Application of RSA Cryptosystem and Linear Congruential Generator to Enhance Security in JSON Web Tokens for Storing User's Credentials

Nayaka Ghana Subrata - 13523090[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13523090@mahasiswa.itb.ac.id, nayakaghana39@gmail.com

***Abstract*— With the vast development of technology, the internet has become an essential part of human life and its basic needs. One of its purposes is to develop and access websites or web-based applications. However, sensitive and confidential information needs to be secured, which led to the introduction of JSON Web Tokens (JWT). Nevertheless, JWT has many weaknesses and variations, and if not implemented correctly, the security of data stored using JWT can be compromised. Therefore, an effectively modified hashing algorithm is needed to enhance the security of JWT. This paper introduces a solution to that problem, investigates the theory behind hash and JSON Web Token and its implementation with RSA and LCG in the encryption and decryption process. Experimental results demonstrate the server encrypting and decrypting the credentials using JWT, showcasing the algorithm used in this paper.**

***Keywords*—RSA, LCG, JWT, Hashing algorithm**

I. Introduction

With the vast developments of technology, the internet has become a crucial part of human life and its basic needs. One of its uses is to develop and access websites or web-based applications. The web roles as a media to exchange user's information and data with the server. However, sensitive and confidential information need to be secured, so there is an urgent need for systems that can securely store data in a way that is resistant to breaches and unauthorized access.

To face this challenge, the concept of JSON Web Tokens (JWT) is introduced. JWT uses various hash algorithms to store information in an encoded string that contains data wrapped in JSON format. However, certain hashing algorithms have been shown to be vulnerable, making it easier to compromise the security of the stored data. This vulnerability increases the need for stronger algorithms to ensure the confidentiality and integrity of server-side data.

To solve these issues, the writer proposes an approach that integrates RSA (Rivest-Shamir-Adleman) encryption and the Linear Congruential Generator (LCG) into the JWT mechanism. The implementation of RSA aims to enhance the encryption effectiveness, making it more difficult to breach. Meanwhile, LCG is integrated to generate random codes (salts) that can be used for padding data within JWTs. This combination used to enhance the overall security of JWTs by utilizing both RSA and LCG.

This paper also discusses the design and implementation of the modified JWT system, exploring how RSA and LCG are incorporated into the existing JWT framework. Furthermore, the outcomes of this implementation will be analyzed to demonstrate its effectiveness in addressing the security vulnerabilities inherent in conventional JWT mechanisms.

This paper aims to provide a comprehensive exploration of the application of Rivest-Shamir-Adleman (RSA) algorithm and Linear Congruential Generator (LCG) to enhance the effectiveness of JSON Web Token (JWT) hashing algorithm.

The paper has been organized as follows: this section provides the overview and the introduction, Section 2 provides the theoretical framework, Section 3 provides the hashing scheme, Section 4 provides the implementation, Section 5 provides the test and the result, and Section 6 provides the conclusion followed by references.

II. Theoretical Framework

A. Cryptosystem

Cryptosystem is an entire set of cryptographic systems needed necessary for the provision of a certain security services, such as data confidentiality and hiding data's crucial information (encryption-decryption process). This can also be defined as converting plaintext to ciphertext to encrypt and decrypt message securely.

In general, cryptosystem consists of three main algorithms: key generation, encryption, and decryption. The basic model of cryptosystem is depicted in the figure below:

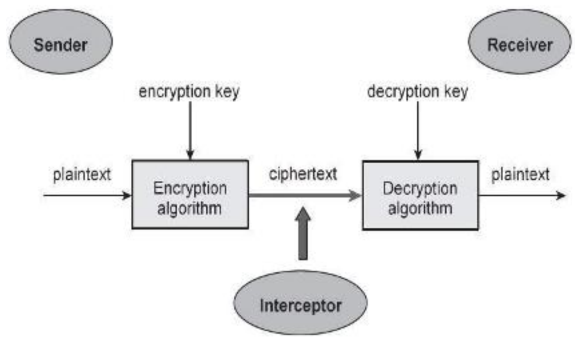


***Fig. 2.1** Basic cryptosystem model*
(Source: Adapted from [3])

Typically, there are two kinds of cryptosystems based on its key-generation process; the first kind of the cryptosystem is symmetric key cryptography, and the second kind of the cryptosystem is asymmetric key cryptography. Symmetric key cryptography is a cryptography process that uses same keys for encryption and decryption process. A well-known example that uses this cryptosystem are Advanced Encryption Standard (AES), Data Encryption Standard (DES), International Data Encryption Algorithm (IDEA), Blowfish, and Rivest Cipher. Example for this encryption can be seen in Fig 2.2.

