**Task 1: RSA key generation**

**Scenario:**  
As a cybersecurity engineer at DataSafe Bank, you are tasked with securing communication channels between the bank's internal systems and its customers. To achieve this, you must generate RSA keys that will be used for encrypting and decrypting sensitive information transmitted over insecure channels.

**Task:**  
Generate a 2048-bit RSA private key and its corresponding public key. Save the private key as private\_key.pem and the public key as public\_key.pem.

**Step 1: Generate an RSA private key**

**Hint:**  
To generate the RSA private key:  
• Use the OpenSSL genpkey command.  
• Specify the algorithm as RSA and set the key length to 2048 bits.

**Command:**

openssl genpkey -algorithm RSA -out private\_key.pem -pkeyopt rsa\_keygen\_bits:2048

**Expected output:**  
• private\_key.pem file containing the RSA private key.

**Action required:**  
Capture a screenshot of the command you used to generate the terminal's private key and confirmation output.



**Step 2: Extract the RSA public key from the private key**

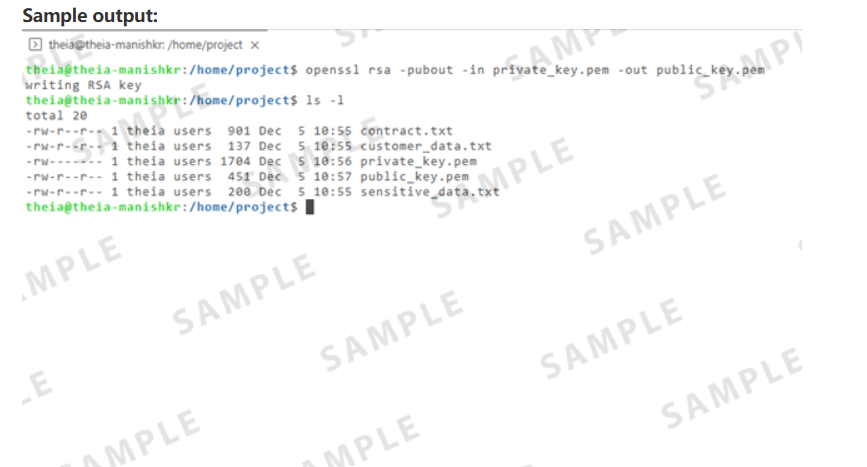
**Hint:**  
To extract the RSA public key from the private key:  
• Use the OpenSSL rsa command.  
• Use the -pubout option to specify that the output should be the public key.

**Command:**

openssl rsa -pubout -in private\_key.pem -out public\_key.pem

**Expected output:**  
• public\_key.pem file containing the RSA public key.

**Action required:**  
Capture a screenshot of the command you used to extract the public key and the confirmation output in the terminal.



**Task 2: RSA key encryption and decryption**

**Scenario:**  
Once the RSA keys have been generated, you must encrypt sensitive customer data (such as account numbers) to ensure secure transmission. The customer data should be encrypted with the public key, and the recipient (bank server) should decrypt it using the private key.

**Task:**  
Using the RSA keys generated in Task 1, encrypt a sensitive customer data file (customer\_data.txt) and decrypt it back to its original form.

**Step 1: Encrypt customer data using the public key**

**Hint:**  
To encrypt the customer data:  
• Use the OpenSSL pkeyutl command with the -encrypt option.  
• Specify the input file as customer\_data.txt and the output file as encrypted\_data.bin.  
• Use the -pubin option to indicate that the key is a public key.

openssl pkeyutl -encrypt -inkey public\_key.pem -pubin -in customer\_data.txt -out encrypted\_data.bin

**Expected output:**  
• encrypted\_data.bin file containing the encrypted data.

**Action required:**  
Capture a screenshot of the command you used to encrypt the customer data and the confirmation output in the terminal.

**Step 2: Decrypt the data using the private key**

**Hint:**  
To decrypt the encrypted data:  
• Use the OpenSSL pkeyutl command with the -decrypt option.  
• Specify the input file as encrypted\_data.bin and the output file as decrypted\_data.txt.

openssl pkeyutl -decrypt -inkey private\_key.pem -in encrypted\_data.bin -out decrypted\_data.txt

**Expected output:**  
• decrypted\_data.txt file containing the decrypted data should match the original customer\_data.txt.

