Project Report for DDWD3343

Report computer security

Student name	Position
NG JUN KAI	Group leader
LOH LOG TIAN	Member
NUR ARIFSHAH BIN ROHIZAM	Member
LUCAS LEE KWONG YEK	Member

Contents

Project objective

1) Bluetooth Communication:
   - Enable seamless message and file exchange between devices via Bluetooth.
   - Ensure compatibility with various devices to maximize accessibility.

2) Message Confidentiality:
   - Use AES (Advanced Encryption Standard) for encrypting messages and files.
   - Use RSA for encrypting message and key.
   - ensuring only intended recipients can access the content.

3) Message Integrity:
   - Employ SHA-256 for hashing, allowing the receiver to verify that the content has not been tampered with during transmission.

4) Secure Large File Transfer:
   - AES (Advanced Encryption Standard) Leverage efficient encryption and compression techniques to support fast and secure transfer of large files.

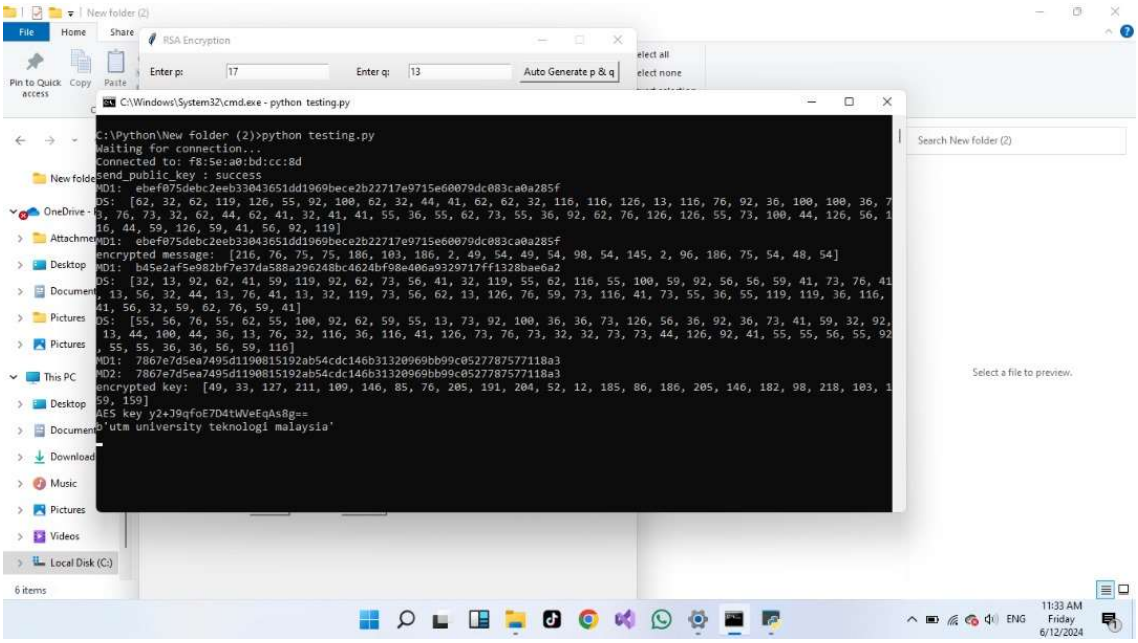
5) Authentication and Digital Signature:
   - Use RSA to generate digital signatures, providing authenticity and non-repudiation for messages.
   - Combine RSA with SHA-256 for signing, ensuring robust security.

6) Real-Time Performance:
   - Optimize cryptographic operations to maintain real-time messaging and fast file transfers without noticeable delays.

