CNS Report: Project 2

Riwaaz Ranabhat, Elias El Bouzidi
University of Antwerp

December 8, 2024

1 Asymetric Encryption

1.1 RSA Implementation

RSA is one of the most important cryptographic techniques that is based upon public and private key's. The private key is kept secret and known only to the creator of the key pair, while the public key is available to anyone. The public key is used for encryption, while the private key of the creator is used for decryption. This algorithm was invented in 1977 by Ron Rivest, Adi Shamir and Len Adleman.
The RSA implementation can be seperated into **three** seperate operations:

- **Key Generation**
- **Encryption**
- **Decryption**

1.1.1 Key Generation

let's start from the beginning how we did it:

§ **P and Q Generation:**
Firstly, we need to start off with generating two prime numbers p and q where each of them are generated with half of the bit-size of the desired RSA modulus.

We initialize integer values **P** and **Q** by the use of **get_prime_number(nr_bits)** function. This function will keep generating an odd-prime number till it passes the **Miller and Rabin test**. At first, we thought that it would be the best if we just checked it in the default way. By this we mean like looping through and checking by the modulus but after writing python unit-tests using very large inputs we saw that it took a very long time to do so.

§ **Miller and Rabin Test Implementation**
To understand the algorithm, we referred to a YouTube video (linked in references). Here's the algorithm explained in simple terms:

1. Suppose $\mathbf{n}$ is the number we want to test for primality.
2. Write $\mathbf{n - 1 = 2^{exp} \cdot m}$

3. Randomly choose an integer **a** such that $\mathbf{1 < a < n - 1}$.
4. Compute $\boldsymbol{b_0 = a^m \mod n}$:
   - If $\mathbf{b_0 = 1}$ or $\mathbf{b_0 = n - 1}$,then **n** passes this round.
5. Suppose $\mathbf{b_0 = x}$ where $\mathbf{x \neq 1 \vee x \neq n - 1}$, calculate $\boldsymbol{b_1 = b_0^2 \mod n}$.
   - Continue squaring until you get $\mathbf{n - 1}$ (probably prime) or **1** (composite). We can generalize the loop in simple equation: $\boldsymbol{b_i = b_i^2 \mod n}$
6. Repeat the above steps **k** times for different random values of **a**.

§ **Calculate n**
Now that we have our values for **P** and **Q** we can proceed to calculated **n** by:

$\mathbf{n = P * Q}$

§ **Calculate Euler's Phi Function(totient)**
This is basically all positive integers < n that are relatively prime to n. Two numbers are relatively prime if and only if the gcd of those 2 is 1.

Calculate by: $\boldsymbol{\phi(n) = (p - 1) * (q - 1)}$.

§ **Generate e**
Generate **e** such that $\boldsymbol{1 < e < \phi(n)}$

§ **Calculate d**
We keep doing the previous step till $\boldsymbol{gcd(e, \phi(n)) = 1}$ This is done by the use of **extended euclidean algorithm** in the $\boldsymbol{get\_d()}$ function. Here we calculate the inverse of **e**.

Remark: this is seperated cause if the RSA modulus is **1024 bits** then we ofcourse used $\mathbf{e = 65537}$ straightforward where we then just had to get the corresponding d value out of $\boldsymbol{get\_d()}$ function.

Now that we have all the values we need we can easily generate public and private keys by the use of the RSA library described in our task.

1.1.2 Understanding the RSAES-OAEP

Before proceeding with the remaining 2 parts of the implementation, we needed to understand the working of the RSAES-OAEP encryption scheme. Similar to our experience with implementing Salsa20, we referred to the specs document for RSAES-OAEP and the RFC8017, as the theory lecture slides provided only a minimal explanation.(see references).

