Blessed

23 minutes to read

Contents

This challenge was made by me for Hack The Box. We are given the Python source of the server that contains the flag:

```
import json

from eth_typing import BLSPrivateKey, BLSPubkey, BLSSignature
from secrets import randbelow
from typing import Dict, Generator, List

from Crypto.PublicKey import ECC

from py_ecc.bls.ciphersuites import G2ProofOfPossession as bls
from py_ecc.bls.g2_primitives import pubkey_to_G1
from py_ecc.bls.point_compression import decompress_G1
from py_ecc.bls.typing import G1Compressed
```

```python
from py_ecc.optimized_bls12_381.optimized_curve import add, curve_order, G1, multiply, neg, normalize


try:
    with open('flag.txt') as f:
        FLAG = f.read().strip()
except FileNotFoundError:
    FLAG = 'HTB{f4k3_fl4g_f0r_t3st1ng}'


def rng() -> Generator[int, None, None]:
    seed = randbelow(curve_order)
    Gx = 0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296
    Gy = 0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5
    G = ECC.EccPoint(Gx, Gy, curve='p256')
    B = ECC.generate(curve='p256').pointQ
    W0 = G * seed + B
    Wn = W0

    while True:
        Wn += G
        yield Wn.x >> 32
        yield Wn.y >> 32


class Robot:
    def __init__(self, robot_id: int, verified: bool = True):
        self.robot_id: int           = robot_id
        self.verified: bool          = verified
        self.pk:       BLSPubkey     = BLSPubkey(b'')
        self._sk:      BLSPrivateKey = BLSPrivateKey(0)

        if self.verified:
            self._sk = BLSPrivateKey(randbelow(curve_order))
            self.pk = bls.SkToPk(self._sk)

    def json(self) -> Dict[str, str]:
        return {'robot_id': hex(self.robot_id)[2:], 'pk': self.pk.hex()}
```

```
class SuperComputer:
    def __init__(self, n: int):
        self.rand:   Generator[int, None, None] = rng()
        self.robots: List[Robot]                = []

        for _ in range(n):
            self.create()

    def _find_robot_by_id(self, robot_id: int) -> Robot | None:
        for r in self.robots:
            if r.robot_id == robot_id:
                return r

    def create(self) -> Dict[str, str]:
        r = Robot(next(self.rand))
        self.robots.append(r)
        return {'msg': 'Do not lose your secret key!', 'sk': hex(r._sk)[2:],
**r.json()}

    def join(self, pk: BLSPubkey) -> Dict[str, str]:
        if not pk:
            return {'error': 'This command requires a public key'}

        r = Robot(next(self.rand), verified=False)
        r.pk = pk
        self.robots.append(r)
        return {'msg': 'Robot joined but not verified', 'robot_id':
hex(r.robot_id)[2:]}

    def verify(self, robot_id: int) -> Dict[str, str]:
        r = self._find_robot_by_id(robot_id)

        if not r:
            return {'error': 'No robot found'}

        if r.verified:
            return {'error': 'User already verified'}
```

```
        print(json.dumps({'msg': 'Prove that you have the secret key that
corresponds to your public key: pk = sk * G1'}))

        Pk = pubkey_to_G1(r.pk)

        for _ in range(64):
            C_hex = input('Take a random value x and send me C = x * G1
(hex): ')
            C = decompress_G1(G1Compressed(int(C_hex, 16)))

            if next(self.rand) & 1:
                x = int(input('Give me x (hex): '), 16)

                if normalize(multiply(G1, x)) != normalize(C):
                    return {'error': 'Proof failed!'}
            else:
                sk_x = int(input('Give me (sk + x) (hex): '), 16)

                if normalize(add(multiply(G1, sk_x), neg(Pk))) !=
normalize(C):
                    return {'error': 'Proof failed!'}

        r.verified = True
        return {'msg': 'Robot verified'}

    def list(self, robot_id: int, sig: BLSSignature) -> Dict[str, str] |
List[Dict[str, str]]:
        if not sig:
            return {'error': 'This command requires a signature'}

        r = self._find_robot_by_id(robot_id)

        if not r:
            return {'error': 'No robot found'}

        if not bls.Verify(r.pk, b'list', sig):
            return {'error': 'Invalid signature'}

        return [r.json() for r in self.robots]
```

```
    def unveil_secrets(self, agg_sig: BLSSignature) -> Dict[str, str]:
        agg_pk = [r.pk for r in self.robots if r.verified]

        if not agg_sig:
            return {'error': 'This command requires an aggregated
signature'}
        elif bls.FastAggregateVerify(agg_pk, b'unveil_secrets', agg_sig):
            return {'msg': 'Secrets have been unveiled!', 'flag': FLAG}
        else:
            return {'error': 'Invalid aggregated signature'}

    def help(self) -> Dict[str, str]:
        return {
            'help':           'Show this panel',
            'create':         'Generate a new robot, already verified',
            'join':           'Add a new robot, given a public key and a
signature',
            'verify':         'Start interactive process to verify a robot
given an ID',
            'list':           'Return a list of all existing robots',
            'unveil_secrets': 'Show the secrets given an aggregated
signature of all registered robots',
            'exit':           'Shutdown the SuperComputer',
        }

    def run_cmd(self, data: Dict[str, str]) -> Dict[str, str] |
List[Dict[str, str]]:
        cmd      = data.get('cmd')
        pk       = BLSPubkey(bytes.fromhex(data.get('pk', '')))
        sig      = BLSSignature(bytes.fromhex(data.get('sig', '')))
        robot_id = int(data.get('robot_id', '0'), 16)

        if cmd == 'create':
            return self.create()
        elif cmd == 'join':
            return self.join(pk)
        elif cmd == 'verify':
            return self.verify(robot_id)
        elif cmd == 'list':
            return self.list(robot_id, sig)
```

```
        elif cmd == 'unveil_secrets':
            return self.unveil_secrets(sig)
        elif cmd == 'exit':
            return {'error': 'exit'}

        return self.help()


def main():
    print('Welcome! You have been invited to use our SuperComputer, which is very powerful and totally secure. Only sophisticated robots are able to use it, so you need to create a robot to interact with the SuperComputer or maybe join an existing one. The key to our success is that critical operations need the approval of all registered robots. Hackers cannot beat our security!\n')

    crew = {
        'Architects/Engineers',
        'Explosives Experts/Demolition Specialists',
        'Hackers',
        'Stealth/Infiltration specialists',
        'Scavengers',
    }

    sc = SuperComputer(len(crew - {'Hackers'}))  # No hackers here...
    print(json.dumps(sc.help(), indent=2), end='\n\n')

    while True:
        res = sc.run_cmd(json.loads(input('> ')))
        print(json.dumps(res), end='\n\n')

        if 'error' in res:
            break


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

Source code analysis

The server defines an instance of `SuperComputer`, and we are allowed to interact with it using these commands:

```
    def help(self) -> Dict[str, str]:
        return {
            'help':           'Show this panel',
            'create':         'Generate a new robot, already verified',
            'join':           'Add a new robot, given a public key and a
signature',
            'verify':         'Start interactive process to verify a robot
given an ID',
            'list':           'Return a list of all existing robots',
            'unveil_secrets': 'Show the secrets given an aggregated
signature of all registered robots',
            'exit':           'Shutdown the SuperComputer',
        }
```

For that, we will need to create a robot, because we need a secret key to sign our commands:

```
    def create(self) -> Dict[str, str]:
        r = Robot(next(self.rand))
        self.robots.append(r)
        return {'msg': 'Do not lose your secret key!', 'sk': hex(r._sk)[2:],
**r.json()}
```

The previous method returns an instance of `Robot`:

```
class Robot:
    def __init__(self, robot_id: int, verified: bool = True):
        self.robot_id: int           = robot_id
        self.verified: bool          = verified
        self.pk:       BLSPubkey     = BLSPubkey(b'')
        self._sk:      BLSPrivateKey = BLSPrivateKey(0)

        if self.verified:
```

```
            self._sk = BLSPrivateKey(randbelow(curve_order))
            self.pk = bls.SkToPk(self._sk)

    def json(self) -> Dict[str, str]:
        return {'robot_id': hex(self.robot_id)[2:], 'pk': self.pk.hex()}
```

Observe that the robot ID is the output of this PRNG:

```
def rng() -> Generator[int, None, None]:
    seed = randbelow(curve_order)
    Gx = 0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296
    Gy = 0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5
    G = ECC.EccPoint(Gx, Gy, curve='p256')
    B = ECC.generate(curve='p256').pointQ
    W0 = G * seed + B
    Wn = W0

    while True:
        Wn += G
        yield Wn.x >> 32
        yield Wn.y >> 32
```

Also, notice that the `SuperComputer` already has a total of 4 robots:

```
    def __init__(self, n: int):
        self.rand:   Generator[int, None, None] = rng()
        self.robots: List[Robot]                = []

        for _ in range(n):
            self.create()
```

Because it is created as follows:

```
    crew = {
        'Architects/Engineers',
        'Explosives Experts/Demolition Specialists',
```

```
        'Hackers',
        'Stealth/Infiltration specialists',
        'Scavengers',
    }

    sc = SuperComputer(len(crew - {'Hackers'}))  # No hackers here...
```

Once we have a robot, we can use the secret key to sign a command. For instance, we can run `list`:

```
    def list(self, robot_id: int, sig: BLSSignature) -> Dict[str, str] |
List[Dict[str, str]]:
        if not sig:
            return {'error': 'This command requires a signature'}

        r = self._find_robot_by_id(robot_id)

        if not r:
            return {'error': 'No robot found'}

        if not bls.Verify(r.pk, b'list', sig):
            return {'error': 'Invalid signature'}

        return [r.json() for r in self.robots]
```

This method will return the ID of all robots registered in the `SuperComputer`. This could be needed to crack the PRNG.

Moreover, we can run `join`:

```
    def join(self, pk: BLSPubkey) -> Dict[str, str]:
        if not pk:
            return {'error': 'This command requires a public key'}

        r = Robot(next(self.rand), verified=False)
        r.pk = pk
        self.robots.append(r)
```

```
        return {'msg': 'Robot joined but not verified', 'robot_id':
hex(r.robot_id)[2:]}
```

With this method we can add an existing robot to the `SuperComputer` using its public key. The difference with `create` is that the new robot is not verified, so we need to run `verify`:

```
    def verify(self, robot_id: int) -> Dict[str, str]:
        r = self._find_robot_by_id(robot_id)

        if not r:
            return {'error': 'No robot found'}

        if r.verified:
            return {'error': 'User already verified'}

        print(json.dumps({'msg': 'Prove that you have the secret key that
corresponds to your public key: pk = sk * G1'}))

        Pk = pubkey_to_G1(r.pk)

        for _ in range(64):
            C_hex = input('Take a random value x and send me C = x * G1
(hex): ')
            C = decompress_G1(G1Compressed(int(C_hex, 16)))

            if next(self.rand) & 1:
                x = int(input('Give me x (hex): '), 16)

                if normalize(multiply(G1, x)) != normalize(C):
                    return {'error': 'Proof failed!'}
            else:
                sk_x = int(input('Give me (sk + x) (hex): '), 16)

                if normalize(add(multiply(G1, sk_x), neg(Pk))) !=
normalize(C):
                    return {'error': 'Proof failed!'}
```

```
            r.verified = True
            return {'msg': 'Robot verified'}
```

This method runs an interactive verification process to ensure that the public key of the robot is valid, but without disclosing the associated secret key. This is known as zero-knowledge proof (ZKP).

The target command we want to run to solve the challenge is `unveil_secrets`:

```
    def unveil_secrets(self, agg_sig: BLSSignature) -> Dict[str, str]:
        agg_pk = [r.pk for r in self.robots if r.verified]

