

Extracting and Managing Keys from QKD to Enhance Cryptographic Techniques for File Encryption (Lab)

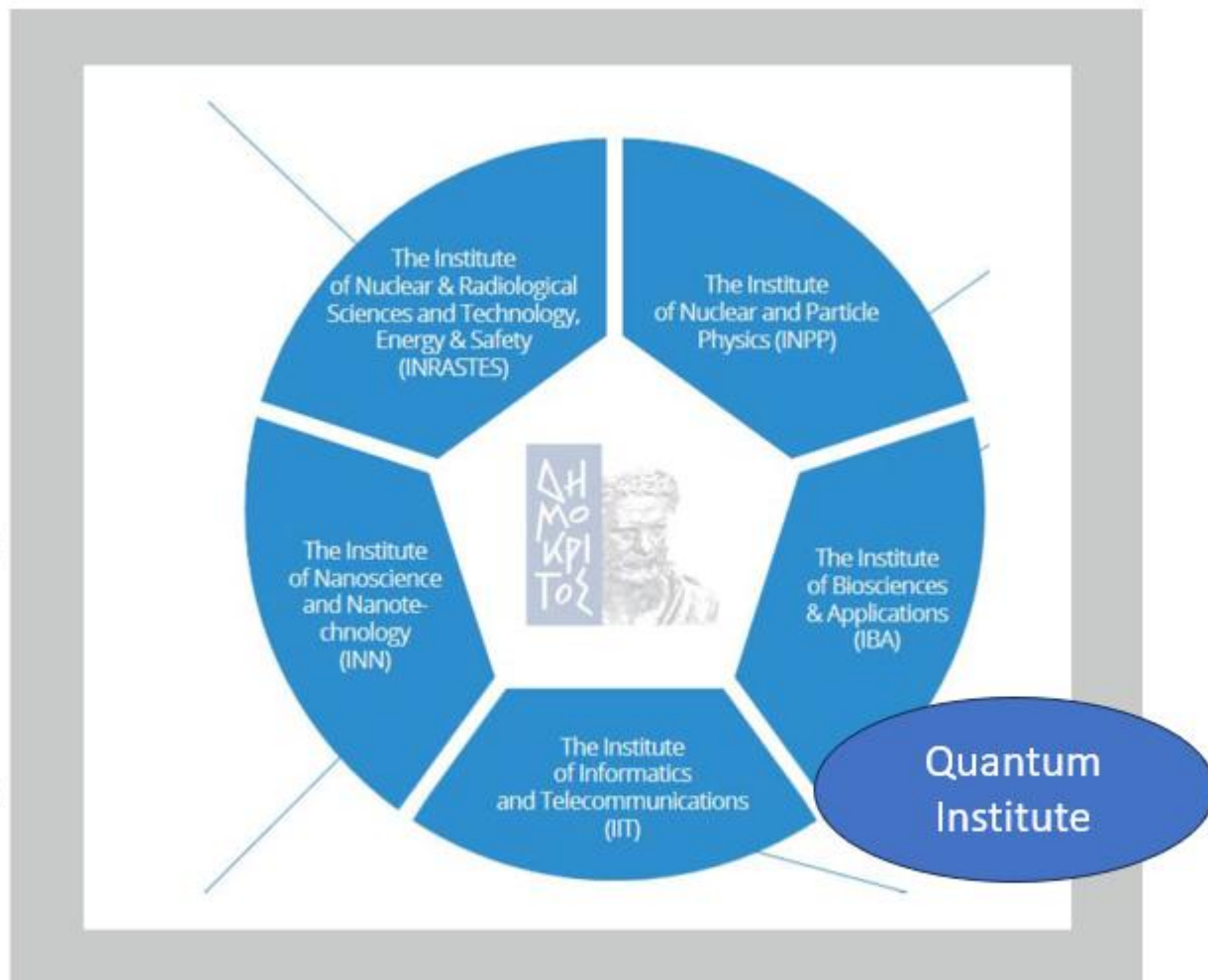
Dr. Homer Papadopoulos, NCSRDI

Antonis Korakis, NCSRDI

HellasQCI Third Training Event, Crete, 04-05 September 2024

- ✓ Symmetric key Cryptography
- ✓ Public/symmetric key Cryptography
- ✓ Hashing, Digital Signatures and Certificates
- ✓ Demo: Communicate with QKD and GET keys

Who we are - NCSR Demokritos - Syndesis Ltd



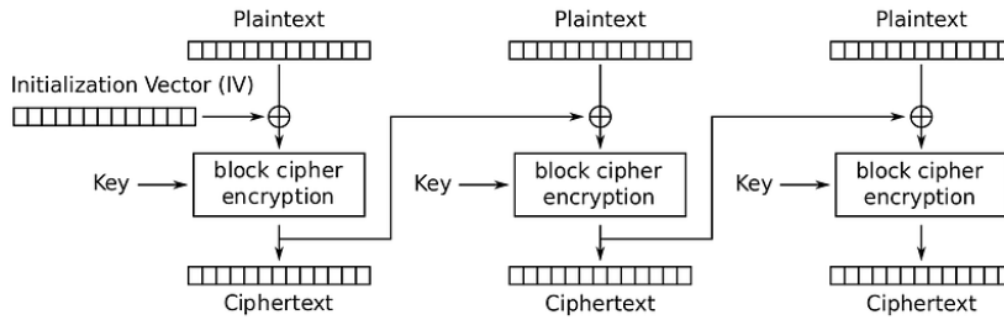
Cryptography is the science of securing information by transforming it into a format that is unreadable to anyone except those who have the key to decrypt it.

The primary goals of cryptography are:

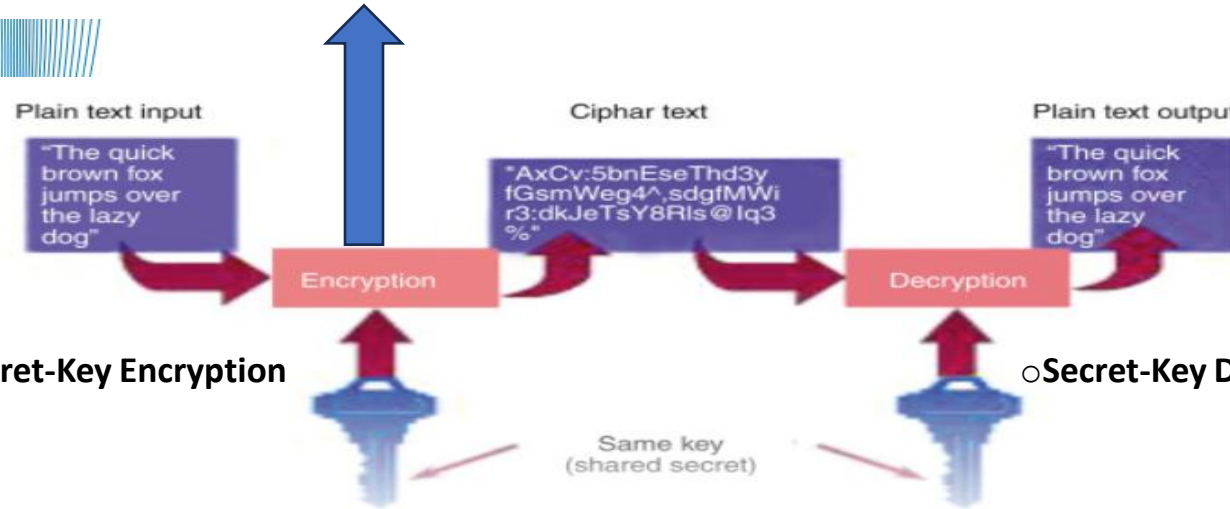
- 1. Confidentiality:** Ensuring that information is only accessible to those who are authorized to view it.
- 2. Integrity:** Protecting information from being altered by unauthorized parties during transmission.
- 3. Authentication:** Verifying the identity of the parties involved in communication.
- 4. Non-repudiation:** Ensuring that a party cannot deny the authenticity of their signature on a document or the sending of a message.



Symmetric Key Cryptography (AES)



Cipher Block Chaining (CBC) mode encryption

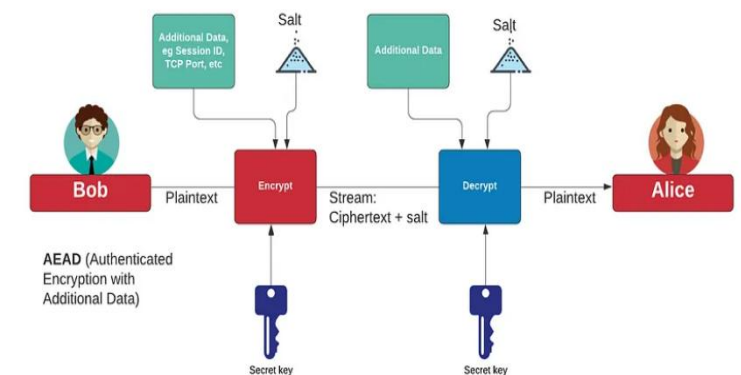


○ Secret-Key Encryption

○ Secret-Key Decryption

There are multiple chipper modes are available in AES:

- ECB mode: Electronic Code Book mode
- **CBC mode: Cipher Block Chaining mode**
- CFB mode: Cipher Feedback mode
- OFB mode: Output FeedBack mode
- CTR mode: Counter mode
- **GCM mode: Galois/Counter mode**



Symmetric encryption schemes

Feature	OTP	ChaCha20	AES
Security Level	Information-Theoretic	Computational	Computational
Authentication	Not inherent, requires additional mechanisms (MAC, digital signature)	Not inherent, often used with Poly1305 for authenticated encryption	Not inherent, often used with GCM or CBC-MAC for authenticated encryption
Key Management	Impractical for long messages	Practical, with a shorter key	Practical, with standardized key lengths eg 256 bits
Performance	High key management overhead	Fast and efficient	Efficient, but can be complex to implement correctly
Common Use Cases	Highly sensitive, low-volume communications (e.g., diplomatic, military messages). Eg for one page of 1000 characters there is need for a key length of 8.000 bit	Real-time communications , secure communications (e.g., TLS 1.3)	Data encryption , secure communications (e.g., HTTPS, VPNs)

A stream cipher can be considered a "pseudo-One-Time Pad" (pseudo-OTP) because it mimics the functionality of the One-Time Pad (OTP) in a practical way, using a pseudorandom number generator (PRNG) to create a keystream rather than relying on a truly random, one-time-use key.

Symmetric Key Cryptography (AES)

AES Encryption / Decryption Tool

AES Encryption

Encryption Text

Secret Key

Encryption Key Size

128 Bits 192 Bits 256 Bits

Encryption Mode

CBC ECB

IV (optional)

Output format

Base64 HEX

Encrypt

Encrypted Text

AES Decryption

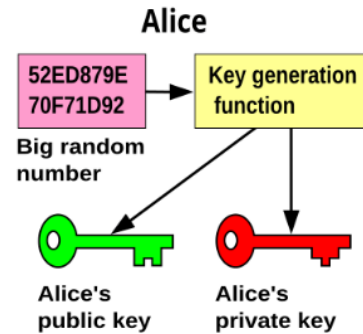
Encrypted Text

Decrypted Text

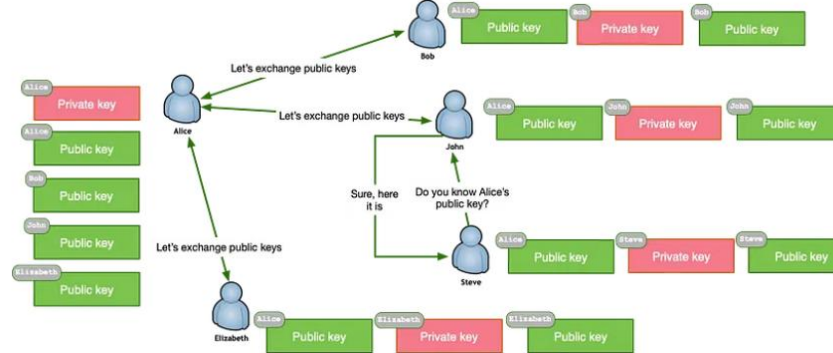
Symmetric Cryptography Weaknesses:

- The secure distribution of the key
- Trust problem - Authentication

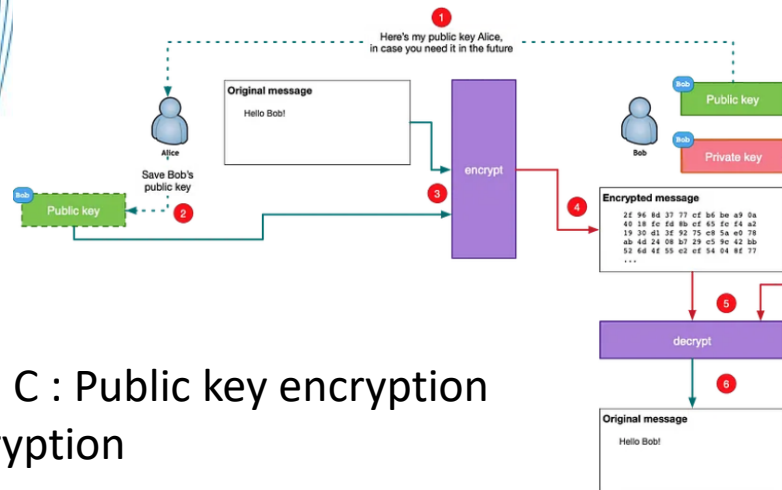
Public or Asymmetric Key Cryptography (Public-Private Key)



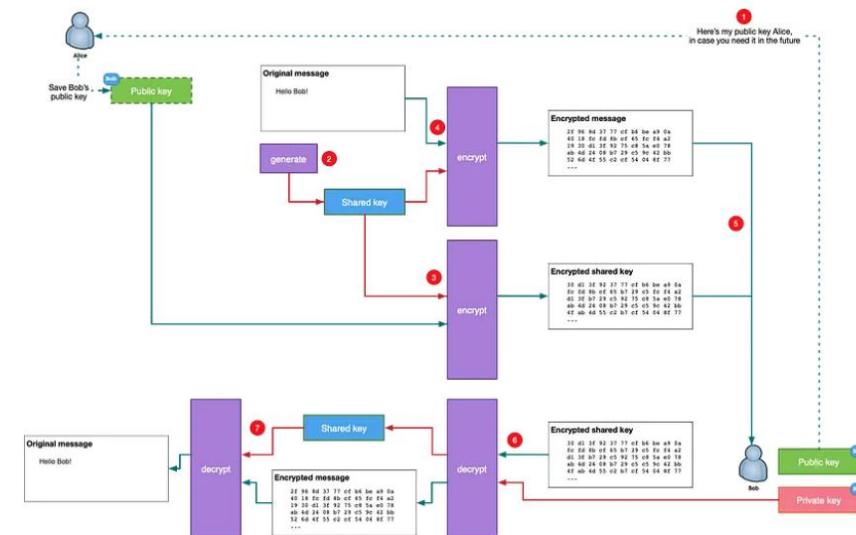
Step A : Generate public-private keys



Step B: Users maintain key-pairs and exchange public keys



Step C : Public key encryption decryption



Step D : Hybrid Encryption

<https://betterprogramming.pub/an-introduction-to-public-key-cryptography-3ea0cf7bf4ba>

Public or Asymmetric Key Cryptography (Public-Private Key)

Key Generation

Select p, q p and q both prime
 Calculate $n = p \times q$
 Calculate $\phi(n) = (p-1)(q-1)$
 Select integer e $\gcd(\phi(n), e) = 1; 1 < e < \phi(n)$
 Calculate d $d \equiv e^{-1} \pmod{\phi(n)}$
 Public key $KU = \{e, n\}$
 Private key $KR = \{d, n\}$

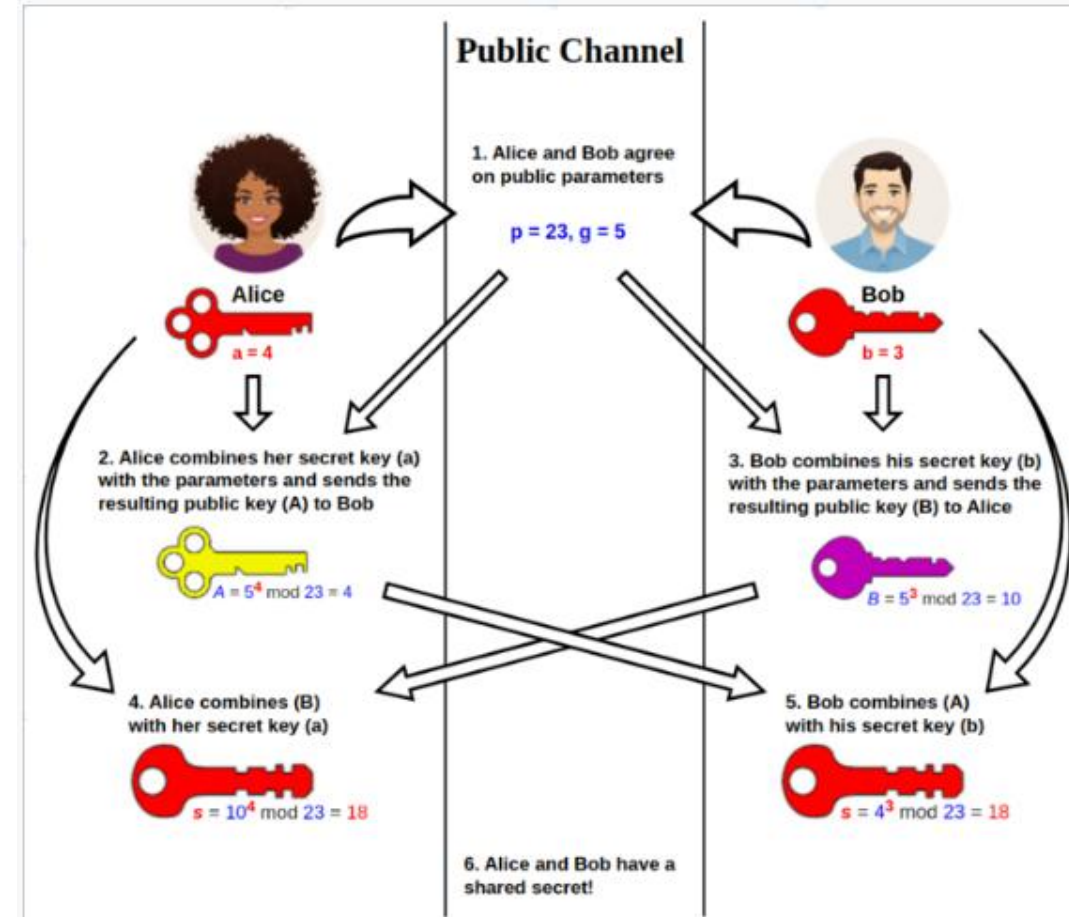
Encryption

Plaintext $M < n$
 Ciphertext $C = M^e \pmod{n}$

Decryption

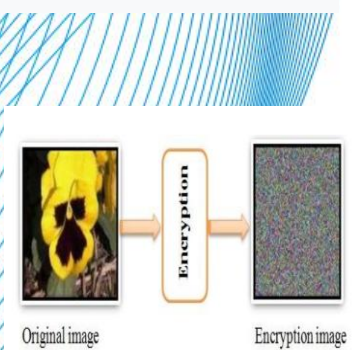
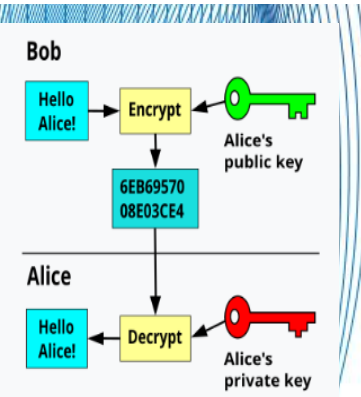
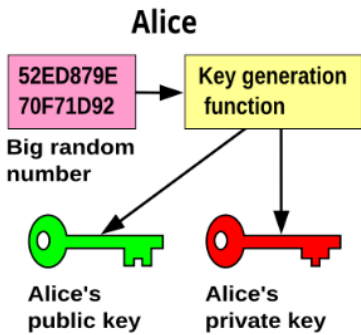
Ciphertext C
 Plaintext $M = C^d \pmod{n}$

RSA (Rivest-Shamir-Adleman) Algorithm for encrypt –decrypt:
https://www.researchgate.net/publication/328828460_Enhancing_Steganography_Techniques_in_Digital_Images



Diffie-Hellman Key Exchange Algorithm for key exchange :
https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange

Public Key Cryptography: Create Keys



RSA Encryption / Decryption Tool

Create RSA public / private keys

Encryption Mode

512 Bits

Public Key

Public Key

Private Key

Private Key

Create public / Private key

RSA Encryption

Encryption Text

Encryption Text

Encrypted Text

Public key

Public key

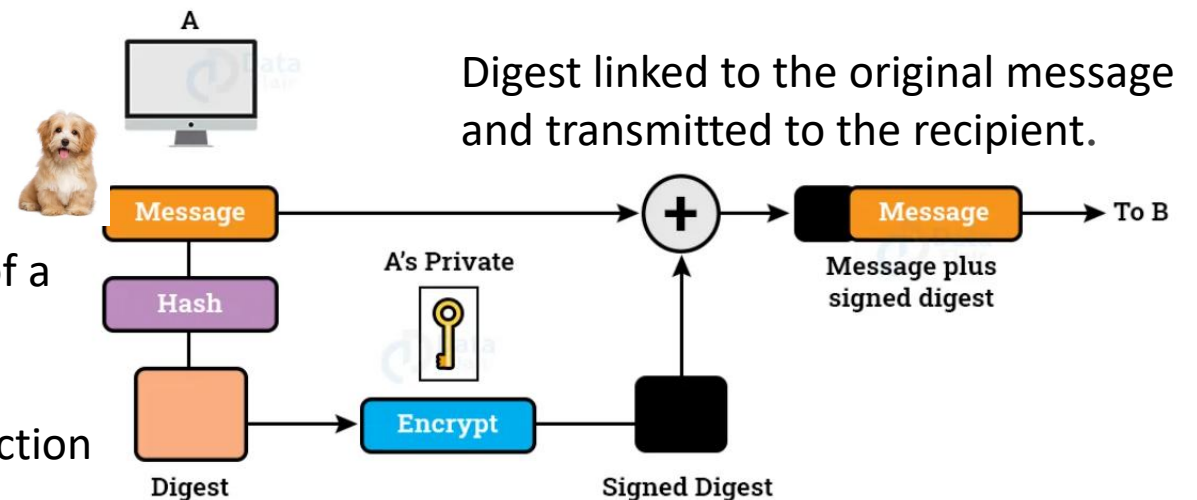
Digital signatures are necessary piece for **Authentication**, because they verify the identity of the sender of a message.

Eg. Alice wants to send Bob a lengthy document, e.g. a book in pdf file, being sure than no one else can claim to be the author. A protocol that can solve her problem is following:

1. Alice creates a one-way **hash** of a document, Alice's **digest (smaller version of the Book)**.
2. Alice encrypts the digest with her private key, thereby signing the document.
3. Alice sends the document, her public key and the signed digest to Bob.

The message digest is generated with the help of a hash function eg MD5, SHA-256 etc.

From the variable-length message, the hash function generates a fixed-size digest.



SHA-256 Hashing Tool

SHA-256 Hash

Text to hash

Hashed output

Create SHA-256 Hash

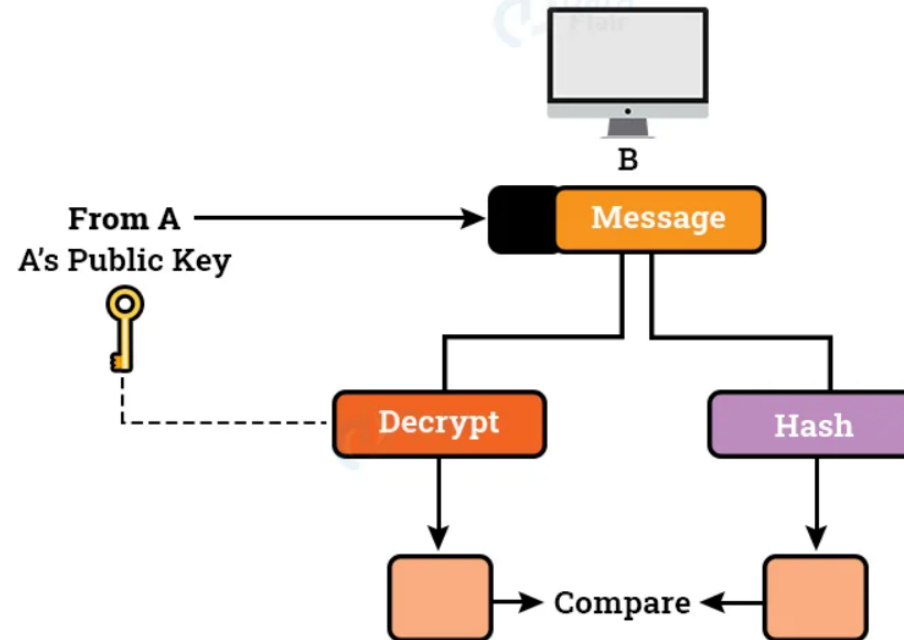
SHA-256 Hash Match

Text to match

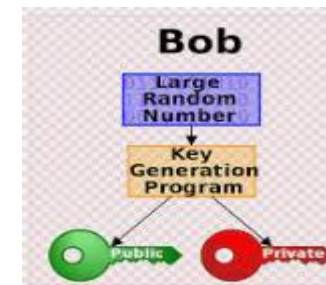
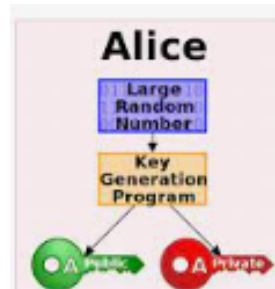
Hash to match

SHA-256 (Secure Hash Algorithm 256-bit) represents a cornerstone in modern cryptography, renowned for its robust data integrity verification capabilities. Developed by the National Security Agency (NSA) in the United States, SHA-256 is a member of the SHA-2 family of cryptographic hash functions. Conceived by the NSA's National Institute of Standards and Technology (NIST) in 2001, SHA-256 generates a fixed-size hash value of 256 bits from input data of any size, ensuring a high level of collision resistance. Its extensive

1. Bob decrypts Alice's digest with her public key.
 2. Bob creates a one-way hash of the document that Alice has sent, Bob's digest.
 3. Bob compares his digest with Alice's to find out if they match
- If the signed hash matches the hash he generated, the signature is valid.