1.1.3 RSA Encryption implementation

The RSAES-OAEP encryption methods doesn't take the message as a whole but instead it takes in messageblocks. To split up our message into seperate blocks, we had to know how much bytes does the RSA modulus had and also what was the length of phash (aka

h_len). We then calculated it by:
block_size = n - 2 * len(phash) - 2
Now that we know how long every messageblock should be, we could start off by giving every block seperately to the **RSAES-OAEP Encryption** function. Let's see step by step how this process goes:

1. **EME-OAEP ENCODE**
   (a) **Padding Generation(PS)**

   Formula:

   $$\boldsymbol{PS = 0x00 \cdot (EM\_length - M\_length - 2 * h\_len - 2)}$$

   where **EM_length** is the length of **n** in bytes checking the key, **M_length** is the length of **M** and **h_len** is the length of the **phash**.

   (b) **DB Generation**

   Now that we have the **phash**, **Paddding** and the **M** we can generate the **DB** by:

   $$\boldsymbol{DB = phash||PS||0x01||M}$$

   (c) **MaskedDB Generation**

   Formula:

   $\boldsymbol{maskedDB = DB \bigoplus MGF(nonce, EM\_length - h\_len - 1)}$ where $\boldsymbol{Z = nonce}$ a

   (d) **MaskedSeed Generation**

   We generate this by using our **maskedDB** to generate new output by the **MGF(Z,l)** where $\boldsymbol{Z = maskedDB}$ and **l = h_len**:

   $\boldsymbol{masked\_seed = nonce \bigoplus MGF(maskedDB, h\_len)}$ where $\boldsymbol{Z = maskedDB}$ and $\boldsymbol{l = h\_len}$.

   (e) **EM Output Generation**

   Now that we have our **MaskedSeed** and the **MaskedDB** we can generate the final output of the **EME-OAEP encoding** method as follows:

   $$\boldsymbol{EM = 0x00||Masked\_seed||Masked\_DB}$$

(f) **Figure**
The previous steps that we described are fully visualized in this picture correctly:

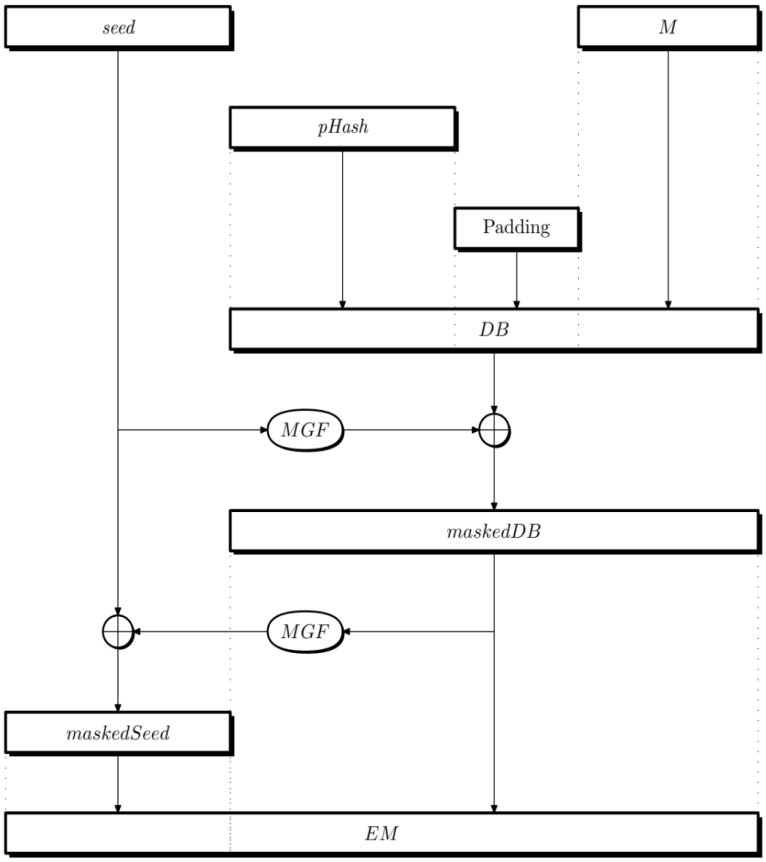

Figure C.1: *EME-OAEP encoding operation. pHash is the hash of the encoding parameters.*

As we know that the given messageblock is a string value, the RSA has primitives that act on integers. So we need to somehow convert a string to nonnegative integer and vice versa. Therefore we have **OS2IP**(Octet-String-to-Integer primitive) and **I2OSP**(Integer-to-Octet-String primitive).