**Action required:**  
Capture a screenshot of the command you used to decrypt the customer data and the confirmation output in the terminal.  
   
**Sample output:**

**Task 3: AES encryption**

**Scenario:**  
DataSafe Bank must securely store sensitive data (for example, transaction logs, user passwords) in its database. To ensure confidentiality, this data is encrypted using AES encryption before storage. The encrypted data can only be decrypted using the correct AES key.

**Task:**  
Encrypt a sensitive data file (sensitive\_data.txt) using AES-256 encryption and decrypt it back using the AES key.

**Step 1: Encrypt data using AES-256-CBC**

**Hint:**  
To encrypt the data using AES-256 encryption:  
• Use the OpenSSL enc command with the -aes-256-cbc option to specify the AES-256-CBC algorithm.  
• Use the -salt option to add random data (salt) to make encryption more secure.  
• Provide the input file (sensitive\_data.txt) and specify the output file (encrypted\_data\_aes.bin).  
• Use a password for encryption with the -pass pass:securepassword option.

**Command:**

openssl enc -aes-256-cbc -salt -in sensitive\_data.txt -out encrypted\_data\_aes.bin -pass pass:securepassword

**Expected output:**  
• encrypted\_data\_aes.bin file containing the AES-encrypted data.

**Action required:**  
Capture a screenshot of the command you used to encrypt the data and the confirmation output in the terminal.

**Sample output:**

**Step 2: Decrypt data using the AES key**

**Hint:**  
To decrypt the data using the AES key:  
• Use the OpenSSL enc command with the -d option for decryption.  
• Specify the encrypted input file (encrypted\_data\_aes.bin) and the output file for the decrypted data (decrypted\_data\_aes.txt).  
• Use the same password used for encryption to decrypt the data.

**Command:**

openssl enc -d -aes-256-cbc -in encrypted\_data\_aes.bin -out decrypted\_data\_aes.txt -pass pass:securepassword

**Expected output:**  
• decrypted\_data\_aes.txt file containing the decrypted data should match the original sensitive\_data.txt.

**Action required:**  
Capture a screenshot of the command you used to decrypt the data and the confirmation output in the terminal.

**Sample output:**

**Task 4: Digital signature creation**

**Scenario:**  
DataSafe Bank needs to ensure the integrity and authenticity of documents (for example, contracts and agreements) transmitted over email. You will create a digital signature using RSA, allowing the recipient to verify the document's integrity and origin.

**Task:**  
Create a digital signature for a document (contract.txt) using your RSA private key. Verify the signature using the corresponding public key.

**Step 1: Generate a digital signature for the document**

**Hint:**  
To generate a digital signature:  
• Use the OpenSSL dgst command with the -sha256 option to specify the SHA-256 hashing algorithm.  
• Create the digital signature with the -sign option with the private key.  
• Specify the document (contract.txt) and the output file for the signature (document\_signature.bin).

**Command:**

openssl dgst -sha256 -sign private\_key.pem -out document\_signature.bin contract.txt

**Expected output:**  
• document\_signature.bin file containing the digital signature.

**Action required:**  
Capture a screenshot of the command you used to generate the digital signature and the confirmation output in the terminal.

**Sample output:**

**Step 2: Verify the digital signature**

**Hint:**  
To verify the digital signature:  
• Use the OpenSSL dgst command with the -sha256 option for the SHA-256 algorithm.  
• Use the -verify option with the public key to verify the signature.  
• Specify the digital signature file (document\_signature.bin) and the original document (contract.txt).

**Command:**

openssl dgst -sha256 -verify public\_key.pem -signature document\_signature.bin contract.txt

**Expected output:**  
• Verification result: "Verified OK" or "Verification Failure".

**Action required:**  
Capture a screenshot of the command you used to verify the terminal's digital signature and confirmation output.

