Task 2:

Q1. Is the hash of the salted password still vulnerable? Why or why not?

A1. Hash of the salted password offers better security as compared to hash of unsalted password. Specifically, it offers safety against rainbow table (a table of precomputed hashes of common passwords) attack and dictionary attack to break passwords of all the accounts simultaneously. But they are still vulnerable to dictionary attack to break password of a single user at a given time. Assuming that the attacker gets users salt using either via SQL injection, data leaks or direct access to database, this attack will look as follows:

```
for user in users:
    PER_USER_SALT = user["salt"]
    for password in LIST_OF_COMMON_PASSWORDS:
        if sha256(PER_USER_SALT + password) in database_table:
            print user["id"], password
```

The above attack becomes infeasible if PER_USER_SALT is not known and is long and random enough that is cannot be guessed.

Q2. What steps could be taken to enhance security in this situation?

A2. The real problem with hash of salted hash is that hash functions like sha256() can be executed on passwords at a rate of 100M+/second (or even faster, by using the GPU). Even though these hash functions were designed with security in mind, they were also designed so they would be fast when executed on longer inputs like entire files. Bottom line: these hash functions were not designed to be used for password storage. Therefore, so as to increase security in this situation use slower hash functions such as bcrypt or pbkdf2. These functions executes internal encryption/hash function multiple times (no of iterations is configurable) and generally take ~1 sec to generate a hash. This makes attacks as described above infeasible.

Task 3:

Q1. What steps did you follow to get the private key?

A1. Security of RSA lies in difficulty in factoring product of 2 “large” prime numbers. But in the number(n) to be factored is just a 64 bit number. Also, we know that:

$n = p * q$:= where $1 < p, q < n$ and both p and q are prime

From above we can infer:

$2 <= p <= \sqrt{n}$ and:

$q = n / p$

So, find p by iterating from 2 to sqrt(n) (only odd no are checked [except 2]) and checking if completely divides n.

Once p and q are found, the private key can be found very easily from the definition of RSA:

d*e = 1 (mod phi(p * q))

d = inv(e) (mod phi(p * q))

Where phi(k) is Euler's totient function which counts positive integers up to a given integer k that are relatively prime to k.

Task 4:

Q1. What makes the key generated in this situation vulnerable?

A1. Keys generated (n1 and n2) in this situation has a common factor. So, if there is way to find this common factor easily, the uncommon factor can also be computed simply be diving n1 and n2. Therefore, we would indirectly factorized both n1 and n2 and broken RSA.

Q2. What steps did you follow to get the private key?

A2. It is very easy to compute the GCD for 2 numbers, however large! If this GCD is not equal to 1, it means there is common factor between the 2 numbers.

g = GCD(n1, n2)

q1 = n1 / g

n1 = g * q1

Using above, private key can be easily computed using definition of RSA:

d = inv(e) (mod phi(g * q1))

Task 5:

Q1. How does this attack work?

A1. In code 3 different attack vectors are considered. But the successful attack in this case is broadcast attack. This attack succeeds because “same message” (no additional randomized bits) is encrypted using multiple public keys. As a result of which we can leverage Chinese Reminder Theorem and get the number before mod operation was performed and then take eth of this number to get original data.

The way to prevent this attack is to add random nonce while encryption.

Q2. What steps did you follow to recover the message?

A2. 3 different attack vectors are considered:

1. Low exponent vulnerability i.e.
   if C1 == C2 or C2 == C3 or C1 == C3,
   then it would mean m = eth_root of corresponding Cn.
2. Same as Task 4 above, where n1 and n2 share a root.
3. Broadcast attack:

   C1 = m**e mod N1

   C2 = m**e mod N2

   C3 = m**e mod N3

   Implies =>

   m**e = N1*i + C1

   m**e = N2*j + C1

   m**e = N3*k + C1

   Where i, j and k are some integers

   Implies =>

   m**e = C1 mod N1

   m**e = C2 mod N2

   m**e = C3 mod N3

   By Chinese Reminder Theorem =>

   m**e = (C1.N2.N3.(inv(N2.N3) mod N1) +

   C2.N1.N3.(inv(N1.N3) mod N2) +

   C3.N1.N2.(inv(N1.N2) mod N3)) mod N1.N2.N3

   m = eth_root(m**e)