20CYS205 – MODERN CRYPTOGRAPHY

Implement and find vulnerabilities in ECDSA

Submitted by

MOTHE ANURAG REDDY – CB.EN.U4CYS22069

NANDANA MAHESH – CB.EN.U4CYS220070

ANAGH SHAJI PLAMOOTTUKADA – CB.EN.U4CYS220071

RUDRA SRI LAKSHMI – CB.EN.U4CYS220072

Under the guidance of

Aravind Vishnu S

Research Scholar

Amrita Vishwa Vidyapeetham Coimbatore

TIFAC-CORE IN CYBER SECURITY

AMRITA SCHOOL OF ENGINEERING

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112

2023

Parameters:

This Python code implements a basic digital signature scheme using the Elliptic Curve Digital Signature Algorithm (ECDSA) with the secp256k1 elliptic curve parameters

- Elliptic Curve Parameters:

  p: The prime modulus defining the finite field over which the elliptic curve operates.

  a and b: Coefficients of elliptic curve equation (y^2 = x^3 + ax + b).

  Gx and Gy: Coordinates of a base point (generator point) on the elliptic curve.

  n: The order of the base point, which is a large prime number.

- Hash Function:

  The sha256_hash function takes a message as input, computes its SHA-256 hash, and converts the hash to an integer.

- Point Addition and Doubling:

  point_add and point_double functions perform point addition and doubling operations on elliptic curve points, respectively.

- Scalar Multiplication:

  The scalar_multiply function computes scalar multiplication of a point on the elliptic curve.

- Key Generation:

  A random private key (private_key) is generated.

  The corresponding public key (public_key) is computed by multiplying the base point by the private key.

- Signing a Message:

  A message is input by the user and hashed.

A random integer k is generated.

The point P is computed as the result of scalar multiplication of the base point by k.

The x-coordinate of P modulo n is used as the signature's r value.

The signature's s value is computed using the private key, k, and the hash of the message.

•Verifying a Signature:

The verification process involves computing values w, u1, and u2.

The public key (public_key) is used in the verification process.

A point V is computed using scalar multiplication and point addition operations.

The signature is considered valid if the x-coordinate of V is equal to the r value from the signature.

PYTHON CODE :-

```
import random
import hashlib

def sha256_hash(message):
    sha256_hash_object = hashlib.sha256()
    sha256_hash_object.update(message.encode('utf-8'))
    hash_result = sha256_hash_object.hexdigest()
    hash_int = int(hash_result, 16)
    return hash_int


# Define the elliptic curve parameters (for example, using the secp256k1
curve)
p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F
a = 0x0000000000000000000000000000000000000000000000000000000000000000
b = 0x0000000000000000000000000000000000000000000000000000000000000007
Gx = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798
Gy = 0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8

n = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141

# Define a point addition function
```

```
def point_add(P, Q):
    if P == (0, 0):
        return Q
    if Q == (0, 0):
        return P

    x1, y1 = P
    x2, y2 = Q

    if P == Q:
        m = (3 * x1 * x1 + a) * pow(2 * y1, -1, p)
    else:
        m = (y2 - y1) * pow(x2 - x1, -1, p)

    x3 = (m * m - x1 - x2) % p
    y3 = (m * (x1 - x3) - y1) % p
    return (x3, y3)

# Define a point doubling function
def point_double(P):
    if P == (0, 0):
        return (0, 0)

    x1, y1 = P
    m = (3 * x1 * x1 + a) * pow(2 * y1, -1, p)
    x3 = (m * m - 2 * x1) % p
    y3 = (m * (x1 - x3) - y1) % p
    return (x3, y3)

# Define a scalar multiplication function
def scalar_multiply(k, P):
    result = (0, 0)
    for i in range(k.bit_length()):
        if (k >> i) & 1:
            result = point_add(result, P)
        P = point_double(P)
    return result

# Generate a random private key
private_key = random.randint(1, n - 1)

# Compute the public key
public_key = scalar_multiply(private_key, (Gx, Gy))

# Signing a message
message = str(input("Enter Message: "))
hash_of_message = sha256_hash(message)
# print(hash_of_message)
```

```
k = random.randint(1, n - 1)
P = scalar_multiply(k, (Gx, Gy))
r = P[0] % n
s = (pow(k, -1, n) * (hash_of_message + r * private_key)) % n

# Verifying a signature
w = pow(s, -1, n)
u1 = (hash_of_message * w) % n
u2 = (r * w) % n
V = point_add(scalar_multiply(u1, (Gx, Gy)), scalar_multiply(u2, public_key))
valid_signature = V[0] == r

print("Public Key:", public_key)
print("Message:", message)
print("Signature (r, s):", (r, s))
print("Signature is valid:", valid_signature)
```