Demo of Encrypt - Decrypt



PGP Tool

A simple and secure online client-side PGP Key Generator, Encryption and Decryption tool. Generate your PGP Key pairs, encrypt or decrypt messages easily with a few clicks.

Generate PGP Keys Sign Verify Encrypt (+Sign) Decrypt (+Verify) FAQ About

Options

Enter your name

Required

Required

Email address: Why it is required?

Required

Required

Required

Required

Passphrase: What is this?

Public Key

Your public key will be generated here.

[Learn More](#)

Private Key

Your private key will be generated here.

[Learn More](#)

PGP Tool

A simple and secure online client-side PGP Key Generator, Encryption and Decryption tool. Generate your PGP Key pairs, encrypt or decrypt messages easily with a few clicks.

Generate PGP Keys Sign Verify Encrypt (+Sign) Decrypt (+Verify) FAQ About

Options

Enter your name

Required

Required

Email address: Why it is required?

Required

Required

Required

Required

Passphrase: What is this?

Public Key

Your public key will be generated here.

[Learn More](#)

Private Key

Your private key will be generated here.

[Learn More](#)

Demo of Encrypt - Decrypt

[Generate PGP Keys](#)
[Sign](#)
[Verify](#)
[Encrypt \(+Sign\)](#)
[Decrypt \(+Verify\)](#)
[FAQ](#)
[About](#)

Receiver's Public Key


Paste the public key here. (RSA only)

[Choose File](#) No file chosen

Signer's Private Key (For signing purpose)

Paste the private key here. RSA or ECC is supported.

[Choose File](#) No file chosen

 Passphrase for private key

Your Message in Plain Text

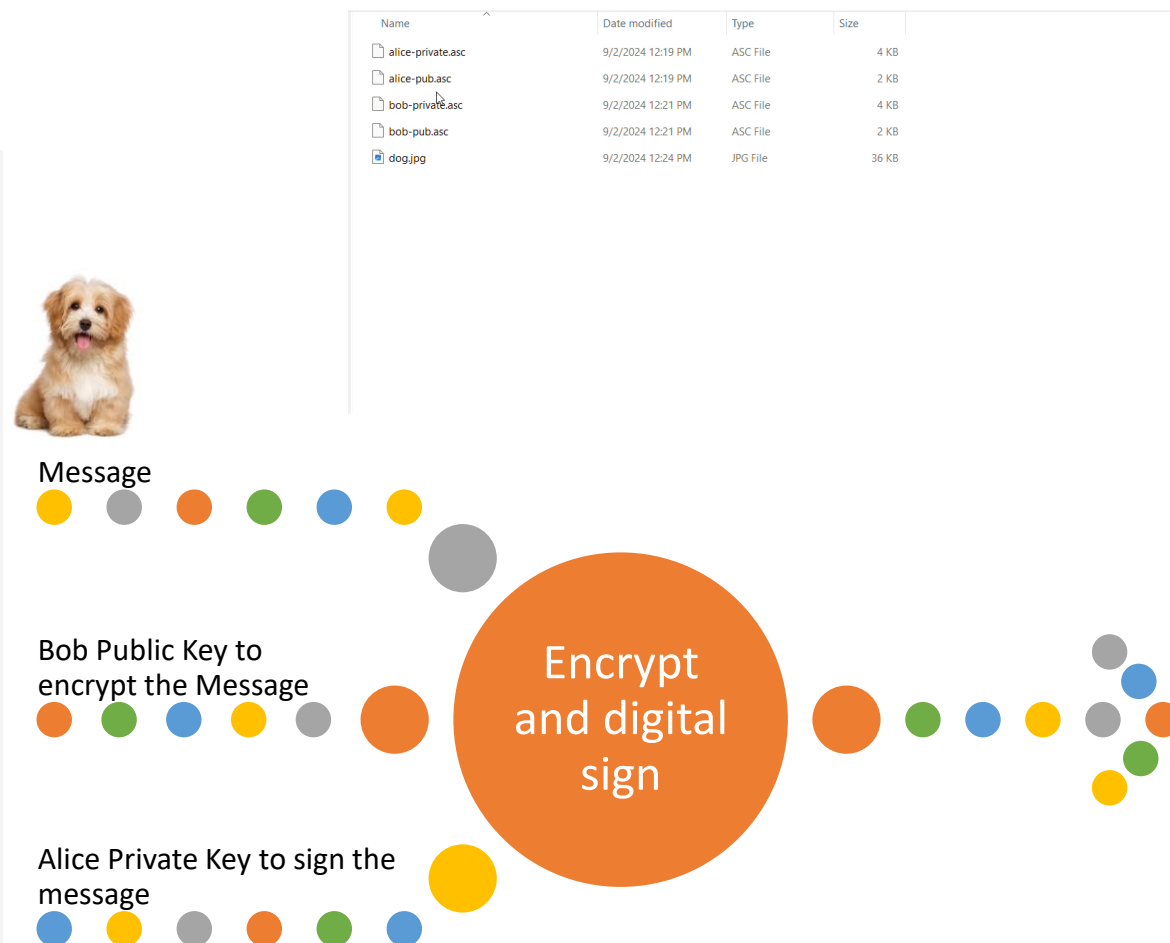
Write your message here.

[Choose File](#) No file chosen

[Encrypt the message](#)

Encrypted PGP Message

Here you'll see the encrypted and signed message.



Demo of Encrypt - Decrypt

Generate PGP Keys
Sign
Verify
Encrypt (+Sign)
Decrypt (+Verify)
FAQ
About

Receiver's Private Key (For decryption purpose)

Paste the private key here to decrypt. RSA key only.

Choose File No file chosen

Passphrase for private key

Signer's Public Key

Paste the signer's public key here if the message is signed. ECC key is supported. (Leave this field if the message is not signed.)

Choose File No file chosen

Encrypted PGP Message

Enter the encrypted message here.

Choose File No file chosen

Decrypt the message

Decrypted Message in Plain Text

Here you'll see the decrypted message.

Encrypted Signed Message



Bob Private Key to Decrypt the Message



Alice Public Key to Verify the Sender



Decrypt and Verify the Sender

Base64 to PNG



Input Base64 ?

Import from file
Save as...
Copy to clipboard

Output PNG

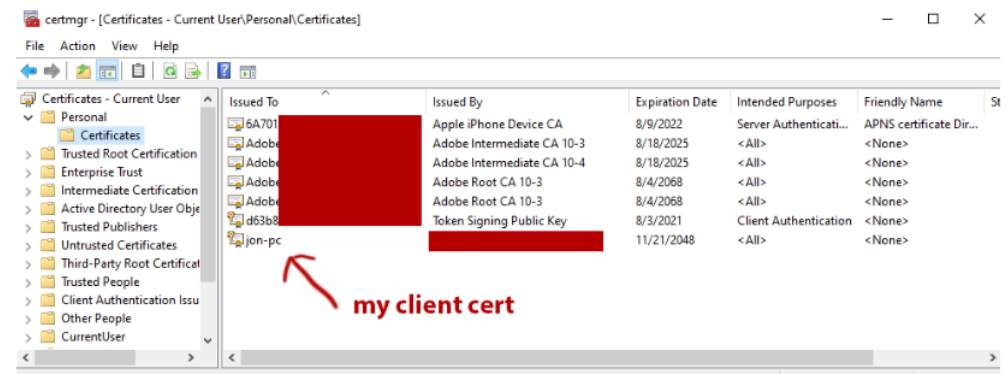
← Start by loading input

Chain with...
Save as...
Copy to clipboard

- **Public-Private Key cryptography is not enough since :**

- The complexity in large systems with numerous public keys can lead to security risks, such as untrusted keys being mistakenly accepted.
- **Need for Trust since** public-private key pairs alone do not verify the identity of the key owner, leading to security breaches if the wrong entities are trusted.
- Attackers can intercept communications and substitute fake public keys, leading to **Man-in-the-Middle** attacks.
- **Need for standard methods eg to replace** compromised keys or ensure that expired keys are no longer used.

Therefore the new concept of Certificates can address these problems by providing a trusted mechanism for verifying identities, preventing attacks, managing keys at scale, revoking compromised keys, and ensuring compliance with security standards. Certificates can be used in smart cards, IoT, SSL/TLS protocols etc.



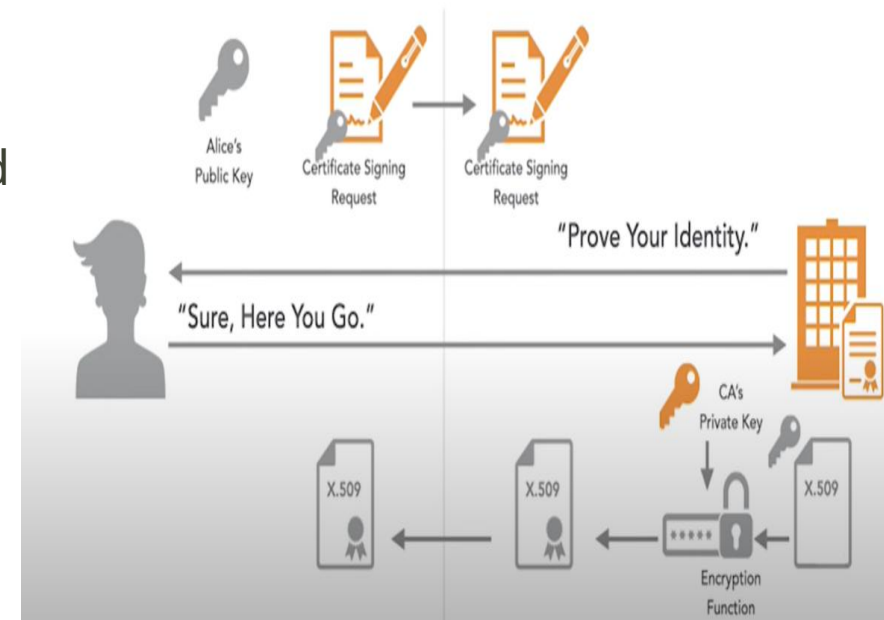
Certificate-Based Authentication (CBA) uses digital certificates to verify the identity of users, devices, or machines, enhancing security beyond traditional methods.

Certificate Authorities (CAs) are trusted entities that issue Digital Certificates, which are electronic documents used to prove the ownership of a public-private key pair.

The popular digital certificate X.509 includes the below elements:

- The public key
- The user or device's name
- The name of the Certificate Authority (CA) that issued the certificate
- The date from which the certificate is valid
- The expiry date of the certificate
- The version number of the certificate data
- A serial number

Creating a Digital Certificate





CLEAR FORM FIELDS

Country Name

State or Province Name

Organization Name

Common Name, the domain

Valid days

Passphrase to protect the private key *(Optional)*

Locality Name *(Optional)*

Organization Unit Name *(Optional)*

Email Address *(Optional)*

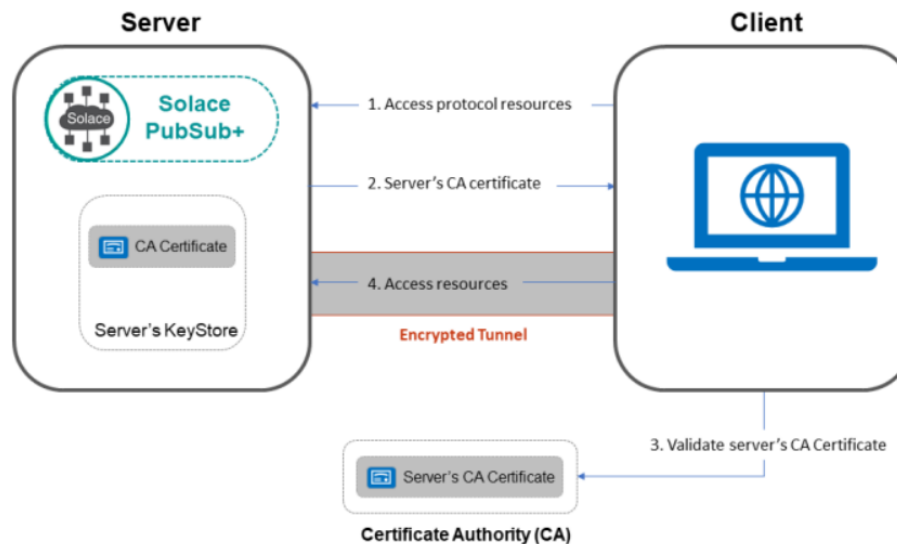
Bits to generate the private key

Digest Algorithm

One-way SSL / Server Certificate Authentication

In one-way SSL authentication (Server Certificate Authentication), only the client validates the server; the server does not verify the client application.