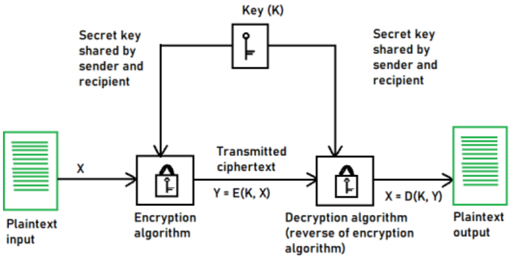


***Fig. 2.2** Basic symmetric key cryptography model*
(Source: Adapted from [1])

Asymmetric key cryptography is a cryptography process that uses different keys for encryption and decryption process. A well-known example that uses this cryptosystem are Rivest-Shamir-Adleman (RSA), Elliptic Curve Cryptography (ECC), Digital Signature Algorithm (DSA), Diffie-Hellman, and Certificate Authorities (CAs). Example for this encryption can be seen in Fig 2.3.

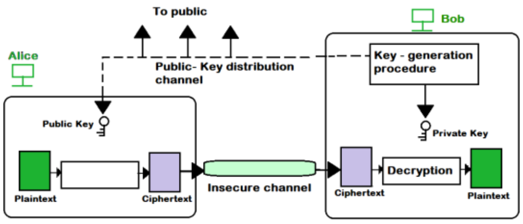


***Fig. 2.3** Basic asymmetric key cryptography model*
(Source: Adapted from [2])

B. Hash Functions

Hash function uses mathematical functions to take various inputs (or we can call it variables), then converting the inputs to fixed-length data. Hash divided into three parts: keys, functions, and hash tables. Keys is the user input, or the data given to the hash system, functions is the hash algorithm functions to convert the keys into its tables, and the tables is to store the hashed values in a table so we can track the outputted value and prevent the hash collision. The process of hashing some data can be seen in Fig 2.4.

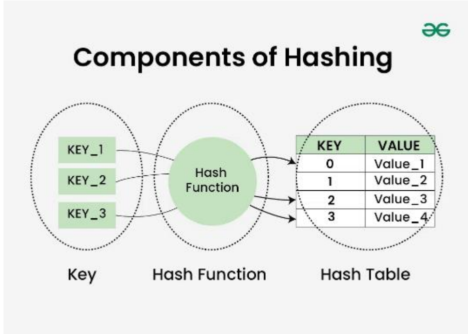


Fig. 2.4 Hash Algorithms
(Source: Adapted from [5])

There are many types of hash functions, the first one is Message Digest (MD), this function is often used to ensure the integrity of the transferred files. The second one is Secure Hash Functions (SHA), this function is often used in applications or network protocols (for example, Secure Socket Layer / SSL). The third one is Cyclic Redundancy Check (CRC), this function is often used for detecting errors in a data transfer process.

C. Rivest-Shamir-Adleman

Rivest-Shamir-Adleman (RSA) algorithm is one of the cryptosystems that uses asymmetric key to encrypt and decrypt the plaintext and the ciphertext. This algorithm is named after its founder: Ron Rivest, Adi Shamir, and Len Adleman in 1977.

***Fig 2.5** (From left to right) Adi Shamir, Ron Rivest, and Len Adleman*
(Source: Adapted from [4])

a. Encryption

The encryption process of this algorithm is quite simple, first pick two primes, or namely p and q. The size of this primes is freely chosen, but it's recommended to pick big primes to make the decryption process more challenging and difficult.

After picking the two primes number (p and q), we can calculate the modulus for the encryption, or namely N. The N value can be calculated using the equations below:

$$N = pq \ldots (1)$$

With N is the modulus value, and pq is the product of the two primes number. Notice that, if we choose big size of integer for the p and q values, the N size is increased significantly too.

After we calculate N value, the next step is to pick the public exponent or sometimes called the encryption key value (e value). In general, we can pick 65537 (or 0x10001 in hexadecimal representation) to be the public exponent. This value picked because of its common compromise between being high, and its cost of raising to the *e*-th power. But keep in mind that the *e* value must be coprime with the Euler's totient value that usually represent in phi (φ) symbol (this totient value will be discussed in the decryption part).

The final step of the RSA encryption process is to convert plaintext to ciphertext, or namely *c*. To calculate the *c* value, we must understand what number theory and modular arithmetic is. The *c* value can be calculated using the equations below:

$$c = m^e \; mod \; N \; ... (2)$$

With *m* is the plaintext representation in its integer value. After we calculate the *c* value, we can share the *N*, *e*, and *c* value to the receiver.

a. Decryption

The decryption process of this algorithm is quite challenging, first, we have to search for prime factors from *N* value (see eq. (1)), if the encryption process is using conventional RSA, we can use Pollard's Rho algorithm to search for the prime factors from *N* (or we're searching for *p* and *q* values). The algorithm can be seen in Fig 2.6.

```
Algorithm 1 Pollard's Rho Algorithm [POL75]

function POLLARD_RHO(N)
    x ← 2
    y ← 2
    d ← 1
    while d = 1 do
        x ← f(x) mod N                Single step
        y ← f(f(y)) mod N             Double step
        d ← GCD(|x - y|, N)
    if d = N then
        return "Failure"
    else
        return d

function f(x)
    return (x² + 1) mod N

function GCD(a, b)
    while b ≠ 0 do
        temp ← b
        b ← a mod b
        a ← temp
    return a
```

