Đại học Quốc gia Thành phố Hồ Chí Minh
Trường Đại học Công nghệ Thông tin (UIT)

BÁO CÁO BÀI TẬP

Môn học: Mật mã học

Kỳ báo cáo: Buổi 04 (Session 04)

Tên chủ đề: Thuật toán mã hoá Elliptic Curve và hàm băm

Ngày báo cáo: 20/05/2023

1. THÔNG TIN CHUNG:

Lớp: NT219.N22.ATCL.1

STT	Họ và tên	MSSV	Email
1	Đoàn Hải Đăng	21520679	21520679@gm.uit.edu.vn
2	Lê Thanh Tuấn	21520518	21520518@gm.uit.edu.vn
3	Phan Thị Hồng Nhung	21521250	21521250@gm.uit.edu.vn

2. NỘI DUNG THỰC HIỆN:

STT	Công việc	Kết quả tự đánh giá	Người đóng góp
1	Bài tập 1	100%	Hồng Nhung
2	Bài tập 2	100%	Hồng Nhung
3	Bài tập 3	100%	Hải Đăng
4	Bài tập 4	100%	Hải Đăng
5	Bài tập 6	100%	Thanh Tuấn
6	Bài tập 7	100%	Thanh Tuấn
7	Bài luyện tập 1	100%	Cả nhóm
8	Bài luyện tập 2	40%	Cả nhóm

Phần bên dưới của báo cáo này là tài liệu báo cáo chi tiết của nhóm thực hiện.

BÁO CÁO CHI TIẾT

1. Bài tập 1

Modify and sign the attached file UIT.png

Idea:

```
string input = "D:/Nam2/MMH/Lab_Run/UIT.png";
string message;

CryptoPP::FileSource file(input.c_str(), true /*pumpAll*/,new CryptoPP::StringSink(message));
string signature;

result = SignMessage( privateKey, message, signature );
assert(true == result );
if (true == result)
    std::cout << "Signed\n";
```

```
string message_bin;
CryptoPP::StringSource source(message, true, new CryptoPP::StringSink(message_bin));
string output = "D:/Nam2/MMH/Lab_Run/output_UIT.png";

// Write the binary data to the output file
ofstream outputFile(output, ios::binary);
if (outputFile.is_open()) {
outputFile.write(message_bin.data(), message_bin.size());
outputFile.close();
std::cout << "Saved the message as a PNG file: " << output << endl;
} else  {
    cout << "Failed to open the output file: " << output << endl;
}


result = VerifyMessage( publicKey, message, signature );
assert(true == result );
if (true == result)
    cout << "Verify Success\n";
```

Result:

```
Modulus:
 14615016373309029182036848327162830196537850593 27.
Coefficient A:
 14615016373309029182036848327162830196537850593 24.
Coefficient B:
 16323579130616811054660491940327157953054834541 3.
Base Point:
 X: 42582623172388835044654159270140906591363556877 0.
 Y: 20352011416290410787399145795734689202798264197 0.
Subgroup Order:
 14615016373309029182036871976068267798846434924 39.
```

```
Cofactor:
 1.

Private Exponent:
 19777264636559979125190020370134251446312323196 8.

Public Element:
 X: 658300593166019772846511580929715267489041084856.
 Y: 648218967620820654744983593807338289473646373927.
Signed
Saved the message as a PNG file: D:/Nam2/MMH/Lab_Run/output_UIT.png
Verify Success
```

2. Bài tập 2

Change SHA1 to SHA256 and compare

After compiling 2 parts, I don't see any differences.

```
 Modulus:
  1461501637330902918203684832716283019653785059327.
 Coefficient A:
  1461501637330902918203684832716283019653785059324.
 Coefficient B:
  1632357913061681105466049194032715795305483454 13.
 Base Point:
  X: 425826231723888350446541592701409065913635568770.
  Y: 203520114162904107873991457957346892027982641970.
 Subgroup Order:
  1461501637330902918203687197606826779884643492439.
Cofactor:
 1.

Private Exponent:
 13802233331148288540683169984763815148861293817 29.

Public Element:
 X: 812613597488831461331801115271565359222503604280.
 Y: 131163982901985874053512515333368840334280239731 5.
Signed
Saved the message as a PNG file: D:/Nam2/MMH/Lab_Run/output2_UIT.png
Verify Success
```

3. Bài tập 3

Load digital signature (r, s) and message m then verify.