        if not agg_sig:
            return {'error': 'This command requires an aggregated
signature'}
        elif bls.FastAggregateVerify(agg_pk, b'unveil_secrets', agg_sig):
            return {'msg': 'Secrets have been unveiled!', 'flag': FLAG}
        else:
            return {'error': 'Invalid aggregated signature'}
```

For this, we need a aggregated signature of all verified robots, which means that every robot must have signed this command. Otherwise, the verification will fail and we will not execute the command successfully.

BLS signatures

The server uses BLS signatures, which involve pairing-based cryptography. The library `py-ecc` makes a nice abstraction of what happens under the hood.

Basically, the BLS signature is using two elliptic curves $\mathbb{G}_1$ and $\mathbb{G}_2$ (known as BLS12-381), whose generator points are G_1 and G_2. These curves are pairing-friendly, which means that a pairing function can be defined here.

The idea is that two points of the elliptic curves can be associated to return another point of another elliptic curve, which is $\mathbb{G}_T$. Particularly, a pairing is a bilinear function:

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$$

The bilinear property means that the following expressions hold for $P, R \in \mathbb{G}_1$ and $Q, S \in \mathbb{G}_2$:

$$\begin{aligned} e(P + R, Q) &= e(P, Q) \cdot e(R, Q) \\ e(P, Q + S) &= e(P, Q) \cdot e(P, S) \end{aligned}$$

As a result, for scalars a and b, the following expressions also hold:

$$\begin{aligned} e(a \cdot P, b \cdot Q) &= e(P, b \cdot Q)^a \\ &= e(a \cdot P, Q)^b \\ &= e(P, Q)^{ab} \\ &= e(b \cdot P, Q)^a \\ &= e(P, a \cdot Q)^b \\ &= e(b \cdot P, a \cdot Q) \\ &= e(ab \cdot P, Q) \\ &= e(P, ab \cdot Q) \end{aligned}$$

This bilinear property allows to define the BLS signature scheme:

- A user takes a random integer $\mathbf{sk}$ and computes its public key as $\mathbf{Pk} = \mathbf{sk} \cdot G_1$

- In order to sign a message m, the message must be hashed to a point (there are several methods to do this), so $H(m) \in \mathbb{G}_2$
- The signature is $\sigma = \mathbf{sk} \cdot H(m) \in \mathbb{G}_2$

The verification process involves the pairing. The verifier only needs to compute $e(\mathbf{Pk}, H(m))$ and compare it to $e(G_1, \sigma)$. This works because:

$$\begin{aligned} e(\mathbf{Pk}, H(m)) &= e(\mathbf{sk} \cdot G_1, H(m)) \\ &= e(G_1, H(m))^{\mathbf{sk}} \\ &= e(G_1, \mathbf{sk} \cdot H(m)) \\ &= e(G_1, \sigma) \end{aligned}$$

The relevance of BLS signatures comes with the fact that signatures can be aggregated. So, instead of verifying a message signature against several public keys, it suffices to verify only an aggregated signature against an aggregated public key:

$$\begin{aligned} \sigma_{\mathbf{agg}} &= \sigma_1 + \sigma_2 + \cdots + \sigma_n \\ \mathbf{Pk}_{\mathbf{agg}} &= \mathbf{Pk}_1 + \mathbf{Pk}_2 + \cdots + \mathbf{Pk}_n \end{aligned}$$

$$e(\mathbf{Pk}_{\mathbf{agg}}, H(m)) \stackrel{?}{=} e(G_1, \sigma_{\mathbf{agg}})$$

However, there is a problem with the aggregation if an attacker can use an arbitrary public key. The attacker is able to forge an aggregated signature for a given message, that is, tell that some victim user has signed a message:

- The attacker uses the following public key:
  $\mathbf{Pk}_{\mathbf{attacker}} = \mathbf{sk}_{\mathbf{attacker}} \cdot G_1 - \mathbf{Pk}_{\mathbf{victim}}$
- The forged aggregated signature is: $\sigma_{\mathbf{forged}} = \mathbf{sk}_{\mathbf{attacker}} \cdot H(m)$

- The verifier will check that $e(\mathbf{Pk}_{\text{victim}} + \mathbf{Pk}_{\text{attacker}}, H(m))$ equals $e(G_1, \sigma_{\text{forged}})$

And it works because

$$\begin{aligned} e(\mathbf{Pk}_{\text{victim}} + \mathbf{Pk}_{\text{attacker}}, H(m)) &= e(\mathbf{Pk}_{\text{victim}} + \mathbf{sk}_{\text{attacker}} \cdot G_1 - \mathbf{Pk}_{\text{victir} \\ &= e(\mathbf{sk}_{\text{attacker}} \cdot G_1, H(m)) \\ &= e(G_1, H(m))^{\mathbf{sk}_{\text{attacker}}} \\ &= e(G_1, \mathbf{sk}_{\text{attacker}} \cdot H(m)) \\ &= e(G_1, \sigma_{\text{forged}}) \end{aligned}$$

This is known as rogue key attack. The way to prevent it is using a “Proof of Posession”, which is to verify that the user that has a public key knows the associated secret key. Normally, the user should provide a signature of the public key, which proves that the user knows the secret key associated to the public key. Other methods might involve zero-knowledge proofs.

For more information about BLS12-381 curves and BLS signatures, refer to BLS12-381 For The Rest Of Us or BLS Signatures & Withdrawals.

Zero-knowledge proof

The way to prevent the rogue key attack in this challenge is using an interactive zero-knowledge proof: We want the user to prove that they know $\mathbf{sk}$ such that $\mathbf{Pk} = \mathbf{sk} \cdot G_1$, but without disclosing $\mathbf{sk}$.

To achieve this, the server tells the user to pick a random integer x and send $C = x \cdot G_1$. Then, the server chooses randomly one of these two questions:

1. Show me x
2. Show me the result of $(\mathbf{sk} + x)$

If the question is 1., then the server can easily check that the given x satisfies that $C = x \cdot G_1$.

If the question is 2., then the server can check that the given value $(\mathbf{sk} + x)$ satisfies that $\mathbf{Pk} + C = (\mathbf{sk} + x) \cdot G_1$.

If this experiment is repeated several times and the user is not able to predict the question, then it is practically impossible to lie on the ZKP protocol:

```
    def verify(self, robot_id: int) -> Dict[str, str]:
        r = self._find_robot_by_id(robot_id)

        if not r:
            return {'error': 'No robot found'}

        if r.verified:
            return {'error': 'User already verified'}

        print(json.dumps({'msg': 'Prove that you have the secret key that
corresponds to your public key: pk = sk * G1'}))

