```
from ecdsa import SigningKey, NIST256p
import hashlib

# Generate ECC private key
private_key = SigningKey.generate(curve=NIST256p)
public_key = private_key.verifying_key

# Fixed nonce (VULNERABILITY)
k_fixed = 1234567890

def sign_message(message, k):
    hash_msg = int(hashlib.sha256(message.encode()).hexdigest(), 16)
    sig = private_key.sign_digest_deterministic(hash_msg.to_bytes(32,
'big'), sigencode=lambda r, s, order: (r, s),
extra_entropy=k.to_bytes(32, 'big'))
    return sig

msg1 = "Hello, ECC!"
msg2 = "Breaking ECDSA!"

sig1 = sign_message(msg1, k_fixed)
sig2 = sign_message(msg2, k_fixed)

print(f"Signature 1: {sig1}")
print(f"Signature 2: {sig2}")
```

```
Signature 1:
(79301806835953528494497374998776186518433086203613116005032526061401151819585,
10757283435450566206271500961997125732660643882937188409168727848940261862711)
Signature 2:
(80936953990508450652543603688920026378899334230576377345702412622510190644 95,
88138007953987146784351103486306725037050879812078490941998461654485366143017)
```

```
from ecdsa.numbertheory import inverse_mod
from ecdsa.ecdsa import generator_256

def recover_private_key(sig1, sig2, msg1, msg2, order):
    r, s1 = sig1
    _, s2 = sig2

    # Convert messages to hash values
    hash1 = int(hashlib.sha256(msg1.encode()).hexdigest(), 16)
    hash2 = int(hashlib.sha256(msg2.encode()).hexdigest(), 16)

    # Calculate private key: d = ((s1 - s2)^-1 * (hash1 - hash2)) mod
n
    s_diff = (s1 - s2) % order
```

```
    h_diff = (hash1 - hash2) % order

    d = (inverse_mod(s_diff, order) * h_diff) % order
    return d

# Curve order
n = generator_256.order()

# Recover private key
private_key_recovered = recover_private_key(sig1, sig2, msg1, msg2, n)
print(f"Recovered Private Key: {private_key_recovered}")
```

```
Recovered Private Key:
534126557677557293934000703110487727761694380515311106982797302579420691039 43
```

```
from ecdsa import SigningKey, NIST256p

recovered_private_key =
9490188564452879580787753250919961296412464318024422249216506408679255312 7509

private_key = SigningKey.from_secret_exponent(recovered_private_key,
curve=NIST256p)

public_key = private_key.verifying_key

message = "Testing private key verification"
signature = private_key.sign(message.encode())

is_valid = public_key.verify(signature, message.encode())

print(f"Signature is valid: {is_valid}")
```

```
Signature is valid: True
```