Write-Up - Vending Machine - Ectario

PwnMe: Crypto - Hard **Author: Ectario**

Table of Contents

1. **Chall Information**
   - Script
2. **Known Infos**
   - Information from the `chall.py`
   - Note on FRP256v1
   - Signature Generation Logic (ECDSA)
   - Encrypted Flag
   - Interactions with the Challenge:
     - Show Credits
     - Show Currency
     - Get Encrypted Flag
     - Request Signatures
     - Buy Credits
3. **Analyse**
   - User input with alea_1 & alea_2
   - Collision with hash function
   - Exploit common unknown prefix on nonce (part 1.)
   - Exploit common unknown prefix on nonce (part 2.)
   - Getting more signatures
4. **Final Exploit**
   - Code implementation
5. **Inspiration**
6. **Annexes**
   - Why the sleep is here?
   - Why this system of currency?
   - Some resources for LLL beginner

Chall Information

Script

From the `chall.py`:

```
#!/usr/bin/env python3
from tinyec.ec import SubGroup, Curve
from Crypto.Util.Padding import pad
from Crypto.Cipher import AES
from json import loads, dumps
from hashlib import sha3_256
from random import choice
from os import urandom
from flag import FLAG
import secrets
import time

class SignatureManager:
    def __init__(self):
        # FRP256v1 Parameters
        self.p = 0xf1fd178c0b3ad58f10126de8ce42435b3961adbcabc8ca6de8fcf353d86e9c03
        self.a = 0xf1fd178c0b3ad58f10126de8ce42435b3961adbcabc8ca6de8fcf353d86e9c00
        self.b = 0xee353fca5428a9300d4aba754a44c00fdfec0c9ae4b1a1803075ed967b7bb73f
        self.Gx = 0xb6b3d4c356c139eb31183d4749d423958c27d2dcaf98b70164c97a2dd98f5cff
        self.Gy = 0x6142e0f7c8b204911f9271f0f3ecef8c2701c307e8e4c9e183115a1554062cfb
        self.n = 0xf1fd178c0b3ad58f10126de8ce42435b53dc67e140d2bf941ffdd459c6d655e1
        self.h = 1

        subgroup = SubGroup(self.p, (self.Gx, self.Gy), self.n, self.h)
        self.curve = Curve(self.a, self.b, subgroup, name="CustomCurve")

        self.P = self.curve.g
        self.d = int.from_bytes(urandom(32), "big") % self.n
        self.Q = self.d * self.P
        self.salt = int.from_bytes(urandom(32), "big") % self.n

    def inverse(self, a, n):
        return pow(a, -1, n)

    def gen_sign(self, m: bytes, alea_1, alea_2):
        a = int(alea_1)
        b = int(alea_2)
        assert a**2 < b**2 < 2**120 - 1
        c = (hash(a) - hash(b)) * int.from_bytes(urandom(32), "big") ^ 0xFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
        randomized_main_part_l = 249
        randomized_part = ""
        for _ in range(256 - randomized_main_part_l):
            randomized_part += choice(bin(c).split("0b")[1])
        parity = int(randomized_part, 2) % 2
        randomized_part = bin(self.salt ^ int(randomized_part, 2))[-(256 - randomized_main_part_
```

```
l):]
        k = 0xFF000000000000000000000000000000000000000000000000000000000000FF
^ int(randomized_part + bin(secrets.randbits(randomized_main_part_l)).split("0b")[1].zfill(random
ized_main_part_l) if parity else bin(secrets.randbits(randomized_main_part_l)) + randomized_par
t, 2)
        e = int.from_bytes(sha3_256(m).digest(), "big")
        R = k * self.P
        r = R.x % self.n
        assert r != 0
        s = (self.inverse(k, self.n) * (e + self.d * r)) % self.n
        return r, s, int.from_bytes(m, "big")

    def verify(self, m: bytes, r: int, s: int):
        e = int.from_bytes(sha3_256(m).digest(), "big")
        assert 0 < r < self.n and 0 < s < self.n
        w = self.inverse(s, self.n)
        u1 = (e * w) % self.n
        u2 = (r * w) % self.n
        P_ = u1 * self.P + u2 * self.Q
        return r == P_.x % self.n


class Server:
    def __init__(self):
        self.signature_manager = SignatureManager()
        self.credits = 1
        self.signatures = []
        self.credit_currency = 0
        key = sha3_256(self.signature_manager.d.to_bytes(32, "big")).digest()[:16]
        self.iv = urandom(16)
        cipher = AES.new(key, IV=self.iv, mode=AES.MODE_CBC)
        self.encrypted_flag = cipher.encrypt(pad(FLAG.encode(), 16)).hex()
        self.used_credit = 0

    def show_credits(self):
        return {"credits": self.credits}

    def show_currency(self):
        return {"currency": self.credit_currency}

    def get_encrypted_flag(self):
        return {"encrypted_flag": self.encrypted_flag, "iv": self.iv.hex()}

    def get_new_signatures(self, alea_1, alea_2):
        if self.credits > 0:
            self.credits -= 1
            self.used_credit += 1
            new_signatures = []
            for i in range(10):
```

```
            m = sha3_256(b"this is my lovely loved distributed item " + str(i+10*self.used_credit).e
ncode()).digest()
            r,s,_ = self.signature_manager.gen_sign(m, alea_1, alea_2)
            new_signatures.append((r, s))
            self.signatures.append((m.hex(), r, s))
        # ...Yeah, it's long, but it's just like vending machines... the cans take forever to drop, it's
maddening...
        time.sleep(90)
        return {"signatures": new_signatures}
    else:
        return {"error": "Not enough credits."}

  def verify_proof_of_ownership(self, owner_proofs):
    owner_proofs = [tuple(item) for item in owner_proofs]
    if len(set(owner_proofs)) != self.credit_currency:
        return False
    for owner_proof in owner_proofs:
        if not self.signature_manager.verify(bytes.fromhex(owner_proof[0]), owner_proof[1], own
er_proof[2]) or owner_proof in self.signatures:
            return False
    return True

  def buy_credit(self, owner_proofs):
    if self.verify_proof_of_ownership(owner_proofs):
        self.credits += 1
        # each credit cost more and more proofs to ensure you are the owner
        self.credit_currency += 5
        return {"status": "success", "credits": self.credits, "credit_currency": self.credit_currency}
    else:
        return {"error": f"You need {self.credit_currency} *NEW* signatures to buy more credit
s."}


def main():
  server = Server()
  print("Welcome to the signatures distributor, this is what you can do:")
  print("1. Show credits")
  print("2. Show currency")
  print("3. Get encrypted flag")
  print("4. Get signatures")
  print("5. Buy credit")
  print("6. Exit")

  while True:
    try:
        command = loads(input("Enter your command in JSON format: "))
        if "action" not in command:
            print({"error": "Invalid command format."})
            continue
```

```
            action = command["action"]

            if action == "show_credits":
                print(server.show_credits())

            elif action == "show_currency":
                print(server.show_currency())

            elif action == "get_encrypted_flag":
                print(server.get_encrypted_flag())

            elif action == "get_signatures":
                if "alea_1" not in command or "alea_2" not in command:
                    print({"error": "Invalid command format."})
                alea_1 = command["alea_1"]
                alea_2 = command["alea_2"]
                print(server.get_new_signatures(alea_1, alea_2))

            elif action == "buy_credit":
                if "owner_proofs" not in command:
                    print({"error": "Invalid command format."})
                print(server.buy_credit(command["owner_proofs"]))

            elif action == "exit":
                print({"status": "Goodbye!"})
                break

            else:
                print({"error": "Unknown action."})
        except Exception as e:
            print({"error": str(e)})

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

Known Infos

Information from the `chall.py`