        Pk = pubkey_to_G1(r.pk)

        for _ in range(64):
            C_hex = input('Take a random value x and send me C = x * G1
(hex): ')
            C = decompress_G1(G1Compressed(int(C_hex, 16)))

            if next(self.rand) & 1:
                x = int(input('Give me x (hex): '), 16)

                if normalize(multiply(G1, x)) != normalize(C):
                    return {'error': 'Proof failed!'}
            else:
                sk_x = int(input('Give me (sk + x) (hex): '), 16)

                if normalize(add(multiply(G1, sk_x), neg(Pk))) !=
```

```
normalize(C):
                    return {'error': 'Proof failed!'}

        r.verified = True
        return {'msg': 'Robot verified'}
```

If the attacker is able to predict the question, then the attacker is able to cheat the ZKP protocol (that is, show that they know $\mathbf{sk}$ such that $\mathbf{Pk}_{\text{attacker}} = \mathbf{sk} \cdot G_1$, when it is false):

- For question 1., the attacker simply sends the value of x such that $C = x \cdot G_1$
- But for question 2., they can compute a special $C = \mathbf{sk}_x \cdot G_1 - \mathbf{Pk}_{\text{attacker}}$, send it, and then use $\mathbf{sk}_x$ as $(\mathbf{sk} + x)$. The server will check that $\mathbf{Pk}_{\text{attacker}} + C = \mathbf{sk}_x \cdot G_1$, which is true because

$$\mathbf{Pk}_{\text{attacker}} + C = \mathbf{Pk}_{\text{attacker}} + \mathbf{sk}_x \cdot G_1 - \mathbf{Pk}_{\text{attacker}} = \mathbf{sk}_x \cdot G_1$$

EC-LCG

The server uses this PRNG instance to choose the question for the ZKP protocol. So, we will need to crack the PRNG to predict questions and thus cheat the ZKP protocol:

```
def rng() -> Generator[int, None, None]:
    seed = randbelow(curve_order)
    Gx = 0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296
    Gy = 0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5
    G = ECC.EccPoint(Gx, Gy, curve='p256')
    B = ECC.generate(curve='p256').pointQ
    W0 = G * seed + B
    Wn = W0
```

