Security Audit Report

2025 Q1 - Stride stBGT Updates

Authors: Darko Deuric, Marko Juric

Last revised 7 February, 2025

Table of Contents

Audit Overview

The Project

Stride engaged Informal Systems between January 27 and February 7, 2025, to conduct a security audit of the stBGT updates. These updates were driven by a modification in Berachain's unbonding time, which introduced delayed dropBoost functionality. Previously, dropBoost was instant, but the new implementation requires it to be queued and executed after 8191 blocks, similar to staking / boosting.

This change had effects on the stBGT protocol, particularly in the following areas:

- User redemptions (queueRedemption and completeRedemption)
- Processing boost cycles (processBoostCycle)
- Validator rebalancing (setValidators, queueRebalancing, and completeRebalancing)

The audit focused on ensuring that these modifications did not introduce inconsistencies, vulnerabilities, or logical flaws in the system.

Scope of this Review

The primary focus of this audit was to evaluate:

1. New Accounting State
    - The audit closely examined changes to accounting-related state variables (some of those including boostPeriodRewards, boostPeriodRedemptions, redemptionsProcessing, redemptionsFinished), ensuring they were correctly manipulated:
    - Special attention was given to edge cases introduced by setValidators / rebalancing.
2. Threat Modeling & Invariants
    - We developed a **threat model** and formulated properties and invariants to ensure the system behaves correctly under different execution scenarios.
    - The analysis focused on redemption timing, validator rebalancing, and processBoostCycle consistency.
3. Static Analysis
    - Slither and Aderyn were used to analyze the contracts for potential vulnerabilities, gas optimizations, and dead code.
    - These tools did not find any major issues but identified a few informational findings.
4. Impact of Rebalancing on Boost Cycles & Redemptions
    - Since rebalancing resets queue states and prevents boost cycles from executing, we evaluated whether global accounting remained consistent when rebalancing was invoked.
5. Code Coverage Analysis
    - The test coverage for the StrideManager, StrideEngine, and StBGTStaking contracts was excellent, ensuring confidence in the correctness of the implementation.
    - The LCOV coverage report (see image below) further validated the extensive test coverage.

Filename	Line Coverage			Functions	
StBGTStaking.sol		84.8 %	28 / 33	81.8 %	9 / 11
StrideEngine.sol		99.5 %	182 / 183	95.5 %	21 / 22
StrideManager.sol		97.6 %	201 / 206	94.3 %	33 / 35

Conclusions

Overall, the stBGT updates successfully adapted to Berachain's new unbonding model without introducing security risks or breaking core functionality.

- No major vulnerabilities were found in the implementation.
- Global state consistency was maintained, even in complex interactions involving rebalancing, redemptions, and boost cycles.
- Static analysis and threat modeling confirmed correctness, with only minor gas optimization suggestions.

- Test coverage was high, confirming the reliability of the changes.

System Overview

The stBGT protocol is designed to facilitate automated staking, claiming, and redemption of BGT while integrating with Berachain's modified dropBoost mechanics. The protocol consists of three core contracts:

- **StrideManager**: Handles user-facing functions such as deposits, withdrawals, redemptions, and claims. It interacts with StrideEngine for staking and rebalancing operations.
- **StrideEngine**: Manages validator delegation, boosting, unboosting, process boost cycles, and rebalancing. All global accounting variables and validator state are stored here, while user-specific state remains in the manager.
- **StBGTStaking**: Allows users to stake stBGT to earn HONEY rewards, which are collected from protocol fees.

How It Works

Depositing and Staking

- Users deposit LP tokens (receipt tokens) via the StrideManager.
- If it is the user's first deposit for a given vault, a dedicated agent contract is deployed.
- The LP tokens are transferred to the manager and staked in a vault to generate BGT rewards.

Claiming and Minting stBGT

- Users claim BGT rewards through the StrideManager.
- The manager mints stBGT at a 1:1 ratio to the claimed BGT.
- The StrideEngine queues and later activates BGT boosts across validators.

Unstaking and Withdrawing LP Tokens

- Users can unstake and withdraw LP tokens at any time, which stops stBGT rewards from accruing.

Redeeming stBGT for BERA

- Users burn stBGT to redeem BGT, which is then converted into BERA.
- Since dropBoost now requires queuing before execution, redemptions occur in two steps:
  - **queueRedemption**: Transfers stBGT to the StrideManager and updates redemption tracking.
  - **completeRedemption**: After 1-2 cycles (without rebalancing in between), the system finalizes dropBoost, burns stBGT, and redeems BERA for the user.

