Users Can Be Permanently Locked Out of Staked Funds Due to Transfer Quota Enforcement

```
function unstake(uint256 scheduleIndex, uint256 nonce, bytes memory userSig)
external whenNotPaused protected nonReentrant {
        _isPayoutDisabled();
        require(0 <= scheduleIndex && scheduleIndex < s.stakeSchedulesCount,
"Invalid schedule index");

        StakeSchedule storage schedule = s.stakeSchedulesMap[scheduleIndex];
        address userAddress = schedule.userAddress;
        address tokenAddress = schedule.tokenAddress;
        uint256 amount = schedule.amount;

        _useNonce(userAddress, nonce);
        _verifySignature(userAddress, userSig, abi.encode(TYPEHASH_UNSTAKE,
scheduleIndex, nonce));

        require(schedule.isUnstaked == false, "Stake has been unstaked");
        schedule.isUnstaked = true;

        _tokenTransferOutQuoteCheck("unstake", tokenAddress, amount);
        require(IERC20(tokenAddress).transfer(userAddress, amount), "Transfer
failed");
        s.totalStakingAmountMap[tokenAddress] -= amount;
        s.userStakingAmountMap[tokenAddress][userAddress] -= amount;

        _removeVotingPowerFromStake(userAddress, scheduleIndex, amount);
        _removeStakeScheduleIndex(userAddress, scheduleIndex);

        emit Unstaked(userAddress, tokenAddress, amount, nonce, scheduleIndex);
    }
```

```
function _tokenTransferOutQuoteCheck(string memory context, address tokenAddress,
uint256 amount) internal {
        TTOQManagerStorage.Storage storage $ = TTOQManagerStorage.load();
        $.usedTokenTransferOutQuoteMap[tokenAddress] += amount;
        if ($.maxTokenTransferOutQuoteMap[tokenAddress] <
$.usedTokenTransferOutQuoteMap[tokenAddress]) {
            revert ExceedsMaxTokenTransferOutQuote(
                context,
                tokenAddress,
                amount,
                $.usedTokenTransferOutQuoteMap[tokenAddress],
                $.maxTokenTransferOutQuoteMap[tokenAddress]
            );
```

```
        }
        emit TTOQUpdated(context, tokenAddress, amount,
$.usedTokenTransferOutQuoteMap[tokenAddress], msg.sender);
    }
}
```

---

1) Root cause

`unstake()` marks the schedule `isUnstaked = true` **before** performing the transfer and before verifying the *remaining transfer quota* is sufficient. The quota check (`_tokenTransferOutQuoteCheck`) can revert when the global `maxTokenTransferOutQuote[token]` is smaller than the individual scheduled amount. Because `isUnstaked` was already set to `true`, the schedule is logically consumed even though transfer reverted — leaving the user with funds they can no longer withdraw (and the contract thinks that stake was unstaked).

---

2) Why this deadlocks users (attack/accident scenario)

- Operator sets `maxTokenTransferOutQuote[token]` to a value smaller than some existing stakes (or a malicious admin lowers it).
- A user with stake > `max` calls `unstake()`:
  - Function sets `isUnstaked = true`.
  - `_tokenTransferOutQuoteCheck` increments `used` and sees it exceeds `max` → reverts.
  - Entire tx reverts, but since the code set `isUnstaked` *before* the revert path, the contract state may still end up inconsistent in some flows (or future code assumes schedule is consumed). Even if revert fully rolls back, subsequent admin actions (e.g., setting max lower) and logic can cause the schedule to become unreachable (depending on other code paths). More importantly, the contract *design* allows a situation where valid unstake attempts will always fail when `max < stake`, and there is no recovery path for users whose stake exceeds a lowered quota.
- Result: users who staked large amounts before quota enforcement cannot withdraw — **funds locked**.

(Practically: even if a single revert keeps state unchanged, the policy of a strict quota lower than some stakes creates an inescapable condition where all future attempts will revert until the quota is raised — effectively locking funds.)

---

3) Immediate mitigations (fast actions)

These are things you can do right now if you control the contract/admin:

1. **Raise the** `maxTokenTransferOutQuote[token]` at least to the size of the largest single stake for that token (or to `s.totalStakingAmountMap[token]`) so future unstake calls succeed.
   - Quick emergency fix, but not a long-term design fix.
2. **Add an emergency owner operation** to process (or allow) forced withdrawals for affected schedules (owner-initiated manual withdrawal), so stuck users can be recovered.
3. **Pause quota changes temporarily** (if contract has a pause) until you deploy a proper fix/migration.
4. **Notify users / frontends** immediately if you operate the system so users don't continue interacting and get stuck.

Vulnerability Walkthrough

1. **User stakes tokens before quota**
   - Suppose Alice stakes **1000 USDC** into the system.
   - At this time, `maxTokenTransferOutQuoteMap[USDC]` might be unset or set to a high value.
   - Her schedule looks like:

     ```
     schedule.amount = 1000
     schedule.isUnstaked = false
     ```
2. **Protocol later reduces quota**
   - Admin (or governance) sets

     ```
     maxTokenTransferOutQuoteMap[USDC] = 100
     ```
   - This limit now applies globally for `unstake`.
3. **Alice tries to unstake**
   - Flow enters `_tokenTransferOutQuoteCheck("unstake", USDC, 1000)`
   - It increments usage:

     ```
     usedTokenTransferOutQuoteMap[USDC] += 1000
     ```
   - Then checks:

     ```
     if (1000 > 100) revert ExceedsMaxTokenTransferOutQuote
     ```
   - ❌ Reverts immediately → Alice is **permanently locked**.
4. **Irrecoverable State**
   - `schedule.isUnstaked` was already set to `true` **before** `_tokenTransferOutQuoteCheck`.
   - That means even if quotas are raised later, Alice can't retry — her stake is "burned" logically but not paid out.
   - Funds are stuck forever in the contract.