2. **OS2IP**

   We gave the **EM Output** value to this function where it will generate the corresponding nonnegative integer for it.

3. **RSAEP**

   This function takes as input the public key **(n,e)** and a positive integer $\boldsymbol{m < n}$ and returns the integer $\boldsymbol{c = m^e \mod n}$

4. **I20SP**

   The output of RSAEP function is needed to generate a corresponding string which

will be our output of length $\mathbf{l = n}$ where n is given in the key.

1.1.4 RSA Decryption implementation

The Decryption Implementation is exactly the reverse of what we have to do of encryption. This mean't we decrypt the cipher by the primitives till we get our initial input values of the EME-OAEP Encoding scheme.

1. **OS2IP**

   Generate the corresponding nonnegative integer value by using the contentblock as input for this function where contentblock is the cipherblock that we have to process.

2. **RSADP**

   This function takes as input the private key and an integer $\boldsymbol{c < n}$ to return the integer $\boldsymbol{m = c^d \mod n}$ where d is an integer.

3. **I2OSP**

   The output of RSADP function is needed to generate a corresponding string which will be our output of length $\boldsymbol{l = n}$ where **n** is given in the key.

4. **EME-OAEP-DECODE**

   As we said earlier we are going to do the reverse of what we did in the encryption, we are going to get our initial values back again how we started to encrypt this message.

   (a) **Fetch MaskedSeed and MaskedDB**

   As we now in the encryption the EM was made by $\boldsymbol{0x00||MaskedSeed||MaskedDB}$. We can easily fetch them out again with the use of **h_len**.

   (b) **Fetch Seed(nonce)**

   To get nonce back again, we need to generate the output value of the function $\boldsymbol{MGF(MaskedDB, h\_len)}$. The none should be then again:

   $$\boldsymbol{Seed = MaskedSeed \bigoplus MGF(MaskedDB, h\_len)}$$

   (c) **Fetch DB**

   In order to generate the **DB** back again we need our **seed(nonce)**.

For this to happen we need the outputvalue of $\boldsymbol{MGF(Seed, EM\_len - h\_len)}$. Then , **DB** can be regenerated as follows:

$$\boldsymbol{DB = MaskedDB \bigoplus MGF(Seed, EM\_len - h\_len)}$$

(d) **Fetch Padding(PS) and M**

As we now in the encryption section, the **DB** was generated by concatenating $\boldsymbol{phash||Padding||0x01||M}$.
These values can be fetched out again with the use of **h_len** together with knowing that **0x01** seperates the **Padding** with **M**.

Now, we have got the decrypted value of for the one of the message blocks.

(e) **Figure**
As like in encryption, the previous steps that we described are fully visualized in this picture correctly:

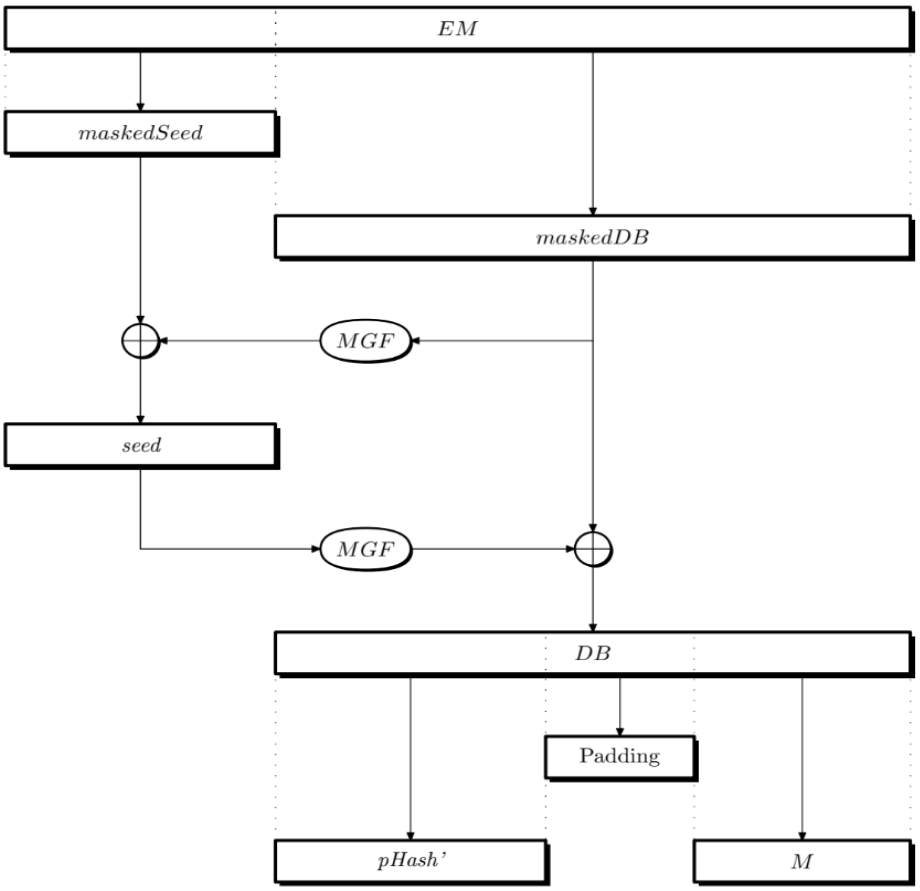

Figure C.2: *EME-OAEP decoding operation. If pHash' equals the hash of the encoding parameters, then decryption is considered successful.*

2 How we use RSA encryption

We use rsa encryption for the client ack in the handshake. We use rsa to encrypt the 2 session keys using the public key of the server and then send it to the server. So that the server can decrypt the session keys using its own private key. The client can also just use RSA as the encryption method for the traffic after we have established the handshake. We then encrypt the content using the public key of the server again and decrypt it with the server private key. We use here again the encryption header to store the nonce and the key id, wich is the session id if we use sessions otherwise for the preshared key we use the "cns_client" or "cns_flaskr" depending on who the receiver is.

3 Certificates

3.1 What are Certificates

Certificates are used in HTTP traffic (primarily with HTTPS) to establish secure communication between a client and a server. Here's how they work:

- **Authentication:** The server presents a digital certificate, issued by a trusted Certificate Authority (CA), to prove its identity to the client. This ensures that the client is communicating with the legitimate server and not an imposter.
- **Encryption:** Once the server's identity is verified, a secure session is established using public-key cryptography. The certificate contains the server's public key, which the client uses to encrypt a shared secret. This secret is then used to encrypt all subsequent communication (symmetric encryption).
- **Data Integrity:** Certificates help ensure that data exchanged between the client and server has not been altered or tampered with during transit.

3.2 What is a CA

A Certificate Authority (CA) is a trusted organization that issues digital certificates to verify the identity of entities, such as websites, individuals, or organizations, on the internet.

3.2.1 Key Roles of a CA

- **Identity Verification:** Before issuing a certificate, the CA verifies the identity of the requesting entity to ensure it is legitimate.
- **Certificate Issuance:** The CA generates and signs digital certificates using its private key. These certificates include the entity's public key, identity information, and the CA's signature.
- **Trust Establishment:** Certificates issued by a well-known CA are inherently trusted by browsers and operating systems, which maintain a list of trusted CAs.
- **Revocation:** If a certificate is compromised or no longer valid, the CA can revoke it and add it to a Certificate Revocation List (CRL) or update the Online Certificate Status Protocol (OCSP) to alert clients.