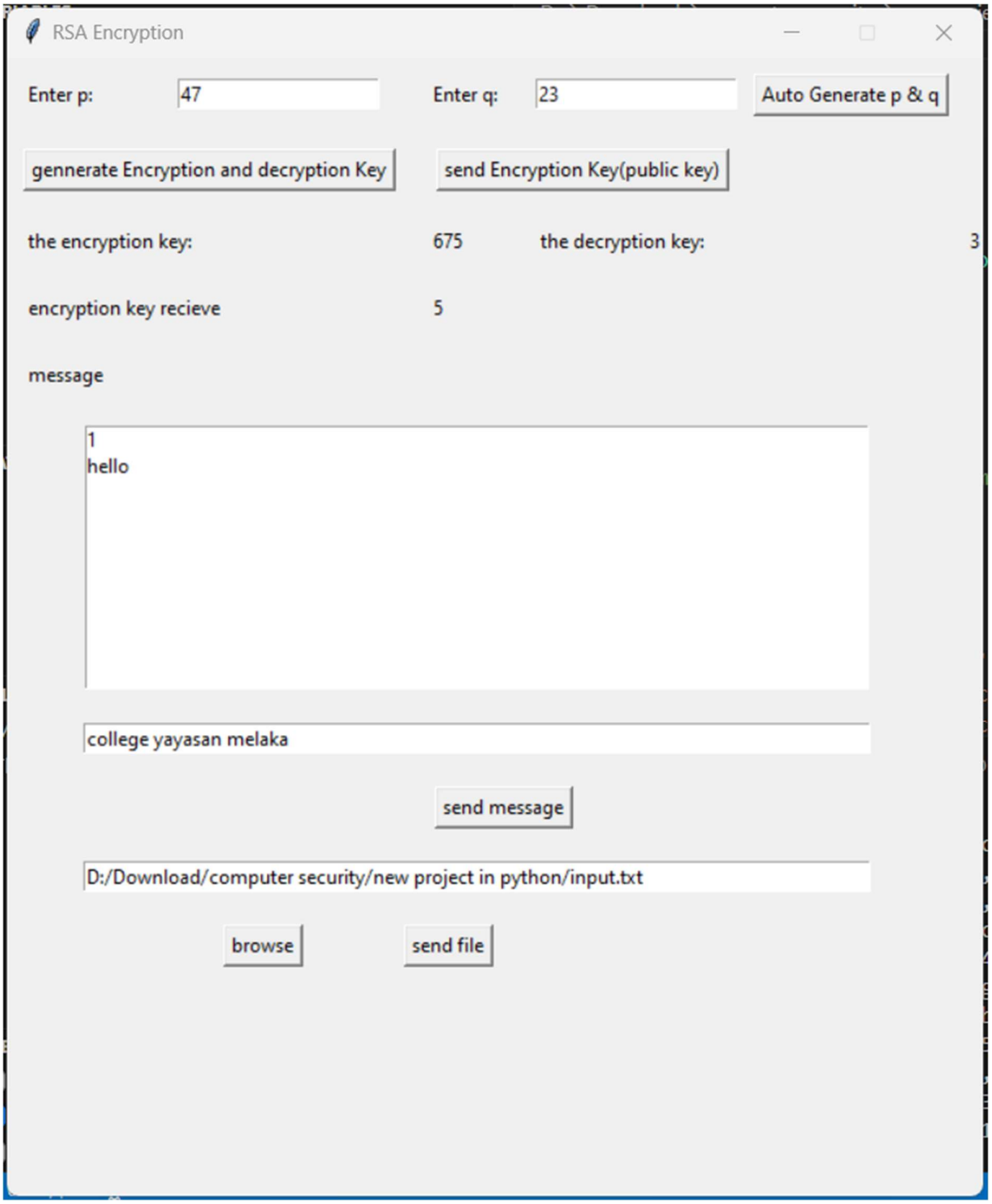
7) Potential Use Cases:
   - Secure peer-to-peer communication in sensitive applications such as business, healthcare, or education.
   - Quick sharing of encrypted files between collaborators in a secure environment.
   - Enhancing privacy for Bluetooth-enabled IoT devices.

