MintClub-Staking

Investigation

Summary

On August 25th, 2025, the MintClub team reported a precision loss issue in staking reward calculations for pools using USDC (6 decimals) as the reward token.

The vulnerability stems from precision loss in the calculation of `accRewardPerShare`, which tracks accumulated rewards per token. When the `stakingToken` amount becomes extremely large, it can result in `accRewardPerShare` no longer increasing, while `totalAllocatedRewards` continues to accumulate. This leads to a situation where these rewards cannot be distributed and become locked in the contract.

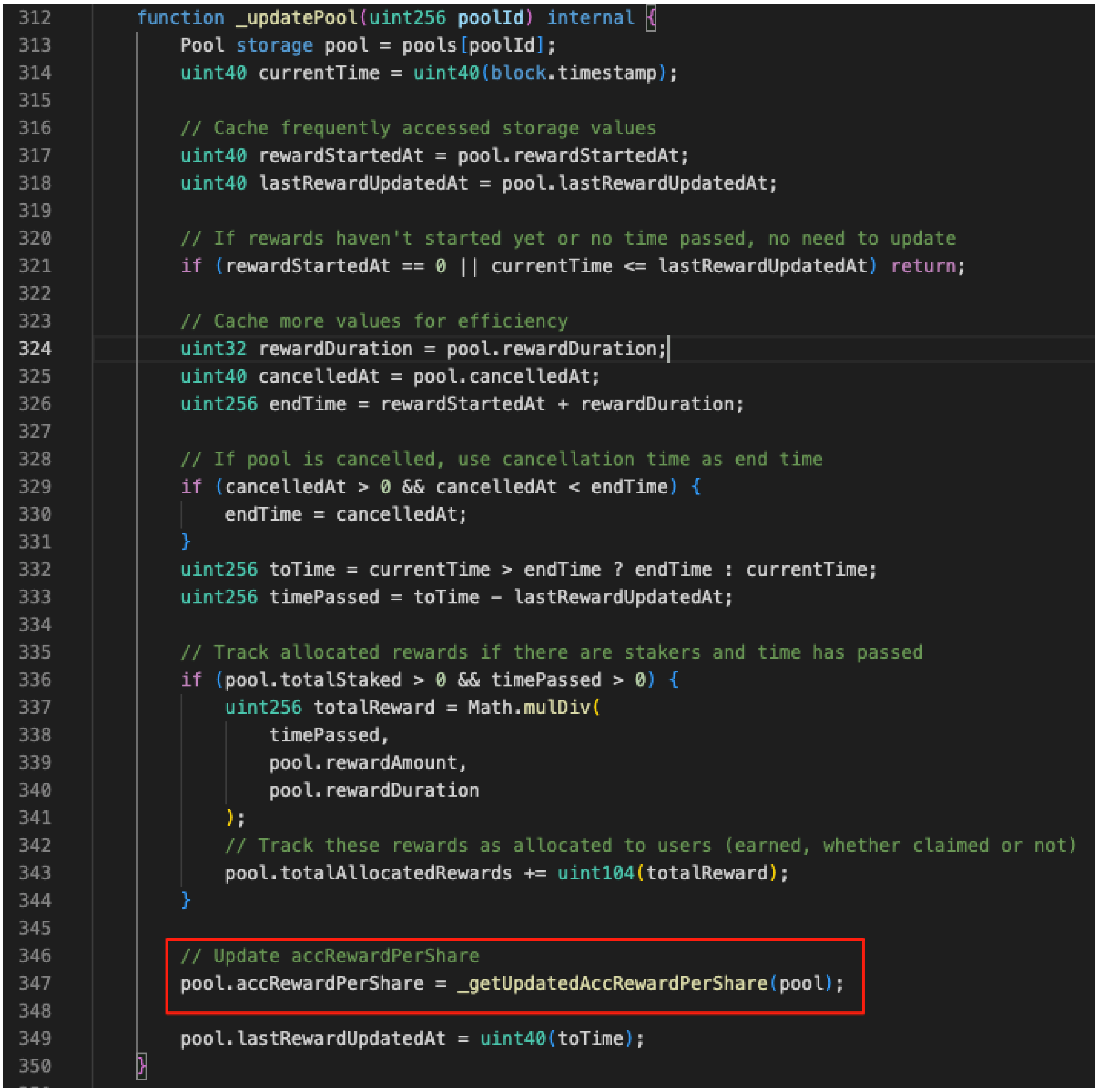
Timeline

- 07/28/2025: The team provided the updated contracts for us to review:
    - https://github.com/Steemhunt/mint.club-v2-contract/blob/3c98fa7fe649c641bbec91edd3728f57592c1ccf/contracts/Stake.sol
- 07/30/2025: The addendum preliminary report was delivered.
- 08/01/2025: The team fixed the issues in the following commits:
    - 08/01-bc4268f
    - 08/04-9c7f5d
    - 08/06-1f1298
    - 08/07-e935a1
- 08/07/2025: The final report was delivered.
- 08/25/2025: The team reported the issue related to precision loss.

