Problem 2

In Problem 1, we exploit two vulnerabilities in TwoDrive's implementation:

1. ECDSA repeated-nonce flaw (Part 1.1).
   - TwoDrive uses only one fresh random byte when generating the per-signature nonce k in ECDSA. Because k must remain unique, reusing k (or collisions in r) allows for a standard repeated-nonce attack on ECDSA.
   - By collecting enough signatures from the server, we eventually see two signatures (r, s_1) and (r, s_2) that share the same r. From these two signatures, we solve for k and then the server's ECDSA private key sk.
   - Concretely, if we have two distinct messages m_1, m_2 with hashes e_1, e_2 but the same r value in the signatures, then$k = (e_1 - e_2) \times ((s_1 - s_2)^{-1} \text{mod} N)$ and $sk = (s_1 \cdot k - e_1) \times (r^{-1} \text{mod} N)$,where N is the order of the elliptic-curve group.
2. Handshake misuse / failing to verify the user's identity (Part 1.2).
   - Once we have the server's ECDSA private key, we notice that the server also reuses the same random seed for its (static) Diffie–Hellman key in the handshake. In the assignment's simplified setting, we exploited the fact that we could impersonate the server by crafting a malicious handshake message pretending to come from the registered user—but passing the server's static key as if it were ours.
   - Because the server's implementation uses the "K" pattern (only the server's identity is known in advance) and does not confirm that the client's ephemeral key truly corresponds to the same static public key, an attacker can produce a handshake message that the server accepts as if it originated from the legitimate user.

Putting these pieces together:

- Recover the Server's ECDSA Private Key.
- Convert ECDSA Key to an X25519 Private Key.
- Create a Responder-like Handshake Object and Initialize.
  - We supply our "fake" server static key (server_static_kp) as s.
  - We use the legitimate user's public key (params.client_static_pk) as rs.
  - This object has everything needed to mimic what a real initiator would produce in the K pattern.
- Forge the Initiator's Handshake Message.
  - Generate an ephemeral key and append it to msg_buffer.
  - Perform the es Diffie–Hellman and update the chaining key.
  - Perform the ss Diffie–Hellman and update the chaining key.
  - Finally, encrypt target_data using the derived handshake keys.
- Send the Forged Message to the Server.

Because the handshake code does not properly verify the client's identity, the server accepts this forged update. That is how we modify the victim's storage even though we are not a registered user.

Problem 3

A likely cause here is a lack of replay protection in the protocol. Although the server now fixes its ECDSA implementation, they may have left the handshake and update mechanisms vulnerable to replay attacks. For example:

1. An attacker records a valid handshake message and encrypted payload (an “older update”) from the legitimate user at some earlier time.
2. Even after the server fixes the ECDSA nonce bug, it may still accept that exact older handshake message if it is replayed.
3. Because there is no sequence number or version checking and no expiration of ephemeral keys, the old handshake message and ciphertext are still considered valid.
4. The server “updates” the cloud storage with the old data, effectively rolling back the user’s data to a past version.

Hence, the attacker’s approach is to simply capture a previously valid handshake+payload, then replay it to the server. Without additional integrity checks such as anti-replay counters or versioning at the application layer, the server mistakenly processes this old update.

Problem 4

Recall the difference:

- K pattern: The initiator knows the responder’s static key, but the responder does not know the initiator’s static key in advance (the user is “K”); the user’s static is only mixed in as a pre-message on the server side. This is simpler to break if the server does not authenticate the client’s static key properly.
- KK pattern: Both initiator and responder have each other’s static keys from the start. As part of the handshake, both sides effectively prove knowledge of their private key.

In most correct KK implementations, you need to prove possession of your static private key. Simply forging ephemeral keys is no longer enough, because the server will check whether the ephemeral handshake computations match the user’s known static public key. You do not have that user’s static private key, so you cannot produce the correct handshake values.

Hence, under the KK pattern—provided it is implemented correctly—an attacker who only knows the server’s private key cannot masquerade as the user. The attacker would fail to complete the user’s side of the handshake because the server expects a match between the ephemeral handshake data and the known user static key.

Therefore, our Part 1.2 approach would not succeed in the KK pattern. The server would see that the ephemeral handshake values do not match the legitimate user’s static key (unless we also somehow stole the user’s private key, which was out of scope).