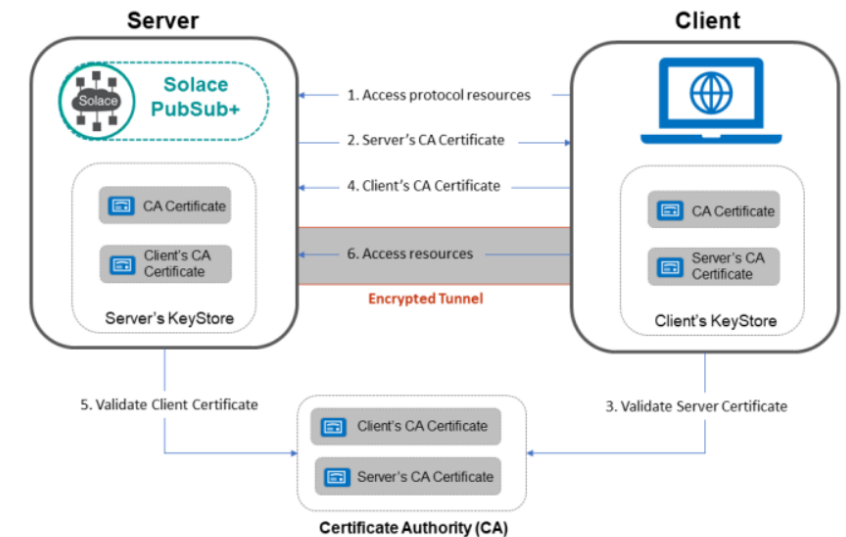
When implementing one-way SSL authentication, the server application shares its public certificate with the client.



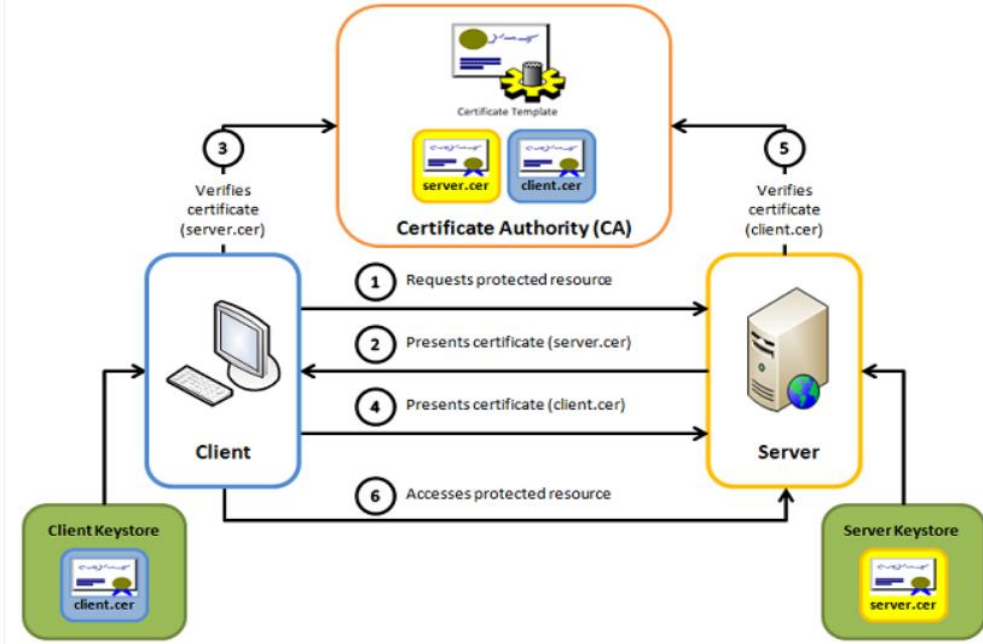
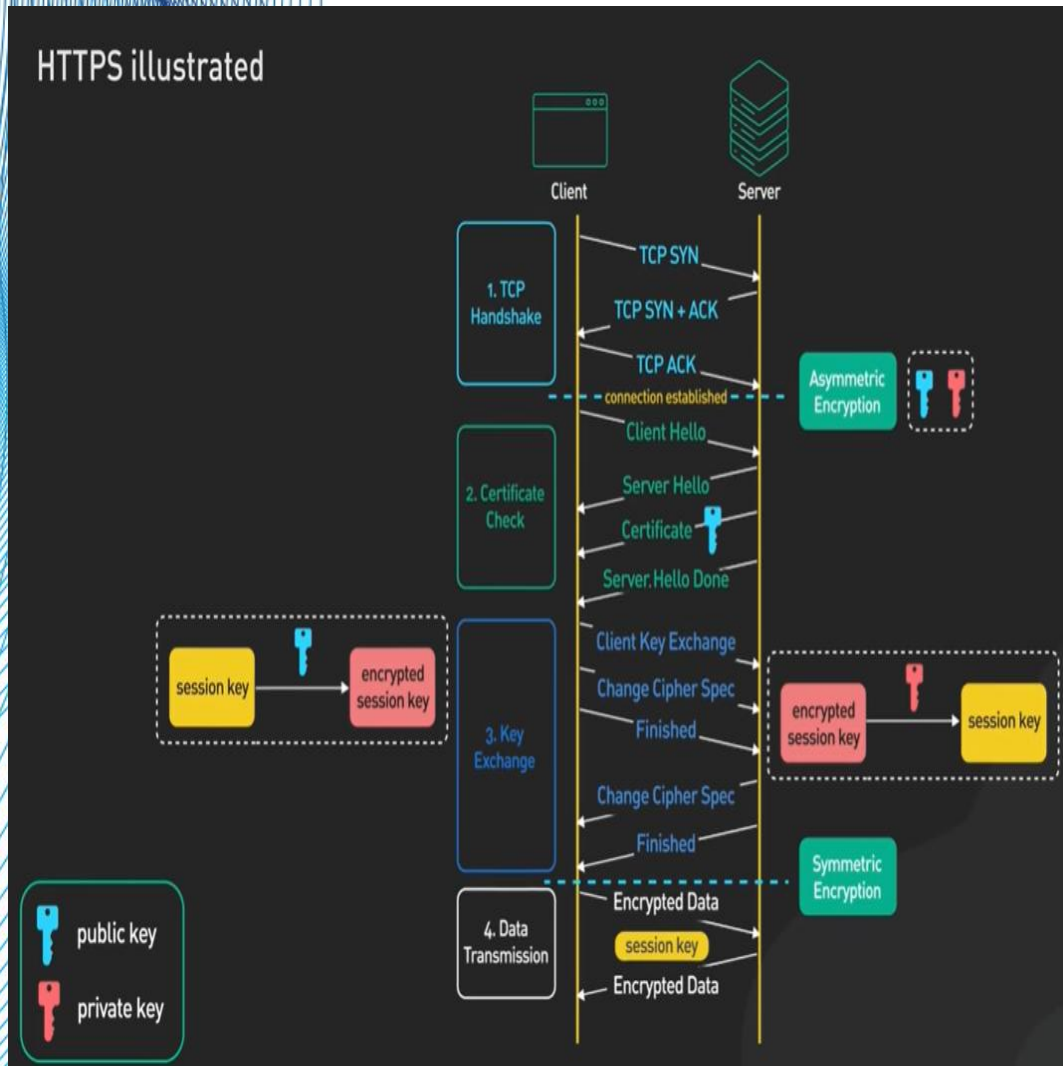
Two-way SSL/ Client Authentication

In two-way SSL authentication, the client application verifies the identity of the server application, and then the server application verifies the identity of the client application. Both parties share their public certificates, and then validation is performed. Two-way SSL authentication works with a mutual handshake by exchanging the certificates.

Similar to Encryptor-QKD handshake



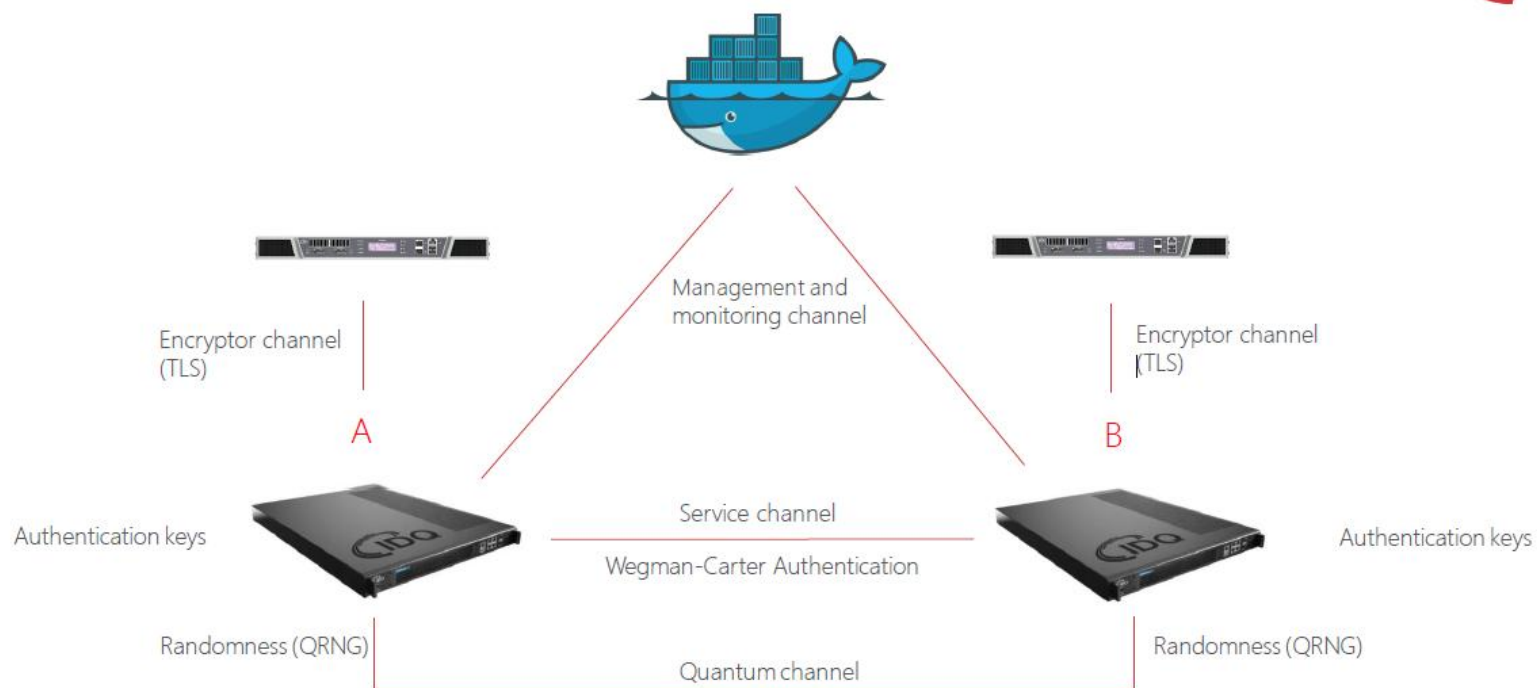
Cryptography with Certificates -examples



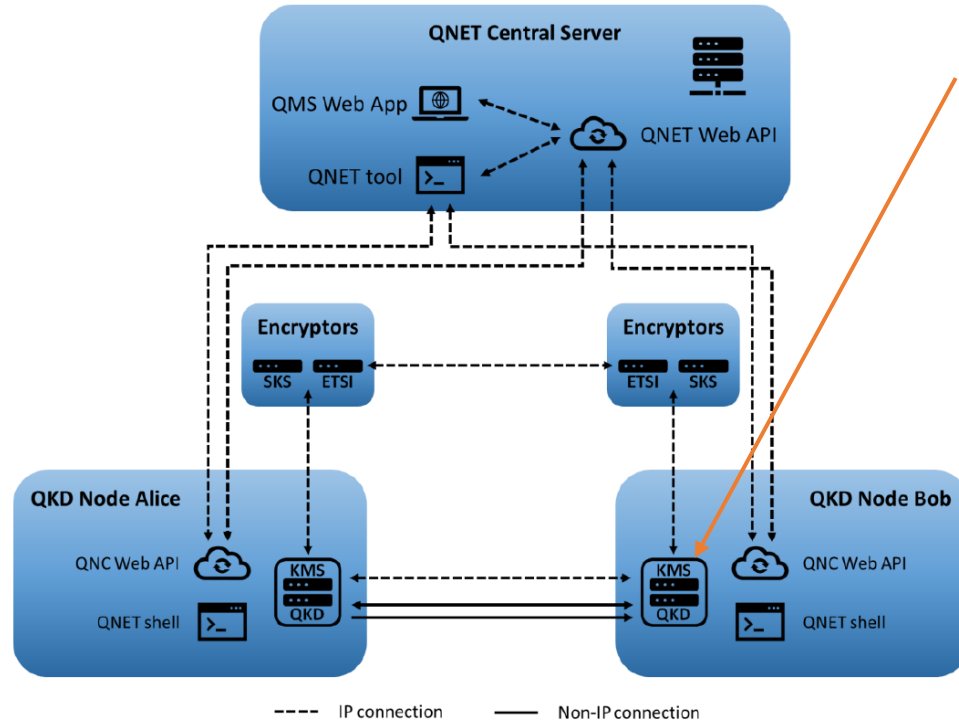
Mutual SSL authentication / Certificate based mutual authentication

1. A client requests access to a protected resource in the server.
2. The server presents its certificate to the client.
3. The client here verifies the server's certificate.
4. If successful, the client sends its certificate to the server.
5. The server verifies the client's credentials.
6. If successful, the server gives access to the protected resource in the server.