Project implementation and interface

Receiver



Sender side




```
encrypted message:  [624, 108, 347, 347, 166]
plaintext :  b'utm university teknologi malaysia'
AES_key b'\xcbo\x89\xf6\xa7\xe8\x13\xb0\xf8\xb5e^\x12\xa0,\xf2'
AES_MD1:  7867e7d5ea7495d1190815192ab54cdc146b31320969bb99c0527787577118a3
AES_DS:  [55, 56, 76, 55, 62, 55, 100, 92, 62, 59, 55, 13, 73, 92, 100, 36, 36, 73, 126, 56, 36, 92, 36, 73, 41, 59, 32, 92, 13, 44, 100, 44, 36, 13, 76,
 32, 116, 36, 116, 41, 126, 73, 76, 73, 32, 32, 73, 73, 44, 126, 92, 41, 55, 55, 56, 55, 92, 55, 55, 36, 36, 56, 59, 116]
ciphertext zTMgUrpeQDRRIKoHAgerkRMOlg1P6ju5H7p9nkcJFSMU18CIq5v4qLCZ5u4W+AZc
```

```
MD1:  ebef075debc2eeb33043651dd1969bece2b22717e9715e60079dc083ca0a285f
DS:  [62, 32, 62, 119, 126, 55, 92, 100, 62, 32, 44, 41, 62, 62, 32, 116, 116, 126, 13, 116, 76, 92, 36, 100, 100, 36, 73, 76, 73, 32, 62, 44, 62, 41, 32
, 41, 41, 55, 36, 55, 62, 73, 55, 36, 92, 62, 76, 126, 126, 55, 73, 100, 44, 126, 56, 116, 44, 59, 126, 59, 41, 56, 92, 119]
MD1:  b45e2af5e982bf7e37da588a296248bc4624bf98e406a9329717ff1328bae6a2
DS:  [32, 13, 92, 62, 41, 59, 119, 92, 62, 73, 56, 41, 32, 119, 55, 62, 116, 55, 100, 59, 92, 56, 56, 59, 41, 73, 76, 41, 13, 56, 32, 44, 13, 76, 41, 13,
 32, 119, 73, 56, 62, 13, 126, 76, 59, 73, 116, 41, 73, 55, 36, 55, 119, 119, 36, 116, 41, 56, 32, 59, 62, 76, 59, 41]
MD1:  b45e2af5e982bf7e37da588a296248bc4624bf98e406a9329717ff1328bae6a2
encrypted message:  [624, 108, 347, 347, 166]
```

RSA encryption and decryption

RSA introduction

RSA algorithm is an asymmetric cryptography algorithm. Asymmetric refers to the fact that it operates on both the public key and the private key.

The pair (n, e) is the public key, and the pair (n, d) is the private key.

Encryption: When a message is encrypted, the sender calculates the ciphertext, which is equal to m^e mod n, using the recipient's public key (n, e). The plaintext message is indicated by the m.

Decryption: To decode an RSA-encrypted message, the recipient computes the plaintext message using their private key (n, d), where the plaintext message is equal to c^d mod n.

The foundation of RSA is the difficulty of factorizing large integers. The private key is obtained from the same two prime numbers, while the public key is made up of two numbers, one of which is the product of two huge prime numbers. Let p and q be the two selected prime numbers. The algorithm then determines their product, which is represented by n = p * q. The modulus n, which is used for both public and private keys, must be made public, but the values of p and q should remain confidential.
The integer e, whose value serves as the public exponent, is then chosen after the Carmecheals' totient function is calculated using p and q. The value of d, which serves as the private exponent, must then be determined.

The key size determines the encryption strength, and the encryption strength grows exponentially if the key size is doubled or tripled. Although RSA keys are usually 1024 or 2048 bits long, experts predict that 1024-bit keys may soon be cracked.

RSA example

Generating Public Key:

Select two prime no's. Suppose **P = 53 and Q = 59.**

Now First part of the **Public key : n = P*Q = 3127.**

We also need a small exponent say e :

But e Must be An integer.

Not be a factor of Φ(n).

1 < e < Φ(n)

Let us now consider it to be equal to 3.

