key, AES.MODE_CBC, iv=iv).decrypt(ct)
```

```
flag = unpad(pt, 16)
print(flag)
```

11 Summary

The RNG output is SHA256(`str(cnt)`) and the counter update is linear mod 2^{256}. This lets us "rewind" the counter using a modular knapsack, force nonce reuse, recover k from two signatures, and finally decrypt the AES-CBC ciphertext from `exchange` using $(kQ).x$.