Q1-)

In the provided code, an RSA padding oracle attack is implemented against a given server. The script retrieves an encrypted message (`c`) and server's RSA public parameters (`N`, `e`). It then selects a random number `r` that is relatively prime to `N`, and creates a new modified ciphertext `c\_` by multiplying `c` with `r` raised to the power of `e` modulo `N`. This modified ciphertext is sent to the oracle, which returns the decrypted plaintext `m\_`. Finally, the original plaintext is computed by multiplying `m\_` with the modular inverse of `r`, and the result is checked using the server's checking oracle.

My output is:

{'c': 478062955999489464723129106262901943901259644443714128151232674102666628019227902477926915467645399353339160477569997015890937095316508473966531815214009654683254149894868859770024379735436653381818375098573631408792893364973192239646098314039025111823868855498344000585538833082635792572846977212143062679507554411498290293628996402780325496054991465944061563709132070620409909009720640227475486348746686720764378916802662053413891135573267141358093909984695938566226402574072183132135801275646828633741424333237583036357491954338933191256705272321780709392081312905545081531292628800655194696725846560965608025225, 'N': 16939269462313198277725089002524968140769895904731797247921693446825361577176513265093962001473249149493514478809355121972226854971742591412020947781343582572258409348854687792672350790838706795045105779549770778632847428289027972956461589642862664604406205649788062899700494060455208957227204718515101331592572887762868780071870612248382350488623041960895952847347193036811161635348404536654480418700288618890111185617389009319953809419282210422869145631879305871459779170673590029863798266603861263916454445940018625253525716638055179599474296505004768081393212277916373461197482274715508530005125354616487101316387, 'e': 65537}

{'m\_': 12407491179956912115634319563325746932816392325625814759792017831013031874392971286747642877026856214651833354145477545104818886830099372173297111581452069276370667867303801453137529072230549332291861565106486745260041329019820720818410124877797756337239478657145397613128959114107869651726224373040935765486472696617777331954877105596910643583811242819435854835200297237906659006213549045109798392798526815825586073399915532578345017797491571390755757506084065952833542976007410285185055627335277666251973009110156591554457883768462524282568268420923323886788027880082874499071677249223517917612521422141206268573067}

Congrats

You can check RSA\_Oracle\_client.py for my solution

Q2-)

I implemented a brute-force approach to find the correct PIN by iterating over all possible four-digit PINs (0000 to 9999) and all possible 8-bit values of **R** (0 to 255). For each combination of PIN and **R**, the code encrypts the PIN using RSA OAEP encryption and compares the resulting ciphertext with the provided ciphertext. If a match is found, the correct PIN is printed.

Output:

Found PIN: 1308

You can check my code on RSA\_OAEP.py

Q3-)I found K by using brute force range is [1,2^16-1) and my decryption formula is m=t X (h^kmodp)modp

So I found : 'Be yourself, everyone else is already taken.' You can chechk ElGamal.py for my code.

Q4-)

In the provided solution, I exploited the flaw in the ElGamal encryption where the same random number `r` was used for encrypting both messages. By using the known first message (m1), its ciphertext components (r1, t1), and the ciphertext components of the second message (r2, t2), I calculated the second message (m2). This was achieved by deriving a relationship between the ciphertexts of m1 and m2 and then applying modular arithmetic to recover m2 in its original byte format.

Recovered m2: b'A person can change, at the moment when the person wishes to change.'

You can check my solution on hw4\_q4.py

Q5-)

To recover the secret key `a` in the DSA scheme, I exploited the known relationship between the nonces used in two signatures, where `k\_2 = 3k\_1 mod q`. By analyzing the given signatures `(r\_1, s\_1)` and `(r\_2, s\_2)` for two different messages, along with their respective hashes, I set up and solved an algebraic equation that relates these signatures and their nonces. This approach leveraged the flaw in nonce generation to algebraically derive the private key `a`, demonstrating how nonce reuse or predictability in DSA can lead to a compromise of the key's confidentiality.

Recovered secret key (a): 11216356259205076775220144248676143661414712556656426573701019179325

You can check DSA.py for my solution.