3.3 Creating Certificates

To be able to exchange session keys securely between cns_client and cns_flaskr, certifcates will have to be generated for a CA and cns_flaskr.

3.3.1 Generating keys

We first have to give the client, server and CA 2048 bit keys. We can do this by using our function that we implemented "generate_keys". This will generate a public and private key in the given paths. For the CA we only need to generate a private key, we can do this with the function "make_private_key" this will generate a private key in the given path. We have to generate these keys only once.

3.3.2 Setting up the CA

Now that we have the keys we can start setting up the CA:

- First we have to create a self signed certificate for the CA using its private key, we did this with the command: **"openssl req -x509 -new -nodes -key {PrivateKey} -sha1 -days 365 -out certificate.pem"**, where {privateKey} is the path of the private key of the CA. The certificate will be in the file certificate.pem and will be valid for 365 days.
- After this command we filled in the fields with the data given in the assignment.

3.3.3 Creating the server certificate and signing it

- Now we will create a certificate for the cns_flaskr. The command we used is **"openssl req -new -key {PrivateKey} -sha1 -out cns_flaskr.csr"**. Where {privateKey} is the path to the private key of the server.
- Now we have to fill in the fields using the data in the assignment
- Now we have to sign the servers certificate using the CA. We used the command **"openssl x509 -req -in {certificate_request} -CA {ca_certificate} -CAkey {ca_private_key} -out {certificate} -days 365 -sha1"**. Where we give the original certificate and the CA certificate and the CA private key. This signed certificate will also be valid for 365 days and will be in {certificate}

We do this whole process only once. Now we can verify if we are talking with the actual server.

3.4 Key fields in an X.509 Certificate

- **Version:** Specifies the version of the X.509 standard being used.
- **Serial Number:** A unique number assigned by the CA to each certificate.
- **Signature Algorithm:** The cryptographic algorithm used by the CA to sign the certificate. In this case it is sha1.
- **Issuer:** Identifies the entity (CA) that issued the certificate. Contains information such as the organization name, country, and common name (CN) of the CA.

- **Validity Period:** Specifies the time frame during which the certificate is valid. Contains two subfields, Not Before: The start date and time, Not After: The expiration date and time.
- **Subject:** Identifies the entity the certificate is issued to. Contains details such as, Common Name (CN): Domain name or entity name, Organization (O): Name of the organization, Country (C): The country where the entity resides.
- **Public Key:** The public key associated with the entity described in the Subject field.
- **extensions:** Additional fields for enhanced functionality. contains: Subject Key Identifier, Authority Key Identifier, Basic Constraints.
- **Signature:** The digital signature of the certificate, created using the CA's private key.

4 Sessions

4.1 What is a session

A session is a temporary interaction between a client and a server that persists across multiple requests. It allows the server to remember data about the client over time. In our case a session lasts 1 minute, where we can send traffic without changing the keys and verifying the server.

4.2 Key Exchange

Using the same key over and over again is not secure, definitely not with the compute power available nowadays. Therefore, each time a connection is made between client and server, a handshake is performed to exchange session keys that are used for securing the communication. By frequently changing the session keys, it is much harder for an attacker to figure out the session key and listen in on other people's communication. The handshake consists of 4 parts:

- **Client Hello:** The client will request the latest certificate of the server to set up a session. This request does not need to be encrypted
- **Server Hello:** The server responds to the Client Hello with its certificate, which is signed by a CA.
- **Client Ack:** The client checks the server's certificate using the public key of the CA, which is known by all parties. If the certificate checks out, it will use RSA encryption to send the two 100 character strings as random session keys to the server. We put the 2 keys in 1 variable content and encrypt this using rsa, where we use the public key of the server that we got out the certificate of the server.
- **Server Ack:** The server will decrypt the two session keys using its private key, which is only known to him. A session is created that is valid for 1 minute. Using the encryption key with AES and authentication key with HMAC, the server will

respond to the client with an encrypted session id and end timestamp in unix format (not rounded). For this message, the server will always use encrypt-then-authenticate.