```
# Elliptic Curve Parameters (FRP256v1)
p = 0xf1fd178c0b3ad58f10126de8ce42435b3961adbcabc8ca6de8fcf353d86e9c03
a = 0xf1fd178c0b3ad58f10126de8ce42435b3961adbcabc8ca6de8fcf353d86e9c00
b = 0xee353fca5428a9300d4aba754a44c00fdfec0c9ae4b1a1803075ed967b7bb73f
n = 0xf1fd178c0b3ad58f10126de8ce42435b53dc67e140d2bf941ffdd459c6d655e1
h = 1
```

Note on FRP256v1

FRP256v1 is an elliptic curve recommended by the ANSSI (Agence Nationale de la Sécurité des Systèmes d'Information), the French national cybersecurity agency. It is designed to provide a high level of security for cryptographic operations and is similar to other widely used curves such as NIST P-256. The parameters of FRP256v1 have been carefully selected to avoid known vulnerabilities, making it a trusted choice for cryptographic applications. Therefore, unless implementation mistakes are made, the curve itself is considered secure against standard cryptographic attacks.

Signature Generation Logic (ECDSA)

The signature process relies on private and public key pairs:

- **Private Key (`d`)**: A randomly generated scalar.
- **Public Key (`Q`)**: Derived as:

$$Q = d \cdot G$$

The signature for a message `m` consists of two values `r` and `s`:

$$(k.G).x \equiv r \mod (n)$$
$$k^{-1}(H(m) + r.d) \equiv s \mod (n)$$

Where:

- `n = 0xf1fd178c0b3ad58f10126de8ce42435b53dc67e140d2bf941ffdd459c6d655e1` , this is the curve order
- `k`, a random nonce
- `H(m)`, the hash of the message m
- `d`, the private key
- `G` , generator point of the curve
- `.x` means the x-coordinates of `k.G`

Encrypted Flag

The flag is encrypted using **AES-CBC** with a key derived from the private key `d`:

```
key = sha3_256(d.to_bytes(32, "big")).digest()[:16]
cipher = AES.new(key, IV=iv, mode=AES.MODE_CBC)
encrypted_flag = cipher.encrypt(pad(FLAG.encode(), 16)).hex()
```

Recovering `d` will allow decryption of the flag.

Interactions with the Challenge

1. Show Credits

Request:

```
{"action": "show_credits"}
```

Description:

This action retrieves the number of credits the user currently has. Credits are necessary to request new signatures. If the user has 0 credits, they won't be able to perform certain operations (like requesting new signatures).

Expected Output:

```
{"credits": <number>}
```

Where `<number>` indicates the number of credits remaining. Initially, this is set to `1`.

2. Show Currency

Request:

```
{"action": "show_currency"}
```

Description:

This action retrieves the "credit currency," which indicates how many *proofs of ownership* are required to purchase additional credits. The value starts at `0` and increases by `5` each time a credit is successfully purchased.

Expected Output:

```
{"currency": <number>}
```

Where `<number>` is the current number of proofs required to buy credits.

3. Get Encrypted Flag

Request:

```
{"action": "get_encrypted_flag"}
```

Description:

This action retrieves the encrypted flag and the initialization vector (IV) used for AES encryption. The encryption key is derived from the server's private key `d`, making it essential to recover `d` to decrypt the flag.

Expected Output:

```
{
  "encrypted_flag": <hex-string>,
  "iv": <hex-string>
}
```

- `encrypted_flag`: The flag encrypted with AES-CBC.
- `iv`: The initialization vector used for encryption.

4. Request Signatures

Request:

```
{"action": "get_signatures", "alea_1": "value1", "alea_2": "value2"}
```

Description:

This action generates **10 signatures** for predefined messages. The parameters `alea_1` and `alea_2` are required and are used internally for randomness. Each signature consists of `(r, s)` values, along with the corresponding hash of the message.

Conditions:

- The user must have at least 1 credit.
- Each call consumes **1 credit**.

Expected Output:

```
{"signatures": [[r1, s1], [r2, s2], ..., [r10, s10]]}
```

- The response contains 10 signature pairs, `(r, s)`, which are valid for their respective predefined messages.

5. Buy Credits

Request:

```
{"action": "buy_credit", "owner_proofs": [...]}
```

Description:

This action allows the user to buy additional credits by providing valid **proofs of ownership**. Each proof is a tuple `(m, r, s)` corresponding to a message `m` and its signature `(r, s)`. Proofs must:

1. Be valid signatures for their respective messages.
2. Not match any previously issued signatures.
3. Equal the required number of proofs (`credit_currency`).

Conditions:

- `owner_proofs`: An array of proofs, where each proof is structured as:

  ```
  [<message>, <r>, <s>]
  ```

Expected Output:

If successful:

```
{"status": "success", "credits": <new_credits>, "credit_currency": <new_currency>}
```

- `credits`: The updated number of credits.
- `credit_currency`: The new number of proofs required for the next credit purchase (increases by 5 each time).

If unsuccessful:

```
{"error": "You need <currency> *NEW* signatures to buy more credits."}
```

Analyse

User input with alea_1 & alea_2

The first thing to note is that **k** is generated based on user inputs, which inherently introduces a bias.

However, there are conditions on these inputs:

```
assert a**2 < b**2 < 2**120 - 1
```

This means that **a** and **b** cannot be equal, and neither **a** nor **b** can exceed 60 bits in length. Looking further into the logic, we can see that the variable **c** is computed as the product of the difference between the hashes of **a** and **b**.

