Think Staking Security Review

Reviewed by: windhustler, ether_sky, 0x37, 0xGojoArc

8th August - 11th August, 2025

Think Staking Review Report

August 12, 2023

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any vulnerabilities. Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

About Burra Security

Burra Sec offers security auditing and advisory services with a special focus on cross-chain and interoperability protocols and their integrations.

About THINK Token Staking System

The focus of the security review were the `StakingVault` and `StakingStorage` contracts.

StakingVault: The main contract that handles ERC20 token staking and unstaking with timelock periods and access controls.

StakingStorage: The data storage contract that tracks all stake records, user balances, and historical snapshots for the staking system.

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - The technical, economic, and reputation damage from a successful attack

Likelihood - The chance that a particular vulnerability gets discovered and exploited

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Security Assessment Summary

review commit hash **- baa506653779e2f44805397a10ca28b31e961f12**

Scope

The following smart contracts were in the scope of the audit:

- src/StakingStorage.sol
- src/StakingVault.sol

Findings Summary

ID	Title	Severity	Status
H-01	Missing `daysLock` restriction enables infinite reward farming	High	-

ID	Title	Severity	Status
M-01	Potential `totalStakesCount` inflation via micro stakes	Medium	-
M-02	Daily lock time doesn’t reflect a full day of staking	Medium	-
L-01	The `stakesCounter` overflow risk in `StakerInfo`	Low	-
L-02	Permanent `_allStakers` growth without removal	Low	-
I-01	The `batchUnstake` reverts entire operation on invalid stake ID Instead of skipping	Info	-
I-02	Redundant condition in `isActiveStake`	Info	-
I-03	The `getStakersPaginated` should return empty array instead of reverting when no results	Info	-
I-04	Remove redundant imports	Info	-
I-05	Missing sanity check on `batchUnstake` input size	Info	-
I-06	Duplicate staked and unstaked event emissions for a single action	Info	-
I-07	The `getStakerStakeIds` returns all generated IDs, including non-existent or unstaked ones	Info	-
I-08	Inefficient checkpoint search when target day is current day	Info	-
I-09	Remove unused fields from the `StakerInfo` struct	Info	-
I-10	`lastCheckpointDay` recorded twice during staking	Info	-

Detailed Findings

[H-01] Missing `daysLock` restriction enables infinite reward farming

Target

- StakingVault.sol

Severity

- Impact: High
- Likelihood: High

Description

In `StandardStakingStrategy.sol`, and `SimpleUserClaimableStrategy.sol`, when `calculateReward`, the effective staking period is calculated like this, StandardStakingStrategy.sol#L70

```
function calculateReward(
    address user,
    IStakingStorage.Stake memory stake,
    uint256 startDay,
    uint256 endDay
) external view override returns (uint256) {

    if (stake.unstakeDay != 0 && stake.unstakeDay < endDay) {
        // User withdrew during the pool.
        // We need to check the re-staking policy.
        if (!isReStakingAllowed) {
            return 0; // Not eligible if re-staking is disallowed and they withdrew.
        }
        // If re-staking is allowed, the weight is calculated only for the active period.
@>      endDay = stake.unstakeDay;
    }

    if (stake.stakeDay > endDay || startDay > endDay) {
        return 0; // Stake is not active within the calculation period.
    }

    uint256 effectiveStart = stake.stakeDay > startDay
        ? stake.stakeDay
        : startDay;
@>  uint256 effectiveDays = endDay - effectiveStart + 1;

    // The final weight is the stake amount multiplied by the number of days it was active in the pool.
    return stake.amount * effectiveDays;
}
```

And note that there is no restriction on `dayLock` when create stake, StakingVault.sol#L95.

```
function stake(
    uint128 amount,
    uint16 daysLock
) external whenNotPaused nonReentrant returns (bytes32 stakeId) {
    require(amount > 0, InvalidAmount());

    address staker = msg.sender;

    // Transfer tokens
    token.safeTransferFrom(staker, address(this), amount);

    // Create stake in storage and get the generated ID
    stakeId = stakingStorage.createStake(
        staker,
        amount,
@>      daysLock,
        EMPTY_FLAGS
    );
    ...
}
```

This means that even if a stake is created and immediately unstaked on the same day, it is counted as **1 effective day** toward rewards.