StrideManager and StrideEngine Architecture

Contract Structure

- StrideManager handles all user interactions and retains custody of stBGT and receipt tokens.
- StrideEngine manages boosting, unboosting, process boost cycles, validator delegation, and redemptions.
- The engine never directly interacts with users; all interactions flow through the manager.
- The engine receives BGT from the manager and manages all staking logic.
- During redemptions, the engine executes dropBoost and issues back BERA once the unbonding period completes.

Initialization Flow

1. StrideEngine is deployed and initialized first.

2. StrideManager is deployed with the engine’s address passed as a parameter.
3. StrideManager calls `setManager` in StrideEngine, finalizing the setup.
4. The engine cannot be used until `setManager` is called, ensuring proper initialization.

Process Boost Cycle and Rebalancing

- processBoostCycle runs every ~5 hours ~ 8191 blocks = 1 cycle, finalizing queued (drop) boosts, queuing new (drop) boosts, and distributing HONEY.
- The transition from instant dropBoost to a queued system means that rebalancing must now wait for drops to complete before reassigning stake.
- Occasionally, admin-triggered rebalancing redistributes staked BGT among validators based on updated weights.
- Rebalancing temporarily halts processBoostCycle, extending redemption wait times by ~2 cycles.

Updated Rebalancing and Validator Management

With the dropBoost change, rebalancing now accounts for unbonding delays before redistributing stake. The updated algorithm executes in three stages:

1. **SetValidators**
    - Creates a copy of the old validator list.
    - Updates new validator weights.
    - Adds new validators to the main validator list in state.
2. **QueueRebalance**
    - Calculates target weights based on active BGT only.
    - Cancels all queued boosts and drops across validators.
    - Stores target boosts for each validator.
    - Queues up all surplus removals.
3. **CompleteRebalancing**
    - Executes dropBoost for validators with surplus stake.
    - Reassigns BGT to validators with a deficit.
    - Removes validators that were dropped from the set.

This ensures that stake is properly redelegated after unbonding, maintaining balance across the validator set.

HONEY Rewards

- Users can stake stBGT in the StBGTStaking contract to earn HONEY rewards.
- HONEY rewards come from fees generated by Bex, Bend, and Berps and are distributed based on the time staked.
- The StrideEngine receives HONEY rewards and transfers them to StBGTStaking during processBoostCycle.
- Once Stride integrates with Dolemite’s lending market, HONEY rewards will be redirected there, and StBGTStaking will no longer distribute rewards.

Audit Dashboard

Target Summary

- **Type**: Protocol and Implementation
- **Platform**: Solidity
- **Artifacts:**
    - StrideEngine.sol
    - StrideManager.sol
    - StBGTStaking.sol

Auditors

- Darko Deuric
- Marko Juric

Engagement Summary

- **Dates**: 27 Jan 2025 - 07 Feb 2025.
- **Method**: Code Review, Protocol Analysis

Severity Summary

Finding Severity	#
Critical	0
High	0
Medium	0
Low	0
Informational	7
Total	**7**

Threat Analysis

Properties for redemption functionality

1. ✓ **A user's redemption must eventually be possible to complete**
   - Redemptions are delayed by rebalancing but cannot be indefinitely blocked.
   - Each redemption is tracked in `UserRedemption` with an `endPeriod`, which determines the minimum waiting period.
     - If `boostPeriod ≈ 5 hours`, a redemption will be available in 1-2 boost cycles (5-10 hours).
   - Frequent rebalancing **extends** redemption periods but does **not prevent them from completing**.
     - Since rebalancing prevents boost cycles and delays redemptions for at least two additional cycles, the wait time extends to 15-25 hours.
   - `processBoostCycle` ensures that redemptions eventually transition from processing to finished by increasing `boostPeriod`.
2. ✓ **Users can redeem anytime after the `endPeriod` has passed and cannot redeem before the `endPeriod`.**
   - The function `completeRedemption()` verifies that `boostPeriod >= endPeriod` before allowing redemption.
   - Admin can disable redemptions - `redemptionsEnabled` is an admin-controlled state variable, allowing the admin to block all redemptions. ❗
3. ✓ **If a user has an ongoing redemption, a new redemption appends to the previous one and extends the waiting period.**
   - `queueRedemption()` appends the new redemption amount to the existing `UserRedemption` record and updates `endPeriod = boostPeriod + 2`.