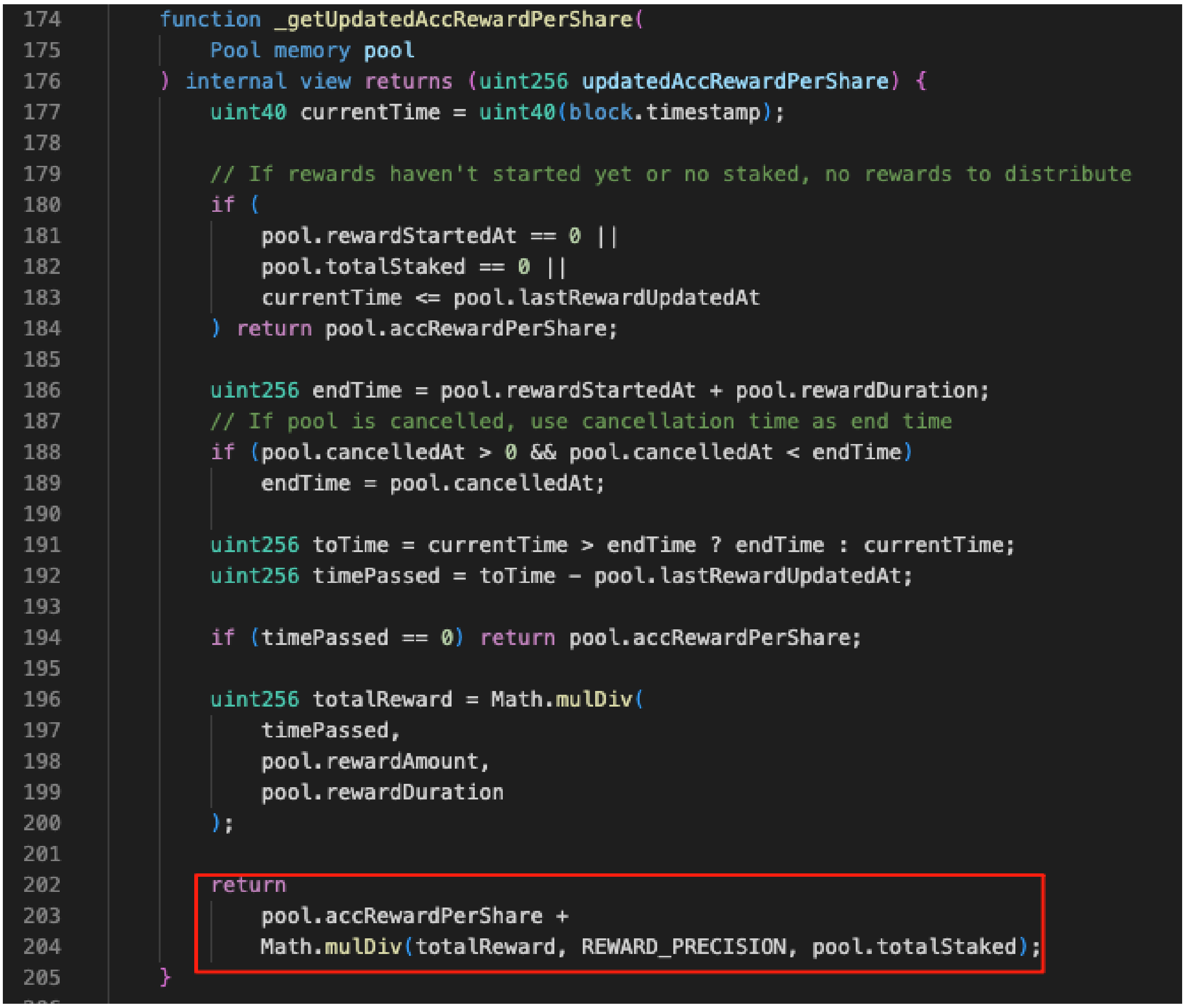
Staking Issue

Description

In the Stake contract, the `accRewardPerShare` value is calculated with a multiplier of `1e18`, which is insufficient to handle the precision requirements of staking pools with reward tokens that have lower decimal precision (e.g., 6 decimals for USDC). This precision loss results in `accRewardPerShare` being rounded down to zero while total rewards continue to accumulate whenever users staked, unstaked, or claimed. As a result, reward tokens are never distributed and remain stuck in the contract.



```
    function _updatePool(uint256 poolId) internal {
        Pool storage pool = pools[poolId];
        uint40 currentTime = uint40(block.timestamp);

        // Cache frequently accessed storage values
        uint40 rewardStartedAt = pool.rewardStartedAt;
        uint40 lastRewardUpdatedAt = pool.lastRewardUpdatedAt;

        // If rewards haven't started yet or no time passed, no need to update
        if (rewardStartedAt == 0 || currentTime <= lastRewardUpdatedAt) return;

        // Cache more values for efficiency
        uint32 rewardDuration = pool.rewardDuration;
        uint40 cancelledAt = pool.cancelledAt;
        uint256 endTime = rewardStartedAt + rewardDuration;

        // If pool is cancelled, use cancellation time as end time
        if (cancelledAt > 0 && cancelledAt < endTime) {
            endTime = cancelledAt;
        }
        uint256 toTime = currentTime > endTime ? endTime : currentTime;
        uint256 timePassed = toTime - lastRewardUpdatedAt;

        // Track allocated rewards if there are stakers and time has passed
        if (pool.totalStaked > 0 && timePassed > 0) {
            uint256 totalReward = Math.mulDiv(
                timePassed,
                pool.rewardAmount,
                pool.rewardDuration
            );
            // Track these rewards as allocated to users (earned, whether claimed or not)
            pool.totalAllocatedRewards += uint104(totalReward);
        }

        // Update accRewardPerShare
        pool.accRewardPerShare = _getUpdatedAccRewardPerShare(pool);

        pool.lastRewardUpdatedAt = uint40(toTime);
    }
```



```
function _getUpdatedAccRewardPerShare(
    Pool memory pool
) internal view returns (uint256 updatedAccRewardPerShare) {
    uint40 currentTime = uint40(block.timestamp);

    // If rewards haven't started yet or no staked, no rewards to distribute
    if (
        pool.rewardStartedAt == 0 ||
        pool.totalStaked == 0 ||
        currentTime <= pool.lastRewardUpdatedAt