Because there is **no restriction on `daysLock`** during stake creation, a malicious user can:

1. Stake a large amount with `daysLock = 0`.
2. Unstake immediately the same day.
3. Receive rewards for 1 effective day.
4. Repeat this process multiple times, claiming rewards repeatedly and disproportionately.

This design enables infinite reward farming without any real lockup or risk.

Recommendation

Introduce a **minimum `daysLock` requirement** to prevent instant stake/unstake abuse.

[M-01] Potential `totalStakesCount` inflation via micro stakes

Target

- StakingStorage.sol

Severity

- Impact: Medium
- Likelihood: Medium

Description

The `DailySnapshot` IStakingStorage.sol#L31-L34 struct tracks the number of stakes and total staked amount.

```
struct DailySnapshot {
    uint128 totalStakedAmount;
    uint16 totalStakesCount;
}
```

When creating or removing a stake, `totalStakesCount` is updated in an `unchecked` block, StakingStorage.sol#L302.

```
unchecked {
    if (deltaSign == Sign.POSITIVE) {
@>      snapshot.totalStakesCount++;
        snapshot.totalStakedAmount += deltaAmount;
        _currentTotalStaked += deltaAmount;
    } else {
        snapshot.totalStakesCount--;
        snapshot.totalStakedAmount = _safeSubtract(snapshot.
           totalStakedAmount, deltaAmount);
        _currentTotalStaked = _safeSubtract(_currentTotalStaked,
           deltaAmount);
    }
}
```

However:

- `totalStakesCount` is a `uint16` (max 65,535),
- There is **no minimum stake amount** restriction,

a malicious user can create **thousands of micro-stakes** (e.g., 1 wei each) in a single transaction. Once `totalStakesCount` exceeds 65,535, it **wraps to zero** and continues counting from there without reverting.

This can result in:

- **Artificially inflated `totalStakesCount`**, breaking analytics or UI displays.
- Potential downstream logic errors if other functions rely on an accurate count.

Recommendation

- Require a minimum stake value to discourage micro-spam stakes.
- Consider using uint32 for `totalStakesCount`

Client

BurraSec

[M-02] Daily lock time doesn't reflect a full day of staking

Target

- StakingVault.sol#L124-L129

Severity

- Impact: High
- Likelihood: Low

Description

The staking system uses a day-based calculation (`block.timestamp / 1 days`) to determine stake maturity, which creates a vulnerability where users can stake and unstake tokens within seconds while the system considers it a full day of staking.

The issue occurs in the `StakingVault::unstake` function where the maturity check compares the current day with the stake day plus lock duration. Since days are calculated as `block.timestamp / 1 days`, a user can stake near the end of one day and unstake at the beginning of the next day.

Proof of Concept

1. User calls `StakingVault::stake` at 23:59:59 on day n with `daysLock = 1`
2. The stake is recorded with `stakeDay = n` and `daysLock = 1`
3. A few seconds later (00:00:01 on day n+1), user calls `StakingVault::unstake`
4. The maturity check calculates:
   - `currentDay = n + 1`
   - `matureDay = n + 1` (stakeDay + daysLock)
   - Since `currentDay >= matureDay`, the stake is considered mature
5. User successfully unstakes after locking tokens for only seconds instead of a full day

```
## StakingVault.sol

function unstake(bytes32 stakeId) public whenNotPaused nonReentrant {
    // ...
    uint16 currentDay = _getCurrentDay();
    uint16 matureDay = _stake.stakeDay + _stake.daysLock;
    require(
        currentDay >= matureDay,
        StakeNotMatured(stakeId, currentDay, matureDay)
    );
    // ...
}

function _getCurrentDay() internal view returns (uint16) {
    return uint16(block.timestamp / 1 days);
}
```

This vulnerability is particularly concerning for reward systems that may calculate rewards based on stake duration, as they could be manipulated to claim rewards for minimal actual staking time.