Fig 2.6 *Pollard's Rho algorithm*
(Source: Writer's archive)

After getting the *p* and *q* values, calculate the Euler's totient, Euler's totient is a function to determine how much numbers are coprime relative to the *N* value (or suppose that the number is *k*, $1 \leq k \leq N$, greatest common divisor of *k* and *N* must be equal to 1). Euler's totient is multiplicative function, meaning that if we have two coprime numbers, for example a and b, then:

$$\varphi(ab) = \varphi(a)\varphi(b) \; \dots (3)$$

If n-set of numbers ($\{a_1, a_2, \dots, a_n\}$) are pair-wisely coprime, then:

$$\varphi\left(\prod_{i=1}^{n} a_i\right) = \prod_{i=1}^{n} \varphi(a_i) \; ... (4)$$

From eq. (3), if b is a prime number, then $\varphi(b) = b - 1$. Notice that a and b are different prime numbers because a and b is coprime. From these results, we can get:

$$\varphi(ab) = \varphi(a)\varphi(b) \; \varphi(ab) = (a - 1) \; (b - 1) \; \dots (5)$$

With $\varphi(ab)$ is the Euler's totient value that we'll use to calculate the private key. After calculating the Euler's totient value, we can calculate the private key value, namely *d*. To calculate *d*, we will use the equivalencies below:

$$d \equiv e - 1 \; mod \; (\varphi(N)) \; \dots (6)$$

From eq. (6), calculate *d* using modular inverse concept, after we get the *d* value, we can convert ciphertext to its plaintext using this equation below:

$$m = c^d \; mod \; N \; ... (7)$$

With *c* is the ciphertext representation in its integer value. After we calculate the *m* value, convert it to its string value to get the plaintext.

D. Linear-Congruential Generator (LCG)

Before we start to discuss Linear Congruential Generator, first we have to know the recursive concept and the number theory. Linear Congruential Generator uses recursive functions to generate the random number, so it will be explained below.

Recursive functions are a function that always call itself until its reaching its basis. The idea of this concept is to solve problems by breaking it to smaller problem, so we can find similar problems related to it and it makes more easier to working with.

There are two parts of recursive functions, the first part is the basis. Basis is the base case of the problems, making the recursive function stops when it reaches the basis conditions. The second part is the recurrence, the recurrence is the steps needed to solve the problems, the recurrence will define itself until it meets the basis.

Problems that use this concept for example linear congruential generator, tree, greatest common divisor, Fibonacci, and fractal image.

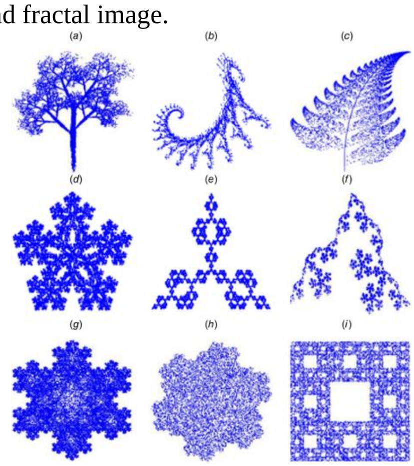


Fig 2.7 *Fractal Geometry*
(Source: Adapted from [https://www.researchgate.net/figure/Nine-well-known-fractal-geometries-a-tree-b-seahorse-c-fern-leaf-d-h_fig2_370681485])

With recursive approach, we can generate random number using linear congruential generator (LCG). LCG is an algorithm to generate random pseudo number. The equation of lcg can be defined in equations below:

$$X_n = (aX_{n-1} + c) \ mod \ m \ldots (8)$$

From eq. (8), a is the multiplier factor, c is increment, m is modulus, and X_n is the n-th random pseudo number. Before we start, we have to define its seed first (X_0) so the recursive can meet the basis and stop when it reaches the basis.

D. JSON Web Token (JWT)

JSON Web Tokens (JWT) are compact, URL-safe tokens used for securely transmitting information between parties as a JSON object. They are commonly used for authentication and information exchange[6]. The token is mainly composed of header, payload, and signature which each part is separated by dots ('.').

The header part is commonly used to describe the cryptosystem applied to JSON Web Token and contains the data of content that we are likely to send. The next parts are the payload part. The payload is the part where all of user's data is added, this part commonly stored the user's data such as credentials but take a note that the information is readable by anyone so we must carefully store the data in the JWTs. The last part is signature part. This part is to verify if the authenticity of the token is valid, so only the authored one can access using this token.

***Fig 2.8** Decrypted JWT Structure*
(Source: Adapted from [7])

In this case, the paper will use one of the JWT hash algorithm, that is RS-256. RS-256 is an asymmetric algorithm that uses public and private key to hash the JWTs. The identity provider has the private key to create the signature. The JWT recipient uses the public key to validate the JWT signature. The public key used to verify, and the private key used to sign the token are related because they are created as a pair.