Our Public Key is made of n and e

Generating Private Key:

We need to calculate Φ(n) :

Such that **Φ(n) = (P-1)(Q-1)**

so, Φ(n) = 3016

Now calculate Private Key, **d :**

d = (k*Φ(n) + 1) / e for some integer k

For k = 2, value of d is 2011.

Now we are ready with our – **Public Key (n = 3127 and e = 3) and Private Key(d = 2011)**
Now we will encrypt "**HI**":

Convert letters to numbers : **H = 8 and I = 9**

Thus, **Encrypted Data c = (89^e)mod n**

Thus our Encrypted Data comes out to be **1394**

Now we will decrypt **1394** :

Decrypted Data = (c^d)mod n

Thus our Encrypted Data comes out to be **89**

8 = H and I = 9 i.e. "HI".

RSA vulnerability

SHA introduction

SHA256withRSA is a hybrid cryptographic algorithm that leverages the SHA-256 hashing algorithm and the RSA digital signature scheme. It utilizes SHA-256 to generate a hash value for the data and then signs the hash using RSA with a private key. The result signature can be verified with a corresponding public key, ensuring the integrity and authenticity of the data.

SHA-256 is a member of the SHA2 family of secure hash functions, and there are not currently any cryptographic weaknesses publicly known for SHA2. It might be less secure than SHA-512, but 256 bits is already completely impractical to brute force. The only viable attacks would require finding a weakness in the hash algorithm itself, and it's not necessarily the case that SHA-512 would be more resistant to such an attack than SHA-256. There is a new SHA3 standard, but it's not yet widely implemented, so browsers probably wouldn't be able to verify the certificate's signature at all if they used SHA3 in the signing algorithm.

RSA is a current standard for public-key cryptography, and a properly generated 2048-bit RSA key is strong enough to resist factoring for decades. We can use a 4096-bit key if we want to (it'll take a lot longer to generate and slightly longer to use, but once the certificate's signature is verified, that doesn't matter anymore), and that would take even longer to break. However, neither certificate is valid for more than two years anyhow.

SHA example

Step 1: Convert Message to Binary

Input: "hello world"

ASCII: 104 101 108 108 111 32 119 111 114 108 100

Binary: 01101000 01100101 01101100 01101100 01101111 00100000 01110111 01101111 01110010 01101100 01100100

Step 2: Padding

SHA-256 requires the input length to be a multiple of 512 bits:

Append a 1 bit:
...01110010 01101100 01100100 1

Add 0 bits to make the total length 448 bits (mod 512):
...01110010 01101100 01100100 10000000 00000000 ...

Append the original message length (in bits) as a 64-bit binary:
Length = 88 bits (11 characters × 8).
Append: 00000000 00000000 00000000 00000000 00000000 00000000 01011000.

Final padded message (512 bits):

01101000 01100101 01101100 01101100 01101111 00100000 01110111 01101111

01110010 01101100 01100100 10000000 00000000 ... 01011000

Step 3: Initialize Constants

SHA-256 uses 8 initial hash values (H0 to H7) and 64 constants (K0 to K63). These values are derived from fractional parts of square roots of prime numbers.

Initial hash values (H):

H0 = 6a09e667, H1 = bb67ae85, H2 = 3c6ef372, ...

Round constants (K):

K0 = 428a2f98, K1 = 71374491, ...

Step 4: Process Each Chunk

The padded message is divided into 512-bit chunks. For each chunk:

1. **Prepare Message Schedule**:

   Expand the 512-bit chunk into 64 words (W0 to W63) of 32 bits each.

   For the first 16 words: Use the chunk's original 512 bits.