Properties for rebalancing functionality

1. ✓ **After `_queueRebalancing`, all queues from the current cycle are canceled.**
   - `_cancelBoostQueues` is called for each validator, resetting all active boost/drop queues.
   - This ensures that queued boosts or queued drop boosts from currect cycle do not interfere with the rebalancing process.
   - When `processBoostCycle` is called, if there were any `excessRewards`, they were queued for boost.
   - `boostedQueue[engine][val_pubkey].balance` is increased by the appropriate amount (proportional to validator weight) for every validator.
   - The sum of all `boostedQueue[engine][val_pubkey].balance` must equal `excessRewards` for the current cycle
   - When `setValidators` is triggered during rebalancing, `cancelBoost` is called for each validator, reducing `boostedQueue[engine][val_pubkey].balance` back to 0.
   - The sum of all `boostedQueue[engine][val_pubkey].balance` after `_finishBoostQueues` should be 0. This is true because `activateBoost` resets `boostedQueue[engine][val_pubkey].balance` to 0.

  - The boost queues should be reduced for the same total amount (`excessRewards`) as they were initially increased after `processBoostCycle's _queueBoostAcrossValidators` **,** if rebalance happens**.**
  - The same conclusions hold for `dropBoostQueue` .
2. ✓ **The validators that had a surplus dropped BGT to align with their target value, and deficit validators entered the queue for boosts up to their target value.**
   - Immediately after `completeRebalancing` , validators are not fully balanced because boosts have not yet been activated.
   - After `_finishBoostQueues` in the next `processBoostCycle` , all validators should be balanced and remain so until the next rebalancing.
3. ✓ **Only** `staked` (boosted) BGT is rebalanced.
   - Unboosted / queued BGT is not affected by rebalancing.
4. ✓ **Rebalancing cannot be completed if it was not started.**
   - `require(rebalancingInProgress, "No rebalancing in progress");` prevents unauthorized rebalancing completion.
5. ✓ **Only one rebalancing process can be ongoing at the same time.**

Properties for process boost cycle

- ✓ `boostPeriod` **can be incremented only by** `processBoostCycle` **.**
  - Ensures the cycle number (`boostPeriod`) is advanced exclusively within `processBoostCycle()` , preventing accidental increments by any other function.
- ✓ `boostPeriod` **only increments after** `boostDelay` **blocks have passed since** `boostPeriodStartBlock` **.**
- ✓ **No function (besides rebalancing) can forcibly reset** `boostPeriodStartBlock` **in a way that breaks normal time progression.**
  - `processBoostCycle` resets `boostPeriodStartBlock` only after `boostDelay` blocks.
  - `rebalancing` also resets it, but simultaneously sets `rebalancingInProgress = true` (blocking further `processBoostCycle` calls), so the system's schedule is still coherent.
- ✓ **Every "cycle" state variable resets at the beginning/end of that period:**
  - `boostPeriodRedemptions` and `boostPeriodRewards` are set to 0 in `processBoostCycle()` after caching their old values.
  - `redemptionsProcessing` is also cleared (moved to `redemptionsFinished`) or adjusted before a new cycle.
- ✓ `rebalancingInProgress` **blocks** `processBoostCycle` **.**
  - `require(!rebalancingInProgress, "Cannot process boost cycle when a rebalancing is in progress");`
  - Prevents partial or conflicting state updates while rebalancing logic is mid-flow.
- ✓ **If** `excessRewards > 0` **,** `_queueBoostAcrossValidators(...)` **properly enqueues new boosts.**
  - Confirms leftover rewards are staked in the current cycle.
- ✓ **If** `excessRedemptions > 0` **,** `_queueDropBoostAcrossValidators(...)` **queues the necessary drops.**
  - Ensures unboosting the appropriate amount from validators.

- ✓ **If** `rewards==redemptions` **, no action with validators is needed.**
  - Ensures all redemptions are immediatelly considered finished.

Invariants for `processBoostCycle` and system integrity

1. ✓ The **total** BGT amount on the **Engine contract** at the start of `processBoostCycle` must be: total BGT= (previous cycle rewards) + ∑ queued boost balance
   - **Ensures:** No BGT is lost or created unexpectedly between cycles.
2. ✓ `redemptionsFinished` **can't go negative**
   - `redemptionsFinished` is `uint256`, so underflows are not possible, but attempting to decrement beyond zero would revert.
   - `redemptionsFinished` decreases in `completeRedemption()` and increases in `processBoostCycle()`.
   - **Ensures:** Redemptions can never exceed available finished redemptions.
3. ✓ `redemptionsProcessing` == sum of all validator queued drop boost balances
   - The queued **dropBoost** across all validators must exactly match the **redemptionsProcessing** amount stored in state.
   - **Ensures:** The system correctly tracks redemptions in progress and doesn't allow mismatches.
4. ✓ **All boosted BGT is proportionally (by weight) distributed across validators**
   - **Ensures:** No validator unexpectedly holds more or less than its proportionate share.
5. ✓ **Imbalances can only be Introduced via** `setValidators`
   - Changing the validator set (with `setValidators`) can momentarily disrupt the proportion of BGT across validators.
   - **Ensures:** The only *permitted* imbalance is during rebalancing, and it's corrected afterward.
6. ✓ **After** `completeRebalancing` **and the next** `processBoostCycle` **, boosted BGT distribution remains balanced until the next** `setValidators` **call**
   - Once a rebalance completes and the next PBC occurs (finishing any queue boosts), validator shares are balanced according to their weights and this state maintains until the next validator update.
   - **Ensures:** The system maintains integrity between validator set updates.