**Sample output:**

**Task 5: Diffie-Hellman key exchange**

Scenario:  
DataSafe Bank must establish a secure, shared secret between two parties (an employee and a bank server) without transmitting the secret over the network. To achieve this, the Diffie-Hellman key exchange protocol will be used.

Task:  
Implement Diffie-Hellman key exchange to securely exchange keys between two parties (Alice and Bob)

**Step 1: Generate DH parameters and generate keys**

**Hint:**  
To generate the Diffie-Hellman parameters:  
• Use the OpenSSL dhparam command.  
• Specify the key length (2048 bits) to generate secure parameters.  
• Generate public and private keys for Alice and Bob

**Commands:**

* **Generate DH parameters:**

openssl dhparam -out dhparam.pem 2048

* **Generate Alice's private key:**

openssl genpkey -paramfile dhparam.pem -out alice\_private.pem

* **Generate Alice's public key:**

openssl pkey -in alice\_private.pem -pubout -out alice\_public.pem

* **Generate Bob's private key:**

openssl genpkey -paramfile dhparam.pem -out bob\_private.pem

* **Generate Bob's public key:**

openssl pkey -in bob\_private.pem -pubout -out bob\_public.pem

**Expected outputs:**  
• dhparam.pem file containing the DH parameters.  
• Command and outputs for the public and private keys for both parties

**Action required:**  
Capture a screenshot of the command you used to generate the DH parameters and the confirmtion of output, along with the keys for both parties.

**Sample output:**

**Step 2: Compute and verify the shared secret keys**

**Hint:**  
To generate the private and public keys for Alice and Bob, and then compute the shared secret:  
• Alice and Bob exchange public keys, and each computes the shared secret using their private and public keys.

**Commands:**  
**Compute the shared secret for Alice:**

openssl pkeyutl -derive -inkey alice\_private.pem -peerkey bob\_public.pem -out alice\_shared\_secret.bin

**Compute the shared secret key for Bob:**

openssl pkeyutl -derive -inkey bob\_private.pem -peerkey alice\_public.pem -out bob\_shared\_secret.bin

**Expected output:**  
• alice\_shared\_secret.bin and bob\_shared\_secret.bin files containing the shared secret (both should match).

**Action required:**  
Capture a screenshot that displays the shared secret keys. The screenshot should include the successful output showing that both shared secret keys match.

**Sample output:**

**Task 6: Public Key Infrastructure (PKI) setup**

**Scenario:**  
DataSafe Bank requires the setup of a Certificate Authority (CA) to sign and issue certificates for internal services and servers. You are tasked with creating a root certificate and signing a server certificate for datasafebank.com.

**Task:**  
Set up a Certificate Authority (CA) using OpenSSL. Generate a root certificate and a signed certificate for datasafebank.com.

**Step 1: Generate the root private key and create the root certificate**

**Hint:**  
To generate the root private key:  
• Use the OpenSSL genrsa command.  
• Specify the key length (2048 bits) for security.  
To create the root certificate:  
• Use the OpenSSL req command with the -x509 option to generate a self-signed certificate.  
• Specify the root private key (ca\_private\_key.pem) and set the certificate validity (3650 days).

**Command:**

* To generate the root private key:

openssl genrsa -out ca\_private\_key.pem 2048

* To create root certificate:

openssl req -x509 -new -key ca\_private\_key.pem -out ca\_cert.pem -days 3650

**Expected output:**  
• ca\_private\_key.pem file containing the root private key.  
• ca\_cert.pem file containing the root certificate.

**Action required:**  
Capture a screenshot of the command you used to generate the root private key and the confirmation output in the terminal, along with the command you used to create the terminal's root certificate and confirmation output.

**Step 2: Generate a CSR for the server**

**Hint:**  
To generate the certificate signing request (CSR) for the server:  
• Use the OpenSSL req command with the -new option.  
• Specify the private key for datasafebank.com and the output CSR file (datasafebank\_csr.csr).

**Command:**

openssl req -new -key private\_key.pem -out datasafebank\_csr.csr

**Expected output:**  
• datasafebank\_csr.csr file containing the server's CSR.

**Action required:**  
Capture a screenshot of the command you used to generate the CSR and the confirmation output in the terminal.