Private and public key are load from file using the following code:

```
// Generate private key
    ECDSA<ECP, SHA256>::PrivateKey privKey;
    LoadPrivateKey("ec.private.key", privKey);
    privKey.Save( privateKey );

    // Create public key
```

```
ECDSA<ECP, SHA256>::PublicKey pubKey;
LoadPublicKey("ec.public.key", pubKey);
pubKey.Save( publicKey );
```

Load message m from UIT.png

```
string filename = "UIT.png";
string message;
CryptoPP::FileSource file(filename.c_str(), true,
                          new CryptoPP::StringSink(message));
```

Result:

```
Algorithm: ECDSA/EMSA1(SHA-256)
Signature: 0098FE4DCFA0CBA6C1F65F66B6B6E466D926D20C13001F26D39B2CFEEF3D3E00E39B7CCAF7C59208CE1B
Verified signature on message
```

4. Bài tập 4

Switch case MD5, SHA224, SHA256, SHA384, SHA512, SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE128, SHAKE256.

- Input enter manually from user. Message = *"Lab4 Elliptic Curve Digital Signature Algorithm & Hash"*
- Print output on screen
- Result:

```
Name: MD5
Message: Lab4 Elliptic Curve Digital Signature Algorithm & Hash
Digest: 4EC95811BCDDB4E04E041C9A8F1CC3AB
```

```
Name: SHA-224
Message: Lab4 Elliptic Curve Digital Signature Algorithm & Hash
Digest: C785B8045B3A53E2EC9C73D5E2B587B715B340AEF8D0802735CC04A8
```

```
Name: SHA-256
Message: Lab4 Elliptic Curve Digital Signature Algorithm & Hash
Digest: 2E23B3975E6B7D02BF4F20F67F418E63D88479170D01E5FD20F16CF125543A35
```

```
Name: SHA-384
Message: Lab4 Elliptic Curve Digital Signature Algorithm & Hash
Digest: DDA6FC8B06A080062CD9F31DAC45199A74707039573BC189C8798CBE2821845D56C1D8154D60E79B5332E98211C530B6
```

```
Name: SHA-512
Message: Lab4 Elliptic Curve Digital Signature Algorithm & Hash
Digest: 7E49A10925D76B7A0FB718B3E942632394CE72ED994E5C23D4DFC83F258F7DC262FE3B42739D4FE3530EDFC70AD2B3239A6A5AABC2DB90B91192B3B1E7E29E87
```

```
Name: SHA3-224
Message: Lab4 Elliptic Curve Digital Signature Algorithm & Hash
Digest: C285FDB15EC99390F8ABC87FB3A2E0319078B282256B541CB6810BA5
```

```
Name: SHA3-256
Message: Lab4 Elliptic Curve Digital Signature Algorithm & Hash
Digest: 970A5247C3DB07E02D2809B74AE910B4E861D37E8794100119A0CF2956135CD1
```

```
Name: SHA3-384
Message: Lab4 Elliptic Curve Digital Signature Algorithm & Hash
Digest: 5210B3B16989C4E92BEFD5A6940C9410885584DFF3C32B36699F75A16FFA23053F6760E41A7ED680B73E5DCE4E8DBF58
```

```
Name: SHA3-512
Message: Lab4 Elliptic Curve Digital Signature Algorithm & Hash
Digest: 410D8E3A94D796CE42EB934EADE9390171EC2FA0821BC46777D8896332265C743FD00D65DBFDC745EC934B84D27F26E024D3BA4A30E99F14E578687C599AA931
```

```
Name: SHAKE-128
Message: Lab4 Elliptic Curve Digital Signature Algorithm & Hash
Digest: F8ACB32CD79AFA96FB83E6E3937B6D8EEE75BFC317C8E40FFFA68CC0446BC6A5
```

```
Name: SHAKE-256
Message: Lab4 Elliptic Curve Digital Signature Algorithm & Hash
Digest: 7AFAE17E3C3A7056D54F68A95A9BBA33D192150FA2EA27237BD3CA9F640F31A337957F8F03F9D694CEBB260BE266F4FDCA806BD4FA2F1957935DECFFDA74E84E
```

5. Bài tập 6

Collision attack in hash

I. Collision attack in hash function

Idea: Attacker searches for two different pieces of data that, when hashed through the hash function, will result in the same hash value. In strong hash functions like SHA-256, collision attacks are highly complex and require significant computational time, but in weaker hash functions like MD5, collision attacks can be performed with lower complexity.

