Storage Clash via Delegatecall

Explainer

When a contract performs a `delegatecall` to another contract:

- The *code* runs from the target,
- But **storage is read and written in the caller's context**.

If the implementation (callee) has variables in **different storage slot positions** than the proxy (caller), storage gets **corrupted**.

This is known as a **storage layout clash**. It causes:

- Unexpected values in `msg.sender`, `owner`, balances, etc.
- Broken access control
- Stuck tokens
- Protocol-wide bricking

> Even a one-slot misalignment (like adding a `bool` at the top) can break everything.

Cause

- `delegatecall` reuses the *calling contract's* storage.
- If the implementation contract has a **different layout** (e.g., added vars, changed order), state variables **overwrite wrong slots**.
- Happens in:
  - Proxy patterns (e.g., UUPS, Transparent, Diamond)
  - Plugin systems (e.g., DAOs, modules)
  - Upgradable contracts with inline delegate logic

Where to Look (General)

Critical zones:

1. **Any use of `delegatecall()`**

   ```
   (bool success, ) = impl.delegatecall(data);
   ```

2. **Proxy → Implementation interactions**
   - Does the implementation assume ownership or balance that isn't stored in proxy?
3. **Upgradeable contracts with changed layout**
   - Initial layout:

     ```
     uint256 a;
     address b;
     ```

   - Upgraded version:

     ```
     bool hacked;     // ❌ changes slot alignment
     uint256 a;
     address b;
     ```

4. **Diamond proxies with multiple facets**
   - Each facet's layout must be aligned carefully
5. **Inline plugins, strategies, or modules using delegatecall**

Why This Happens

- Developers modify implementations without respecting **strict storage layout**
- Solidity doesn't warn or detect storage layout mismatch
- Upgradeable contracts **look like normal Solidity**, but behave like assembly
- Small additions (e.g., a single `bool`) shift the entire layout

---

General Recommended Solutions

Use storage gap pattern (OpenZeppelin standard):

```
uint256[50] private __gap;
```

- Reserves slots for future variables without breaking layout

Always **inherit** from the same base contracts in same order

Lock proxy storage layout using a **shared base contract**:

```
contract StorageLayout {
    address internal owner;
    uint256 internal totalSupply;
}
```

In Diamonds: isolate storage by hashing a unique ID:

```
bytes32 constant STORAGE_SLOT = keccak256("my.facet.storage");

struct Data { address x; uint256 y; }

function getStorage() internal pure returns (Data storage ds) {
    assembly { ds.slot := STORAGE_SLOT }
}
```

NEVER reorder or prepend variables in upgraded contracts

Use Hardhat's `@openzeppelin/hardhat-upgrades` or Foundry plugins to detect unsafe upgrades

Write upgrade tests that:

- Deploy v1
- Store data
- Upgrade to v2
- Assert that old data remains unchanged

Example Incidents in the Wild

Protocol	**Bug Description**
Parity Wallet (2017)	Delegatecall corrupted ownership, allowing self-destruct
UUPS proxy forks	Admin slot overwritten via layout clash
Diamond proxies	Facet storage clashed due to layout assumptions