4.3 The Attack: Man-in-the-Middle

The simplified handshake is vulnerable to a "Man-in-the-Middle attack", primarily due to the absence of certain key steps in the handshake process that ensure mutual authentication and protection against tampering.

- **Lack of Mutual Authentication:** The client verifies the server's certificate, but the server does not verify the identity of the client. This allows an attacker to impersonate the client and intercept communication after the session keys are sent.
- **Session Key Exchange Vulnerability:** The session keys are sent from the client to the server encrypted using RSA with the server's public key. If an attacker intercepts the Client Ack message and can replace the server's public key with their own (e.g., during an earlier step or via DNS spoofing), they can decrypt the session keys using their private key and then re-encrypt the traffic to forward it to the server. The attacker now acts as an intermediary, able to read and modify all communication.
- **No Integrity Checks During Key Exchange:** There's no mechanism to verify the integrity of the session key exchange or to ensure the session keys weren't tampered with.

4.3.1 How to avoid the attack

- **Implement Mutual Authentication:** Use client certificates or another method to authenticate the client during the handshake, ensuring both parties can verify each other.
- **Use Perfect Forward Secrecy:** Replace RSA encryption with a key exchange mechanism like Diffie-Hellman Ephemeral or Elliptic Curve Diffie-Hellman Ephemeral. PFS ensures that even if an attacker intercepts the session key exchange, they cannot retroactively decrypt the session if the private key is compromised.
- **Ensure Proper Certificate Validation:** The client must rigorously validate the server's certificate using the CA's public key. Any invalid or self-signed certificates should result in the termination of the handshake.
- **Include Message Authentication During Key Exchange:** Digitally sign all exchanged messages (e.g., Client Hello, Server Hello, and Client Ack) to ensure their authenticity and integrity. This prevents attackers from tampering with the key exchange.
- **Session Key Confirmation:** The server and client should explicitly confirm the derived session keys after the exchange by exchanging hashes or MACs (Message Authentication Codes) of the session keys, signed using their respective private keys.

5 How everything works together

When we go to the website cns_flaskr we get in the client proxy. Here we first check if the client supports sessions by looking if the sessions key is present in its configuration file. If so we then first check if there is already an active session. If so we then get the encryption key, auth key and session id out. If there is no active session we then will perform the handshake. So the client is going to do a client hello to ask for the certificate of the server. The server responds with a server hello with its certificate. The client than first checks the certificate using the ca to check if it is signed. If so we then go to the client ack where we encrypt the 2 session keys with rsa using the public key of the serve that we got out of the certificate. We then send it to the server to ask to create a session. The server decrypts the session keys with its own private key. It then creates a session with a unique session id and an end time (+60 sec). The server responds then with a server ack where we encrypt the session id and end time using aes and then authenticate it using hmac. So it is encrypt first and then authenticate. Now the client will decrypt the session id and end time using authorize first and then decrypt. Now the client will also save a copy of the session in its sessions folder. If the client does not support sessions we then use the preshared key as encrypt key and auth key. Now we have our keys so we can start sending traffic using the client method and the keys we got. If we used sessions we will have to make another one after a minute has passed. We use a garbage collection for this to delete expired sessions. In the server request and server response and client response we also check if sessions is supported if so we take the keys out the session otherwise we use preshared key.

References

[1] RFC8017
`https://datatracker.ietf.org/doc/html/rfc8017`

[2] RSA-OAEP-SPEC
`https://www.inf.pucrs.br/calazans/graduate/TPVLSI_I/RSA-oaep_spec.pdf`

[3] Miller & Rabin
`https://www.youtube.com/watch?v=8i0UnX7Snkc`