Lab4 Report

Rishi Vardhan Majji
24B0969

September 2025

1 Challenge 1: Public Encryption

You are given the RSA-OAEP private and public keys of a user in pub.pem and priv.pem respectively. The flag encrypted using the user's public key is provided to you in cipher.bin. Use openssl to decrypt the ciphertext to recover the flag!

1.1 Approach

Using openssl, this qsn can be solved using the terminal.

```
openssl pkeyutl -decrypt -inkey priv.pem \
-in cipher.bin -pkeyopt \
rsa_padding_mode:oaep
```

we get the flag **cs409{r54_043p_3t_0p3n_55l}**

2 Challenge 2: Is This Certified?

2.1 Description

Who digitally certified https://www.cse.iitb.ac.in?
The flag for this challenge is the name of the issuer of the digital certificate of https://www.cse.iitb.ac.in

2.2 Approach

Open the browser, go to the required website (cse.iitb.ac.in)
And then click on the icon left to the search bar
(the one that shows site information)

Then click on the option "Connection is secure", then on "certificate is valid" And then at the section issued by, you can see its common name as RapidSSL TLS RSA CA G1

hence the flag is **cs409{RapidSSL_TLS_RSA_CA_G1}**

3 Challenge 3: ECDSA Nonce Reuse

3.1 Description

A nonce should only be used once. Repeating nonce across signatures can lead to disastrous consequences. In this challenge, you'll be given two signatures generated using the same nonce.
You need to recover the nonce and in fact, the private key of the signer!

3.2 Approach

We are given two messages, m_1 and m_2, and their respective signatures, (r_1, s_1) and (r_2, s_2). We observe that $r_1 = r_2$, confirming nonce reuse. We can just use $r = r_1$.

Let $z_1 = H(m_1)$ and $z_2 = H(m_2)$. We have:

1. $s_1 = k^{-1}(z_1 + r \cdot d) \pmod{n}$
2. $s_2 = k^{-1}(z_2 + r \cdot d) \pmod{n}$

By solving for k:

$$
\begin{aligned}
k \cdot s_1 &= z_1 + r \cdot d \pmod{n} \\
k \cdot s_2 &= z_2 + r \cdot d \pmod{n} \\
\Rightarrow k(s_1 - s_2) &= z_1 - z_2 \pmod{n}
\end{aligned}
$$

From this:

$$k = (z_1 - z_2) \cdot (s_1 - s_2)^{-1} \pmod{n}$$

Once k is found, we can substitute it back into the first equation to solve for the private key d:

$$d = (k \cdot s_1 - z_1) \cdot r^{-1} \pmod{n}$$

We will implement these two formulas to recover both the nonce and the private key.
The code is as follows.................

First, we get the curve order n from `ecdsa.SECP256k1`.

```
n = ecdsa.SECP256k1.order
```

Next, we hash the two messages using SHA-256 and convert their hex digests to integers. These are our z_1 and z_2 values (here named `h1` and `h2`). We also set $r = r_1$.

```
h1 = int(hashlib.sha256(msg1.encode()).hexdigest(), base=16)
h2 = int(hashlib.sha256(msg2.encode()).hexdigest(), base=16)
r = r_1
```

We implement the formula for k. We compute the difference of s values and the difference of h values, modulo n.

```
# k = (h1- h2) * (s1- s2)^-1 mod n
s_diff = (s_1 - s_2) % n
h_diff = (h1 - h2) % n
```

We find the modular inverse of `s_diff` and multiply by `h_diff` to get the nonce.

```
# (s1- s2)^-1 mod n
inv_s_diff = inverse(s_diff, n)
nonce_rec = (h_diff * inv_s_diff) % n # Got the nonce
```

Now we implement the formula for d. We first find the modular inverse of r.

```
# d = (k*s1- h1) * r^-1 mod n
# r^-1 mod n
inv_r = inverse(r, n)
```

Finally, we compute the numerator $(k \cdot s_1 - h_1) \pmod{n}$ and multiply it by `inv_r` to recover the private key.

```
k_s1 = (nonce_rec * s_1) % n
k_s1_h1 = (k_s1 - h1) % n
privkey_rec = (k_s1_h1 * inv_r) % n # Got the private key
```

These two values, `nonce_rec` and `privkey_rec`, are sent to the server, which then provides the flag.

```
# flag-->cs409{n0nc3_5h0uld_b3_u53d_0nc3}
```

4 Challenge 4: EdDSA Variants

4.1 Description

One of the several digital signature schemes is EdDSA
In this challenge, you are provided with two insecure custom variants of EdDSA. Figure out how you can exploit these variants to forge signatures on arbitrary message

4.2 Approach

We will attack each variant separately to recover its private key, then use that key to sign the server's challenge. All calculations are modulo q, the order of the curve.