```
    while True:
        Wn += G
        yield Wn.x >> 32
        yield Wn.y >> 32
```

This algorithm is an EC-LCG that uses curve P256, denoted by $E(\mathbb{F}_p)$, in the following way:

$$\begin{cases} W_0 & = \text{seed} \cdot G + B \\ W_n & = W_{n-1} + G \\ r_{2n-1} & = (W_n)_\text{x} \gg 32 \\ r_{2n} & = (W_n)_\text{y} \gg 32 \end{cases}$$

Where $B, G \in E(\mathbb{F}_p)$, $\text{seed} \in \mathbb{Z}$ and $n > 0$. The outputs of the PRNG are r_i:

$$\begin{cases} r_1 & = (W_1)_\text{x} \gg 32 \\ r_2 & = (W_1)_\text{y} \gg 32 \\ r_3 & = (W_2)_\text{x} \gg 32 \\ r_4 & = (W_2)_\text{y} \gg 32 \\ \dots \end{cases}$$

The same can be expressed as follows:

$$\begin{cases} W_n & = (\text{seed} + n) \cdot G + B \\ r_{2n-1} & = (W_n)_\text{x} \gg 32 \\ r_{2n} & = (W_n)_\text{y} \gg 32 \end{cases}$$

The key to break this EC-LCG implementation is that $W_{n+1} - W_n = G$. However, we are not given the exact values of $(W_n)_\text{x}$ and $(W_n)_\text{y}$, so we

cannot simply find a point W_n to crack the EC-LCG. Instead, we can write some equations with the information we know.

Let's use the following notation for the known outputs:

$$u_n = r_{2n-1} = (W_n)_{\mathrm{x}} \gg 32 \qquad v_n = r_{2n} = (W_n)_{\mathrm{y}} \gg 32$$

And these for the unknowns:

$$a_n = W_n - \left((W_n)_{\mathrm{x}} \gg 32\right) \ll 32 \qquad b_n = W_n - \left((W_n)_{\mathrm{y}} \gg 32\right) \ll 3$$

We know the curve parameters and the generator point G (P256). We know that $(u_n + a_n, v_n + b_n) \in E(\mathbb{F}_p)$, so:

$$(v_n + b_n)^2 = (u_n + a_n)^3 + a \cdot (u_n + a_n) + b \mod p$$

Moreover, we know that

$$G = W_{n+1} - W_n = (u_{n+1} + a_{n+1}, v_{n+1} + b_{n+1}) - (u_n + a_n, v_n + b_n)$$

This can be expressed using point addition formulas:

$$
\begin{cases}
(x_1, y_1) + (x_2, y_2) = (x_3, y_3) \\
\\
\lambda = (y_2 - y_1) \cdot (x_2 - x_1)^{-1} \mod p \\
x_3 = \lambda^2 - x_1 - x_2 \\
y_3 = \lambda \cdot (x_1 - x_3) - y_1
\end{cases}
$$

We can get rid of λ in the formula for x_3:

$$
\begin{aligned}
x_3 &= \lambda^2 - x_1 - x_2 \iff \\
x_3 &= \left((y_2 - y_1) \cdot (x_2 - x_1)^{-1}\right)^2 - x_1 - x_2 \iff \\
x_3 &= (y_2 - y_1)^2 \cdot (x_2 - x_1)^{-2} - x_1 - x_2 \iff \\
0 &= x_3 - (y_2 - y_1)^2 \cdot (x_2 - x_1)^{-2} + x_1 + x_2 \iff \\
0 &= x_3 \cdot (x_2 - x_1)^2 - (y_2 - y_1)^2 + (x_1 + x_2) \cdot (x_2 - x_1)^2 \iff \\
0 &= (x_1 + x_2 + x_3) \cdot (x_2 - x_1)^2 - (y_2 - y_1)^2
\end{aligned}
$$

And the same with the formula for y_3:

$$
\begin{aligned}
y_3 &= \lambda \cdot (x_1 - x_3) - y_1 \iff \\
y_3 &= \left((y_2 - y_1) \cdot (x_2 - x_1)^{-1}\right) \cdot (x_1 - x_3) - y_1 \iff \\
0 &= y_3 - \left((y_2 - y_1) \cdot (x_2 - x_1)^{-1}\right) \cdot (x_1 - x_3) + y_1 \iff \\
0 &= y_3 - (y_2 - y_1) \cdot (x_2 - x_1)^{-1} \cdot (x_1 - x_3) + y_1 \iff \\
0 &= y_3 \cdot (x_2 - x_1) - (y_2 - y_1) \cdot (x_1 - x_3) + y_1 \cdot (x_2 - x_1) \iff \\
0 &= (y_3 + y_1) \cdot (x_2 - x_1) - (y_2 - y_1) \cdot (x_1 - x_3)
\end{aligned}
$$

Given the fact that we have more unknowns than equations, but the linear unknowns are bounded to 2^{32}, we can use a lattice to solve a Shortest Vector Problem using LLL.

We will use a total of 6 PRNG outputs (u_1, v_1, u_2, v_2, u_3, and v_3), that is, 3 points (W_1, W_2 and W_3), and we can get 7 independent equations:

$$\begin{cases} W_1 \in E(\mathbb{F}_p) \\ W_2 \in E(\mathbb{F}_p) \\ W_3 \in E(\mathbb{F}_p) \\ \\ W_2 - W_1 = G \quad (\text{for x and y}) \\ W_3 - W_2 = G \quad (\text{for x and y}) \end{cases}$$

$$\begin{cases} (u_1 + a_1)^3 + a \cdot (u_1 + a_1) + b - (v_1 + b_1)^2 = 0 \mod p \\ (u_2 + a_2)^3 + a \cdot (u_2 + a_2) + b - (v_2 + b_2)^2 = 0 \mod p \\ (u_3 + a_3)^3 + a \cdot (u_3 + a_3) + b - (v_3 + b_3)^2 = 0 \mod p \\ \\ ((u_1 + a_1) + (u_2 + a_2) + G_{\text{x}}) \cdot ((u_2 + a_2) - (u_1 + a_1))^2 - ((v_2 + b \\ ((u_2 + a_2) + (u_3 + a_3) + G_{\text{x}}) \cdot ((u_3 + a_3) - (u_2 + a_2))^2 - ((v_3 + b \\ \\ (G_{\text{y}} - (v_1 + b_1)) \cdot ((u_2 + a_2) - (u_1 + a_1)) - ((v_2 + b_2) + (v_1 + b_1)) \\ (G_{\text{y}} - (v_2 + b_2)) \cdot ((u_3 + a_3) - (u_2 + a_2)) - ((v_3 + b_3) + (v_2 + b_2)) \end{cases}$$

The signs colored in red are to indicate that we are expressing $W_{n+1} - W_n = G$, therefore $-W_n = (u_n + a_n, -v_n - b_n)$.

We can get rid of $\mod p$ by adding some integers $k_1, \ldots, k_7$ multiplied by p:

$$
\begin{cases}
(u_1+a_1)^3+a\cdot(u_1+a_1)+b-(v_1+b_1)^2+k_1\cdot p=0\\
(u_2+a_2)^3+a\cdot(u_2+a_2)+b-(v_2+b_2)^2+k_2\cdot p=0\\
(u_3+a_3)^3+a\cdot(u_3+a_3)+b-(v_3+b_3)^2+k_3\cdot p=0\\
\\
((u_1+a_1)+(u_2+a_2)+G_{\mathrm{x}})\cdot((u_2+a_2)-(u_1+a_1))^2-((v_2+l\\
((u_2+a_2)+(u_3+a_3)+G_{\mathrm{x}})\cdot((u_3+a_3)-(u_2+a_2))^2-((v_3+l\\
\\
(G_{\mathrm{y}}-(v_1+b_1))\cdot((u_2+a_2)-(u_1+a_1))-((v_2+b_2)+(v_1+b_1))\\
(G_{\mathrm{y}}-(v_2+b_2))\cdot((u_3+a_3)-(u_2+a_2))-((v_3+b_3)+(v_2+b_2))
\end{cases}
$$

Obviously, we don't have only 6 unknowns, because there are interactions between variables (i.e. $a_1\cdot a_2$ or a_1^3). However, these variables are also bounded and short when compared to p. As a result, we can use a lattice basis like this to determine the value of a_1, b_1, a_2, b_2, a_3 and b_3:

$$
\begin{bmatrix}
\begin{pmatrix}
p & & & \\
& p & & \\
& & \ddots & \\
& & & p
\end{pmatrix}
&
\begin{pmatrix}
\mathrm{eq}_1 & \mathrm{coefficients}\\
\mathrm{eq}_2 & \mathrm{coefficients}\\
& \cdots\\
\mathrm{eq}_7 & \mathrm{coefficients}
\end{pmatrix}\\
&
\begin{pmatrix}
1 & & & & \\
& 1 & & & \\
& & \ddots & & \\
& & & 1 & \\
& & & & 2^{256}
\end{pmatrix}
\end{bmatrix}
$$

And the target vector is $(k_1, \ldots, k_7, \ldots, a_1, b_1, a_2, b_2, a_3, b_3, 2^{256})$, which is possible to get using LLL on the lattice basis matrix.

Once we have a_1, b_1, a_2, b_2, a_3 and b_3, we can find W_1, W_2 and W_3 to crack the PRNG.

Solution

So, this is the outline to solve the challenge:

- We must run `unveil_secrets`, which needs an aggregated signature of all verified robots
- We can use a rogue key attack to forge the signature
- For this, we must provide a malicious public key, so we will be running `join`
- We need to verify the robot, that is, we need to prove that we have the secret key associated to the malicious public key
- It is not easy to find a secret key that can generate the malicious public key, so we need to cheat in order to complete the zero-knowledge proof
- For this, we need to know exactly what question is the server going to ask, so we need to crack the EC-LCG PRNG
- We need 6 outputs, so we can create a new robot (5^{th} output), use it to list existing robots, and then join another robot with the malicious public key (6^{th} output)

Implementation

The interaction with the server is using JSON (except for the verification process), so we can use this function to send and receive JSON data:

```python
def sr(data):
    io.sendlineafter(b'> ', json.dumps(data).encode())
    return json.loads(io.recvline().decode())
```

First, we create a robot and use it to list the rest:

```
res = sr({'cmd': 'create'})
sk = int(res.get('sk'), 16)
robot_id = int(res.get('robot_id'), 16)

cmd = 'list'
sig = bls.Sign(sk, cmd.encode())
res = sr({'cmd': cmd, 'robot_id': hex(robot_id), 'sig': sig.hex()})

ids, Pks = [], []

for r in res:
    ids.append(int(r.get('robot_id'), 16))
    Pks.append(decompress_G1(G1Compressed(int(r.get('pk'), 16))))
```

With the public keys, we can craft the malicious public key for the rogue key attack (`py-ecc` makes it very easy to implement):

```
sk = 1337
cmd = 'unveil_secrets'
pk = bls.SkToPk(sk)
sig = bls.Sign(sk, cmd.encode())
Pk = pubkey_to_G1(pk)

Pk_prime = add(Pk, neg(reduce(add, Pks, Z1)))
pk_prime = G1_to_pubkey(Pk_prime)
assert normalize(add(reduce(add, Pks), Pk_prime)) == normalize(Pk)
io.success('Forged signature!')
```

Now, we join this malicious public key and get the 6^{th} PRNG output:

```
res = sr({'cmd': 'join', 'pk': pk_prime.hex()})
robot_id = int(res.get('robot_id'), 16)
```