UID CODE : -

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>ECDSA Signature Generator</title>
  <script
src="https://cdnjs.cloudflare.com/ajax/libs/elliptic/6.5.3/elliptic.min.js"></
script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/3.1.9-
1/crypto-js.js"></script>
  <style>
    body {
      font-family: Arial, sans-serif;
      background-color: #f4f4f4;
      margin: 0;
      padding: 0;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
    }

    #container {
      background-color: #fff;
      border-radius: 8px;
```

```
      box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
      padding: 20px;
      width: 800px;
      height: 500px;
      text-align: center;
    }

    label {
      display: block;
      margin-bottom: 8px;
      font-weight: bold;
    }

    input {
      width: 100%;
      padding: 8px;
      margin-bottom: 16px;
      box-sizing: border-box;
    }

    button {
      background-color: #007BFF;
      color: #fff;
      border: none;
      padding: 10px 20px;
      font-size: 16px;
      cursor: pointer;
      border-radius: 4px;
    }

    button:hover {
      background-color: #0056b3;
    }
  </style>
</head>
<body>
  <div id="container">
    <h2>ECDSA Signature Generator</h2>
    <label for="privateKey">Private Key:</label>
    <input type="text" id="privateKey" placeholder="Enter private key">

    <label for="message">Message:</label>
    <input type="text" id="message" placeholder="Enter message">

    <button id="generateBtn">Generate Signature</button>

    <h3>Signature (r, s):</h3>
    <p id="signatureOutput"></p>
```

```
    <h3>Signature Verification:</h3>
    <p id="verificationOutput"></p>
  </div>

  <script>
    document.getElementById('generateBtn').addEventListener('click',
generateSignature);

    function generateSignature() {
      const privateKeyHex = document.getElementById('privateKey').value;
      const message = document.getElementById('message').value;

      // Validate input
      if (!privateKeyHex || !message) {
        alert('Please enter both private key and message.');
        return;
      }

      try {
        // Convert the private key from hex to bytes
        const privateKeyBytes = hexToBytes(privateKeyHex);

        // Create an elliptic curve object using the secp256k1 curve
        const ec = new elliptic.ec('secp256k1');

        // Create a key pair from the private key
        const key = ec.keyFromPrivate(privateKeyBytes);

        // Hash the message using SHA-256
        const hashedMessage =
CryptoJS.SHA256(CryptoJS.enc.Utf8.parse(message)).toString(CryptoJS.enc.Hex);

        // Sign the hashed message
        const signature = key.sign(hashedMessage);

         /* Modify the signature to make it invalid (for example, increment s)
         signature.s = signature.s.add(ec.n); // This is just an example, you
can modify it differently*/

        // Display the signature in hex format
        const signatureOutput = `(${signature.r.toString(16)},
${signature.s.toString(16)})`;
        document.getElementById('signatureOutput').innerText =
signatureOutput;

        // Verification logic
        const validSignature = key.verify(hashedMessage, signature);
```

```
        // Display the verification result
        const verificationOutput = `Is Signature valid? : ${validSignature}`;
        document.getElementById('verificationOutput').innerText =
verificationOutput;
      } catch (error) {
        alert('Error: ' + error.message);
      }
    }