```
c = (hash(a) - hash(b)) * int.from_bytes(urandom(32), "big") ^ 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

This implies that if **a** and **b** collide (i.e., their hashes are identical), then **c** will remain constant (and equal in binary to `111...111`), even though the factor was originally designed to be highly random.

Later in the code,

```
randomized_main_part_l = 249
randomized_part = ""
    for _ in range(256 - randomized_main_part_l):
        randomized_part += choice(bin(c).split("0b")[1])
```

the variable **randomized_part** is created by randomly selecting bits from **c**. However, since **c** is effectively equal to `111...111` (as it is XORed with `0xfff...fff`), **randomized_part** will consist entirely of 1s in binary.

It's also important to observe that parity is handled **before** the XOR operation with the salt. This means that, given a collision in the hashes of **a** and **b**, the parity will always resolve to 1.

```
    def gen_sign(self, m: bytes, alea_1, alea_2):
        a = int(alea_1)
        b = int(alea_2)
        assert a**2 < b**2 < 2**120 - 1
        c = (hash(a) - hash(b)) * int.from_bytes(urandom(32), "big") ^ 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
        randomized_main_part_l = 249
        randomized_part = ""
        for _ in range(256 - randomized_main_part_l):
            randomized_part += choice(bin(c).split("0b")[1])
--------> parity = int(randomized_part, 2) % 2 <--------------------------
        randomized_part = bin(self.salt ^ int(randomized_part, 2))[-(256 - randomized_main_part_l):]
        k = 0xFF0000000000000000000000000000000000000000000000000000000000000000FF ^ i
        e = int.from_bytes(sha3_256(m).digest(), "big")
```

```
        R = k * self.P
        r = R.x % self.n
        assert r != 0
        s = (self.inverse(k, self.n) * (e + self.d * r)) % self.n
        return r, s, int.from_bytes(m, "big")
```

Finally, after the XOR with the salt (which is generated only **once** at the start of the program), the **randomized_part** will always result in a constant value for every signature.

This constant value serves as a prefix, by the parity and the following line:

```
k = 0xFF0000000000000000000000000000000000000000000000000000000000000FF ^ int(ra
```

So, a collision between the hashes of **a** and **b** would result in a consistent 7-bit prefix (unknown because of the salt) for each nonce.

Collision with hash function

In Python, the native hash function exhibits a specific behavior: if the input value is smaller than Python's internal modulo threshold, the function simply returns the input itself as the hash. This behavior bypasses any additional hashing computation, making the function effectively act as an identity function for values below the threshold. So where can we find the collision since `a` and `b` are less than 60 bits and the architecture used is 64 bits so the modulo is bigger than both `a` and `b`?

Here: https://github.com/python/cpython/blob/v3.6.0/Objects/longobject.c#L3004

The hash of -1 is equivalent to the hash of -2 because (after some research) it seems that -1 as a return value for the hash function is reserved for errors in some cpython things. Therefore, it is explicitly designed to make the function return -2 instead of -1. *Weird behavior..*

Now that we know the 7-bit prefix is fixed and we know how to make it fixed, let's see how we can exploit this on the ECDSA scheme.

Exploit common unknown prefix on nonce (part 1.)

Alright, let's start by taking the equation we previously mentioned (by using the `s` value relation from the ECDSA scheme).

The goal is to transform the problem into an HNP instance, as outlined in the paper that served as inspiration (link to paper chapter 4 & this insane video). To provide better understanding, I will just add a lil' detail about the intuition behind the paper/video for the lattice construction:

$$k^{-1}(H(m) + r.d) \equiv s \mod (n)$$

$$\iff (H(m) + r.d) \equiv k.s \mod (n)$$

$$\iff s^{-1}(H(m) + r.d) \equiv k \mod (n)$$

$$\iff s^{-1}(H(m) + r.d) - k \equiv 0 \mod (n)$$

$$\iff k - s^{-1}r.d - s^{-1}H(m) \equiv -0 \mod (n)$$

Using the latest expression, we can construct a lattice (by matching the equation to that of the HNP) which will be used to find the nonces (that is well described in the video).