4.2.1 Variant 1: Deterministic Nonce

The vulnerability is that the nonce k (which we'll call `rKnown`) is not random, but is computed as $k = H(m \,||\, P_K.x)$. The signature equation is $s = k + e \cdot d \pmod q$, where h (which we'll call `hKnown`) is $h = H(R \,||\, P_K.x \,||\, m)$.

Since we can calculate k and h for any known signature, we can rearrange the equation to solve for the private key d (which we'll call `privkeyRec1`):

$$d = (s - k) \cdot h^{-1} \pmod q$$

Once we have d, we can forge a signature for the challenge message m_c by following the same rules.
The code goes as follows...........
First, we get the curve order q and take the first message and signature from the server.

```
q = ecdsa.NIST256p.generator.order()
G = ecdsa.NIST256p.generator
msgKnown = msgs[0].encode()
Rknown, Sknown = sigs[0]
```

We calculate the deterministic nonce `rKnown` (our k) for the known message.

```
rKnownHashInp = msgKnown + str(VARIANT1_PUBKEY.x()).encode()
rKnown = int(hashlib.sha256(rKnownHashInp).hexdigest(), base=16) % q
```

We calculate the challenge hash `hKnown` (h) for the known signature.

```
hKnownHashInp = str(Rknown.x()).encode() + str(VARIANT1_PUBKEY.x()).
 encode() + msgKnown
hKnown = int(hashlib.sha256(hKnownHashInp).hexdigest(), base=16) % q
```

Now we solve for the private key `privkeyRec1`.

```
hKnownInv = inverse(hKnown, q)
privkeyRec1 = ((Sknown-rKnown)*hKnownInv) % q
```

We forge the challenge signature by following the same flawed process:

```
# Calculate challenge nonce kc, point Rc, hash ec
kc_hash_inp = challenge_msg_1 + str(VARIANT1_PUBKEY.x()).encode()
kc = int(hashlib.sha256(kc_hash_inp).hexdigest(), base=16) % q
Rc = kc * G
ec_hash_inp = str(Rc.x()).encode() + str(VARIANT1_PUBKEY.x()).encode() + challenge_msg_1
ec = int(hashlib.sha256(ec_hash_inp).hexdigest(), base=16) % q

# Calculate final signature s = (kc + ec * d) mod q
s = (kc + ec * privkeyRec1) % q
R = Rc
```

4.2.2 Variant 2: Colliding Nonce via Message Prefix

The vulnerability here is that the nonce k is computed based on the private key and only the first half of the message: $k = H(\text{msg}[:\text{len}/2] \,||\, d)$. If we send two messages with the *same first half*, the server computes the same k for both, causing a nonce reuse.

This gives us $R_a = R_b$ and lets us solve for d:

$$s_a - s_b = (h_a - h_b) \cdot d \implies d = (s_a - s_b) \cdot (h_a - h_b)^{-1} \pmod q$$

With d, we can forge a signature by following the server's flawed logic.
The code is as follows...........
We get two signatures from same-prefix messages and find their hashes.

```
msgAbytes = msgs[0].encode()
msgBbytes = msgs[1].encode()
R_a , s_a = sigs[0]
R_b, s_b = sigs[1]
Rknown = R_a # R is the same

h_aHashInp = str(Rknown.x()).encode() + str(VARIANT2_PUBKEY.x()).
 encode() + msgAbytes
h_a = int(hashlib.sha256(h_aHashInp).hexdigest(), base=16) % q

h_bHashInp = str(Rknown.x()).encode() + str(VARIANT2_PUBKEY.x()).
 encode() + msgBbytes
h_b = int(hashlib.sha256(h_bHashInp).hexdigest(), base=16) % q
```

We implement the formula to recover the private key `privkeyRec2`.

```
sDiff = (s_a- s_b) % q
hDiff = (h_a- h_b) % q
hDiffInv = inverse(hDiff, q)
privkey = (sDiff * hDiffInv) % q
```

