Problem 1:

IC2: If the commander is loyal, then every loyal lieutenant obeys the order it sends.

Initially, we assume that the commander is loyal, and he sends command out to all lieutenants. In each of $m+1$ rounds, lieutenants forward all received and authenticated messages, appending their own signatures. Traitors may try to forge or alter messages. Due to message authentication, traitors cannot forge messages from loyal nodes. Any message purportedly from a loyal node can be verified for authenticity. Therefore, only loyal lieutenants can legitimately propagate the commander's order.

Let l be the number of loyal lieutenants, where $l \geq 2m$. In $m+1$ rounds, any loyal lieutenant receives messages that have traversed multiple disjoint paths from the commander. Apparently, there are at least $l - m > m$ paths are loyal. After $m+1$ rounds, each loyal lieutenant collects all received orders along with their signature chains and selects the majority command.

By contradiction, we suppose that two loyal lieutenants l_1 and l_2 decide on two different commands. Since they're both loyal, they must have received a majority of messages supporting their respective decisions. Given $l > 2m$, the overlap of loyal paths supporting both orders must be at least $l - m > m$. This implies that there's at least one common path supporting both orders. However, as the commander is loyal, he would never send inconsistent commands, which contradicts the assumption that l_1 and l_2 are following distinct orders.

Therefore, all loyal lieutenants obey the order sent by commander if he's loyal.

Problem 2:

Yes. Consider the scenario where there are $n = 3$ sites: A, B and C. When $m = n - 1 = 2$, we can assume that B is honest, while A and C are traitors. Suppose the commander of log slot 0 is A ($0 \bmod 3 = 0$). If B wants to execute *put key value*, it sends a proposal message to A, requesting A to initialize the protocol. However, A may never start the initialization process. Eventually, B won't receive a consistent message set that leads to a final decision before the timeout. It will simply print *unable to execute*. From B's perspective, slot 0 remains unfilled. In this scenario, despite the honest site's attempt to fill the log slot, the slot remains undecided and effectively "unfilled" for that honest site because the designated commander is traitor, and no mechanism exists to circumvent that.

Problem 3:

To allow multiple proposals, we will update the *handle_propose* method by add all proposed commands to the set rather than overwriting or assuming a single command.

The following code snippet

```
# Commander: Immediately start dolev-strong at round 0 with this command
if not self.slot_state[slot]["executing"]:
    self.slot_state[slot]["executing"] = True
    # ... existing code ...
    self.slot_state[slot]["round_values"][0].add(command)
```

will be changed to

```
# Commander: Immediately start dolev-strong at round 0
if not self.slot_state[slot]["executing"]:
    self.slot_state[slot]["executing"] = True
    # If multiple proposals are allowed, we might have already added others.
    # Just ensure we add this new proposed command as well.
    self.slot_state[slot]["round_values"][0].add(command)
# If this slot was already executing, and another propose ar-
rives, also add it:
else:
    self.slot_state[slot]["round_values"][0].add(command)
```

This ensures that if multiple "propose" messages arrive for the same slot, all proposed commands accumulate in round_values[0]. To ensure all honest sites select the same command from multiple candidates, we'll implement a deterministic tie-breaking rule by changing the following code snippet in *run_dolev_strong*

```
all_values = set()
for r in range(self.m+1):
    all_values.update(self.slot_state[slot]["round_values"][r])
decided_value = None
if all_values:
    decided_value = sorted(all_values)[0]
```

to

```
all_values = set()
for r in range(self.m+1):
    all_values.update(self.slot_state[slot]["round_values"][r])
decided_value = None
if all_values:
    original_command = self.proposed_commands.get(slot)

    if original_command in all_values and not self.is_traitor:
        decided_value = original_command
    else:
        decided_value = sorted(all_values)[0]
```

In *lieutenant_dolev_strong*, we will modify the following section to ensures that no matter how many proposals were received, all honest nodes pick the same final command

```
Vi = self.slot_state[slot]["values_set"]
decided_value = None
if len(Vi) == 1:
    decided_value = list(Vi)[0]
else:
    decided_value = "append sneak attack"
```

```
Vi = self.slot_state[slot]["values_set"]
decided_value = None
if len(Vi) > 0:
    # Apply the same deterministic tie-breaking rule used by the commander:
    original_command = self.slot_state[slot].get("proposer_command", None)
    if original_command in Vi and not self.is_traitor:
        decided_value = original_command
    else:
        decided_value = sorted(Vi)[0]
else:
    # If no values somehow, fallback:
    decided_value = "append sneak attack"
```

Problem 4:

IC1: All loyal lieutenants obey the same order.

We have shown that all loyal lieutenants follow the commander's initial order if he's loyal in Problem 1. Hence, we only need to consider the case where the commander is a traitor.

When the commander is a traitor, he may send inconsistent orders to different lieutenants. However, as lieutenants forward these orders, the recursive message passing ensures that the majority of messages received by any lieutenant come from loyal sources who have relayed consistent orders. Since $l > 2m$, the loyal lieutenants' consistent messages outweigh the conflicting messages from traitors, leading all loyal lieutenants to converge on the same order.

As a result, all loyal lieutenants obey the same order.