II. Description of collision attack process using MD5

The collision attack process in MD5 involves the following steps:

1. Choosing two different messages
2. Creating data blocks for each message
3. Computing hash values for each message
4. Searching for a pair of identical hash values
5. Creating a new message

The attacker creates a new message by combining the data blocks from the two different messages such that the hash values of the corresponding blocks are identical. This new message will have the same hash value as the two original different messages.

III. Code demo

Below is an example of how to perform a collision attack in the MD5 hash function using the Python language:

```python
import hashlib

def md5_collision():
    message1 = b'hello'
    message2 = b'world'

    hash1 = hashlib.md5(message1).digest()
    hash2 = hashlib.md5(message2).digest()

    for i in range(1, 50):
        for j in range(1, 50):
            padded_message1 = message1 + b'\x80' + b'\x00'*(i-1) + hash1
            padded_message2 = message2 + b'\x80' + b'\x00'*(j-1) + hash2

            new_hash1 = hashlib.md5(padded_message1).digest()
            new_hash2 = hashlib.md5(padded_message2).digest()

            if new_hash1 == new_hash2:
                print("Collision found:")
                print(f"Message 1: {padded_message1}")
                print(f"Message 2: {padded_message2}")
                print(f"Hash: {new_hash1.hex()}")
                return

md5_collision()
```

The result of the program:

```
Collision found:
Message 1: b'hello\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
Message 2: b'world\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
Hash: 8f78456d9c9a9ddefa85e0c15cec15d5
```

When we find a collision pair of messages, we print out information about those messages and their hash values. In this case, we print out the pair of messages "hello" and "world" along with their hash values.

6. Bài tập 7

Length extension attack in hash

I. Length extension attack in hash

This attack works on the basis that the SHA256 hash function has a specific structure and the input is divided into blocks of a certain length. When calculating the hash value of a message, the SHA256 function will calculate on each block in order and use the hash value of the previous block as the input for the next block. To calculate the hash value of M', the attacker will calculate the hash value of M up to the last block, then use this hash value as the input for the next block that the attacker creates using a new element P and the length of M.

II. Description of the process of using the SHA256 hash function:

Idea: In the given code, we calculate the SHA256 hash value of an initial message. Then, we create a new message by adding padding to it and calculate its SHA256 hash value as well. The padding ensures that the length of the new message and padding is divisible by 64 bytes. We save the new hash value in the variable h2. By comparing the values of variables h and h2, we can determine whether the hash values of the original and new messages are different.

```python
import hashlib

message = b'This is the original message'
h = hashlib.sha256(message).hexdigest()

new_message = b'This is the new message'
length = len(message) + len(new_message)
padding = b'\x80' + b'\x00' * (55 - (length + 1) % 64) + length.to_bytes(8, byteorder='big')

h2 = hashlib.sha256(new_message + padding).hexdigest()

print(f"h: {h}")
print(f"h2: {h2}")
```

The result of the program:

```
h1: 0x82a22410b0e15d9a11ce7fdd28558fc2d7cfcdb3ab8f1c95d527c7f1d954fe96
h2: 0x13ad46e4995b334f63fdeac08520c5bc30d8a09028bc8ba5b33a1f43b8b65c52
```