    // Hex to Bytes conversion
    function hexToBytes(hex) {
      const bytes = [];
      for (let i = 0; i < hex.length; i += 2) {
        bytes.push(parseInt(hex.substr(i, 2), 16));
      }
      return new Uint8Array(bytes);
    }
  </script>
</body>
</html>
```

USER MANUAL:-

USER MANUAL FOR ECDSA SIGNATURE

What is ECDSA?

ECDSA is a cryptographic algorithm used to generate digital signatures. It provides authenticity and integrity for data by utilizing elliptic curve cryptography. A signature acts like a digital fingerprint that proves a message was created by a specific entity (the signer) and has not been tampered with.

Introduction

Welcome to the ECDSA Signature Generator, a web application designed to facilitate the generation and verification of ECDSA (Elliptic Curve Digital Signature Algorithm) signatures. This cryptographic algorithm uses the secp256k1elliptic curve and is widely used for secure data authentication. This manual will guide you through the process of using the application effectively.

2. Getting Started

For Accessing the Website,
Open your web browser and visit the ECDSA Signature Generator website.Then, you get to Experience an intuitive design with a centered container, offering a clean and organized interface. Input fields for the private key and message, a signature generation button, and result sections are neatly presented.

3. Generating ECDSA Signature

Type your message:
In the "Message" field, type the message you want to sign. This can be any text, data, or document you want to authenticate.

Enter your private key:
In the "Private Key" field, type your ECDSA private key in hexadecimal format. This key should be kept secret and secure, as it is used to generate signatures.

Generate Signature:
Click the "Generate Signature" button to initiate the signature generation process.

View Signature:
Witness the generated signature (r, s) presented in the corresponding section.

4.Understanding the Output:

Signature (r, s):
After generating the signature, you will see the signature displayed in the "Signature" section. It will be in the format of "(r, s)," where r and s are two hexadecimal values representing the signature components.

Signature Verification:
This section displays whether the generated signature is valid for the entered message. It will say "Signature is valid: true" if the signature is valid and "false" if it's not.

Tips:

Store your private key safely and securely, as anyone who possesses it can generate signatures for your messages.
Be cautious when sharing your private key with any third-party applications or services.
Use strong and unique passwords for any accounts related to your private key.

Conclusion:

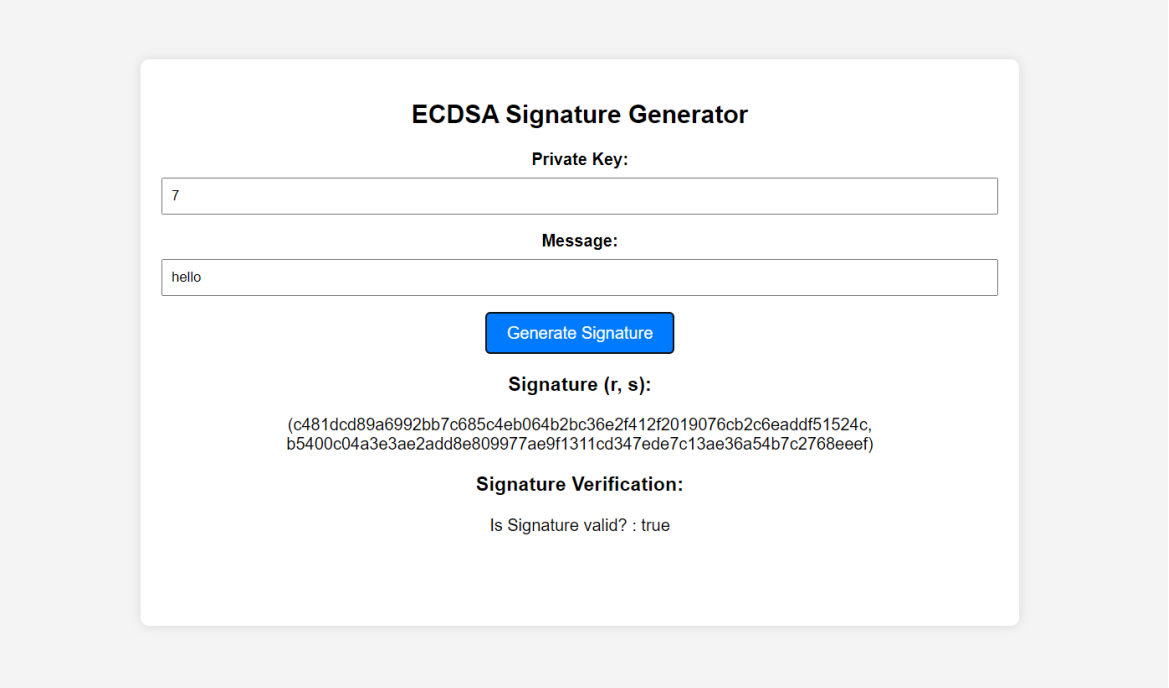
This ECDSA Signature Generator provides a convenient way to generate and verify signatures for your data. I hope this user manual clarifies the functionality and usage of the ECDSA Signature Generator. Feel free to ask any further questions you may have!

Feedback Mechanism:

Provide valuable feedback, suggestions, or report issues to the website administrator for prompt assistance.

THANK YOU!

OUTPUT SCREENSHOTS :-



ERRORS :-

The curve that we chose is the secp256k1 curve. It is considered to be a secure elliptic curve and doesn’t have any vulnerabilities. secp256k1 was constructed in a special non-random way, which allows for especially efficient computation and its constants were selected in a predictable way which significantly reduces the possibility that the curve's creator inserted any sort of backdoor into the curve.

Selecting another curve which is not so secure would lead to vulnerabilities and invalid signatures.