Satoshi Farm Security Review

Reviewer
billh (@hibillh)

Draft v1: Jan 29, 2025
Draft v2: Feb 04, 2025
Final Version: February 12, 2025

1 Executive Summary

Over the course of 4 days in total, Satoshi Protocol engaged with billh to review Satoshi Farm.

A total of 6 issues have been found with Satoshi Farm.

Repository	Commit
satoshi-farm	1c3e5c2adc5dbbb79ee96af440742382897f2f28

Summary

Type of Project	CDP Protocol
Timeline	Jan 23, 2025 - Jan 29, 2025
Methods	Manual Review

Total Issues

High Risk	1
Medium Risk	0
Low Risk	2
Informational	3

Contents

Disclaimer: This security review is not a guarantee against hacking. This is the result of a review within a time range based on specific commits.

2 Findings

2.1 High Risk

2.1.1 `computeInterval` Underflow Causing Reverts Post `rewardEndTime`

Severity: High

Context: FarmMath.sol#L102-L104

```
    function computeInterval(
        uint256 currentTime,
        uint256 lastUpdateTime,
        uint256 startTime,
        uint256 endTime
    )
        internal
        pure
        returns (uint256)
    {
        if (currentTime < startTime) {
            return 0;
        }
        if (currentTime > endTime) {
            return endTime - Math.max(lastUpdateTime, startTime);
        }
        return currentTime - Math.max(lastUpdateTime, startTime);
    }
```

Description:

In `computeInterval`, the line `endTime - Math.max(lastUpdateTime, startTime)` can underflow if `lastUpdateTime` exceeds `endTime`. Practically, if any update or reward calculation occurs after `rewardEndTime`, and `lastUpdateTime` is beyond `endTime`, calls to this function will revert instead of returning zero.

As a result, user actions that rely on `FarmMath.computeInterval` (such as withdraw, or any operation that calls reward logic) may fail unexpectedly once `rewardEndTime` has passed, leading to locked funds or an inability to complete certain transactions.

Recommendation:

Returns zero if `lastUpdateTime` is already beyond `endTime`.

Status:

Fixed.

2.2 Low Risk

2.2.1 Potential Collision of `claimId` Can Cause Unexpected Revert

Severity: Low

Context: Farm.sol#L492-L495

```
        bytes32 claimId = keccak256(abi.encode(amount, owner, receiver,
        ↪  claimableTime));
        ClaimStatus claimStatus = _claimStatus[claimId];
        if (claimStatus != ClaimStatus.NONE) revert
        ↪  InvalidStatusToRequestClaim(claimStatus);
```

Description: The `claimId` is generated via `keccak256(abi.encode(amount, owner, receiver, claimableTime))`. If the same user calls the function multiple times in the same block with the same parameters (for example, a user might want to have two claims with the same amount for `executeClaim` and `forceExecuteClaim`), it would produce the identical `claimId`, causing collisions and reverting with `InvalidStatusToRequestClaim`. Another potential scenario is when `claimDelayTime` is updated after a claim is made, two identical `claimableTime` and `claimId` might be generated in two different blocks.

Recommendation:

Append an additional nonce to differentiate claim requests.

Status:

Fixed by adding nonce.

2.2.2 Missing Checks in `forceExecuteClaim` (Ignoring `isClaimable()` and `isForceClaimEnabled()`)

Severity: Low

Context: Farm.sol#L574-L589

```
    function _beforeForceExecuteClaim(
        uint256 amount,
        address owner,
        address receiver,
        uint256 claimableTime,
        bytes32 claimId
    )
        internal
        view
    {
        _checkClaimId(amount, owner, receiver, claimableTime, claimId);

        ClaimStatus claimStatus = _claimStatus[claimId];

        if (claimStatus != ClaimStatus.PENDING) revert
        ↪  InvalidStatusToForceExecuteClaim(claimStatus);
    }
```

Description: Within the `forceExecuteClaim` process, `_beforeForceExecuteClaim` does not call `_checkIsClaimable()` and `_checkIsForceClaimEnabled()`. This means users can still force-execute a claim even when the time window has passed or if the system has disabled force-claiming.

This causes inconsistencies in the protocol as `forceClaim` checks both conditions. If `_checkIsClaimable` is not needed, then `forceClaim` should not enforce this restriction as well. If `_checkIsForceClaimEnabled` is not added, users can bypass this check while achieving `forceClaim` by calling `requestClaim` and `forceExecuteClaim` instead, which basically has the same effect.

Recommendation: In `_beforeForceExecuteClaim`, also invoke:

```
_checkIsClaimable();
_checkIsForceClaimEnabled();
```

to maintain consistency with `forceClaim`.

Status:

Fixed.

2.3 Informational

2.3.1 `depositCapPerUser` Might Be Redundant or Conflicting

Severity: Informational

Context: Farm.sol#L381-L393

```
    function _beforeDeposit(uint256 amount, address, address receiver) internal
    ↪  {
        if (amount == 0) revert InvalidZeroAmount();

        if (_totalShares + amount > farmConfig.depositCap) revert
        ↪  DepositCapExceeded(amount, farmConfig.depositCap);

        if (_shares[receiver] + amount > farmConfig.depositCapPerUser) {
            revert DepositCapPerUserExceeded(amount,
            ↪  farmConfig.depositCapPerUser);
        }

        _checkIsDepositEnabled();

        _updateReward(receiver);
    }
```

Description: The contract can enforce a per-user deposit cap (`depositCapPerUser`). However, this can be easily bypassed by utilizing multiple receiver addresses. Furthermore, when combined with whitelisting, since the whitelist check is enforced on the depositor address, a whitelisted user can make multiple deposits to different receivers to have a total deposit beyond the cap.

Recommendation:

- Keep the logic but verify it aligns with business requirements.
- Otherwise, simplify by removing whichever mechanism is not absolutely necessary.

Status:

Acknowledged.

2.3.2 `recoverNativeAsset` Using `transfer(...)` Could Fail for Contract Owner

Severity: Informational

Context: FarmManager.sol#L189

```
payable(owner()).transfer(amount);
```

Description: Using `.transfer` imposes a 2300 gas limit on the receiver's fallback function. If `owner()` is a multisig or another smart contract, the fallback/receive might need more than 2300 gas, causing a revert.

Recommendation: Switch to `.call{ value: amount }("")` with a proper success check. If certain the owner is always an EOA, the current setup is acceptable, but `.call` is more flexible.

Status:

Fixed.

2.3.3 Including `block.timestamp` in Events Might Be Redundant

Severity: Informational

Context: Farm.sol#L329

```
revert InvalidDepositTime(block.timestamp, farmConfig.depositStartTime,
↪  farmConfig.depositEndTime);
```

Description: There are multiple events that include a `block.timestamp` field. Because events structure naturally tie to the block's timestamp, explicitly embedding `block.timestamp` in the event is somewhat redundant.

Recommendation:

Omit the explicit `block.timestamp` if it's not strictly required. The block's timestamp can be retrieved from the chain context anyway.

Status:

Fixed.