Threat Analysis & Conclusions

1. Operator Control Over Key Functions

- **Threat:** Since only the operator can trigger `processBoostCycle`, `setValidators`, and `completeRebalancing`, they can arbitrarily delay or accelerate these calls.
- **Impact:** Users might experience delays in redemptions.
- **Conclusion:** The entire `stBGT` system has centralized aspects (pause functionality, redemption controls, and potential redemption delays), but the Stride team is aware and accepts this design choice. ✓

2. Longer Redemption Windows Due to Timing

- **Threat:** Some users might face longer redemption windows purely due to unlucky rebalancing timing.
- **Impact:** Redemption windows can range between **5 to 25 hours**.
- **Conclusion:** Not a significant concern. ✓

3. Inconsistent `UserRedemption.endPeriod`

- **Threat:** A process other than the user's redemption could modify the `endPeriod`.

- **Conclusion:** Not possible— `endPeriod` is only modified inside `queueRedemption`. ✓

4. Validators Queuing Both Boost and Drop Boost in the Same Cycle

- **Threat:** Some validators might queue a boost while others queue a drop boost within the same cycle.
- **Conclusion:** Not possible—either `boostPeriodRedemptions > boostPeriodRewards` or `boostPeriodRedemptions < boostPeriodRewards`, but both conditions cannot be true simultaneously. ✓

5. Partial or Double Drops of BGT Due to Rebalancing Timing

- **Threat:** If rebalancing occurs after queuing a boost/drop boost, partial or double drops of BGT could happen.
- **Conclusion:** Not possible—
  - When rebalancing happens, the full `excessRedemptions` amount is canceled from the drop queue.
  - `boostPeriodRedemptions` is increased for `excessRedemptions` (delaying it to the next cycle), and `redemptionsProcessing` is decreased by the same amount (canceling the assignment in `processBoostCycle`).
  - If `boostPeriodRedemptions < boostPeriodRewards`, the full `excessRewards` amount is canceled from the queue boost, and `boostPeriodRewards` is increased by `excessRewards` to be processed after rebalancing.
  - **During** `_queueRebalancing`, some active BGT is queued for drop.
  - **During** `completeRebalancing`, some BGT is queued for staking.
  - What started with `queueDropBoost` and `queueBoost` will be **unqueued (dropped/activated)** during `_finishBoostQueues` (`dropBoost` and `activateBoost` respectively), preventing any double or partial drops/boosts. ✓

6. Double Reset of `boostPeriodStartBlock`

- **Threat:** Both `processBoostCycle` and rebalancing calls (`_queueRebalancing`, `completeRebalancing`) may reset `boostPeriodStartBlock`.
- **Conclusion:** The first reset (inside `_queueRebalancing`) has no impact— `processBoostCycle` can proceed regardless of whether it's reset, as `rebalancingInProgress = true`. ✓

8. `stBGT` Can Be Stolen During Claim

- **Threat:** A user could potentially steal `stBGT` while claiming.
- **Conclusion:** Not possible—
  - Users can only claim on their own behalf (`msg.sender`).
  - Users can only access **BGT rewards** through their **own** dedicated vault agent.
  - There is **no** way for a user to access another user's vault. ✓

9. Berachain BGT Contract Affects Boost Time

- **Threat:** The Berachain BGT contract could cause `processBoostCycle` to revert by increasing/decreasing boost time.
- **Conclusion:** `processBoostCycle` could revert, but the operator can **easily** adjust the `boostDelay` state variable using `setBoostDelay`. ✓

Findings

Finding	Severity	Status
Gas Optimization in _cancelBoostQueues by Avoiding Zero-Value State Updates	INFORMATIONAL	ACKNOWLEDGED
Gas Optimization in _cancelBoostQueues to Avoid State Variable Updates in Every Loop Iteration	INFORMATIONAL	DISPUTED
Missing address(0) validation in address assignments	INFORMATIONAL	RESOLVED
Using rebalancingStartBlock variable may be unnecessary	INFORMATIONAL	RESOLVED
Ineffective event emission check in processBoostCycle	INFORMATIONAL	REPORTED
Lack of event emissions for state variable changes	INFORMATIONAL	DISPUTED
Various code quality improvements	INFORMATIONAL	RESOLVED

Gas Optimization in _cancelBoostQueues by Avoiding Zero-Value State Updates

Project

Stride 2025 Q1: stBGT Updates

Project	Stride 2025 Q1: stBGT Updates
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	MEDIUM
Status	ACKNOWLEDGED

Involved artifacts

- src/StrideEngine.sol

Description

The `_cancelBoostQueues` function updates the state variables `boostPeriodRewards`, `boostPeriodRedemptions`, and `redemptionsProcessing` even when their corresponding values are zero. This results in unnecessary storage writes, increasing gas costs.