Needs at the QKD level



Main Concept of a QKD network architecture



KMS plays two roles as Consumer and Provider to Provider and Consumer respectively. That is, KMS is seen as Consumer when collecting keys from Provider and as Provider when dispensing keys to Consumer.

IDQ QKD solution for a point-to-point deployment

ETSI GS QKD 014 V1.1.1 (2019-02)

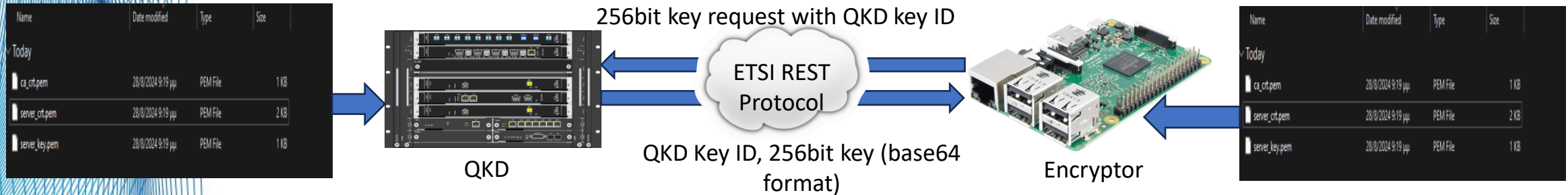


Quantum Key Distribution (QKD);
Protocol and data format of REST-based key delivery API

https://www.etsi.org/deliver/etsi_gs/QKD/001_099/014/01.01.01_60/gs_qkd014v010101p.pdf

Access a QKD device

We use QKD technology to provide keys to encryptors (ETSI 014) to facilitate network communication.

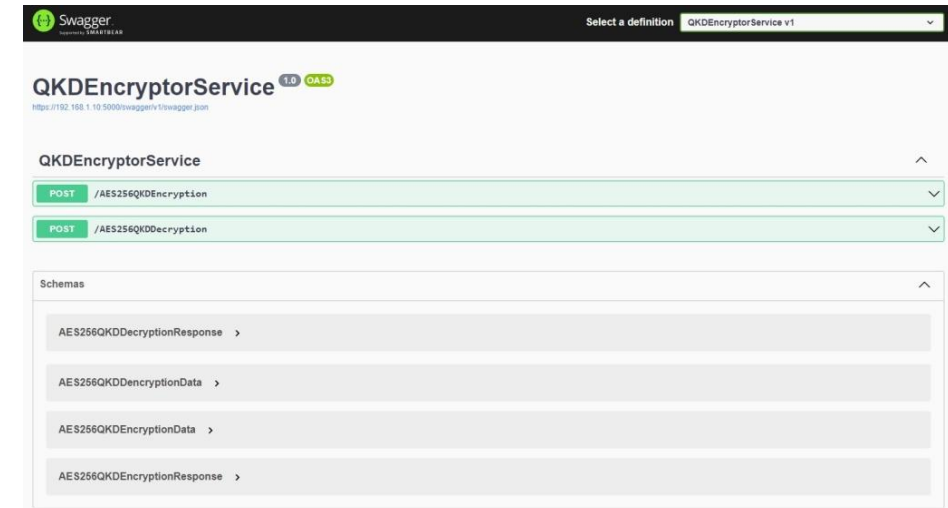


REQUEST A UNIQUE 256bit key from the QKD KMS

- Encryptor sends an authorized (via SSL Certificate) https post request (in JSON format) to the QKD's ETSI REST Interface requesting a 256bit key.
 - `curl -sS --cert server.crt.pem --key server.key.pem --cacert ca.crt.pem -k https://[Alice<MEIP]/api/v1/keys/[Bob SAE]/enc_keys?size=256`
 - `curl -sS --cert server.crt.pem --key server.key.pem --cacert ca.crt.pem -khttps://192.168.0.4/api/v1/keys/192.168.0.1/enc_keys?size=256`
- QKD accepts the authorized request from the Encryptor and returns a 256bit key (base64 format) together with the corresponding QKD Key ID (in JSON format).
- Encryptor can use the received QKD 256bit key for synchronous encryption.
 - 1.A client eg encryptor HSM, Raspberry pi etc requests access to a protected resource eg QKD keys.
 - 2.The server here the QKD KMS server presents its certificate to the client - encryptor.
 - 3.The client encryptor here verifies the server's certificate QKD KMS system.
 - 4.If successful, the client –encryptor sends its certificate to the server -QKD.
 - 5.The server-QKD verifies the client's-encryptor credentials.
 - 6.If successful, the server-QKD grants access to the protected resource -Keys requested by the client-encryptor.

Encryptor / Decryptor

- QKD Encryptor provides two JSON REST web services for encryption and decryption.
 - ✓ AES256QKDEncryption
 - ✓ AES256QKDDecryption
- Only authorized users can request data encryption/decryption.
- The data send to the QKD Encryptor has a specific JSON format (described in the Schemas)
- QKD Encryptor communicates with the QKD device via the [ETSI Protocol](#)



```
#!/bin/sh
CURRENTDIR=/home/qkd/opt/ldq/config/$0
RMSM_IP=192.168.20.21443
RMSM_IP=192.168.20.31443
while true
do
date
echo "Get key round"
echo "  Get New Key from master"
NewKey=$(curl -sS --cert ETSIA.pem --key ETSIA-key.pem --cacert ChrisCA.pem -k https://$RMSM_IP/api/v1/keys/ETSIA/enc_keys)
echo $NewKey | jq .keys[0].key_ID | cut -d '"' -f 2
KeyID=$(echo $NewKey | jq .keys[0].key_ID | cut -d '"' -f 2)
echo "  Found KEYID ALICE: $KeyID"
echo "  Found NEWKEY ALICE: $NewKey"
echo "  Get Key with ID from slave"
Rep=$(curl -sS --cert ETSIB.pem --key ETSIB-key.pem --cacert ChrisCA.pem -X POST -H 'Content-Type:application/json' -d '{"key_ID": "'$KeyID'"}' -k https://$RMSM_IP/api/v1/keys/ETSIA/dec_keys)
echo $Rep
Key=$(echo $Rep | jq .keys[0].key | cut -d '"' -f 2)
KeyIDSlave=$(echo $Rep | jq .keys[0].key_ID | cut -d '"' -f 2)
echo "  Found BOB Key: $Key"
echo "  Found BOB ID: $KeyIDSlave"
echo "  Waiting 5 seconds for new request"
sleep 5
done
echo "DONE"
```


The provided QKD simulator webservice simulates the QKD ETSI 014 protocol requests that QKD devices accept. Two methods are provided:

- **enc_keys** (request new encryption keys from QKD device)
- **dec_keys** (request encryption keys using QKD key IDs)

The webservice is hosted in: <https://qkdsimulator.iwelli.com:3333> and requires a client certificate authentication (the certificate files are provided and they are active for 3 months. You need to communicate with us to get new certificate files to continue using the QKD simulator).

Request new QKD keys (GET Request):

[https://qkdsimulator.iwelli.com:3333/\[QKD 1\]/api/v1/keys/\[QKD 2\]/enc_keys?size=256&number=1](https://qkdsimulator.iwelli.com:3333/[QKD 1]/api/v1/keys/[QKD 2]/enc_keys?size=256&number=1)

Example Response:

```
{
  "keys": [
    {
      "key_ID": "bc490419-7d60-487f-adc1-4ddcc177c139",
      "key": "wHHVxRwDJs3/bXd38GHP3oe4svTuRpZS0yCC7x4Ly+s="
    }
  ]
}
```

Request QKD keys using the QKD IDs (POST Request)

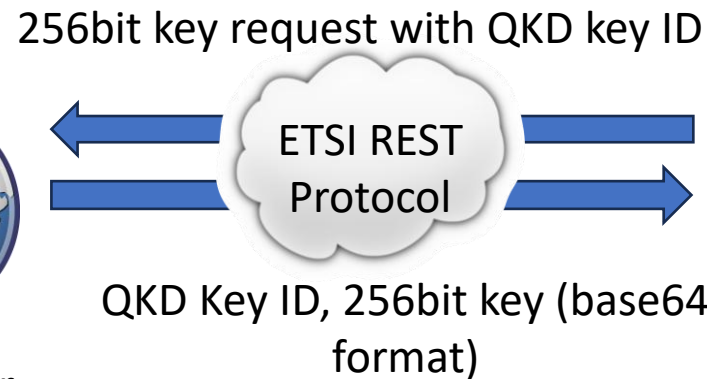
[https://qkdsimulator.iwelli.com:3333/\[QKD 2\]/api/v1/keys/\[QKD 1\]/dec_keys](https://qkdsimulator.iwelli.com:3333/[QKD 2]/api/v1/keys/[QKD 1]/dec_keys)

Request Body (JSON): {
 "key_IDs": [
 {
 "key_ID": [KeyID]
 }
]
}

QKD 1 and QKD 2 pair must be the same as when the user created the new keys. QKD 2 is the QKD device that the user requests the key using the key_ID. QKD 1 is the QKD device that the user creates the initial key QKD 1 key ID.

Example response: {
 "keys": [
 {
 "key_ID": "bc490419-7d60-487f-adc1-4ddcc177c139",
 "key": "wHHVxRwDJs3/bXd38GHP3oe4svTuRpZS0yCC7x4Ly+s="
 }
]
}

Simulate communication with a QKD device



QKDETSIClient
(Encryptor)



REQUEST A UNIQUE 256bit key from the QKD ETSI Simulator webservice

- The QKDETSIClient software sends an authorized (via SSL Certificate) https post request (in JSON format) to the QKD ETSI Simulator Webservice requesting a 256bit key.
- QKD ETSI Simulator Webservice accepts the authorized request from the QKDETSIClient software and returns a 256bit key (base64 format) together with a the corresponding QKD Key ID (in JSON format).
- QKDETSIClient software can use the received QKD 256bit key for synchronous encryption.