   For W16 to W63:

   $W_t = \sigma1(W_{t-2}) + W_{t-7} + \sigma0(W_{t-15}) + W_{t-16}$

   where:

   $\sigma0(x) = ROTR7(x) \text{ XOR } ROTR18(x) \text{ XOR } SHR3(x)$

   $\sigma1(x) = ROTR17(x) \text{ XOR } ROTR19(x) \text{ XOR } SHR10(x)$

2. **Initialize Working Variables**:

   a = H0, b = H1, c = H2, ..., h = H7

3. **Compression Loop** (64 rounds): For each round t (0 to 63):

   $T1 = h + \Sigma1(e) + Ch(e, f, g) + K_t + W_t$

   $T2 = \Sigma0(a) + Maj(a, b, c)$

   h = g, g = f, f = e, e = d + T1

   d = c, c = b, b = a, a = T1 + T2

   where:

   $\Sigma0(x) = ROTR2(x) \text{ XOR } ROTR13(x) \text{ XOR } ROTR22(x)$

   $\Sigma1(x) = ROTR6(x) \text{ XOR } ROTR11(x) \text{ XOR } ROTR25(x)$

   $Ch(x, y, z) = (x \text{ AND } y) \text{ XOR } ((\text{NOT } x) \text{ AND } z)$

   $Maj(x, y, z) = (x \text{ AND } y) \text{ XOR } (x \text{ AND } z) \text{ XOR } (y \text{ AND } z)$

4. **Update Hash Values**:

   H0 = H0 + a, H1 = H1 + b, ..., H7 = H7 + h

Step 5: Output Final Hash

After processing all chunks, concatenate H0 to H7 to produce the final 256-bit hash:

b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9

RSA limitation

AES introduction

Encrypting big files directly with techniques like RSA is wasteful and virtually impossible since RSA is meant to encrypt little quantities of data, such as cryptographic keys or brief communications. RSA uses difficult mathematical operations (modular exponentiation) over big integers, making it computationally costly and limiting the amount of data that can be encrypted to a fraction of the RSA key size (for example, a 2048-bit RSA key can only encrypt data up to around 256 bytes).

AES is a symmetric method that employs the same 128, 192, or 256-bit key for encryption and decryption (the security of an AES system grows exponentially with key length). With a 128-bit key, cracking AES by testing each of the 2128 possible key values (a "brute force" assault) is so computationally demanding that even the fastest supercomputer would take more than 100 trillion years to complete. In reality, AES has never been broken and, according to current technical trends, is anticipated to stay safe for many years to come.

The AES algorithm performs a sequence of mathematical modifications on each 128-bit data block. Because to its minimal processing needs, AES may be utilized with consumer computing devices like laptops and smartphones, as well as to swiftly encrypt massive volumes of data. For example, the IBM z14 mainframe series employs AES to allow ubiquitous encryption, which encrypts all data in the system, whether at rest or in transit.

Large files are encrypted using AES (Advanced Encryption Standard). AES is ideal for encrypting huge datasets since it processes data in blocks and is computationally lighter than RSA. To combine the strengths of RSA and AES, a hybrid encryption method is utilized.

AES example

Encrypt “hello world”

First step padding input into 16 byte and generate random key in 16 byte

1. Convert "hello world" to bytes:

   ASCII: hello world → [104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100]

   Pad to 16 bytes: [104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100, 5, 5, 5, 5, 5]

2. Represent as a 4x4 matrix (AES state):

   [104, 111, 114, 5]

   [101, 32, 108, 5]

   [108, 119, 100, 5]

   [108, 111, 5, 5]

3. Key (example, 16 bytes):

   [0x2b, 0x7e, 0x15, 0x16]

   [0x28, 0xae, 0xd2, 0xa6]

   [0xab, 0xf7, 0x32, 0xfe]

   [0xf9, 0x19, 0x59, 0xc7]

AES Steps

Step 1: Add Round Key

Each byte in the state is XORed with the corresponding byte of the key:

Example XOR of 104 (0x68) with 0x2b: 0x68 ^ 0x2b = 0x43.

Resulting state:

[0x43, 0x1d, 0x61, 0x13]

[0x55, 0x9e, 0xba, 0xa3]

[0x87, 0x86, 0x6a, 0xfb]

[0xb1, 0x18, 0x5c, 0xc2]

Step 2: SubBytes

Each byte is substituted using the AES S-Box. For example:

0x43 → S_BOX[0x43] = 0xa5.