Recommendation

Implement a minimum lock duration based on actual time elapsed rather than day boundaries.

Client

BurraSec

[L-01] The `stakesCounter` overflow risk in `StakerInfo`

Target

- StakingStorage.sol

Severity

- Impact: High
- Likelihood: Low

Description

The `StakerInfo` struct contains a `stakesCounter` field stored as a `uint16`, IStakingStorage.sol#L22-L29.

```
struct StakerInfo {
    uint128 totalStaked;
    uint128 totalRewarded;
    uint128 totalClaimed;
@>  uint16 stakesCounter;
    uint16 activeStakesNumber;
    uint16 lastCheckpointDay;
}
```

`stakesCounter` is incremented in `createStake` but **never decremented**, StakingStorage.sol#L90:

```
_stakerInfo.stakesCounter++;
```

Because `uint16` has a maximum value of **65,535**, once a staker creates 65,536 stakes over their lifetime, the **next stake attempt will revert**.

Additionally, **there is no limit for `daysLock`**, so users can call `createStake` multiple times in the same day, accelerating the counter's growth.

This means a single address can permanently lock itself from staking after enough stake operations - even if all stakes have been **unstaked**.

Recommendation

- Use `uint32` for `stakesCounter`
- If `uint16` is intentional for gas savings, clearly state in the documentation that each address can have a maximum of 65,535 stakes over the lifetime of the contract.

Client

BurraSec

[L-02] Permanent `_allStakers` growth without removal

Target

- StakingStorage.sol

Severity

- Impact: Low
- Likelihood: Medium

Description

The `_allStakers` set is updated in `createStake` to include the `staker` address, StakingStorage.sol#L81.

```
if (_allStakers.add(staker)) {
    _stakers[staker].lastCheckpointDay = today;
}
```

However, **there is no corresponding removal** in `removeStake`.
This means an address is permanently recorded in `_allStakers` after creating a single stake, even if they later have **zero active stakes**.

A malicious or careless user can exploit this by - Staking and unstaking immediately with **multiple addresses** (bots or new wallet addresses). - Permanently inflating `_allStakers` with inactive or one-time users.

This impacts:

1. **`getStakersPaginated`** - The returned list will include stale, inactive addresses, making pagination less meaningful and more gas-heavy for off-chain processing.
2. **`getTotalStakersCount`** - Will over-report "active" staker count, misrepresenting protocol usage.

Recommendation

Remove inactive stakers in `removeStake` when `activeStakesNumber` reaches `0`.

```
function removeStake(
    address staker,
    bytes32 id
) external onlyRole(CONTROLLER_ROLE) {
    Stake storage stake = _stakes[id];
    ...
    // Update staker info
    _stakers[staker].totalStaked -= amount;
    _stakers[staker].activeStakesNumber--;

+   if (_stakers[staker].activeStakesNumber == 0) {
```

```
+       _allStakers.remove(staker);
+   }
    ...
}
```

Client

BurraSec

[I-01] The `batchUnstake` reverts entire operation on invalid stake ID Instead of skipping

Target

- StakingVault.sol

Description

The current `batchUnstake` implementation calls `unstake` in a loop, StakingVault.sol#L144-L148.

```
function batchUnstake(bytes32[] calldata stakeIds) external {
    for (uint256 i = 0; i < stakeIds.length; i++) {
        unstake(stakeIds[i]);
    }
}
```

If **any** `unstake(stakeId)` call reverts (e.g., due to invalid ID, already unstaked stake, or not yet matured), **the entire transaction reverts** and **none of the stakes are unstaked** even if there are valid IDs.

This means:

- A single bad element invalidates the whole batch.
- Users pay for reverted gas costs without achieving any partial progress.