Also, to gain some intuition about the desired vector, here's an interesting observation:

The left side vector of the matrix contains the coefficients, which produce the vector on the right-hand side (RHS) that indeed holds the nonces. This resulting vector (RHS), therefore, belongs to the **space spanned by the following matrix**:

$$\begin{bmatrix} v_1 & v_2 & \dots & d & 1 \end{bmatrix} \cdot \begin{bmatrix} n & 0 & 0 & 0 & 0 \\ 0 & n & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots \\ r_1.{s_1}^{-1} & r_2.{s_2}^{-1} & \dots & B/n & 0 \\ H(m)_1.{s_1}^{-1} & H(m)_2.{s_2}^{-1} & \dots & 0 & B \end{bmatrix}$$

$$= \begin{bmatrix} v_1.n + 0 + d.r_1.s_1^{-1} + H(m)_1.{s_1}^{-1} = s_1^{-1}(H(m)_1 + r_1.d) \mod n \\ 0 + v_2.n + d.r_2.s_2^{-1} + H(m)_2.{s_2}^{-1} = s_2^{-1}(H(m)_2 + r_2.d) \mod n \\ \dots \\ d.\frac{B}{n} \\ B \end{bmatrix} = \begin{bmatrix} k_1 \\ k_2 \\ \dots \\ d.\frac{B}{n} \\ B \end{bmatrix}$$

Where:

- `v_i` are the modulo factor
- `B` the weight (used for Kannan's embedding, which is used here to translate the CVP from HNP into a SVP)
- others variables are the same as before

So, in principle *(i.e., applying the paper logic)*, by performing a reduction (using LLL or BKZ) on the lattice, we can obtain the vector we are interested in.

The issue, however, is that it requires the nonce to be small relative to `n` *(i.e., it must have a certain number of zero bits when represented in binary and left-padded to 256 bits)*. In our challenge, we only know that the nonce has a static prefix for each instance, but this prefix is not necessarily zero.

So, how should we proceed?

Exploit common unknown prefix on nonce (part 2.)

Let's revisit the information that makes the implementation vulnerable.

We know that the nonce has a FIXED 7-bit prefix, which means that if we have, for example *(they're small here for educational purposes, but in the actual implementation, they are 256 bits long)*, three nonces `K_a`, `K_b`, and `K_c`, with:

$$k_a = 100101110101010110$$
$$k_b = 100101100110001010$$
$$k_c = 100101111111000111$$

We know that we have a method (from part 1) that works if we aim to obtain values with an MSB entirely filled with 0s.

To be in this state, we can focus not on the nonces themselves but on their differences, such as `K_a - K_b` and `K_c - K_b` :

$$k_a - k_b = 000000001111001100$$
$$k_c - k_b = 000000011000111101$$
$$k_b = 100101100110001010$$

We observe that by focusing not on the nonces themselves, but on the difference between each nonce and one of the nonces (any one of them), it effectively sets the shared prefix to 0.

The objective now is to find a matrix representing one of the bases of the lattice, which allows us to recover these differences. To achieve this, we need to formulate the expression of a nonce difference to align it with an HNP instance. So we already have (from part 1.):

$$s_1^{-1}(H(m)_1 + r_1.d) \equiv k_1 \mod (n)$$
$$s_2^{-1}(H(m)_2 + r_2.d) \equiv k_2 \mod (n)$$

$$\iff s_1^{-1}(H(m)_1 + r_1.d) - s_2^{-1}(H(m)_2 + r_2.d) \equiv k_1 - k_2 \mod (n)$$
$$\iff s_1^{-1}H(m)_1 + s_1^{-1}r_1.d - s_2^{-1}H(m)_2 - s_2^{-1}r_2.d \equiv k_1 - k_2 \mod (n)$$
$$\iff (s_1^{-1}H(m)_1 - s_2^{-1}H(m)_2) + (s_1^{-1}r_1 - s_2^{-1}r_2).d \equiv k_1 - k_2 \mod (n)$$

And to map this to an HNP instance, it is sufficient to reformulate it as:

$$\iff -(k_1 - k_2) + (s_1^{-1}H(m)_1 - s_2^{-1}H(m)_2) + (s_1^{-1}r_1 - s_2^{-1}r_2).d \equiv 0 \mod (n)$$
$$\iff -(k_1 - k_2) + (s_1^{-1}r_1 - s_2^{-1}r_2).d + (s_1^{-1}H(m)_1 - s_2^{-1}H(m)_2) \equiv 0 \mod (n)$$
$$\iff (k_2 - k_1) - (s_1^{-1}r_1 - s_2^{-1}r_2).d - (s_1^{-1}H(m)_1 - s_2^{-1}H(m)_2) \equiv -0 \mod (n)$$

Lil' reminder (again, from this video), an HNP instance can be described as:

$$k_i - t_i.d - a_i \equiv 0 \mod (n)$$

Where:

- `k_i` and `d` are unknown
- `t_i` , `n` and `a_i` are known

So the mapping here is:

$$k_i = (k_2 - k_1)$$
$$t_i = (s_1^{-1}r_1 - s_2^{-1}r_2)$$
$$d = d$$
$$a_i = (s_1^{-1}H(m)_1 - s_2^{-1}H(m)_2)$$

Let's reconstruct the matrix!