**Step 3: Sign the CSR with the root certificate**

**Hint:**  
To sign the CSR using the root certificate:  
• Use the OpenSSL x509 command with the -req option to sign the CSR.  
• Specify the root certificate (ca\_cert.pem) and the root private key (ca\_private\_key.pem).  
• Set the server certificate's validity period (for example , 365 days).

**Command:**

openssl x509 -req -in datasafebank\_csr.csr -CA ca\_cert.pem -CAkey ca\_private\_key.pem -CAcreateserial -out datasafebank\_cert.pem -days 365

**Expected output:**  
• datasafebank\_cert.pem file containing the signed server certificate.

**Action required:**  
Capture a screenshot of the command you used to sign the CSR and the confirmation output in the terminal.

**Task 7: Hashing with SHA-256**

**Scenario:**  
DataSafe Bank needs to store passwords securely. Instead of storing passwords in plain text, you will hash them using SHA-256 to ensure they are stored securely.

**Task:**  
Hash a password using SHA-256 and verify its integrity.

**Step 1: Hash the password using SHA-256**

**Hint:**  
To hash the password using SHA-256:  
• Use the OpenSSL dgst command with the -sha256 option to create a SHA-256 hash.  
• Specify the input password file (password.txt) and the output file for the hashed password (hashed\_password.txt).

**Command:**

openssl dgst -sha256 -out hashed\_password.txt password.txt

**Expected output:**  
• hashed\_password.txt file containing the SHA-256 hash of the password.

**Action required:**  
Capture a screenshot of the command you used to hash the password and the confirmation output in the terminal.

**Step 2: Verify the hash**

**Hint:**  
To verify the integrity of the hashed password:  
• Use the OpenSSL dgst command again to check if the hash matches the original file.  
• Ensure that the -verify option and the appropriate public key (public\_key.pem) are used for verification.

**Command:**

openssl dgst -sha256 -verify public\_key.pem -signature hashed\_password.txt password.txt

**Expected output:**  
• The original and verified hash values.

**Action required:**  
Capture a screenshot of the command you used to verify the terminal's hash and confirmation output.

**Sample output:**

**Task 8: Digital certificate generation**

**Scenario:**  
DataSafe Bank needs to issue digital certificates to its employees and internal systems for secure communication. You will use OpenSSL to generate a CSR (Certificate Signing Request) and sign it with the root certificate.

**Task:**  
Create a CSR for a server and generate a self-signed certificate.

**Step 1: Generate a private key for the server**

**Hint:**  
To generate the private key for the server:  
• Use the OpenSSL genpkey command with the -algorithm RSA option.  
• Specify the key size (2048 bits).

**Command:**

openssl genpkey -algorithm RSA -out server\_private\_key.pem -pkeyopt rsa\_keygen\_bits:2048

**Expected output:**  
• server\_private\_key.pem file containing the private key for the server.

**Action required:**  
Please capture a screenshot of the command you used to generate the server's private key and the confirmation output in the terminal.

**Step 2: Create a CSR**

**Hint:**  
To create the CSR for the server:  
• Use the OpenSSL req command with the -new option.  
• Specify the private key (server\_private\_key.pem) and the output CSR file (server\_csr.csr).

**Command:**

1. 1
2. openssl req -new -key server\_private\_key.pem -out server\_csr.csr

Copied!

**Expected output:**  
• server\_csr.csr file containing the server's CSR.

**Action required:**  
Capture a screenshot of the command you used to create the CSR and the confirmation output in the terminal.

**Step 3: Create a self-signed certificate**

**Hint:**  
To create a self-signed certificate:  
• Use the OpenSSL x509 command with the -req option to sign the CSR using the server's private key.  
• Specify the certificate's validity period (for example, 365 days).

**Command:**

openssl x509 -req -in server\_csr.csr -signkey server\_private\_key.pem -out server\_cert.pem -days 365

**Expected output:**  
• server\_cert.pem file containing the self-signed certificate.

**Action required:**  
Capture a screenshot of the command you used to create the self-signed certificate and the confirmation output in the terminal.

**Sample output:**