```
import random
from math import ceil
from decimal import Decimal

FIELD_SIZE = 10**5


def reconstruct_secret(shares):
    """
    Combines individual shares (points on graph)
    using Lagranges interpolation.

    `shares` is a list of points (x, y) belonging to a
    polynomial with a constant of our key.
    """
    sums = 0
    prod_arr = []

    for j, share_j in enumerate(shares):
        xj, yj = share_j
        prod = Decimal(1)

        for i, share_i in enumerate(shares):
            xi, _ = share_i
            if i != j:
                prod *= Decimal(Decimal(xi)/(xi-xj))

        prod *= yj
        sums += Decimal(prod)

    return int(round(Decimal(sums), 0))


def polynom(x, coefficients):
    """
    This generates a single point on the graph of given polynomial
    in `x`. The polynomial is given by the list of `coefficients`.
    """
    point = 0
    # Loop through reversed list, so that indices from enumerate match the
    # actual coefficient indices
    for coefficient_index, coefficient_value in enumerate(coefficients[::-1]):
        point += x ** coefficient_index * coefficient_value
    return point


def coeff(t, secret):
    """
    Randomly generate a list of coefficients for a polynomial with
    degree of `t` - 1, whose constant is `secret`.

    For example with a 3rd degree coefficient like this:
        3x^3 + 4x^2 + 18x + 554

        554 is the secret, and the polynomial degree + 1 is
        how many points are needed to recover this secret.
        (in this case it's 4 points).
    """
    coeff = [random.randrange(0, FIELD_SIZE) for _ in range(t - 1)]
    coeff.append(secret)
    return coeff


def generate_shares(n, m, secret):
    """
    Split given `secret` into `n` shares with minimum threshold
    of `m` shares to recover this `secret`, using SSS algorithm.
    """
    coefficients = coeff(m, secret)
    shares = []

    for i in range(1, n+1):
        x = random.randrange(1, FIELD_SIZE)
        shares.append((x, polynom(x, coefficients)))

    return shares

def generate_ascii(secret):
    secret_ascii = ""
    for s in secret:
        secret_ascii += str(ord(s))
    return secret_ascii


# Driver code
if __name__ == '__main__':

    # (3,5) sharing scheme
    t, n = 4, 6
    username = "JOHN"
    password = "ABCDEF"
    salt = "S1"
    secret = username + password + salt
    print(f'Original Secret: {secret}')
    secret_ascii = generate_ascii(secret)
    # Phase I: Generation of shares
    shares = generate_shares(n, t, int(secret_ascii))
    print(f'Shares: {", ".join(str(share) for share in shares)}')

    #Signin
    sign_username = "JOHN"
    sign_password = "ABCDEF"
    sign_salt = "S1"
    sign_secret = sign_username + sign_password + sign_salt
    print(f'Sign in Original Secret: {sign_secret}')
    sign_secret_ascii = generate_ascii(sign_secret)
    # Phase I: Generation of shares
    sign_shares = generate_shares(n, t, int(sign_secret_ascii))
    print(f'Shares: {", ".join(str(share) for share in sign_shares)}')
    # Phase II: Secret Reconstruction
    # Picking t shares randomly for
    # reconstruction
    pool = random.sample(sign_shares, t)
    print(f'Combining shares: {", ".join(str(share) for share in pool)}')
```

```
reconstructed_secret = reconstruct_secret(pool)
print(f'Reconstructed secret: {reconstructed_secret}')
#check signin details
if(int(secret_ascii) ^ reconstructed_secret):
    print('Sign in details incorrect')
else:
    print('Sign in sucessful')
```