After substitution:

[0xa5, 0x69, 0x85, 0x7c]

[0xfe, 0x91, 0x3a, 0x6b]

[0xc6, 0xca, 0x9d, 0xb2]

[0x47, 0x3f, 0x72, 0xde]

Step 3: Shift Rows

The rows are shifted left by 0, 1, 2, and 3 positions:

[0xa5, 0x69, 0x85, 0x7c]

[0x91, 0x3a, 0x6b, 0xfe]

[0x9d, 0xb2, 0xc6, 0xca]

[0xde, 0x47, 0x3f, 0x72]

Step 4: Mix Columns

Each column is multiplied by a fixed matrix in GF(2^8). Example for the first column:

[a5, 91, 9d, de] × [2, 3, 1, 1]

Result (simplified):

[0x57, 0x4c, 0x19, 0x73]

After Mix Columns:

[0x57, 0xb1, 0x34, 0x9f]

[0x4c, 0x6d, 0xab, 0x3a]

[0x19, 0x82, 0xd7, 0x1e]

[0x73, 0x5e, 0xf8, 0xa9]

Steps 5-10: Repeat for 9 More Rounds

The above steps are repeated with additional key scheduling for each round. In the final round, **Mix Columns** is skipped.

Result

The encrypted output is:

[0x47, 0x95, 0x2c, 0x7e]

[0x6d, 0x8b, 0x91, 0x4f]

[0xa8, 0xf3, 0x4e, 0x76]

[0x1d, 0xe9, 0x32, 0xab]

Decrypt Ciphertext

AES Decryption Steps

Step 1: Add Round Key (Last Round Key)

XOR the ciphertext with the final round key (computed during key schedule). Example for the first byte:

0x47 ^ 0x2b = 0x6c.

Resulting state:

[0x6c, 0xe1, 0x39, 0x68]

[0x45, 0x25, 0x43, 0xe9]

[0x03, 0x04, 0x7c, 0x88]

[0x24, 0xf0, 0x6b, 0x6c]

Step 2: Reverse Shift Rows

Reverse the row shifts:

- Row 0: No shift.
- Row 1: Shift right by 1.
- Row 2: Shift right by 2.
- Row 3: Shift right by 3.

Reversed:

[0x6c, 0xe1, 0x39, 0x68]

[0xe9, 0x45, 0x25, 0x43]

[0x7c, 0x88, 0x03, 0x04]

[0x6c, 0x24, 0xf0, 0x6b]

Step 3: Reverse SubBytes

Use the **inverse S-Box** to substitute bytes back to their original values. For example:

0x6c → INV_S_BOX[0x6c].

Resulting state:

[0xa5, 0x69, 0x85, 0x7c]

[0xfe, 0x91, 0x3a, 0x6b]

[0xc6, 0xca, 0x9d, 0xb2]

[0x47, 0x3f, 0x72, 0xde]

Step 4: Reverse Mix Columns

Apply the **inverse Mix Columns** transformation. For example:

[0xa5, 0xfe, 0xc6, 0x47] × Inverse Mix Matrix.

Resulting state:

[0x43, 0x1d, 0x61, 0x13]

[0x55, 0x9e, 0xba, 0xa3]

[0x87, 0x86, 0x6a, 0xfb]

[0xb1, 0x18, 0x5c, 0xc2]

Steps 5-10: Repeat for Previous Rounds

Repeat steps **Add Round Key**, **Shift Rows**, **SubBytes**, and **Mix Columns** for all 10 rounds in reverse order, applying the corresponding round keys.

Final Step: Remove Padding

After decrypting all rounds, the state returns to:

[104, 101, 108, 108]

[111, 32, 119, 111]

[114, 108, 100, 5]

[5, 5, 5, 5]

Convert back to bytes and remove padding (0x05):

[104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100]

Convert to ASCII: "hello world".