We will also carefully verify that the matrix has a linear combination that allows generating a vector containing the nonce differences.

(Here, the differences will always be calculated using the last nonce for simplicity. Note that the generated space loses one dimension because the (`j-2`)-th column, with `j` being the number of signatures, will equal 0)

$$\begin{bmatrix} v_1 & v_2 & \dots & d & 1 \end{bmatrix} \cdot \begin{bmatrix} n & 0 & 0 & 0 & 0 \\ 0 & n & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots \\ s_1^{-1}r_1 - s_j^{-1}r_j & r_j.s_j^{-1} - s_j^{-1}r_j & \dots & B/n & 0 \\ H(m)_1.s_1^{-1} - s_j^{-1}H(m)_j & H(m)_j.s_j^{-1} - s_j^{-1}H(m)_j & \dots & 0 & B \end{bmatrix}$$

$$= \begin{bmatrix} v_1.n + 0 + d.(s_1^{-1}r_1 - s_j^{-1}r_j) + H(m)_1.s_1^{-1} - s_j^{-1}H(m)_j \\ 0 + v_2.n + d.(s_2^{-1}r_j - s_j^{-1}r_j) + H(m)_2.s_2^{-1} - s_j^{-1}H(m)_j \\ \dots \\ d.\frac{B}{n} \\ B \end{bmatrix}$$

$$= \begin{bmatrix} (s_1^{-1}H(m)_1 - s_j^{-1}H(m)_j) + (s_1^{-1}r_1 - s_j^{-1}r_j).d \mod n \\ (s_2^{-1}H(m)_2 - s_j^{-1}H(m)_j) + (s_2^{-1}r_2 - s_j^{-1}r_j).d \mod n \\ \dots \\ d.\frac{B}{n} \\ B \end{bmatrix}$$

$$= \begin{bmatrix} k_1 - k_j \\ k_2 - k_j \\ \dots \\ d.\frac{B}{n} \\ B \end{bmatrix}$$

As before, we simply need to apply LLL or BKZ to the matrix to obtain, among the resulting short vectors, the one containing the nonce differences.

From the difference in nonces, only a bit of modular arithmetic is needed to derive the private key:

$$\begin{aligned} & (s_1^{-1}H(m)_1 - s_j^{-1}H(m)_j) + (s_1^{-1}r_1 - s_j^{-1}r_j).d \equiv k_1 - k_j \mod (n) \\ \iff & (s_1^{-1}r_1 - s_j^{-1}r_j).d \equiv (k_1 - k_j) - (s_1^{-1}H(m)_1 - s_j^{-1}H(m)_j) \mod (n) \\ \iff & d \equiv (s_1^{-1}r_1 - s_j^{-1}r_j)^{-1}.((k_1 - k_j) - (s_1^{-1}H(m)_1 - s_j^{-1}H(m)_j)) \mod (n) \end{aligned}$$

All's well that ends well... or is it?

We are allowed 10 signatures because we start with 1 credit. Luckily, since the currency initially has a value of 0, we can regain a credit and obtain 10 additional signatures. **BUT WAIT!** Once reduced, the matrix does not seem to reveal any nonce differences. It appears there aren't enough constraints between the basis vectors for the reduced matrix to yield, as a short vector, the one containing the nonce difference.

How can we increase the dimension of the matrix, then?