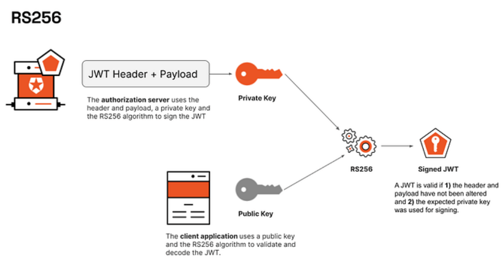


***Fig 2.9** RS256 Algorithm*
(Source: Adapted from [6])

III. IMPLEMENTATION

This program is developed using Python as its primary programming language due to its simplicity and versatility in mathematical processing. The libraries included are Crypto.Util and cryptography to help with RSA process, flask to make the dummy server, base64 to do base64 encryption and decryption, hashlib to do hash, json to make the JWTs, request to make the client dummies, time to calculate the seed for lcg, and datetime to calculate the expiration date of the JWT token (because JWT can be used once to prevent security breach). Several limitations have been incorporated into the implementation to ensure its feasibility.

The limitations are no databases provide in this implementation, so all of data is stored locally in the server and will be deleted if the program is terminated. The source code of the program can be accessed in appendix sections.

The code is divided into two parts, servers.py, and clients.py. servers.py will do the encryption and generating the JWTs, storing the user's data, verifying the JWTs signature from the client, and decrypting the credentials stored in the JWTs to login into the server. Clients.py will do the request to server, registering some data, and login attempt (or to check whether the program is successful or not).

A. Servers.py

This file will do the core algorithm of the proposed hash idea. First the servers will generate the private key and public key to verify the JWTs, it uses 2048 bits of modulus value (N) and static public exponent (e=65537). After generating the private key and public key for verifying JWTs, then the program will generate the LCG parameters, with a value of 1664525, c value of 1013904223, modulus value of 2^{32}, and seed value of the current time.

After initiating the RSA and LCG parameters, the next step is to generate the salts for the password using e.q. (8). The usage of these salts is to make the password is harder to decode, and making the hash result is more random. The hashing process of the password will be using SHA256 algorithm, and the password will be padded with its salts, then the salt data and password will also be stored in the server's database.

After encrypting the password, server will generate the JWTs, the header contains the type (JWT) and the algorithm (RS256), the payload contains username, token expiration date, the encoded password, and some hashed data, and the last is the JWTs signature. The implementation of these scheme can be seen below:

```python
import json
import time
import base64
import hashlib
from flask import Flask, request, jsonify
from Crypto.Util.number import getPrime, GCD
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import rsa, padding

class UserManager:
    def __init__(self):
        self.users = {}

        while True:
            p = getPrime(1024)
            q = getPrime(1024)
            e = 65537

            if p == q:
                continue

            n = p * q
            phi = (p - 1) * (q - 1)

            if GCD(e, phi) == 1:
                break

        d = pow(e, -1, phi)

        self.private_key = rsa.RSAPrivateNumbers(
            p=p,
            q=q,
            d=d,
            dmp1=d % (p - 1),
            dmq1=d % (q - 1),
            iqmp=pow(q, -1, p),
            public_numbers=rsa.RSAPublicNumbers(
                e=e,
                n=n
            )
        ).private_key()

        self.public_key = self.private_key.public_key()

        self.private_pem = self.private_key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.PKCS8,
            encryption_algorithm=serialization.NoEncryption()
        )
        self.public_pem = self.public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo
        )

        # LCG parameters
        self.lcg_a = 1664525
        self.lcg_c = 1013904223
        self.lcg_m = 2**32
        self.lcg_seed = int(time.time())

    def get_public_key(self):
        return self.public_pem.decode()
```

***Fig 3.1** Server – RSA and LCG initiation*
(Source: Writer's archive)

```python
def generate_salt(self, length=16):
    salt = bytearray()
    for _ in range(length):
        self.lcg_seed = (self.lcg_a * self.lcg_seed + self.lcg_c) % self.lcg_m
        salt.append(self.lcg_seed % 256)
    return bytes(salt)

def register_user(self, username, password):
    if username in self.users:
        raise ValueError("Username already exists")

    salt = self.generate_salt()
    hashed_password = hashlib.pbkdf2_hmac(
        'sha256',
        password.encode(),
        salt,
        100000
    )

    encrypted_password = self.public_key.encrypt(
        hashed_password,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    self.users[username] = {
        'encrypted_password': base64.b64encode(encrypted_password).decode(),
        'salt': salt.hex()
    }

    return True
```

***Fig 3.2** Server – User's credential encryption process*
(Source: Writer's archive)

```python
def verify_user(self, username, password):
    if username not in self.users:
        return False

    user_data = self.users[username]
    salt = bytes.fromhex(user_data['salt'])

    input_hash = hashlib.pbkdf2_hmac(
        'sha256',
        password.encode(),
        salt,
        100000
    )

    try:
        stored_hash = self.private_key.decrypt(
            base64.b64decode(user_data['encrypted_password']),
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )
        return input_hash == stored_hash
    except:
        return False
```