Recommendation

Implement fault-tolerant batch processing using **`try`**/**`catch`**

```
function batchUnstake(bytes32[] calldata stakeIds) external {
    for (uint256 i = 0; i < stakeIds.length; i++) {
-       unstake(stakeIds[i]);
+       try this.unstake(stakeIds[i]) {
+       } catch {
+           // emit failure for invalid id, skip to next
+       }
    }
}
```

[I-02] Redundant condition in `isActiveStake`

Target

- StakingStorage.sol

Description

The condition `stake.amount > 0` in the return statement is redundant because you already check it with `require` just before, StakingStorage.sol#L153.

```
function isActiveStake(bytes32 id) external view returns (bool) {
    Stake memory stake = _stakes[id];
    require(stake.amount > 0, StakeNotFound(id));
@>  return stake.amount > 0 && stake.unstakeDay == 0;
}
```

Removing it simplifies code and avoids slight gas cost for the redundant check.

Recommendation

```
function isActiveStake(bytes32 id) external view returns (bool) {
    Stake memory stake = _stakes[id];
    require(stake.amount > 0, StakeNotFound(id));
-   return stake.amount > 0 && stake.unstakeDay == 0;
+   return stake.unstakeDay == 0;
}
```

[I-03] The `getStakersPaginated` should return empty array instead of reverting when no results

Target

- StakingStorage.sol

Description

`getStakersPaginated` reverts if `offset >= total`. If there are no stakers (`total == 0`) and caller passes `offset == 0`, `offset < total` fails and reverts either. For frontend pagination, it is preferable to return an empty array instead of reverting to improve UX, StakingStorage.sol#L207. For frontend pagination, it is preferable to return an empty array instead of reverting to improve UX, StakingStorage.sol#L207.

```
function getStakersPaginated(
    uint256 offset,
    uint256 limit
) external view returns (address[] memory) {
    uint256 total = _allStakers.length();
@>  require(offset < total, OutOfBounds(total, offset));

    uint256 end = offset + limit;
    if (end > total) end = total;

    address[] memory result = new address[](end - offset);
    for (uint256 i = offset; i < end; i++) {
        result[i - offset] = _allStakers.at(i);
    }
    return result;
}
```

Recommendation

Return an empty array if `offset >= total` instead of reverting.

[I-04] Remove redundant imports

Target

- StakingVault.sol

Description

`import "./StakingFlags.sol";` is duplicated. This is harmless but unnecessary, StakingVault.sol#L14

```
import "./StakingFlags.sol";
import "./StakingFlags.sol";
```

There's an `import "forge-std/console.sol";` inside the `StakingStorage.sol` that should also be removed.

Recommendation

Remove redundant imports.

[I-05] Missing sanity check on `batchUnstake` input size

Target

- StakingVault.sol

Description

The `batchUnstake` function processes an array of stake IDs in a loop without validating the input array length, StakingVault.sol#L144-L149.

```
function batchUnstake(bytes32[] calldata stakeIds) external {
    for (uint256 i = 0; i < stakeIds.length; i++) {
        unstake(stakeIds[i]);
    }
}
```

If called with an empty array (`length == 0`) or an **excessively large** array, it can lead to: - Wasted gas on no-op calls (empty array). - Gas exhaustion or block gas limit exceeded errors (too large arrays), causing DoS.

Recommendation

Add a sanity check on the input array size.

```
function batchUnstake(bytes32[] calldata stakeIds) external {
+   require(stakeIds.length > 0 && stakeIds.length <= MAX_BATCH_UNSTAKE, "InvalidBatchSize");
    for (uint256 i = 0; i < stakeIds.length; i++) {
        unstake(stakeIds[i]);
    }
}
```

[I-06] Duplicate staked and unstaked event emissions for a single action

Target

- StakingVault.sol
- StakingStorage.sol

Description

When a stake is created through the `stake` function, it internally calls `createStake`, which also emits a `Staked` event.

This results in **two `Staked` events** for the same stake action:

1. **First emission** — inside `createStake` (storage layer).
2. **Second emission** — inside `stake` (user-facing function).