A proposed optimization moves these state updates inside their respective `if` conditions, ensuring that the contract only updates storage when necessary.

```
    function _cancelBoostQueues(bytes memory validatorPubKey) internal {
        (, uint128 queuedBoostAmount) = bgt.boostedQueue(address(this),
validatorPubKey);
        (, uint128 queuedDropBoostAmount) = bgt.dropBoostQueue(address(this),
validatorPubKey);
        if (queuedBoostAmount != 0) {
            bgt.cancelBoost(validatorPubKey, queuedBoostAmount);
             boostPeriodRewards += queuedBoostAmount;
        }
        if (queuedDropBoostAmount != 0) {
            bgt.cancelDropBoost(validatorPubKey, queuedDropBoostAmount);
            boostPeriodRedemptions += queuedDropBoostAmount;
            redemptionsProcessing -= queuedDropBoostAmount;
        }
    }
```

Gas Savings

- Tested on Remix with 219 validators (for benchmarking purposes).
- If all validators have `queuedDropBoostAmount == 0`, the gas savings are approximately 13%.
- While the Stride team does not plan to support 219 validators, the optimization is still relevant for reducing gas costs.

Recommendation

Implement the proposed change to minimize redundant state updates and reduce gas consumption in `_cancelBoostQueues`, improving the efficiency of rebalancing transactions.

Status Update

The Stride team confirmed that they will implement this optimization.

Gas Optimization in _cancelBoostQueues to Avoid State Variable Updates in Every Loop Iteration

Project	Stride 2025 Q1: stBGT Updates
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	MEDIUM
Status	DISPUTED

Involved artifacts

- src/StrideEngine.sol

Description

The `_cancelBoostQueues` function is called within a loop over multiple validators. In its current implementation, the function updates the state variables `boostPeriodRewards`, `boostPeriodRedemptions`, and `redemptionsProcessing` inside the loop for each validator. This results in excessive storage writes, leading to high gas consumption.

By modifying `_cancelBoostQueues` to return the accumulated values for `queuedBoostAmount` and `queuedDropBoostAmount`, and updating the state variables once after the loop, significant gas savings could be achieved.

Example and Gas Benchmarking

The experiment involved a minimal Solidity contract to test gas savings when iterating over 100 validators, with and without the optimization. It was deployed and tested using Remix VM (Cancun).

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

contract GasOptimizationTest {
    uint128 public boostPeriodRewards;
    uint128 public boostPeriodRedemptions;
    uint128 public redemptionsProcessing;

    bytes[] public validatorPubKeys;

    constructor() {
```

```
        // 100 validators
        for (uint16 i = 0; i < 100; i++) {
            validatorPubKeys.push(bytes(abi.encodePacked(i)));
        }
        redemptionsProcessing = 1000; // to avoid underflow in for loop
    }

    // Mock function
    function _cancelBoostQueues(bytes memory /*validatorPubKey*/) internal pure
returns (uint128, uint128) {
        return (10, 5); // Mock return values (boostAmount, dropAmount)
    }

    // Loop with state updates inside the loop
    function loopWithoutOptimization() external {
        uint256 numValidators = validatorPubKeys.length;

        for (uint16 i = 0; i < numValidators; i++) {
            (uint128 boostAmount, uint128 dropAmount) =
_cancelBoostQueues(validatorPubKeys[i]);

            boostPeriodRewards += boostAmount;
            boostPeriodRedemptions += dropAmount;
            redemptionsProcessing -= dropAmount;
        }
    }

    // Loop with state update only once, after the loop
    function loopWithOptimization() external {
        uint256 numValidators = validatorPubKeys.length;
        uint128 totalBoostAmount = 0;
        uint128 totalDropBoostAmount = 0;

        for (uint16 i = 0; i < numValidators; i++) {
            (uint128 boostAmount, uint128 dropAmount) =
_cancelBoostQueues(validatorPubKeys[i]);
            totalBoostAmount += boostAmount;
            totalDropBoostAmount += dropAmount;
        }

        boostPeriodRewards += totalBoostAmount;
        boostPeriodRedemptions += totalDropBoostAmount;
        redemptionsProcessing -= totalDropBoostAmount;
    }
}
```

Scenario	Execution Cost	Transaction Cost	Gas	Gas Savings
Without Optimization	**540,206**	**561,270**	**645,461**	**-**
With Optimization	**373,892**	**390,156**	**459,720**	**~30%**

Recommendation

Instead of updating state variables inside `_cancelBoostQueues`, accumulate the values and update the state only once after the loop. Modify `_cancelBoostQueues` to return the queued amounts instead of directly modifying storage:

```
function _cancelBoostQueues(bytes memory validatorPubKey) internal returns (uint128
boostAmount, uint128 dropAmount) {
    (, uint128 queuedBoostAmount) = bgt.boostedQueue(address(this), validatorPubKey);
    (, uint128 queuedDropBoostAmount) = bgt.dropBoostQueue(address(this),
validatorPubKey);

    if (queuedBoostAmount != 0) {
        bgt.cancelBoost(validatorPubKey, queuedBoostAmount);
    }
    if (queuedDropBoostAmount != 0) {
        bgt.cancelDropBoost(validatorPubKey, queuedDropBoostAmount);
    }

    return (queuedBoostAmount, queuedDropBoostAmount);
}
```

Now, in `_queueRebalancing`, accumulate the total amounts before a single state update:

```
function _queueRebalancing() internal {
  uint128 totalActiveBoost = bgt.boosts(address(this));
  if (totalActiveBoost == 0) {
      return;
  }

  delete rebalancingTargetBoosts;
  uint256 numValidators = validatorPubKeys.length; // cache

  uint128 totalBoostAmount = 0;
  uint128 totalDropBoostAmount = 0;

  for (uint16 i; i < numValidators; i++) {
      bytes memory pubKey = validatorPubKeys[i];

      // Cancel boost queues and accumulate amounts
      (uint128 boostAmount, uint128 dropAmount) = _cancelBoostQueues(pubKey);
      totalBoostAmount += boostAmount;
      totalDropBoostAmount += dropAmount;

      uint128 activeBoost = bgt.boosted(address(this), pubKey);
      uint128 targetBoost = getValidatorTargetBoost(pubKey, totalActiveBoost);

      if (targetBoost < activeBoost) {
          uint128 surplus = activeBoost - targetBoost;
          bgt.queueDropBoost(pubKey, surplus);
      }
```

```
      rebalancingTargetBoosts.push(ValidatorTargetBoost({pubKey: pubKey, targetBoost:
targetBoost}));
    }

  // Update state only once after loop completes
  boostPeriodRewards += totalBoostAmount;
  boostPeriodRedemptions += totalDropBoostAmount;
  redemptionsProcessing -= totalDropBoostAmount;
  //...
  }
```

Status Update

The Stride team **disputed** the need for this optimization, given that:

- They prefer keeping the accounting updates within the same function to prevent accidental omissions in state updates.
- The transaction is admin-only and runs once every couple of weeks, making gas costs less of a concern.

While optimization results in ~30% gas savings, the team prioritizes **code safety and clarity over gas efficiency** in this specific case.

Missing address(0) validation in address assignments

Project	Stride 2025 Q1: stBGT Updates
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	NONE
Status	RESOLVED

Involved artifacts

- src/StrideEngine.sol
- src/StrideManager.sol

Description

Several instances within the `StrideEngine` and `StrideManager` contracts assign address state variables without validating that the provided address is non-zero (`address(0)`). This could lead to unintended behavior, such as breaking access control, disrupting contract logic, or preventing expected interactions with external contracts.

The affected locations in each contract are as follows:

StrideEngine:

1. `initialize` function

```
honey = IERC20(_honey);
```

2. `setManager` function

```
manager = _manager;
```

3. `setBeneficiary` function

```
beneficiary = _beneficiary;
```

StrideManager:

4. `initialize` function

```
pauser = _pauser;
```

5. `setPauser` function

```
pauser = _pauser;
```

6. `initialize` function

```
operator = _operator;
```

7. `setOperator` function

```
operator = _operator;
```

Recommendation

For all affected assignments **except** `beneficiary`, add a validation check to ensure the input address is not `address(0)`.

Status Update

Stride team has confirmed that allowing `address(0)` for `beneficiary` is intentional, so no changes are required there.

Using rebalancingStartBlock variable may be unnecessary

Project	Stride 2025 Q1: stBGT Updates
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	NONE
Status	RESOLVED

Involved artifacts

- src/StrideEngine.sol

Description

In the `_queueRebalancing` method, after all drop boosts across validators are queued based on target boosts, two key variables are updated:

1. `rebalancingInProgress` is set to `true` to prevent `processBoostCycle` from being called.
2. `rebalancingStartBlock` is set to `block.number` to mark the start of rebalancing.

Additionally, `boostPeriodStartBlock` is also reset to the current block to reflect the beginning of a new boost window.