    ) return pool.accRewardPerShare;

    uint256 endTime = pool.rewardStartedAt + pool.rewardDuration;
    // If pool is cancelled, use cancellation time as end time
    if (pool.cancelledAt > 0 && pool.cancelledAt < endTime)
        endTime = pool.cancelledAt;

    uint256 toTime = currentTime > endTime ? endTime : currentTime;
    uint256 timePassed = toTime - pool.lastRewardUpdatedAt;

    if (timePassed == 0) return pool.accRewardPerShare;

    uint256 totalReward = Math.mulDiv(
        timePassed,
        pool.rewardAmount,
        pool.rewardDuration
    );

    return
        pool.accRewardPerShare +
        Math.mulDiv(totalReward, REWARD_PRECISION, pool.totalStaked);
}
```

When the `pool.totalStaked` amount becomes extremely large, it can result in `accRewardPerShare` no longer increasing, while `totalAllocatedRewards` continues to accumulate. This leads to a situation where these rewards cannot be distributed and become locked in the contract.

Impact

This issue primarily affects pools where the reward token has lower decimal precision (e.g., USDC with 6 decimals) and the staking token has a higher precision (e.g., 18 decimals). In the Stake contract, there are 3 pools with USDC (6 decimals) and 1 pool with WBTC (8 decimals) as reward tokens. These pools could be affected due to precision loss.

Root Cause

The current `REWARD_PRECISION` with a value of `1e18` is not sufficient to handle the precision mismatch between reward tokens and staking tokens with differing decimal places.

Client's Hotfix

The MintClub team proposed and implemented a hotfix:

- Increasing the multiplier in the `accRewardPerShare` calculation from `1e18` to `1e30`.
- This change significantly reduces the precision loss, ensuring that rewards are distributed properly in most normal cases, particularly for pools where the reward token is USDC (6 decimals) and the staking token is 18 decimals.

Limitations of the Hotfix

While the fix mitigates the issue in most scenarios, it does not completely eliminate the risk of precision loss:

- The current design does not impose restrictions on the `StakingToken`. If the `StakingAmount` is extremely large, precision loss may still occur.
- However, this fix resolves the issue for the majority of real-world use cases.

Recommendations

- Cancel the affected pools on the Base chain to prevent further complications.
- Consider implementing additional safeguards in the staking contract to handle extreme staking amounts or mismatched token precisions more robustly.