***Fig 3.3** Server – User verification*
(Source: Writer's archive)

```python
def generate_token(self, username):
    user_data = self.users[username]

    payload = {
        'username': username,
        'exp': int(time.time()) + 3600,
        'credentials': user_data['encrypted_password']
    }

    payload_str = json.dumps(payload, sort_keys=True)
    payload['hash'] = hashlib.sha256(payload_str.encode()).hexdigest()

    header = {
        'alg': 'RS256',
        'typ': 'JWT'
    }

    header_encoded = base64.urlsafe_b64encode(json.dumps(header).encode()).rstrip(b'=')
    payload_encoded = base64.urlsafe_b64encode(json.dumps(payload).encode()).rstrip(b'=')

    message = header_encoded + b'.' + payload_encoded
    signature = self.private_key.sign(
        message,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )

    signature_encoded = base64.urlsafe_b64encode(signature).rstrip(b'=')

    return b'.'.join([header_encoded, payload_encoded, signature_encoded]).decode()
```

Fig 3.4 *Server – JWTs generator*
(Source: Writer's archive)

```python
app = Flask(__name__)
user_manager = UserManager()

@app.route('/register', methods=['POST'])
def register():
    data = request.get_json()
    username = data.get('username')
    password = data.get('password')

    if not username or not password:
        return jsonify({'error': 'Username and password required'}), 400

    try:
        user_manager.register_user(username, password)
        return jsonify({'message': 'Registration successful'})
    except ValueError as e:
        return jsonify({'error': str(e)}), 400
    except Exception as e:
        return jsonify({'error': 'Registration failed'}), 500

@app.route('/login', methods=['POST'])
def login():
    data = request.get_json()
    username = data.get('username')
    password = data.get('password')

    if not username or not password:
        return jsonify({'error': 'Username and password required'}), 400

    if user_manager.verify_user(username, password):
        token = user_manager.generate_token(username)
        return jsonify({'token': token})
    else:
        return jsonify({'error': 'Invalid credentials'}), 401

@app.route('/public-key', methods=['GET'])
def get_public_key():
    return jsonify({'public_key': user_manager.get_public_key()})

app.run(debug=True, port=1220)
```

Fig 3.5 *Server*
(Source: Writer's archive)

B. Clients.py

This file will do the core algorithm of the algorithm's testing. First the clients will be registering their credentials to the server, so the JWTs can be made. The next step, the clint will login to test if they can login with their credentials or not. If the login succeeds, they can see their decoded JWTs data that stored in the server, but they can't see their own passwords and hash, so the decoded JWTs will be outputted safely. The implementation of these scheme can be seen below:

```python
import os
import json
import time
import base64
import requests
from datetime import datetime
from cryptography.hazmat.primitives import serialization

class Colors:
    HEADER = '\033[95m'
    BLUE = '\033[94m'
    GREEN = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    ENDC = '\033[0m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'
```

Fig 3.6 *Client – Initiation*
(Source: Writer's archive)

```python
class TokenClient:
    def __init__(self, server_url="http://localhost:1220"):
        self.server_url = server_url
        self.public_key = None
        self._fetch_public_key()

    def _fetch_public_key(self):
        try:
            response = requests.get(f"{self.server_url}/public-key")
            if response.status_code == 200:
                pem_data = response.json()['public_key'].encode()
                self.public_key = serialization.load_pem_public_key(pem_data)
            else:
                raise ConnectionError(f"Server error: {response.json().get('error')}")
        except requests.RequestException as e:
            raise ConnectionError(f"Connection error: {str(e)}")

    def register(self, username, password):
        try:
            response = requests.post(
                f"{self.server_url}/register",
                json={'username': username, 'password': password}
            )

            if response.status_code == 200:
                return {'status': 'success', 'message': "Registration successful!"}
            else:
                raise ValueError(response.json().get('error', 'Registration failed'))
        except requests.RequestException as e:
            raise ConnectionError(f"Connection error: {str(e)}")

    def login(self, username, password):
        try:
            response = requests.post(
                f"{self.server_url}/login",
                json={'username': username, 'password': password}
            )

            if response.status_code == 200:
                token = response.json()['token']
                return {'status': 'success', 'token': token}
            else:
                raise ValueError(response.json().get('error', 'Login failed'))
        except requests.RequestException as e:
            raise ConnectionError(f"Connection error: {str(e)}")
```

Fig 3.7 *Client – Register and Login*
(Source: Writer's archive)

```python
def decode_jwt(token):
    parts = token.split('.')
    if len(parts) != 3:
        return "Invalid token format"

    try:
        header = base64.urlsafe_b64decode(parts[0] + '=' * (4 - len(parts[0]) % 4))
        payload = base64.urlsafe_b64decode(parts[1] + '=' * (4 - len(parts[1]) % 4))

        header_data = json.loads(header)
        payload_data = json.loads(payload)

        return {
            'header': header_data,
            'payload': payload_data,
            'signature': parts[2],
            'full_token': token
        }
    except Exception as e:
        return f"Error decoding token: {str(e)}"
```