- Install an FTP Client (e.g. Filezilla) to access the required demonstration files.
- Access the FTP Server using the below information

FTP Details

Host: 143.233.247.77

Port: 21

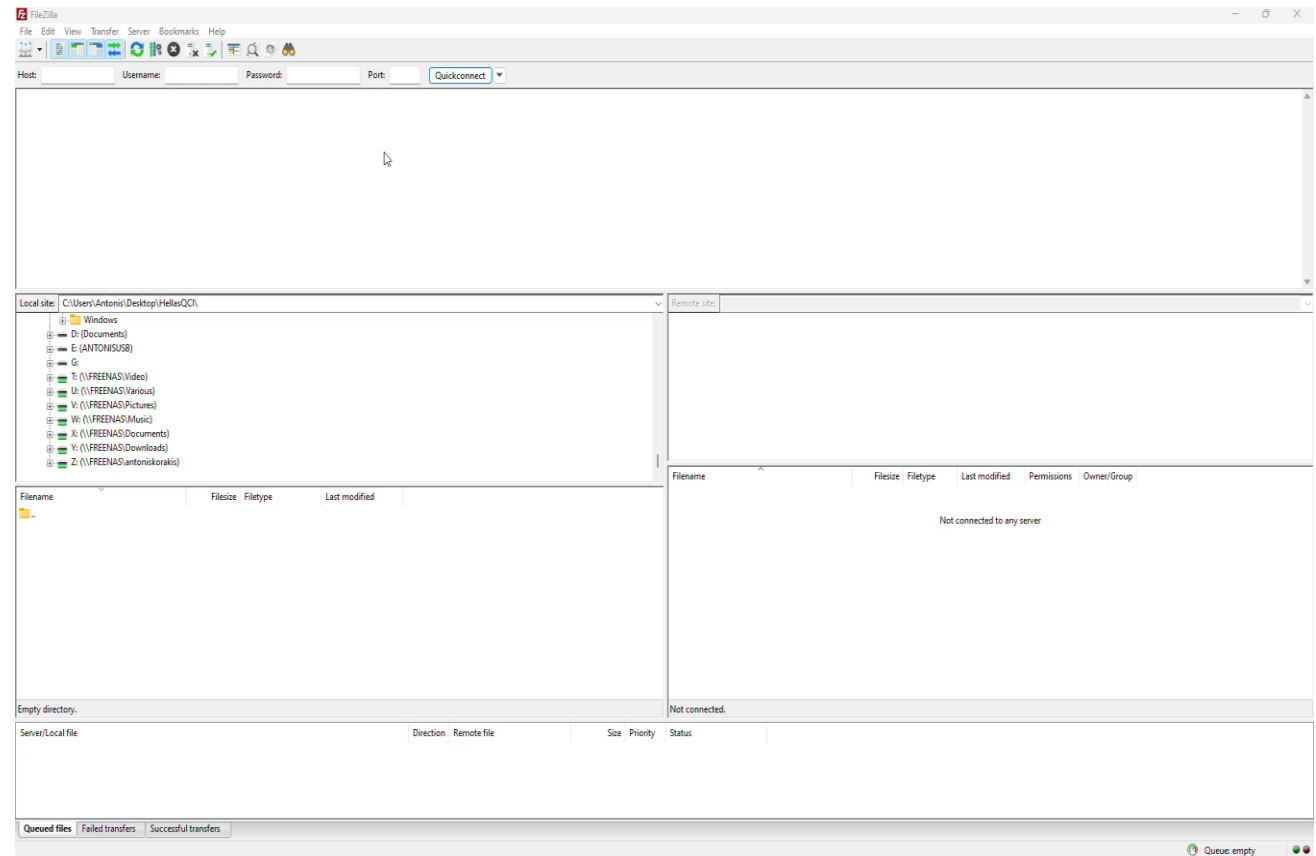
Username: hellasqci

Password: HellasQCI2024Demo

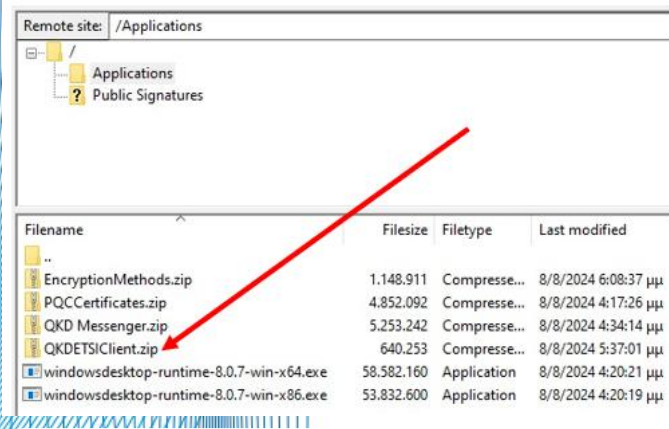
Directory: /Application

HTTP Details

<http://143.233.247.77/hellasqcidemo.zip>

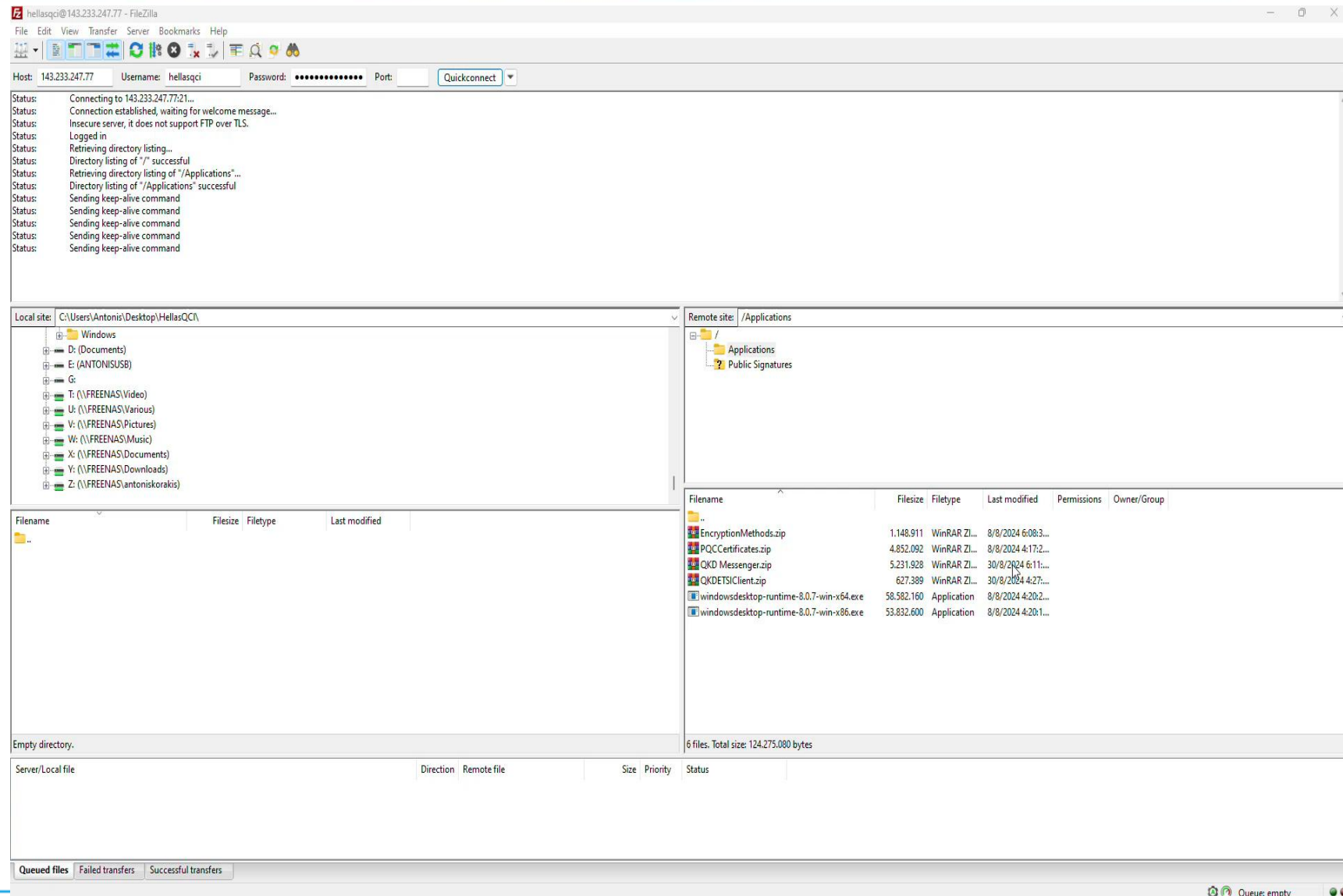


Download the App: QKDETSIClient



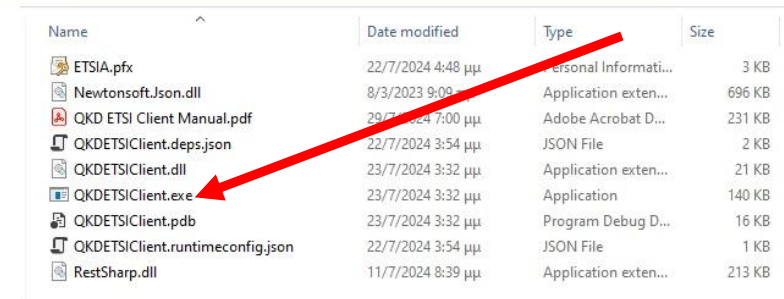
Download the required files

- Download QKDETSIClient.zip
- Download Windowsdesktop-runtime-x64.exe (in case you have a 32bit system install Windowsdesktop-runtime-x86.exe)
- Install Windowsdesktop-runtime-x64.exe (in case you have a 32bit system install Windowsdesktop-runtime-x86.exe)
- Unzip QKDETSIClient.zip to your device

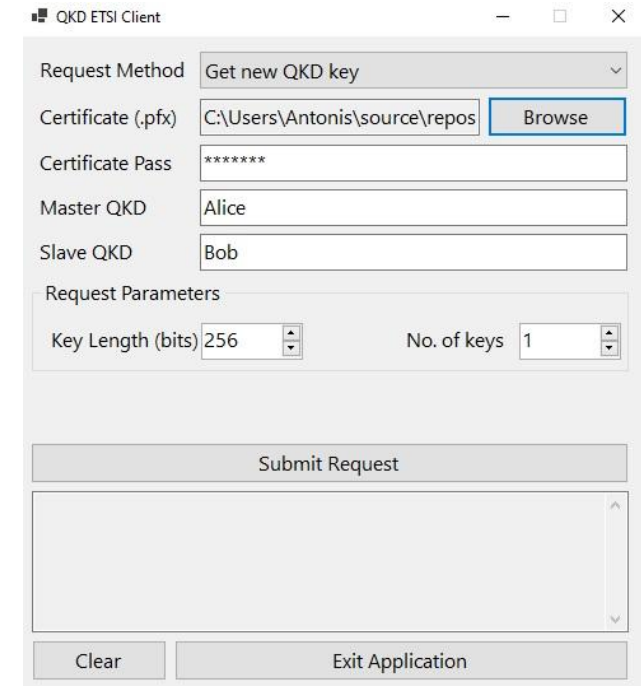
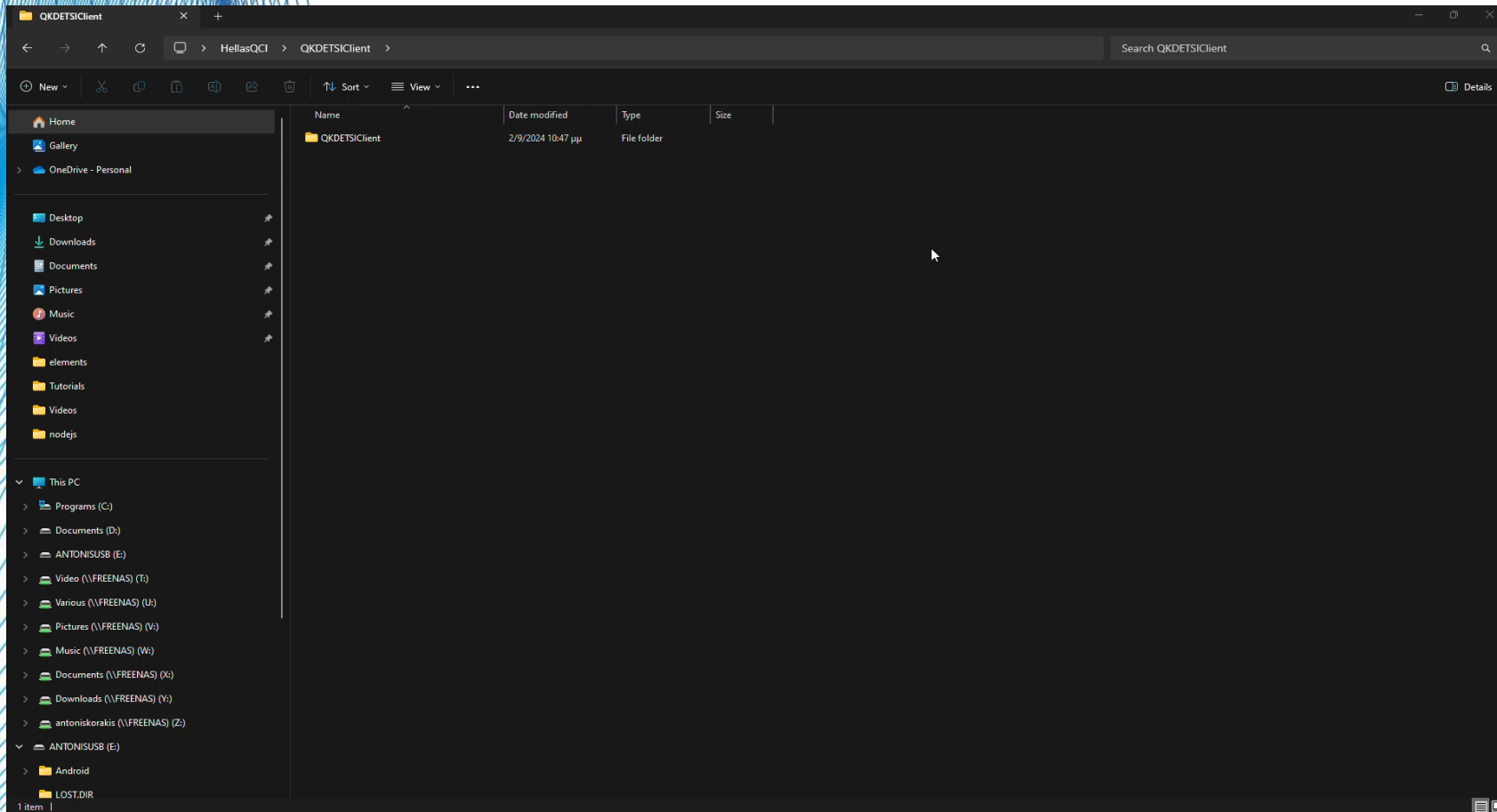