StakingVault.sol#L99

```
function stake(
    uint128 amount,
    uint16 daysLock
) external whenNotPaused nonReentrant returns (bytes32 stakeId) {
    ...
    stakeId = stakingStorage.createStake(
        staker,
        amount,
        daysLock,
        EMPTY_FLAGS
    );

@>  emit Staked(
        staker,
        stakeId,
        amount,
        _getCurrentDay(),
        daysLock,
        EMPTY_FLAGS
    );
}
```

StakingStorage.sol#L97

```
function createStake(
    address staker,
    uint128 amount,
    uint16 daysLock,
    uint16 flags
) external onlyRole(CONTROLLER_ROLE) returns (bytes32 id) {
    ...

@>  emit Staked(staker, id, amount, today, daysLock, flags);
}
```

Event logs contain duplicate entries for the same action, potentially confusing off-chain services, analytics tools, or indexers.

This issue is same for `unstake`.

Recommendation

Emit the `Staked` event **only once** per stake action.

[I-07] The `getStakerStakeIds` returns all generated IDs, including non-existent or unstaked ones

Target

- StakingStorage.sol

Description

`getStakerStakeIds` returns generated ids irrespective of existence, but many may be `unstaked` or not exist, StakingStorage.sol#L100.

```
function getStakerStakeIds(
    address staker
) external view returns (bytes32[] memory) {
    uint32 counter = _stakers[staker].stakesCounter;
    bytes32[] memory stakeIds = new bytes32[](counter);

    for (uint32 i = 0; i < counter; i++) {
        stakeIds[i] = _generateStakeId(staker, i);
    }

    return stakeIds;
}
```

This may confuse off-chain clients expecting only active stakes.

Recommendation

Rename to `getAllGeneratedStakeIds` or document semantics.

Client

BurraSec

[I-08] Inefficient checkpoint search when target day is current day

Target

- StakingStorage.sol

Description

The `_getStakerBalanceAt` function determines a `staker`'s balance at a given `targetDay` using binary search.

However, in all calls from `stake` and `unstake`, `targetDay` is **always the current day**, meaning:

- The most recent checkpoint is guaranteed to be the correct reference point.
- A binary search over historical checkpoints is unnecessary in these cases.

This results in redundant computation and higher gas costs for `stake`/`unstake` transactions without any functional benefit.

Recommendation

```
function _getStakerBalanceAt(
    address staker,
    uint16 targetDay
) internal view returns (uint128) {
    // checkpoints are days when staker had a stake
    uint16[] memory checkpoints = _stakerCheckpoints[staker];
    uint256 nCheckpoints = checkpoints.length;

    // Return 0 if no checkpoints exist
    if (nCheckpoints == 0) return 0;

    // Quick exact match check
    uint128 exactBalance = _stakerBalances[staker][targetDay];
    if (exactBalance > 0) return exactBalance;

    // Handle edge case: target is before first checkpoint
    if (checkpoints[0] > targetDay) return 0;

+   if (targetDay >= checkpoints[nCheckpoints - 1]) return _stakerBalances[staker][checkpoints[nCheckpoints - 1]];

      ...
}
```

Client

BurraSec

[I-09] Remove unused fields from the `StakerInfo` struct

Target

- IStakingStorage.sol#L24-L25

Description

`StakerInfo` has the `totalRewarded` and `totalClaimed` fields that aren’t used.

Recommendation

Consider removing the two fields from the `StakerInfo` struct.

[I-10] `lastCheckpointDay` recorded twice during staking

Target

- StakingStorage.sol#L83

Description

While calling `StakingStorage::createStake` the `_stakers[staker].lastCheckpointDay = today;` gets recorded if the staker is new, and it's also always recorded inside the `_updateStakerCheckpoint` function.

Recommendation

Remove the update of `lastCheckpointDay` that's inside the stake function.

```diff
-        // Add staker to the set. If the staker is new, the .add function returns true.
-        if (_allStakers.add(staker)) {
-            // New staker, set their initial checkpoint day.
-            _stakers[staker].lastCheckpointDay = today; // We set the checkpoint day to the current day.
-        }
+        _allStakers.add(staker);
```