Fig 3.8 *Client – JWTs decrypting process*
(Source: Writer's archive)

```python
def clear_screen():
    os.system('cls' if os.name == 'nt' else 'clear')

def print_banner():
    banner = f"""
{Colors.BLUE}╔════════════════════════════════╗
║          TOKEN CLIENT v1.0          ║
╚════════════════════════════════╝{Colors.ENDC}
"""
    print(banner)

def print_menu():
    menu = f"""
{Colors.BOLD}Available Options:{Colors.ENDC}
{Colors.GREEN}1.{Colors.ENDC} Register New User
{Colors.GREEN}2.{Colors.ENDC} Login
{Colors.GREEN}3.{Colors.ENDC} View Current Token
{Colors.GREEN}4.{Colors.ENDC} Exit
"""
    print(menu)

def print_token_info(token_data):
    print(f"\n{Colors.BLUE}=== Token Information ==={Colors.ENDC}")

    print(f"\n{Colors.BOLD}Full JWT Token:{Colors.ENDC}")
    print(f"{Colors.GREEN}{token_data['full_token']}{Colors.ENDC}")

    print(f"\n{Colors.BOLD}Header:{Colors.ENDC}")
    print(json.dumps(token_data['header'], indent=2))

    print(f"\n{Colors.BOLD}Payload:{Colors.ENDC}")
    payload = token_data['payload']

    if 'exp' in payload:
        exp_time = datetime.fromtimestamp(payload['exp'])
        payload['exp'] = exp_time.strftime('%Y-%m-%d %H:%M:%S')

    print(json.dumps(payload, indent=2))

    print(f"\n{Colors.BOLD}Signature:{Colors.ENDC}")
    print(token_data['signature'])

def loading_animation(duration=1):
    chars = "|/-\\"
    for _ in range(int(duration * 10)):
        for char in chars:
            print(f"\r{Colors.BLUE}Processing {char}{Colors.ENDC}", end='')
            time.sleep(0.1)
    print("\r" + " " * 20 + "\r", end='')
```

Fig 3.9 *Client – GUI and JWTs info*
(Source: Writer's archive)

```python
def interactive_client():
    client = None
    token = None

    try:
        client = TokenClient()
    except ConnectionError as e:
        print(f"\n{Colors.FAIL}✗ Failed to initialize client: {str(e)}{Colors.ENDC}")
        return
    except Exception as e:
        print(f"\n{Colors.FAIL}✗ Unexpected error during initialization: {str(e)}{Colors.ENDC}")
        return

    while True:
        try:
            clear_screen()
            print_banner()
            print_menu()

            choice = input(f"\n{Colors.BOLD}Enter your choice (1-4):{Colors.ENDC} ")

            if choice == "1":
                print(f"\n{Colors.BLUE}=== User Registration ==={Colors.ENDC}")
                username = input(f"{Colors.BOLD}Username:{Colors.ENDC} ")
                password = input(f"{Colors.BOLD}Password:{Colors.ENDC} ")

                if not username or not password:
                    raise ValueError("Username and password cannot be empty")

                loading_animation()
                result = client.register(username, password)
                print(f"\n{Colors.GREEN}✓ {result['message']}{Colors.ENDC}")

            elif choice == "2":
                print(f"\n{Colors.BLUE}=== User Login ==={Colors.ENDC}")
                username = input(f"{Colors.BOLD}Username:{Colors.ENDC} ")
                password = input(f"{Colors.BOLD}Password:{Colors.ENDC} ")

                if not username or not password:
                    raise ValueError("Username and password cannot be empty")

                loading_animation()
                result = client.login(username, password)
                if result['status'] == 'success':
                    token = result['token']
                    print(f"\n{Colors.GREEN}✓ Login successful!{Colors.ENDC}")

            elif choice == "3":
                if token:
                    try:
                        decoded = decode_jwt(token)
                        if isinstance(decoded, str) and "Error" in decoded:
                            raise ValueError(decoded)
                        print_token_info(decoded)
                    except ValueError as e:
                        print(f"\n{Colors.FAIL}✗ Token error: {str(e)}{Colors.ENDC}")
                    except Exception as e:
                        print(f"\n{Colors.FAIL}✗ Unexpected error while decoding token: {str(e)}{Colors.ENDC}")
                else:
                    print(f"\n{Colors.WARNING}⚠ No active token. Please login first.{Colors.ENDC}")

            elif choice == "4":
                print(f"\n{Colors.GREEN}Thank you for using Token Client!{Colors.ENDC}")
                break

            else:
                print(f"\n{Colors.WARNING}⚠ Invalid choice. Please select 1-4.{Colors.ENDC}")

        except ValueError as e:
            print(f"\n{Colors.FAIL}✗ Validation error: {str(e)}{Colors.ENDC}")
        except ConnectionError as e:
            print(f"\n{Colors.FAIL}✗ Connection error: {str(e)}{Colors.ENDC}")
        except KeyboardInterrupt:
            print(f"\n\n{Colors.WARNING}⚠ Operation cancelled by user{Colors.ENDC}")
            break
        except Exception as e:
            print(f"\n{Colors.FAIL}✗ Unexpected error: {str(e)}{Colors.ENDC}")

        try:
            input(f"\n{Colors.BOLD}Press Enter to continue...{Colors.ENDC}")
        except KeyboardInterrupt:
            print(f"\n\n{Colors.WARNING}⚠ Program terminated by user{Colors.ENDC}")
            break

interactive_client()
```