7. Bài luyện tập 1

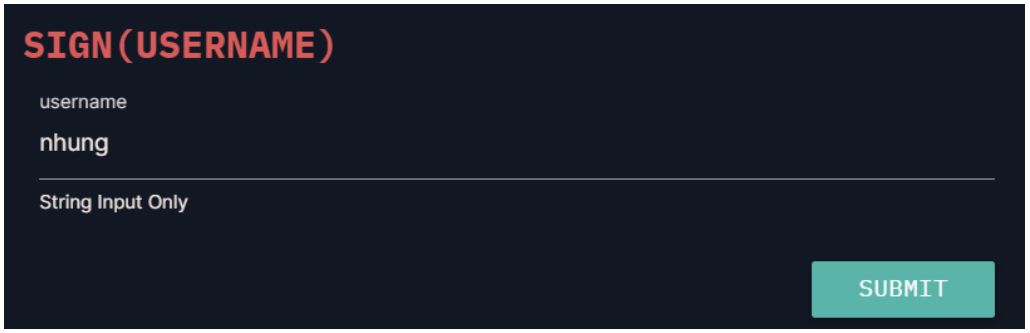
	RSA	ECC
Advantages	• Established Trust: RSA has been in existence and widely used for over 40 years and	• High Computational Efficiency: Improved performance, especially on

	has been reviewed by numerous security experts. • Flexible Key Length: This enables customization of security according to system requirements. • Popularity: One of the most widely public key encryption algorithms and is supported by a wide range of cryptographic libraries and applications.	resource-constrained devices. • Smaller Key Sizes: Reduces the storage requirements and computation time, making it more suitable for constrained environments. • Simplicity of Key Structure: Simplifies key management and implementation.
Disadvantages	• Computational Efficiency: RSA requires more computational resources to perform encryption and decryption operations. • Large Key Sizes: RSA demands larger key sizes to achieve the same level of security as ECC. • Complex Key Structure: RSA's public key involves large prime numbers, whereas ECC's public key involves points on a curve.	• Limited Historical Usage: Its shorter history may introduce some level of uncertainty for long-term trust. • Patent Concerns: Some ECC algorithms were covered by patents, which limited their usage and implementation. • Implementation Complexity: Implementing ECC correctly can be more challenging than RSA due to its specific requirements.

Writeup challenge Digestive on CryptoHack: Signature Verification in Flask Application.

This code generates and verifies signatures for messages, checks the user's privileges through the "admin" attribute in the message, and returns the corresponding results.

When my input username is “nhung”. I will get the output below.



The /digestive/verify/<msg>/<signature>/ route is used to verify a signature. If the verification is successful, the message is decoded from JSON, and the "admin" attribute in the message is checked for a value of True.

```
hal.route('/digestive/verify/<msg>/<signature>/')
f verify(msg, signature):
  try:
      VK.verify(
          bytes.fromhex(signature),
          msg.encode(),
          hashfunc=HashFunc,
      )
  except:
      return {"error": "Signature verification failed"}

  verified_input = json.loads(msg)
  if "admin" in verified_input and verified_input["admin"] == True:
      return {"flag": FLAG}
  else:
      return {"error": f"{verified_input['username']} is not an admin"}
```

So we just add "admin" = true in the msg and copy signature form the last output.

In conclusion, we have the FLAG: crypto{thanx_for_ctf_inspiration_https://mastodon.social/@filippo/109360453402691894}

8. Bài luyện tập 2

	HMAC	MAC
Advantages	• Security: HMAC is designed to be resistant against various cryptographic attacks, including birthday attacks and differential attacks. • Keyed Hashing: uses a secret key – add a layer security. The key ensures that only parties with the correct key can generate and verify the MAC. • Flexibility: can be used with various hash functions, such as MD5, SHA-1, SHA-256, etc., allowing for flexibility in choosing the appropriate	• Efficiency: MAC algorithms compute faster than HMAC since they don't require the additional key processing steps. • Simplicity: MAC algorithms can be implemented with various cryptographic primitives. • Key Management: MAC algorithms often have simpler key management requirements compared to HMAC

	hash function based on security requirements.	
Disadvantages	• Performance Overhead: requires additional computational resources compared to a simple hash function. • Key Management: proper key management practices need to be followed, including secure key storage and distribution.	• Limited Flexibility: MAC algorithms are often tied to specific cryptographic primitives, which can limit the choice of algorithms available. • Vulnerable to Key Reuse: MACs should not be used with the same key for multiple messages, as it can lead to security vulnerabilities if one MAC is compromised.