```
ids.append(robot_id)
assert len(ids) == 6
```

Then, we crack the EC-LCG PRNG:

```
p = 0xffffffff00000001000000000000000000000000ffffffffffffffffffffffff
K = GF(p)
a = K(0xffffffff00000001000000000000000000000000fffffffffffffffffffffffc)
b = K(0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b)
E = EllipticCurve(K, (a, b))
G = E(0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296,
0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5)
E.set_order(0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551
* 0x1)


def crack_ec_lcg(values):
    assert len(values) == 6
    u1, v1, u2, v2, u3, v3 = values
    a1, b1, a2, b2, a3, b3 = PolynomialRing(K, 'a1, b1, a2, b2, a3,
b3').gens()

    ec1 = (v1 + b1) ** 2 - (u1 + a1) ** 3 - a * (u1 + a1) - b
    ec2 = (v2 + b2) ** 2 - (u2 + a2) ** 3 - a * (u2 + a2) - b
    ec3 = (v3 + b3) ** 2 - (u3 + a3) ** 3 - a * (u3 + a3) - b

    ec4 = ((u1 + a1) + (u2 + a2) + G.x()) * ((u2 + a2) - (u1 + a1)) ** 2 -
((v2 + b2) + (v1 + b1)) ** 2
    ec5 = ((u2 + a2) + (u3 + a3) + G.x()) * ((u3 + a3) - (u2 + a2)) ** 2 -
((v3 + b3) + (v2 + b2)) ** 2
    ec6 = (G.y() - (v1 + b1)) * ((u2 + a2) - (u1 + a1)) - ((v2 + b2) + (v1 +
b1)) * ((u1 + a1) - G.x())
    ec7 = (G.y() - (v2 + b2)) * ((u3 + a3) - (u2 + a2)) - ((v3 + b3) + (v2 +
b2)) * ((u2 + a2) - G.x())

    A, v = Sequence([ec1, ec2, ec3, ec4, ec5, ec6,
ec7]).coefficients_monomials(sparse=False)
    A = A.change_ring(ZZ)
```

```
    A = (identity_matrix(7) * p).augment(A)
    A = A.stack(zero_matrix(len(v), 7).augment(identity_matrix(len(v))))
    A[-1, -1] = 2 ** 256

    L = A.T.LLL()
    assert L[-1][-1] == 2 ** 256
    a1, b1, a2, b2, a3, b3 = L[-1][-7:-1]

    W1 = E(u1 + a1, v1 + b1)
    W2 = E(u2 + a2, v2 + b2)
    W3 = E(u3 + a3, v3 + b3)
    return W3
```

With this, we can cheat the ZKP protocol to verify the malicious public key:

```
Wn = crack_ec_lcg([i << 32 for i in ids])
io.success('Cracked EC-LCG!')

prog = io.progress('Cheating ZKP')
sr({'cmd': 'verify', 'robot_id': hex(robot_id)})

for _ in range(64 // 2):
    Wn += G

    for c in Wn.xy():
        if (int(c) >> 32) & 1:
            x = int(os.urandom(16).hex(), 16)
            C = multiply(G1, x)
            assert normalize(multiply(G1, x)) == normalize(C)
            io.sendlineafter(b'Take a random value x and send me C = x * G1
(hex): ', bytes(G1_to_pubkey(C)).hex().encode())
            io.sendlineafter(b'Give me x (hex): ', hex(x).encode())
        else:
            sk_x = int(os.urandom(16).hex(), 16)
            C = add(multiply(G1, sk_x), neg(Pk_prime))
            assert normalize(add(multiply(G1, sk_x), neg(Pk_prime))) ==
normalize(C)
            io.sendlineafter(b'Take a random value x and send me C = x * G1
(hex): ', bytes(G1_to_pubkey(C)).hex().encode())
```

```
            io.sendlineafter(b'Give me (sk + x) (hex): ',
hex(sk_x).encode())

prog.success()
```

Finally, we run the `unveil_secrets` command, with the malicious aggregated signature:

```
res = sr({'cmd': cmd, 'sig': sig.hex()})
sr({'cmd': 'exit'})
io.success(res.get('flag'))
```

Flag

If we run the script, we will solve the challenge and get the flag:

```
$ python3 solve.py 94.237.59.199:57607
[+] Opening connection to 94.237.59.199 on port 57607: Done
[+] Forged signature!
[+] Cracked EC-LCG!
[+] Cheating ZKP: Done
[+] HTB{EC-LCG_cr4ck3r_z3r0_kn0wl3dg3_ch34t3r_4nd_r0gue_k3y_4tt4ck3r!!}
[*] Closed connection to 94.237.59.199 port 57607
```

The full script can be found in here: `solve.py`.