Getting more signatures

So, let's restart with the basics. An ECDSA signature consists of a tuple `(r,s)`, where:

$$\begin{aligned} x &\equiv r \mod (n) \\ k^{-1}(H(m) + r.d) &\equiv s \mod (n) \end{aligned}$$

The public key is `Q = d · G`, where `G` is the curve's generator point. During verification, a point `R` is recomputed:

$$R = u_1 \cdot G + u_2 \cdot Q$$

where:

$$r \cdot s^{-1} \equiv u_2 \mod (n)$$
$$H(m) \cdot s^{-1} \equiv u_1 \mod (n)$$

The signature is valid if:

$$x \equiv r \mod (n)$$

Now, something that will be a game-changer for us:
For any valid signature
`(r,s),` there exists another valid signature `(r,s')`, where:

$$s\prime = n - s$$

Proof of Validity:

$$(s\prime)^{-1} \equiv (n-s)^{-1} \mod (n)$$
$$(n-s)^{-1} \equiv -s^{-1} \mod (n)$$

Let's compute u_1 and u_2 with s':

$$u_1\prime \equiv H(m) \cdot (s\prime)^{-1} \equiv H(m) \cdot (-s^{-1}) \equiv -H(m) \cdot s^{-1} \equiv -u_1 \mod (n)$$
$$u_2\prime \equiv r \cdot (s\prime)^{-1} \equiv r \cdot (-s^{-1}) \equiv -r \cdot s^{-1} \equiv -u_2 \mod (n)$$

Substituting these into the elliptic curve equation for `R`, we find:

$$R\prime = u_1\prime \cdot G + u_2\prime \cdot Q = (-u_1) \cdot G + (-u_2) \cdot Q = -(u_1 \cdot G + u_2 \cdot Q) = -R$$

The point `R'` is the negation of `R`, then its x-coordinate remains unchanged: `R'_x = R_x`. Consequently, the verification condition `R'_x mod n = r` is still **satisfied**.

Eureka! We can actually obtain more than 20 signatures in total:

The first 10 are free (+ we will still have 1 credit left), and the next 10 cost 5 signatures, then 10, then 15, and finally 20. This means we can ultimately obtain **60 signatures**, calculated as follows:

10 (free credit with currency = 0) + 10 (from the starting credit) + 10 (purchased for 5) + 10 (purchased for 10) + 10 (purchased for 15) + 10 (purchased for 20).

Note: *It doesn't seem possible to exceed 60 signatures under this "inflation" system. Even if signatures are saved early on, after the 5 credit purchases (at costs of 0, 5, 10, 15, and 20 currency), there are no longer enough signatures available to forge additional credits.*

From this point, we have enough signatures to exploit the previously explained vulnerability and Capture The Flag.