Install the App: QKD ETSI Simulator

- Open the folder QKDETSIClient
- Execute QKDETSIClient.exe



Name	Date modified	Type	Size
ETSIA.pfx	22/7/2024 4:48 μμ	Personal Informati...	3 KB
Newtonsoft.Json.dll	8/3/2023 9:09 μμ	Application exten...	696 KB
QKD ETSI Client Manual.pdf	29/7/2024 7:00 μμ	Adobe Acrobat D...	231 KB
QKDETSIClient.deps.json	22/7/2024 3:54 μμ	JSON File	2 KB
QKDETSIClient.dll	23/7/2024 3:32 μμ	Application exten...	21 KB
QKDETSIClient.exe	23/7/2024 3:32 μμ	Application	140 KB
QKDETSIClient.pdb	23/7/2024 3:32 μμ	Program Debug D...	16 KB
QKDETSIClient.runtimeconfig.json	22/7/2024 3:54 μμ	JSON File	1 KB
RestSharp.dll	11/7/2024 8:39 μμ	Application exten...	213 KB

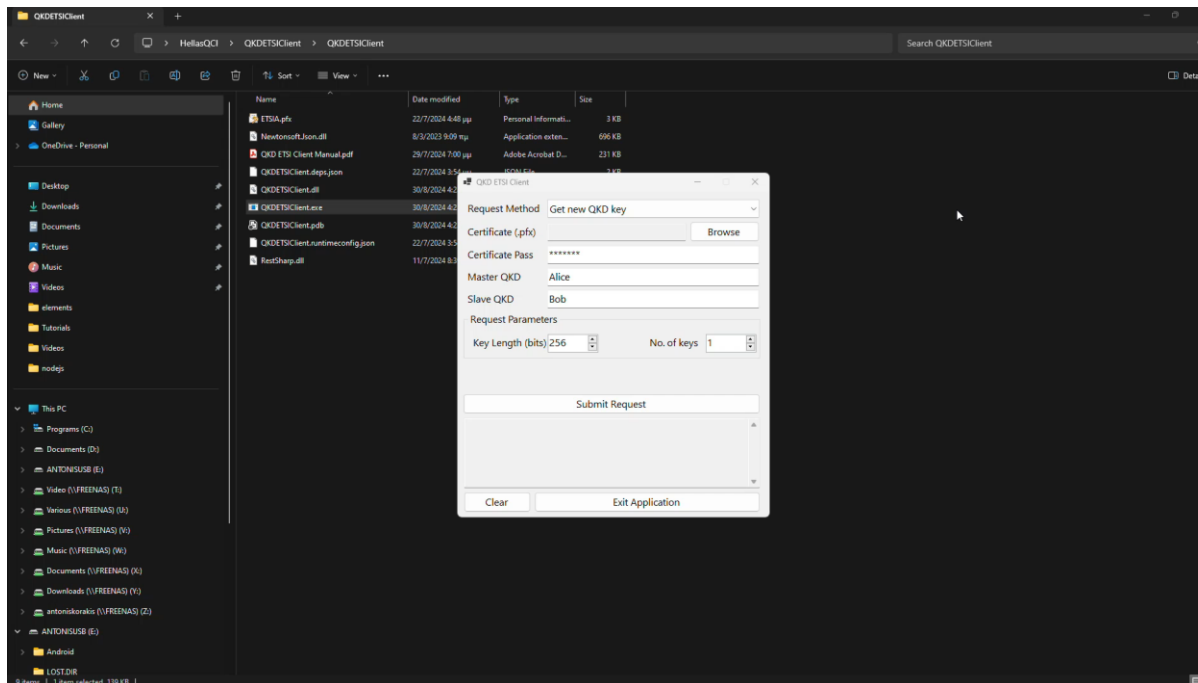


The screenshot shows the 'QKD ETSI Client' application window. It has a light gray background and a title bar with standard Windows window controls. The window contains several input fields and buttons:

- Request Method:** A dropdown menu set to 'Get new QKD key'.
- Certificate (.pfx):** A text field containing 'C:\Users\Antonis\source\repos' and a 'Browse' button.
- Certificate Pass:** A text field containing '*****'.
- Master QKD:** A text field containing 'Alice'.
- Slave QKD:** A text field containing 'Bob'.
- Request Parameters:** A section with two controls: 'Key Length (bits)' set to '256' and 'No. of keys' set to '1'.
- Submit Request:** A large, light gray button.
- Clear:** A small button.
- Exit Application:** A small button.

Get new encryption keys from the QKD

- Select the Get new QKD key option from the dropdown menu
- Select Certificate: Click the browse button to select the appropriate certificate (.pfx file). The .pfx file is located in the application's folder (ETSIA.pfx)
- Enter QKD Details: **Master QKD Name:** Enter the name of the Master QKD (e.g., Alice). **Slave QKD Name:** Enter the name of the Slave QKD (e.g., Bob).
- **Key Length:** Enter the desired key length (maximum 1024 bits).
- **Number of Keys:** Enter the number of keys you want to request (up to 10).
- Submit Request: Press the **Submit Request** button.
- The application will communicate with the QKD simulator and retrieve the requested keys in base64 format.



QKD ETSI Client

Request Method

Get new QKD key

Certificate (.pfx)

C:\Users\Antonis\source\repos

Browse

Certificate Pass

Master QKD

Alice

Slave QKD

Bob

Request Parameters

Key Length (bits)

256

No. of keys

2

Submit Request

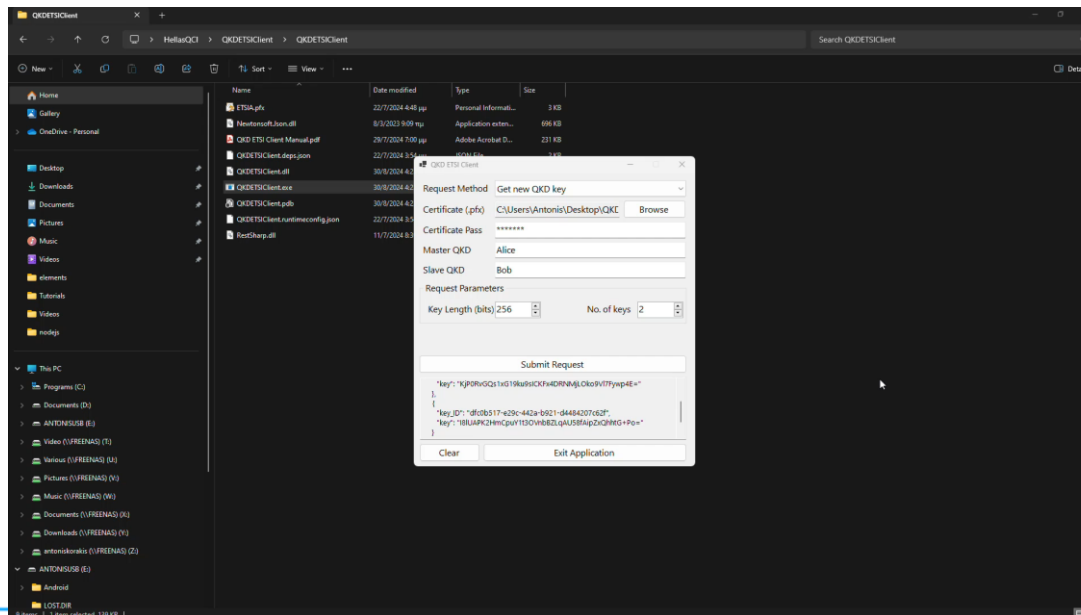
```
{
  "keys": [
    {
      "key_ID": "4412f284-b828-4039-a452-3d3f81b85c6d",
      "key": "Bb6u4tiuOFikX7Q2250U4zgnY1H0utvjal3QRZX+3k="
    }
  ],
}
```

Clear

Exit Application

Requesting Key by ID from Slave QKD

- Select the Get QKD key by ID option from the dropdown menu
- Select Certificate: Click the browse button to select the appropriate certificate (.pfx file). The .pfx file is located in the application's folder (ETSIA.pfx)
- Enter QKD Details: **Master QKD Name:** Enter the name of the QKD that will request the key (e.g., Bob). **Slave QKD Name:** Enter the name of the QKD that originally created the key (e.g., Alice).
- Enter Key ID(s): Enter the QKD key ID(s) of the requested keys.
- Submit Request: Press the **Submit Request** button.
- The application will communicate with the QKD simulator and retrieve the requested keys in base64 format.



QKD ETSI Client

Request Method: Get QKD key by ID

Certificate (.pfx): C:\Users\Antonis\source\repos Browse

Certificate Pass: *****

Master QKD: Bob

Slave QKD: Alice

Request Parameters

QKD Key IDs (comma separated)

a2b77a26-c796-487d-bc5d-1b2d8a82b063,765f75e0-0b5c-4cdf-a736-7ed09904214c

Submit Request

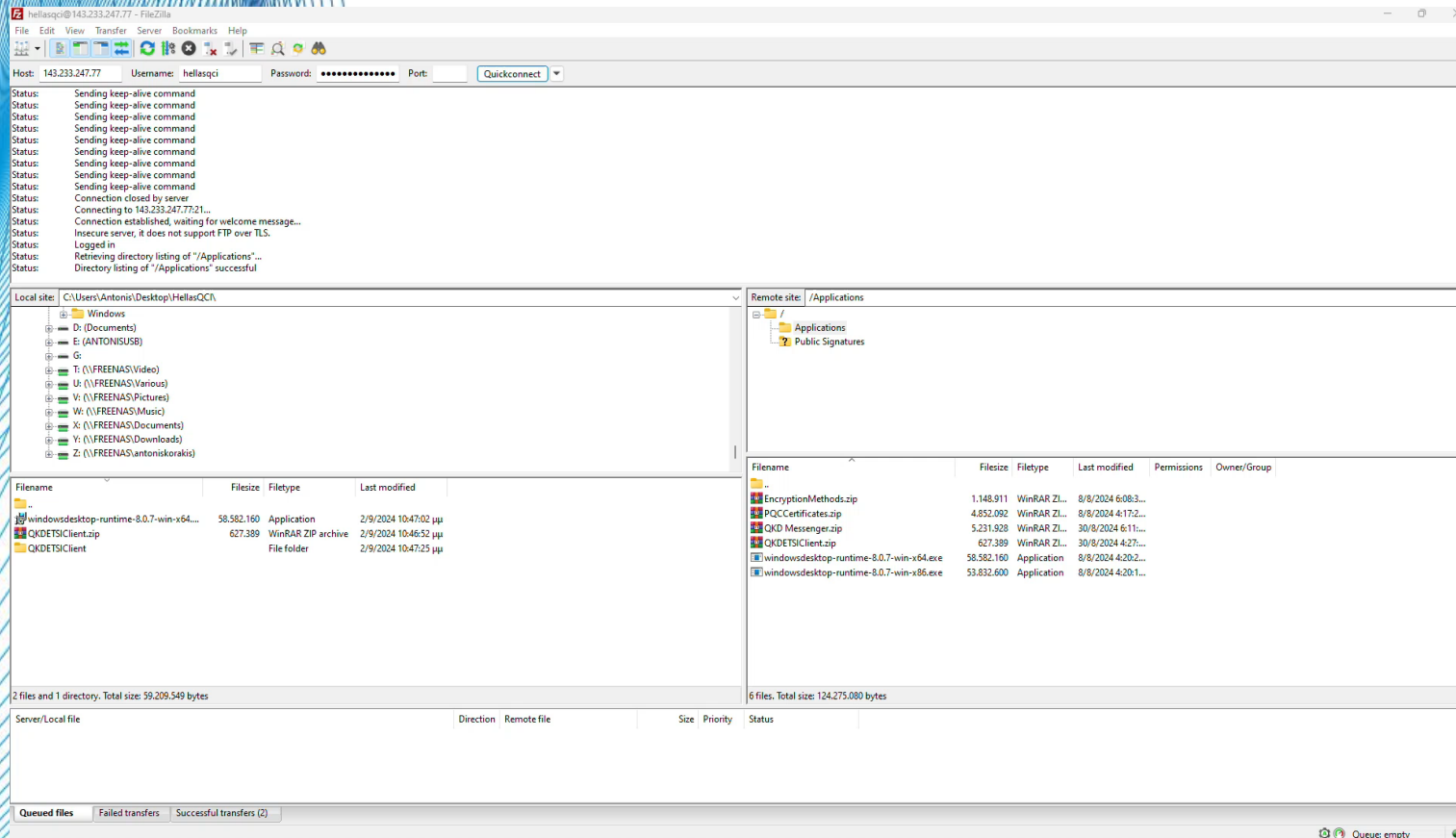
```
{
  "keys": [
    {
      "key_ID": "a2b77a26-c796-487d-bc5d-1b2d8a82b063",
      "key": "BB6DKvEQkCk8j/9gV8nKgWQsnzQ3ZJbQ3YgN5wYe8="
    }
  ]
}
```