Fig 3.10 *Client*
(Source: Writer's archive)

IV. RESULT

Test results are obtained from a series of test cases which covers various scenarios that may be the vulnerabilities in JWTs. This testing process is carried out for testing the functionality and flexibility of the JWTs hashing algorithm. The program starts with giving the input form to the users, there are 4 options: register, login, see token, and exit. First, users have to register their account first to get their JWTs, the program will be asking the users for the username and the password, the process can be seen below:

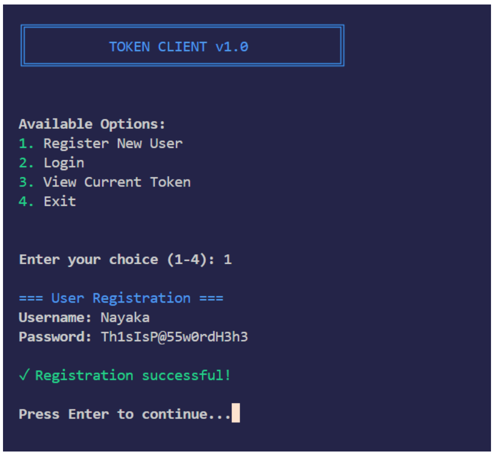


Fig 4.1 *User registration test*
(Source: Writer's archive)

If registration succeeds, the program will output the successful message, but they can't see their current token, because they aren't logged in yet.

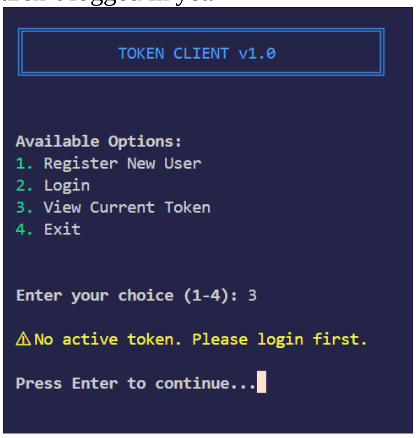


Fig 4.2 *No active token in the server*
(Source: Writer's archive)

To see the JWTs, we have to login first, but in this test, there are several testcase. First if the credentials is false, the second is if the credentials is true, and the third is if we can use other users credentials with our username. For this test case, new users are added to the server, named dummy, and the password is 123.

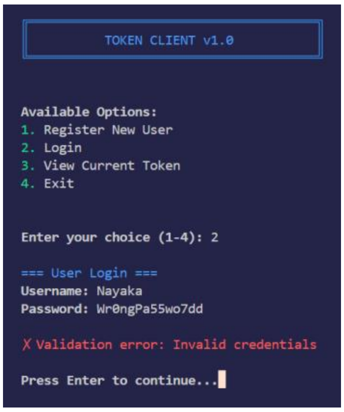


Fig 4.3 *Invalid user's credentials*
(Source: Writer's archive)

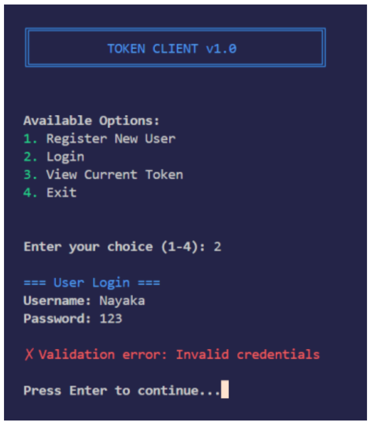


Fig 4.4 *Login attempt with other user's credentials*
(Source: Writer's archive)

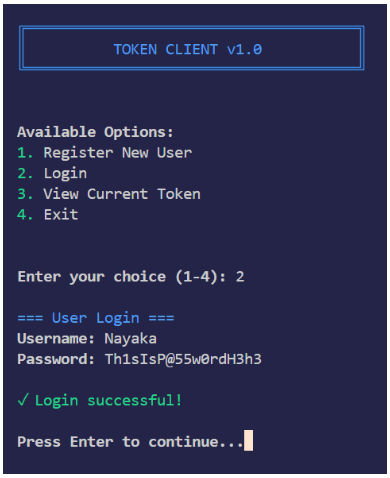


Fig 4.5 *Successful login attempts*
(Source: Writer's archive)

If the user's login is valid, then they can see their JWTs in the option 3, there are the header, payload (that contains the encoded credentials) and its signature. And for the last test, is to check the uniqueness of the generated JWT and the hash, use the dummy's JWT for the comparator to the Nayaka's JWT.