Final Exploit

```
from sage.all import *
from pwn import process, remote
from hashlib import sha3_256
```

```
from Crypto.Util.number import *
from Crypto.Util.Padding import unpad
from Crypto.Cipher import AES
from json import dumps
from tqdm import tqdm
import ast

n = 0xf1fd178c0b3ad58f10126de8ce42435b53dc67e140d2bf941ffdd459c6d655e1

# proc = process("./server.py")
proc = remote("localhost", 1337)
proc.recvuntil(b"format: ")

signatures = []
forged_signatures = []
used_credits = 0

# bytes() with single quoted names in json format
def json_read(inp):
    inp = inp.strip()
    inp = ast.literal_eval(inp.decode())
    return inp

def show_credits():
    proc.sendline(dumps({"action": "show_credits"}))
    res = json_read(proc.recvline())
    clean()
    return res

def show_currency():
    proc.sendline(dumps({"action": "show_currency"}))
    res = json_read(proc.recvline())
    clean()
    return res

def get_encrypted_flag():
    proc.sendline(dumps({"action": "get_encrypted_flag"}))
    res = json_read(proc.recvline())
    clean()
    return res

def get_new_signatures(alea_1, alea_2):
    proc.sendline(dumps({"action": "get_signatures", "alea_1": alea_1, "alea_2": alea_2}))
    res = json_read(proc.recvline())
    clean()
    return res

def wrapper_get_new_signatures():
    global used_credits
```

```
    global forged_signatures
    global signatures
    alea_1 = "-1"
    alea_2 = "-2"
    new_signatures = get_new_signatures(alea_1, alea_2)
    used_credits += 1
    for i, (r, s) in enumerate(new_signatures["signatures"]):
        m = sha3_256(b"this is my lovely loved distributed item " + str(i+10*used_credits).encode()).di
        e = int.from_bytes(sha3_256(m).digest(), "big")
        signatures.append((r, s, e))
        forged_signatures.append((m.hex(), r, n-s))
    return signatures

def buy_credit():
    global forged_signatures
    currency = int(show_currency()["currency"])
    proc.sendline(dumps({"action": "buy_credit", "owner_proofs": forged_signatures[:currency]}))
    forged_signatures = forged_signatures[currency:]
    res = json_read(proc.recvline())
    clean()
    return res

def clean():
    proc.recvuntil(b"format: ")

def decrypt_flag(private_key):
    encrypted_flag = bytes.fromhex(get_encrypted_flag()["encrypted_flag"])
    iv = bytes.fromhex(get_encrypted_flag()["iv"])
    key = sha3_256(private_key.to_bytes(32, "big")).digest()[:16]
    cipher = AES.new(key, IV=iv, mode=AES.MODE_CBC)
    return cipher.decrypt(encrypted_flag)


wrapper_get_new_signatures()
buy_credit()
buy_credit()
wrapper_get_new_signatures()
wrapper_get_new_signatures()
buy_credit()
wrapper_get_new_signatures()
buy_credit()
wrapper_get_new_signatures()
buy_credit()
wrapper_get_new_signatures()

NB_MSG = len(signatures)

last_r = signatures[-1][0]
last_s = signatures[-1][1]
```

```
inv_last_s = inverse(last_s, n)
last_e = signatures[-1][2]

t_values = []
a_values = []

for r, s, e in signatures[:-1]:
    inv_s = inverse(s, n)
    t = ( (inv_s * r) - (last_r * inv_last_s) ) % n
    a = ( (inv_s * e) - (last_e * inv_last_s) ) % n
    t_values.append(t)
    a_values.append(a)

# sage: m = 60
# sage: int(((log(n,2) * (m-1)) / m) - (log(m,2)/2)) from https://eprint.iacr.org/2019/023.pdf
# 248

B = 2**248
f = QQ
NB_MSG -= 1 # last signature used to cancel the unknown prefix

diagonal_block = diagonal_matrix(f, [n] * NB_MSG)
t_vector = Matrix(f, t_values)
a_vector = Matrix(f, a_values)

# Assemble the full matrix
M = block_matrix(f, [
    [diagonal_block, zero_matrix(f, NB_MSG, 2)],
    [t_vector, Matrix(f, [f(B) / f(n), 0])],
    [a_vector, Matrix(f, [0, f(B)])],
])

M_reduced = M.LLL()
s0 = signatures[0][1]
r0 = signatures[0][0]
e0 = signatures[0][2]

for i in range(len(list(M_reduced))):
    k_diff_test = int(M_reduced[i][0])
    d_test = inverse((last_r*s0 - r0*last_s), n) * (e0*last_s - last_e*s0 - s0*last_s*k_diff_test) % n
    test_dec_flag = decrypt_flag(d_test)
    if b"pwnme{" in test_dec_flag.lower():
        print(unpad(test_dec_flag, 16))
        break
```

Inspiration

The idea for this challenge originates from Section 4.3 of this paper: link to paper

Annexes

Why the sleep is here?

The `sleep` function here is intended to discourage players from brute-forcing the 7 bits until they are all zeros (a probability of 0.0078125). By simply brute-forcing, the second part, "Exploit common unknown prefix on nonce (part 2)", becomes unnecessary.

With a 90-second sleep, this means that, on average, it would take in average ~20 hours to get the flag through brute force. Considering the CTF lasts 48 hours and there is a limited number of instances per team, it is technically feasible but would require a significant sacrifice. I didn't set a longer sleep because I consider it a legitimate way to get the flag, as the 7-bit vulnerability step remains (an optional) way to break the system. After all, the probability of all 7 bits being zero isn't that small, but the waiting time serves as a minor penalty—likely making it impossible to achieve first blood, for example.

Why this system of currency?

As explained in the note at the end of "Getting more signatures", it is not possible to obtain more than 60 signatures. This system was designed because, based on simulations, it takes between 50 and 60 signatures for the desired short vector to consistently appear after matrix reduction.

And I wanted to point out the property of ECDSA and the possibility of forging signatures.

***EDIT**: Kudos to `Nikost` for mentionning the unintended! In the end, unfortunately, it was possible to forge way more signatures for the `owner_proof` since all it took was creating variations of the message with uppercase hex characters, which bypassed the check... That is my bad 👀*

Some resources for LLL beginner

Below are some blog posts and papers that I find particularly helpful for anyone who hasn't used LLL yet:

- https://theblupper.github.io/blog/posts/lattices/
- https://magicfrank00.github.io/writeups/posts/lll-to-solve-linear-equations
- https://eprint.iacr.org/2023/032.pdf