Clear Exit Application

Simulator for Encryption Methods

Download the required files

- Download EncryptionMethods.zip
- Unzip EncryptionMethods.zip to your device
- Execute EncryptionMethods.exe



source > repos > EncryptionMeyhods > EncryptionMeyhods > bin > Debug > net8.0-windows > Encr

Name	Date modified	Type	Size
BouncyCastle.Crypto.dll	8/8/2024 8:05 μμ	Application exten...	2,548 KB
EncryptionMethods Manual.pdf	29/7/2024 6:25 μμ	Adobe Acrobat D...	256 KB
EncryptionMethods.deps.json	26/7/2024 6:54 μμ	JSON File	2 KB
EncryptionMethods.dll	26/7/2024 6:54 μμ	Application exten...	16 KB
EncryptionMethods.exe	26/7/2024 6:54 μμ	Application	140 KB
EncryptionMethods.pdb	26/7/2024 6:54 μμ	Program Debug D...	17 KB
EncryptionMethods.runtimeconfig.json	26/7/2024 6:54 μμ	JSON File	1 KB

Remote site: /Applications

Applications

Public Signatures

Filename	Filesize	Filetype	Last modified
EncryptionMethods.zip	1,148,911	Compressed...	8/8/2024 6:08:37 μμ
PQCCertificates.zip	4,852,092	Compressed...	8/8/2024 4:17:26 μμ
QKD Messenger.zip	5,253,242	Compressed...	8/8/2024 4:34:14 μμ
QKDETSIClient.zip	640,253	Compressed...	8/8/2024 5:37:01 μμ
windowsdesktop-runtime-8.0.7-win-x64.exe	58,582,160	Application	8/8/2024 4:20:21 μμ
windowsdesktop-runtime-8.0.7-win-x86.exe	53,832,600	Application	8/8/2024 4:20:19 μμ



Use the QKD keys to encrypt data

- Entering Keys and Message: Enter the two 256-bit keys in base64 format (QKD Key 1 and QKD Key 2).
- Type your message into the plaintext textarea.
- Encrypting the Message: After entering the keys and message, press the **Encrypt** button.

The application will encrypt the message using the following four algorithms: AES 256 CBC, AES 256 GCM, One Time Pad, ChaCha20-Poly1305

Viewing Encrypted Message: The encrypted message will be displayed in the designated area for each encryption algorithm.

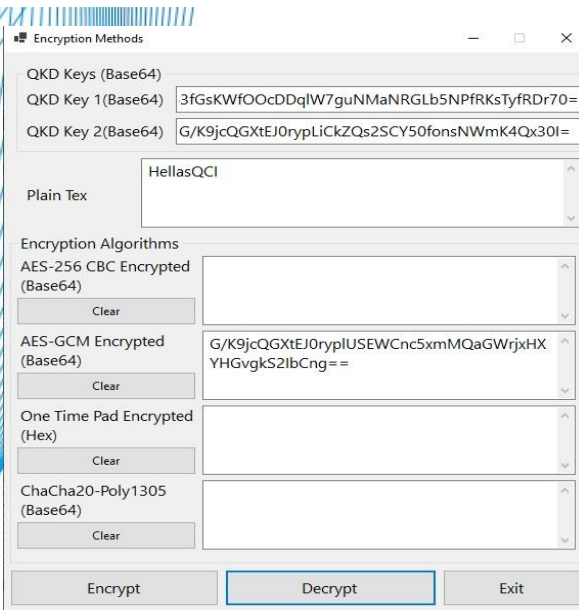
The screenshot shows the 'Encryption Methods' application window. It has a sidebar with file explorer icons. The main area contains fields for 'QKD Keys (Base64)', 'QKD Key 1 (Base64)', 'QKD Key 2 (Base64)', and 'Plain Tex'. Below these are four encryption algorithms: 'AES-256 CBC Encrypted (Base64)', 'AES-GCM Encrypted (Base64)', 'One Time Pad Encrypted (Hex)', and 'ChaCha20-Poly1305 (Base64)'. Each algorithm has a corresponding output field and a 'Clear' button. At the bottom are 'Encrypt', 'Decrypt', and 'Exit' buttons.

This screenshot shows the 'Encryption Methods' application window with a 'QKD ETSI Client' window open on top. The 'Encryption Methods' window is the same as in the previous screenshot. The 'QKD ETSI Client' window has a 'Request Method' dropdown set to 'Get new QKD key', a 'Certificate (.pfx)' field with a 'Browse' button, a 'Certificate Pass' field with asterisks, 'Master QKD' set to 'Alice', and 'Slave QKD' set to 'Bob'. Under 'Request Parameters', 'Key Length (bits)' is 256 and 'No. of keys' is 2. There is a 'Submit Request' button and a text area showing a JSON response with two keys. At the bottom are 'Clear' and 'Exit Application' buttons.

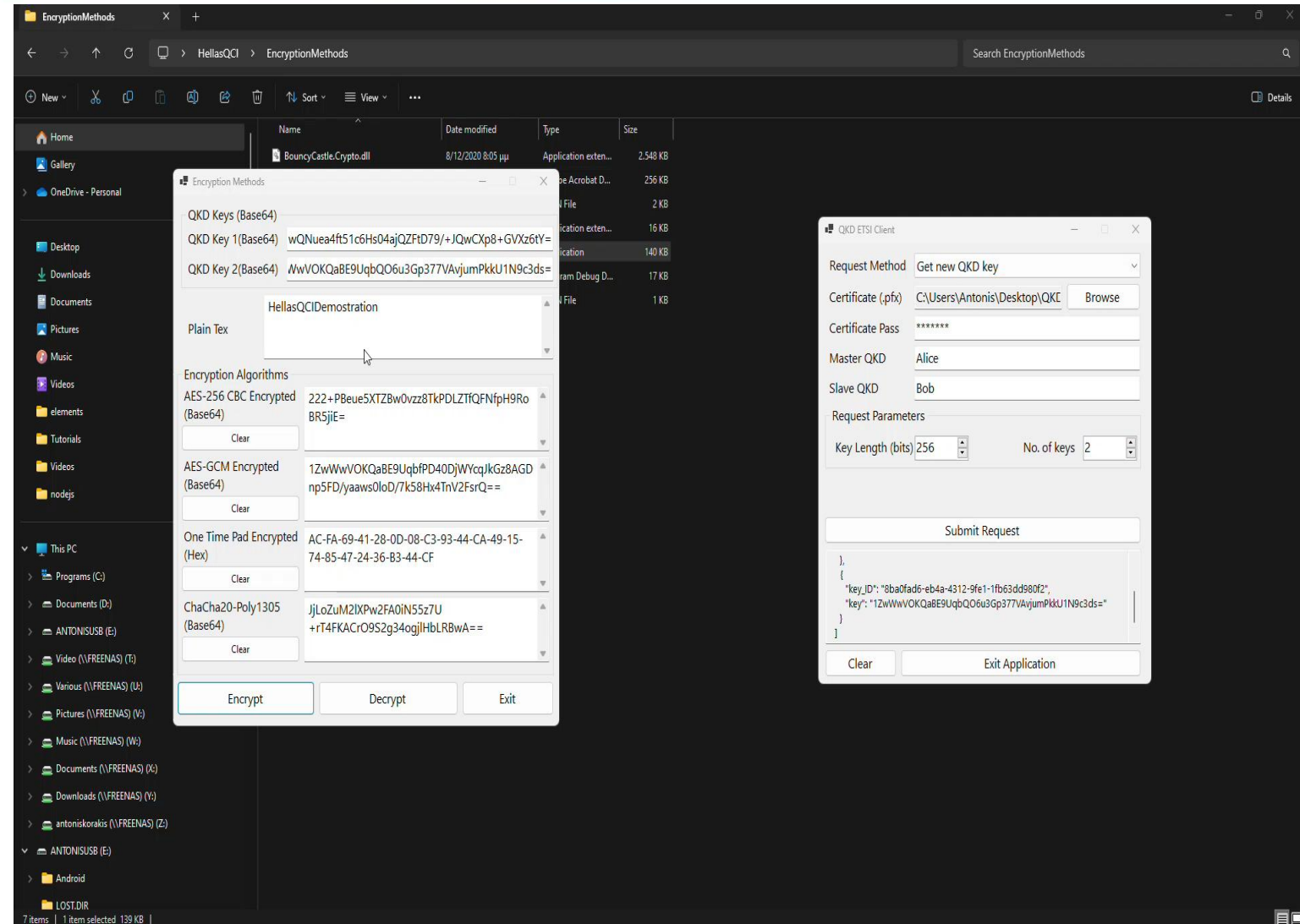
Use the QKD keys to decrypt data

- **Entering Encrypted Message and Keys:**
Enter the encrypted message and the corresponding encryption keys, into the respective fields.
- **Decrypting the Message:**
Press the **Decrypt** button.

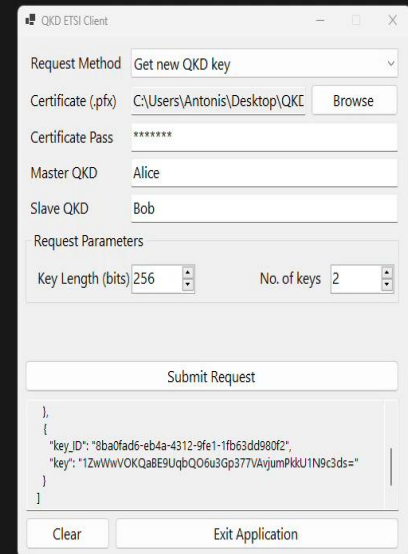
The application will decrypt the message and display the plaintext.



The screenshot shows the 'Encryption Methods' application window. It has a sidebar with navigation options like Home, Gallery, OneDrive, Desktop, Downloads, Documents, Pictures, Music, Videos, elements, Tutorials, and nodejs. The main area contains a 'Plain Text' field with 'HellasQCI' entered. Below it are four encryption algorithm sections: AES-256 CBC Encrypted (Base64), AES-GCM Encrypted (Base64), One Time Pad Encrypted (Hex), and ChaCha20-Poly1305 (Base64). Each section has a 'Clear' button and a text area for the encrypted message. At the bottom, there are three buttons: 'Encrypt', 'Decrypt' (highlighted), and 'Exit'.



This screenshot shows the 'Encryption Methods' application window with a file explorer overlay. The file explorer is displaying the contents of the 'EncryptionMethods' folder, which includes files like 'BouncyCastle.Crypto.dll', 'Acrobat D...', 'Application exten...', 'Application', 'ram Debug D...', and 'File'. The application window is partially obscured by the file explorer. The 'Encryption Methods' window shows the 'Plain Text' field with 'HellasQCIDemostration' entered. The 'Encryption Algorithms' section is expanded, showing details for 'AES-256 CBC Encrypted (Base64)', 'AES-GCM Encrypted (Base64)', 'One Time Pad Encrypted (Hex)', and 'ChaCha20-Poly1305 (Base64)'. Each algorithm has a 'Clear' button and a text area for the encrypted message. At the bottom, there are three buttons: 'Encrypt', 'Decrypt', and 'Exit'.



The screenshot shows the 'QKD ETSI Client' application window. It has a 'Request Method' dropdown set to 'Get new QKD key'. Below it are fields for 'Certificate (.pfx)' (with a 'Browse' button), 'Certificate Pass' (with a masked input), 'Master QKD' (set to 'Alice'), and 'Slave QKD' (set to 'Bob'). There are also 'Request Parameters' for 'Key Length (bits)' (set to 256) and 'No. of keys' (set to 2). At the bottom, there is a 'Submit Request' button and a 'Clear' button. The main area displays a JSON response:

```
{
  "key_ID": "8ba0fad6-eb4a-4312-9fe1-1fb63dd980f2",
  "key": "1ZwWwVOKQaBE9UqbQO6u3Gp377VAvjumPkkU1N9c3ds="
}
```

Thank you

Dr. Homer Papadopoulos, NCSRD

Antonis Korakis, NCSRD