```
// Set the rebalancing lock to prevent processBoostCycle from being called
rebalancingInProgress = true;
rebalancingStartBlock = block.number;

// Reset the boost window
boostPeriodStartBlock = block.number;
```

Later, when `complete_rebalancing` is executed, a delay check ensures that the rebalancing period has passed before proceeding:

```
require(block.number > rebalancingStartBlock + boostDelay, "Rebalancing queue is
still processing");
```

Problem Scenarios

- `rebalancingStartBlock` is always set to the same value as `boostPeriodStartBlock` at the start of rebalancing.
- Since both variables track the same block number and serve similar purposes, storing `rebalancingStartBlock` as a separate state variable may be redundant.
- This leads to unnecessary state storage and gas usage, as a single variable (`boostPeriodStartBlock`) could suffice for the delay check.

Recommendation

Instead of maintaining a separate `rebalancingStartBlock` variable, the contract can reuse `boostPeriodStartBlock` for the delay check in `complete_rebalancing` :

```
require(block.number > boostPeriodStartBlock + boostDelay, "Rebalancing queue is
still processing");
```

By eliminating `rebalancingStartBlock` , the contract can reduce storage costs and simplify logic without changing functionality.

Status Update

The Stride team has removed `rebalancingStartBlock` entirely and now uses `boostPeriodStartBlock` instead.

Ineffective event emission check in processBoostCycle

Project	Stride 2025 Q1: stBGT Updates
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	NONE
Status	ACKNOWLEDGED

Involved artifacts

- src/StrideManager.sol

Description

In the `processBoostCycle` function, the `StrideManager` contract calls `engine.processBoostCycle()`, which increments `boostPeriod`. The function also checks:

```
if (updatedPeriod > currentPeriod) {
    emit BoostCycle(updatedPeriod);
}
```

Problem Scenarios

However, since `processBoostCycle()` in the `engine` contract is designed to increment `boostPeriod` (and revert if an error occurs), it is **guaranteed** that `updatedPeriod > currentPeriod` if execution reaches this point.

If this condition ever fails (i.e., `updatedPeriod <= currentPeriod`), it indicates an unexpected and potentially serious issue in `engine.processBoostCycle()`. In such a case, failing to emit the event is not an adequate safeguard. Instead, the function should explicitly revert to signal an invariant violation.

Recommendation

If maintaining the check is necessary, explicitly revert when the condition is false to prevent silent failures:

```
require(updatedPeriod > currentPeriod, "Invariant violation: boost period did not
increase");
emit BoostCycle(updatedPeriod);
```

This ensures that if an unexpected issue occurs in `engine.processBoostCycle()`, it is caught immediately rather than allowing silent inconsistencies.

Lack of event emissions for state variable changes

Project	Stride 2025 Q1: stBGT Updates
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	NONE
Status	DISPUTED

Involved artifacts

- src/StrideEngine.sol
- src/StrideManager.sol

Description

There are several occurrences in the `StrideEngine` and `StrideManager` contracts where state variables are modified without emitting events. Emitting events for state changes is a best practice, as it allows users, dApps, or external processes to track or respond to changes effectively.

The affected locations in each contract are:

StrideEngine:

1. Set beneficiary (Code Reference)
2. Set boost delay (Code Reference)

StrideManager:

3. Set operator (Code Reference)
4. Set pauser (Code Reference)
5. Enable redemptions (Code Reference)
6. Disable redemptions (Code Reference)

Recommendation

To improve transparency and enable better off-chain monitoring, emit events whenever state variables are modified. This will enhance tracking, improve external integrations, and aid in debugging.

Status Update

Stride team decided not to add event emissions for admin addresses changes since they don't need to index those.

Various code quality improvements

Project	Stride 2025 Q1: stBGT Updates
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	NONE
Status	RESOLVED

Involved artifacts

- src/StrideEngine.sol
- src/StrideManager.sol
- src/StrideVaultAgent.sol
- src/StBGTStaking.sol
- src/interfaces/IStBGTStaking.sol
- src/interfaces/IStrideEngine.sol

Description

During the review, several minor code quality concerns were identified that, while not critical, could improve the maintainability, efficiency, and clarity of the codebase if addressed. These include:

1. Several functions that are not called internally but are marked as `public` should be `external` to optimize gas costs.
    - Affected functions:
        - `initialize` (`StrideEngine` CodeReference)
        - `processBoostCycle` (`StrideEngine` CodeReference)
        - `processBoostCycle` (`StrideManager` CodeReference)
        - `initialize` (`StrideManager` CodeReference)
        - `pause` (CodeReference)
        - `unpause` (CodeReference)
        - `receipts` (CodeReference)
        - `userReceiptTokens` (CodeReference)
        - Other `public view` functions
2. The Solidity version declaration should be fixed rather than open-ended (`^0.8.26`) to avoid unintentional breaking changes in future compiler versions.
    - Affected files: `StrideEngine.sol` , `StBGTStaking.sol` , `StrideManager.sol` , `StrideVaultAgent.sol` , `IStBGTStaking.sol` , `IStrideEngine.sol`

3. The `emit RedemptionCompleted` event currently includes two identical values (`redeemAmount` and `redeemAmount`), where `stBGTAmount == beraAmount` . Emitting only one value is more efficient.
4. Several comments provide incorrect or ambiguous information:
   - `Starts at zero and is incremented by processBoostCycle` comment: The counter actually starts at 1.
   - `If there's no queued or active BGT, no need to rebalance` comment : `bgt.boosts` only returns active boosts, while `queuedBoost` is separate.
   - `Get the validator's queued, active, and target boost amounts` comment : The comment implies that queued boosts are also fetched, but only active and target boosts are .
5. The `StBGTStaking` contract should inherit from `IStBGTStaking` to ensure consistency with its interface.

Recommendation

1. Change applicable `public` functions to `external` where they are not used internally.
2. Specify the Solidity version explicitly instead of using `^0.8.26` .
3. Modify the `emit RedemptionCompleted` statement to remove redundant values.
4. Update misleading comments to reflect the actual behavior of the code.
5. Inherit `IStBGTStaking` in `StBGTStaking` to enforce interface consistency.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability, etc.) set forth in the associated Services Agreement. This report provided in connection with the Services set forth in the Services Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This audit report is provided on an “as is” basis, with no guarantee of the completeness, accuracy, timeliness or of the results obtained by use of the information provided. Informal has relied upon information and data provided by the client, and is not responsible for any errors or omissions in such information and data or results obtained from the use of that information or conclusions in this report. Informal makes no warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of this report. This report should not be considered or utilized as a complete assessment of the overall utility, security or bugfree status of the code.

This audit report contains confidential information and is only intended for use by the client. Reuse or republication of the audit report other than as authorized by the client is prohibited.

This report is not, nor should it be considered, an “endorsement”, “approval” or “disapproval” of any particular project or team. This report is not, nor should it be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts with Informal to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor does it provide any indication of the client’s business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should it be leveraged as investment advice of any sort.

Blockchain technology and cryptographic assets in general and by definition present a high level of ongoing risk. Client is responsible for its own due diligence and continuing security in this regard.

Appendix: Vulnerability Classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of Common Vulnerability Scoring System (CVSS) v3.1, which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the Impact score, and the Exploitability score. The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ CVSS Qualitative Severity Rating Scale, and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

Impact Score	Examples
🟠 **High**	Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic
🟡 **Medium**	Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x)
🟢 **Low**	Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction)
🔵 **None**	Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation

Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

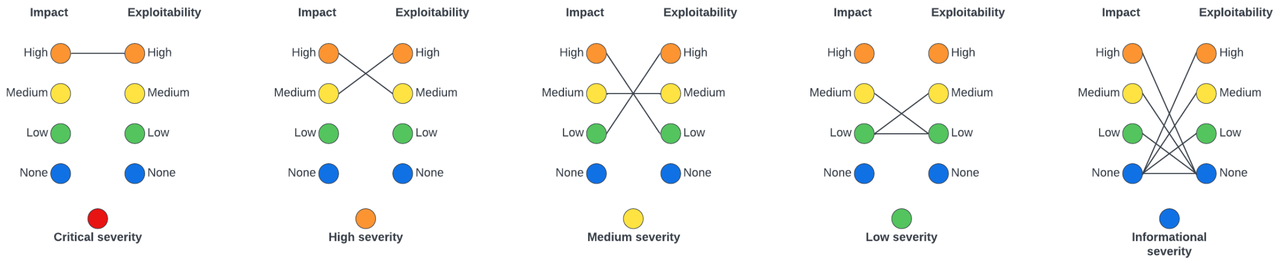
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

  - *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
  - *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for < 3%; *medium* for 3-10%; *large* for 10-33%, *all* for >33%. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

Exploitability Score	Examples
High	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
Medium	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
Low	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
None	illegitimate actions taken in a coordinated fashion by all actors

Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.



As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

Severity Score	Examples
Critical	Halting of chain via a submission of a specially crafted transaction

Severity Score	Examples
High	Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers
Medium	Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users
Low	2x increase in node computational requirements via coordinated withdrawal of all user tokens
Informational	Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary