Note

1) Smooth Criminal

- Problema

```
 from Crypto.Cipher import AES
from Crypto.Util.number import inverse
from Crypto.Util.Padding import pad, unpad
from collections import namedtuple
from random import randint
import hashlib
import os

# Create a simple Point class to represent the affine points.
Point = namedtuple("Point", "x y")

# The point at infinity (origin for the group law).
O = 'Origin'

FLAG = b'crypto{????????????????????????????????}'


def check_point(P: tuple):
    if P == O:
        return True
    else:
        return (P.y**2 - (P.x**3 + a*P.x + b)) % p == 0 and 0 <= P.x < p and 0 <= P.y < p


def point_inverse(P: tuple):
    if P == O:
        return P
    return Point(P.x, -P.y % p)


def point_addition(P: tuple, Q: tuple):
    # based of algo. in ICM
    if P == O:
        return Q
    elif Q == O:
        return P
    elif Q == point_inverse(P):
        return O
    else:
        if P == Q:
            lam = (3*P.x**2 + a)*inverse(2*P.y, p)
            lam %= p
        else:
            lam = (Q.y - P.y) * inverse((Q.x - P.x), p)
            lam %= p
    Rx = (lam**2 - P.x - Q.x) % p
    Ry = (lam*(P.x - Rx) - P.y) % p
    R = Point(Rx, Ry)
    assert check_point(R)
    return R


def double_and_add(P: tuple, n: int):
    # based of algo. in ICM
    Q = P
    R = O
    while n > 0:
        if n % 2 == 1:
            R = point_addition(R, Q)
        Q = point_addition(Q, Q)
        n = n // 2
    assert check_point(R)
    return R


def gen_shared_secret(Q: tuple, n: int):
    # Bob's Public key, my secret int
    S = double_and_add(Q, n)
    return S.x
```

```
    return S.x


def encrypt_flag(shared_secret: int):
    # Derive AES key from shared secret
    sha1 = hashlib.sha1()
    sha1.update(str(shared_secret).encode('ascii'))
    key = sha1.digest()[:16]
    # Encrypt flag
    iv = os.urandom(16)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    ciphertext = cipher.encrypt(pad(FLAG, 16))
    # Prepare data to send
    data = {}
    data['iv'] = iv.hex()
    data['encrypted_flag'] = ciphertext.hex()
    return data


# Define the curve
p = 310717010502520989590157367261876774703
a = 2
b = 3

# Generator
g_x = 179210853392303317793440285562762725654
g_y = 105268671499942631758568591033409611165
G = Point(g_x, g_y)

# My secret int, different every time!!
n = randint(1, p)

# Send this to Bob!
public = double_and_add(G, n)
print(public)

# Bob's public key
b_x = 272640099140026426377756188075937988094
b_y = 51062462309521034358726608268084433317
B = Point(b_x, b_y)

# Calculate Shared Secret
shared_secret = gen_shared_secret(B, n)

# Send this to Bob!
ciphertext = encrypt_flag(shared_secret)
print(ciphertext)
```

- Soluzione

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
import hashlib
import os

def gen_shared_secret(Q: tuple, n: int):
    # Bob's Public key, my secret int
    S = double_and_add(Q, n)
    return S.x

def decrypt_flag(shared_secret: int, iv: str, ciphertext: str):
    # Derive AES key from shared secret
    sha1 = hashlib.sha1()
    sha1.update(str(shared_secret).encode('ascii'))
    key = sha1.digest()[:16]
    # Decrypt flag
    ciphertext = bytes.fromhex(ciphertext)
    iv = bytes.fromhex(iv)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    plaintext = cipher.decrypt(ciphertext)
    return plaintext

p = 310717010502520989590157367261876774703
a = 2
b = 3

E = EllipticCurve(GF(p), [a, b])

G_x = 179210853392303317793440285562762725654
G_y = 105268671499942631758568591033409611165
G = E(G_x, G_y)

B_x = 272640099140026426377756188075937988094
B_y = 51062462309521034358726608268084433317
B = E(B_x, B_y)

Q_A = E(280810182131414898730378982766101210916, 291506490768054478159835604632710368904)

# Calculate discrete logarithm of Q_A to the base G
n_A = G.discrete_log(Q_A)
S = n_A * B
shared_secret = S[0]

flag_enc = {'iv': '07e2628b590095a5e332d397b8a59aa7', 'encrypted_flag': '8220b7c47b36777a737f5ef9caa2814cf20c1c1ef496ec21a9b48

flag = decrypt_flag(shared_secret, flag_enc['iv'], flag_enc['encrypted_flag'])
print(flag)
```

2) Curveball

- Problema

```
import fastecdsa
from fastecdsa.point import Point
from utils import listener

FLAG = "crypto{????????????????????????????????????}"
G = fastecdsa.curve.P256.G
assert G.x, G.y == [0x6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296,
0x4FE342E2FE1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB6406837BF51F5]

class Challenge():
def **init**(self):
self.before_input = "Welcome to my secure search engine backed by trusted certificate library!\n"
self.trusted_certs = {
'www.cryptohack.org': {
"public_key": Point(0xE9E4EBA2737E19663E993CF62DFBA4AF71C703ACA0A01CB003845178A51B859D, 0x179DF068FC5C380641DB2661121E568BB24B
"curve": "secp256r1",
"generator": [G.x, G.y]
},
'www.bing.com': {
"public_key": Point(0x3B827FF5E8EA151E6E51F8D0ABF08D90F571914A595891F9998A5BD49DFA3531, 0xAB61705C502CA0F7AA127DEC096B2BBDC9BD
"curve": "secp256r1",
"generator": [G.x, G.y]
},
'www.gchq.gov.uk': {
"public_key": Point(0xDEDFC883FEEA09DE903ECCB03C756B382B2302FFA296B03E23EEDF94B9F5AF94, 0x15CEBDD07F7584DBC7B3F4DEBBA0C13ECD2D
"curve": "secp256r1",
"generator": [G.x, G.y]
}
}

    def search_trusted(self, Q):
        for host, cert in self.trusted_certs.items():
            if Q == cert['public_key']:
                return True, host
        return False, None

    def sign_point(self, g, d):
        return g * d

    def connection_host(self, packet):
        d = packet['private_key']
        if abs(d) == 1:
            return "Private key is insecure, certificate rejected."
        packet_host = packet['host']
        curve = packet['curve']
        g = Point(*packet['generator'])
        Q = self.sign_point(g, d)
        cached, host = self.search_trusted(Q)
        if cached:
            return host
        else:
            self.trusted_certs[packet_host] = {
                "public_key": Q,
                "curve": "secp256r1",
                "generator": G
            }
            return "Site added to trusted connections"

    def bing_it(self, s):
        return f"Hey bing! Tell me about {s}"

    #
    # This challenge function is called on your input, which must be JSON
    # encoded
    #
    def challenge(self, your_input):
        host = self.connection_host(your_input)
        if host == "www.bing.com":
            return self.bing_it(FLAG)
        else:
            return self.bing_it(host)

import builtins; builtins.Challenge = Challenge # hack to enable challenge to be run locally, see https://cryptohack.org/faq/#
listener.start_server(port=13382)
```

- Soluzione

```
# Taken from https://neuromancer.sk/std/secg/secp256r1
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)

P = E(
    0x3B827FF5E8EA151E6E51F8D0ABF08D90F571914A595891F9998A5BD49DFA3531,
    0xAB61705C502CA0F7AA127DEC096B2BBDC9BD3B4281808B3740C320810888592A,
)
d = 2

# Find inverse of d modulo order of E
d_inv = inverse_mod(d, E.order())

R = d_inv * P

print(R.xy())

# Result: (15520159875205514130255899098025123715054849599936616868365830290232639266390, 3533257396448043298666012267330522258
```

```
from pwn import remote
import json

# Connect to the server
r = remote("socket.cryptohack.org", 13382)

G = [
    0x6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296,
    0x4FE342E2FE1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB6406837BF51F5,
]

d = 2
R = [
    15520159875205514130255899098025123715054849599936616868365830290232639266390,
    35332573964480432986660122673305225849700662492297568815244635356931754804527,
]  # Taken from cryptohack/Curveball/solve.sage

packet = {
    "private_key": d,  # Example private key, adjust as needed
    "host": "www.bing.com",  # Example host, adjust as needed
    "curve": "secp256r1",
    "generator": R,
}

# Send the packet as JSON
packet = json.dumps(packet)

print(packet)

r.sendline(packet)

r.interactive()

# # Receive the response
# response = r.recvline().decode().strip()
# print("Response from server:", response)

# Close the connection
r.close()


# Devi trovare un P ed m qualsiasi tali che R = mP con m != 1
# Cosa puoi fare? Puoi calcolare P = m^(-1)R, dove m^(-1) è l'inverso moltiplicativo di m modulo l'ordine della curva (che è u
```

3) ProSign 3

- Problema

```
#!/usr/bin/env python3

import hashlib
from Crypto.Util.number import bytes_to_long, long_to_bytes
from ecdsa.ecdsa import Public_key, Private_key, Signature, generator_192
from utils import listener
from datetime import datetime
from random import randrange

FLAG = "crypto{?????????????????????????}"
g = generator_192
n = g.order()


class Challenge:
    def __init__(self):
        self.before_input = "Welcome to ProSign 3. You can sign_time or verify.\n"
        secret = randrange(1, n)
        self.pubkey = Public_key(g, g * secret)
        self.privkey = Private_key(self.pubkey, secret)

    def sha1(self, data):
        sha1_hash = hashlib.sha1()
        sha1_hash.update(data)
        return sha1_hash.digest()

    def sign_time(self):
        now = datetime.now()
        m, n = int(now.strftime("%m")), int(now.strftime("%S"))
        current = f"{m}:{n}"
        msg = f"Current time is {current}"
        hsh = self.sha1(msg.encode())
        sig = self.privkey.sign(bytes_to_long(hsh), randrange(1, n))
        return {"msg": msg, "r": hex(sig.r), "s": hex(sig.s)}

    def verify(self, msg, sig_r, sig_s):
        hsh = bytes_to_long(self.sha1(msg.encode()))
        sig_r = int(sig_r, 16)
        sig_s = int(sig_s, 16)
        sig = Signature(sig_r, sig_s)

        if self.pubkey.verifies(hsh, sig):
            return True
        else:
            return False

    #
    # This challenge function is called on your input, which must be JSON
    # encoded
    #
    def challenge(self, your_input):
        if "option" not in your_input:
            return {"error": "You must send an option to this server"}

        elif your_input["option"] == "sign_time":
            signature = self.sign_time()
            return signature

        elif your_input["option"] == "verify":
            msg = your_input["msg"]
            r = your_input["r"]
            s = your_input["s"]
            verified = self.verify(msg, r, s)
            if verified:
                if msg == "unlock":
                    self.exit = True
                    return {"flag": FLAG}
                return {"result": "Message verified"}
            else:
                return {"result": "Bad signature"}

        else:
```

```
            return {"error": "Decoding fail"}


import builtins

builtins.Challenge = Challenge  # hack to enable challenge to be run locally, see https://cryptohack.org/faq/#netcat
listener.start_server(port=13381)
```

- Soluzione

```
 from hashlib import sha1
import json
from datetime import datetime
from Crypto.Util.number import bytes_to_long, long_to_bytes
from Pwn4Sage.pwn import remote, context

p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFF
K = GF(p)
a = K(0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFC)
b = K(0x64210519E59C80E70FA7E9AB72243049FEB8DEECC146B9B1)
E = EllipticCurve(K, (a, b))
G = E(
    0x188DA80EB03090F67CBF20EB43A18800F4FF0AFD82FF1012,
    0x07192B95FFC8DA78631011ED6B24CDD573F977A11E794811,
)
E.set_order(0xFFFFFFFFFFFFFFFFFFFFFFFF99DEF836146BC9B1B4D22831 * 0x1)
q = G.order()
R = Zmod(q)

context.log_level = "error"
r = remote("socket.cryptohack.org", 13381)

packet = {"option": "sign_time"}

response = r.recvline()
r.sendline(json.dumps(packet).encode())
response = r.recvline()
response = json.loads(response)
print(response)

msg = response["msg"]
r1 = R(int(response["r"], 16))
s1 = R(int(response["s"], 16))

hsh = sha1(msg.encode()).digest()
z1 = R(bytes_to_long(hsh))

# Sign the message "unlock"
msg = "unlock"
hsh = sha1(msg.encode()).digest()
z2 = R(bytes_to_long(hsh))

for i in range(1, 60):
    k = i
    d = (k * s1 - z1) * r1 ^ (-1)

    r2 = R((k * G).xy()[0])
    s2 = (z2 + d * r2) * R(k) ^ (-1)

    print(r2, s2)
    packet = {"option": "verify", "msg": "unlock", "r": hex(int(r2)), "s": hex(int(s2))}
    r.sendline(json.dumps(packet).encode())
    response = r.recvline()
    response = json.loads(response)
    print(response)
```

4) Moving Problems

- Problema

```

```

- Soluzione

```python
import random
import hashlib
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import os

FLAG = b"crypto{????????????????????????????????????????}"


def gen_keypair(G, p):
    n = random.randint(1, (p - 1))
    P = n * G
    return n, P


def gen_shared_secret(P, n):
    S = P * n
    return S.xy()[0]


def encrypt_flag(shared_secret: int):
    # Derive AES key from shared secret
    sha1 = hashlib.sha1()
    sha1.update(str(shared_secret).encode("ascii"))
    key = sha1.digest()[:16]
    # Encrypt flag
    iv = os.urandom(16)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    ciphertext = cipher.encrypt(pad(FLAG, 16))
    # Prepare data to send
    data = {}
    data["iv"] = iv.hex()
    data["encrypted_flag"] = ciphertext.hex()
    return data


# Define Curve params
p = 1331169830894825846283645180581
a = -35
b = 98
E = EllipticCurve(GF(p), [a, b])
G = E.gens()[0]

# Generate keypair
n_a, P1 = gen_keypair(G, p)
n_b, P2 = gen_keypair(G, p)

# Calculate shared secret
S1 = gen_shared_secret(P1, n_b)
S2 = gen_shared_secret(P2, n_a)

# Check protocol works
assert S1 == S2

flag = encrypt_flag(S1)

print(f"Generator: {G}")
print(f"Alice Public key: {P1}")
print(f"Bob Public key: {P2}")
print(f"Encrypted flag: {flag}")
```