Fig 4.6 *Nayaka's JWT*
(Source: Writer's archive)

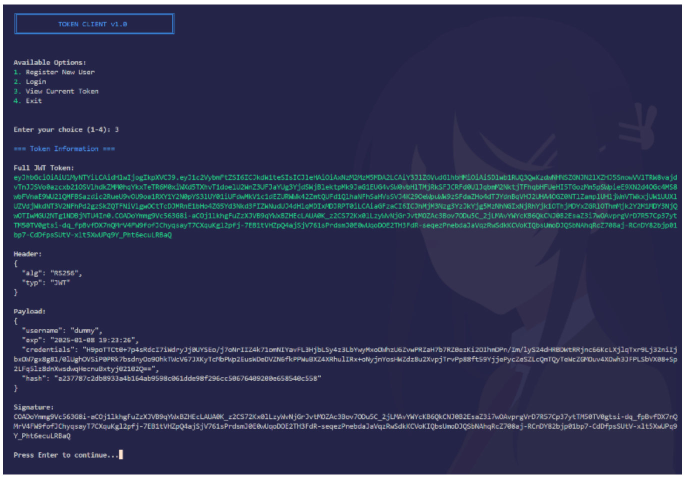


Fig 4.7 *dummy's JWT*
(Source: Writer's archive)

As can be seen in Fig 4.6 and Fig 4.7, the programs are successfully encrypting user's credentials in JWTs. Notice that the JWT, the hash, and the signature is uniquely generated so to it is quite impossible for a hash collision to occur. If the figure provided is not clearly visible, the figure can also be accessed at the link in the appendix sections.

V. Conclusion

With Rivest-Shamir-Adleman and Linear Congruential Generator, we can enhance JWTs security. RSA is used to generate the signature of the JWTs and generate its private key and public key to decode, and Linear Congruential Generator to generate salt for the password, so the hashed password will be unique and harder to decode. From the results, the generated salt, hash, and signature is unique, so it can prevent for hash collision to occur. The generated JWTs is also unique, making it an one-time-usage token, so there are no users can recycle or exploiting this token, the token also resets after 1 hour to prevent these thing happened.

This study lays the groundwork for further development. The current program developed is exclusively encrypt and decrypt small amount of data. Furthermore, the program's limitation to the databases can be broadened, allowing for more user's data and its credentials to be hashed.

VI. Appendix

The program that used in this paper can be seen in this link: https://github.com/Nayekah/JWT

VII. Acknowledgment

All praise and gratitude belong to the Almighty God, Allah Subhanahu wa Ta'ala, for his blessings and grace, enable the writer to complete this paper. The writer also giving sincere thanks to Ir. Rila Mandala, M.Eng., Ph.D., the lecturer for the IF1220 – Discrete Mathematics for his guidance and kindness to the writer. And the writer also appreciates for author's families and friends for their motivational support throughout the process of finishing this paper.

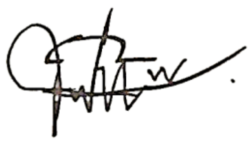
References

[1] Geeksforgeeks, "Symmetric Key Cryptography", https://www.geeksforgeeks.org/symmetric-key-cryptography/, 2024, accessed 6th January 2025, 02.33 UTC+7.
[2] Geeksforgeeks, "Asymmetric Key Cryptography", https://www.geeksforgeeks.org/asymmetric-key-cryptography/, 2024, accessed 6th January 2025, 02.37 UTC+7.
[3] JaneW, "Cryptosystem Model", https://uwillnvrknow.github.io/deCryptMe/pages/cryptosystem.html , 2024, accessed 6th January 2025, 02.43 UTC+7.
[4] M. Rinaldi, "Algoritma RSA", https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2020-2021/Algoritma-RSA-2020.pdf, accessed 6th January 2025, 04.53 UTC+7.
[5] Geeksforgeeks, "Introduction to Hashing", https://www.geeksforgeeks.org/symmetric-key-cryptography/, 2024, accessed 7th January 2025, 16.45 UTC+7.
[6] Chamblee S., "Difference Between RS256 and HS256 JWT Signing Algorithms", https://community.auth0.com/t/difference-between-rs256-and-hs256-jwt-signing-algorithms/58609, 2024, accessed 8th January 2025, 16.29 UTC+7.
[7] M. Dan, "Components of JWTs Explained", https://fusionauth.io/articles/tokens/jwt-components-explained, 2024, accessed 8th January 2025, 16.05 UTC+7.
[8] M. Rinaldi, "Teori Bilangan (Bagian 3)", https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/17-Teori-Bilangan-Bagian3-2024.pdf, 2024, accessed 8th January 2025, 16.58 UTC+7.
[9] M. Jones, J. Bradley, N. Sakimakura "JSON Web Token (JWT)", https://www.rfc-editor.org/rfc/rfc7519.html, 2015, accessed 8th January 2025, 17.26 UTC+7

Statement of Originality

I hereby declare that this paper is my own writing, not an adaptation, or translation of someone else's paper, and not plagiarized.

Bandung, 08 January 2025



Nayaka Ghana Subrata
13523090