Now, we forge the signature for `challenge_msg_2` using the key.

```
# Calculate the challenge nonce 'kc' based on its prefix
msg_c = challenge_msg_2
msg_c_prefix = msg_c[:len(msg_c)//2]
kc_hash_inp = msg_c_prefix + str(privkey).encode()
kc = int(hashlib.sha256(kc_hash_inp).hexdigest(), base=16) % q

# Calculate the challenge point 'Rc'
Rforge = kc * G

# Calculate the challenge hash 'hc'
hc_hash_inp = str(Rforge.x()).encode() + str(VARIANT2_PUBKEY.x()).encode() + msg_c
hc = int(hashlib.sha256(hc_hash_inp).hexdigest(), base=16) % q
```

```
# Calculate the final signature 'sc'
sForge = (kc + hc * privkey) % q
R = Rforge
s= sForge

    # flag-->cs409{3dd54_g0t_m3_t4lk1ng_n0nc353nc3}
```

We get the flag **cs409{3dd54_g0t_m3_t4lk1ng_n0nc353nc3}**

5 Bonus Challenge: Grover's Cipher

The server's challenge is to find a 32-bit (4-byte) message. The normal way to solve this would be to **brute-force**, or guess all 2^{32} possible messages. This is way too slow and we'd run out of time.

So we use an attack called meet-in-the-middle attack that's much faster. Instead of solving one huge 32-bit problem, we'll split it into two easy 16-bit problems.

This is how it goes:

1. **Split the List:** The server gives us a list v of 32 numbers. We split this list into two halves: a `vLeft` (the first 16 numbers) and a `vRight` (the last 16 numbers).

2. **Make a dictionary for the Left Half:** We pre-calculate all 2^{16} possible answers for the `vLeft` list. We store these in our dictionary where the key is the product c_L, and the value is the 16-bit mask m_L that made it.

3. **Search the Right Half:** Now, we loop 2^{16} times, checking every possible mask m_R for the `vRight` list. For each one, we calculate its product c_R.

4. **Find the Match:** We know that $c_L \times c_R = c$ (the final answer). We can rearrange this to ask: "What c_L do I need to find in the dictionary made?" The answer is $c_L = c \times (c_R)^{-1} \pmod{pub}$. We look up this target c_L.

5. The moment we find a target c_L that is in our phone book, we've found our match! We get the `m_L` from the phone book and have the `m_R` from our loop. We just stick them together to make the final 32-bit message: $m = (\texttt{m\_L} \ll 16) | \texttt{m\_R}$.

The code goes as follows...........
First, we split the 32-element vector `v` into two 16-element halves.

```
    n = len(v)
    half_n = n // 2

    vLeft = v[:half_n]
    vRight = v[half_n:]
```

We create the lookup table `leftTable` by iterating through all 2^{16} possible masks for the left half (`m_L`) and storing their products.

```
leftTable = {}
for m_L in range(1 << half_n):
    c_L = 1
    for i in range(half_n):
        # Check the (15-i)-th bit of m_L
        if (m_L >> (half_n - 1 - i)) & 1:
            c_L = (c_L * vLeft[i]) % pub
    leftTable[c_L] = m_L
```

Now we search the right half. We iterate through all 2^{16} masks for the right half (`m_R`) and compute their products `c_R`.

```
for m_R in range(1 << half_n):
    c_R = 1
    for i in range(half_n):
        # Check the (15-i)-th bit of m_R
        if (m_R >> (half_n - 1 - i)) & 1:
            c_R = (c_R * vRight[i]) % pub
```

We check if `c_R` is invertible. If it is, we find its inverse and calculate the target `c_L_target` we need to find in our table.

```
    if gcd(c_R, pub) != 1:
        continue

    c_Rinv = pow(c_R, -1, pub)
    c_L_target = (c * c_Rinv) % pub
```

If the target exists in `leftTable`, we've found our match. We reconstruct the full 32-bit solution and break the loop.

```
    if c_L_target in leftTable:
        m_L_found = leftTable[c_L_target]

        solution_mask = (m_L_found << half_n) | m_R
        break
```

Finally, we convert the 32-bit integer mask into the 4-byte hex string the server expects.

```
n_bytes = (n + 7) // 8
message_bytes = long_to_bytes(solution_mask, n_bytes)
message = message_bytes.hex()

# flag-->cs409{4174_br34k5_unbr34k34bl3_c1ph3r5}
```