



Security Audit Report

Stride 2024 Q4

Authors: Darko Deuric, Mahtab Norouzi

Last revised 25 October, 2024

Table of Contents

Audit Overview	2
The Project	2
Conclusions	2
Audit Dashboard	3
Target Summary	3
Auditors	3
Engagement Summary	3
Severity Summary	3
Threat Inspection	4
Findings	6
Lack of Sufficient Pause Controls in StrideManager Contract	7
Incorrect Balance Check in adminWithdraw Function	9
Missing Checks for ERC20 Transfer Return Values	11
Inefficient Boost Queuing When Unboosted Balance is Zero	13
Unnecessary Validator Weight Validation Loop in assertValidatorWeights	15
Inefficient Gas Usage Due to Uncached validatorPubKeys.length in Loops	17
Inefficient Use of State Variable _validatorTargetBoost in rebalance Function	19
Redundant Use of stBgtCollateralized Modifier in setBoostActivationDelay and addVaultReceiptPair	21
Inefficient Execution Order in deposit Function	22
Informational Miscellaneous Findings	24
Disclaimer	26
Appendix A: Vulnerability Classification	27
Impact Score	27
Exploitability Score	27
Severity Score	28
Appendix B: Slither Findings.....	30

[https://informalsystems.atlassian.net/wiki/spaces/AUDITS/database/325124313?
entryId=2c016548-0840-4df6-8b7a-8ced1c8d766e](https://informalsystems.atlassian.net/wiki/spaces/AUDITS/database/325124313?entryId=2c016548-0840-4df6-8b7a-8ced1c8d766e)

Audit Overview

The Project

In October 2024, Informal Systems conducted a security audit of two Solidity smart contracts: **StrideManager** and **StrideVaultAgent**. The audit focused on evaluating the security, efficiency, and best practices of these contracts. The onboarding process was smooth, with the Stride team providing comprehensive documentation, videos, and diagrams to improve our understanding.

The **StrideManager** contract is heavily reliant on external Berachain (POL) contracts, which we assumed to function correctly as they were not within the scope of this audit. The main focus was on the Stride contracts themselves, ensuring they operate securely and efficiently in various scenarios.

Conclusions

The overall security posture of the Stride contracts was found to be solid, with most of the findings centered around **gas savings and optimization opportunities**. However, one issue was marked as **high priority**: the **Lack of Sufficient Pause Controls in StrideManager**. This finding highlighted the absence of appropriate pause functionality, which could prevent critical operations when the contract is paused by the admin. This issue is significant and should be addressed to ensure robust emergency controls.

Additionally, Stride requested recommendations for improving their boost cycle process. Initially, the **optimizeBoostCycle** function was called at the end of each user transaction, leading to inefficiency and higher gas costs for users. Stride proposed the **processBoostCycle** function, which shifts boost cycle management to an off-chain cron job. We support this approach and further recommend two potential improvements:

1. **Using a decentralized Chainlink service** for automating smart contract functions.
2. **Incentivizing users to trigger the boost cycle process**, offering rewards for calling the function periodically.

We employed **Slither**, a static analysis tool, to identify additional findings, primarily related to code readability and minor gas optimizations. Some of these relevant findings have been included in the **Findings** section of the report, while the full list of Slither findings specific to these two contracts is provided in **Appendix B: Slither Findings**.

Audit Dashboard

Target Summary

- **Type:** Protocol and Implementation
- **Platform:** Solidity
- **Artifacts:**
 - [StrideManager.sol](#)
 - [StrideVaultAgent.sol](#)

Auditors

- Darko Deuric
- Mahtab Norouzi

Engagement Summary

- **Dates:** 07 Oct 2024 - 25 Oct 2024.
- **Method:** Code Review

Severity Summary

Finding Severity	#
Critical	0
High	1
Medium	1
Low	7
Informational	1
Total	10

Threat Inspection

Threat: Initialization issues

Conclusion: OK

- The contracts initialization process has been examined and verified to use the `initializer` modifier, ensuring that it can only be initialized **once**. This prevents accidental re-initialization, which could cause unexpected behavior or vulnerabilities.

Threat: Manager interaction with `bgt`, `stBGT`, and receipts/vaults is insecure

Conclusion: OK (Assumption)

- It is assumed that the `bgt`, `stBgt`, and vault contracts (and their associated tokens) are trusted and reliable. Given that these external contracts are critical to the system's functionality, this assumption is crucial. If these contracts contain vulnerabilities or malicious code, they could compromise the safety of the system. However, under the assumption of their trustworthiness, no issues were found regarding their interaction with the Manager contract.

Threat: Contract is not paused when things go wrong

Conclusion: Fail

- See [finding](#) for more details.

Threat: Return values are not checked

Conclusion: Fail

- See [finding](#) for more details.

Threat: Risk of DoS via small deposits

Conclusion: OK

- The contract has been evaluated for potential denial-of-service (DoS) attacks via small deposits. The design is efficient, using mappings rather than looping over large data sets, which minimizes gas costs and prevents state bloat. Additionally, only the first deposit by a user creates a new agent, meaning that subsequent deposits don't grow the state size unnecessarily. Overall, the gas costs and system behavior make it impractical for an attacker to perform a DoS attack via small deposits.

Threat: Rewards accounting doesn't work

Conclusion: OK

- The system for accounting rewards is sound. When users claim rewards, their `userClaimedRewards` is updated correctly within the `_claimAndMint` function. After rewards are distributed in the `claimAndDistributeRewards` function, the balance is reset to zero, ensuring accurate accounting of rewards. This prevents any discrepancies in reward claims, ensuring users get the correct reward amounts from different vaults.

Threat: `stBGT` is burned, but BERA is not sent to the user

Conclusion: OK

- The redeem process is robust and operates in a chained sequence where burning `stBGT` triggers the burn of `BGT` and results in the transfer of BERA to the user. If any part of this process fails (for example, if `BGT` burning fails), the entire transaction will revert, ensuring that users will not lose their tokens. This approach guarantees the integrity of the redeem process, preventing partial failures that could cause user funds to be lost or stuck.

Threat: Users cannot redeem tokens

Conclusion: OK

- For a successful redemption, the redeem amount must be less than or equal to the combined unboosted balance and active boosted balance. If the combined unboosted balance and active boosted balance are insufficient to cover the redeem amount, the transaction will fail, even if the total balance (including queued boosted tokens) would have been sufficient. The system intentionally excludes queued boosts from redemption calculations to maintain simplicity and avoid potential complications in managing queued boosts during the redemption process.

Threat: Manager contract can deposit an excessively large amount of tokens

Conclusion: OK

- While there is no hard cap on the number of tokens the Manager contract can deposit, there is no need for concern because the deposited tokens are immediately transferred to the BerachainRewardsVault contract. Additionally, the Manager contract only temporarily holds BGT rewards until users claim them, and the minting of `stBGT` reflects the balance accurately. The design ensures that large deposits won't cause state bloat or contract malfunctions.

Threat: After rebalancing, there is still some unboosted amount in the pool (rounding errors due to integer division)

Conclusion: OK

- After rebalancing, minor amounts of unboosted balance may remain in the pool due to rounding errors caused by integer division in Solidity. These discrepancies are typically very small due to the system's 18 decimal precision and do not negatively impact the functionality or operation of the contract. Although the unboosted balance is generally expected to be zero after rebalancing, these small rounding discrepancies do not affect the contract's overall efficiency or the integrity of the rebalancing process.

Threat: Issues can occur while upgrading

Conclusion: OK

- The contract uses a proxy architecture for upgrades, ensuring that only the logic (implementation) is updated, while the state remains intact. This approach is widely used and ensures that contract upgrades can occur without disrupting the system's existing data or user balances. No concerns have been identified with the upgrade process.

Findings

Finding	Severity	Status
Lack of Sufficient Pause Controls in StrideManager Contract	HIGH	ACKNOWLEDGED
Incorrect Balance Check in adminWithdraw Function	MEDIUM	ACKNOWLEDGED
Missing Checks for ERC20 Transfer Return Values	LOW	ACKNOWLEDGED
Inefficient Boost Queuing When Unboosted Balance is Zero	LOW	ACKNOWLEDGED
Unnecessary Validator Weight Validation Loop in assertValidatorWeights	LOW	ACKNOWLEDGED
Inefficient Gas Usage Due to Uncached validatorPubKeys.length in Loops	LOW	ACKNOWLEDGED
Inefficient Use of State Variable _validatorTargetBoost in rebalance Function	LOW	ACKNOWLEDGED
Redundant Use of stBgtCollateralized Modifier in setBoostActivationDelay and addVaultReceiptPair	LOW	ACKNOWLEDGED
Inefficient Execution Order in deposit Function	LOW	ACKNOWLEDGED
Informational Miscellaneous Findings	INFORMATIONAL	ACKNOWLEDGED

Lack of Sufficient Pause Controls in StrideManager Contract

Project	Stride 2024 Q4
Type	IMPLEMENTATION
Severity	HIGH
Impact	HIGH
Exploitability	MEDIUM
Status	RESOLVED

Involved artifacts

[src/StrideManager.sol](#)

Description

The `StrideManager` contract implements the ability to pause and unpause contract operations using OpenZeppelin's `PausableUpgradeable` module. The contract provides two public functions: `pause()` and `unpause()`, which are only accessible by the contract owner. However, there are no additional protections or considerations for sensitive functions that should not be callable when the contract is paused, such as deposit, withdraw, and redemption operations.

Problem Scenarios

If the contract is paused using the `pause()` function, the contract's functionality should ideally be restricted to prevent certain key operations from executing (such as deposit, withdrawal, or redemption). However, if the `whenNotPaused` and `whenPaused` modifiers are not applied to all relevant functions, users may still be able to interact with critical functions while the contract is paused.

This could result in inconsistencies in contract state, allowing users to bypass the intended pause mechanism or create unexpected scenarios where contract funds are moved or manipulated while in a paused state.

For example:

- A user might be able to **deposit** tokens even after the contract is paused, bypassing the emergency stop feature.
- In an emergency where funds or operations should be temporarily halted, an incomplete pause mechanism could expose the contract to attacks or further issues.

We think this is not Critical issue but High severity one, because there's no direct, immediate risk of catastrophic loss or widespread vulnerability. This issue doesn't directly open the contract to attacks or vulnerabilities like reentrancy or unauthorized access.

Recommendation

Apply the `whenNotPaused` **modifier** to sensitive functions such as `deposit()`, `withdraw()`, `redeem()`, and any other function that performs critical state changes or interacts with user funds.

Status Update

Resolved in [\[PR\]](#).

Incorrect Balance Check in adminWithdraw Function

Project	Stride 2024 Q4
Type	IMPLEMENTATION
Severity	MEDIUM
Impact	LOW
Exploitability	HIGH
Status	RESOLVED

Involved artifacts

[src/StrideManager.sol](#)

Description

In the `adminWithdraw` function, there is a logical error in the balance check for withdrawing ERC20 tokens. The condition `require(amount > balance, "Insufficient balance");` incorrectly compares the amount to the contract's token balance, effectively reversing the intended comparison logic. This means the function will allow withdrawals **only** when the `amount` exceeds the balance, which should not be the case.

Problem Scenarios

If the owner tries to withdraw tokens from the contract:

- The function would fail **incorrectly** when attempting to withdraw an amount less than or equal to the available token balance.
- The logic should check that the contract has enough tokens to fulfill the withdrawal request, but instead, it inverts the check.

This issue is highly likely to occur whenever the owner attempts to use the function, due to the incorrect logic, making the exploitability **high**. However, while the owner may be unable to withdraw tokens as intended, the amount trapped in the contract may not be significant (e.g., bribes sent to the contract), and it doesn't directly affect end users. Therefore, the impact is **low**, resulting in an overall severity of **medium**.

Recommendation

Change the balance check to correctly verify that the contract has sufficient tokens before allowing the withdrawal:

```
require(amount <= balance, "Insufficient balance");
```

Status Update

Resolved in [\[PR\]](#).

Missing Checks for ERC20 Transfer Return Values

Project	Stride 2024 Q4
Type	IMPLEMENTATION
Severity	LOW
Impact	MEDIUM
Exploitability	LOW
Status	RESOLVED

Involved artifacts

[src/StrideManager.sol](#)

Description

Several instances in the code, such as `transfer`, `transferFrom`, and `approve` in functions like `withdraw`, `deposit`, `claimAndDistributeRewards`, and `adminWithdraw`, do not check the return values of ERC20 operations. While the [ERC20 standard recommends reverting on failure](#), this is not mandatory, and some tokens may return `false` without reverting when a transfer fails. Not checking the return value can lead to silent failures and incorrect assumptions about successful transfers.

Problem Scenarios

If a token returns `false` rather than reverting, the contract will continue execution, potentially leading to an incorrect state. For example:

In the `claimAndDistributeRewards` function, if `stBgt.transfer` fails silently, the `userClaimedRewards[msg.sender] = 0;` statement would incorrectly reset the user's reward balance, even though no tokens were transferred.

Recommendation

Use OpenZeppelin's [SafeERC20](#) library to ensure that ERC20 operations revert on failure. This will prevent the contract from continuing execution when a `transfer`, `transferFrom`, or `approve` operation fails:

```
using SafeERC20 for IERC20;
IERC20(token).safeTransfer(msg.sender, amount);
IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
IERC20(token).safeApprove(address(vault), amount);
```

Status Update

Resolved in [\[PR\]](#).

Inefficient Boost Queuing When Unboosted Balance is Zero

Project	Stride 2024 Q4
Type	IMPLEMENTATION
Severity	LOW
Impact	LOW
Exploitability	MEDIUM
Status	RESOLVED

Involved artifacts

[src/StrideManager.sol](#)

Description

The `queueUnboostedBalance` function iterates over all validators and attempts to queue boosts even when the `totalUnboostedBalance` is zero. This leads to unnecessary gas consumption since the function will perform pointless iterations and calculations, but no actual boost will be queued. This occurs because the function lacks a check for whether `totalUnboostedBalance` is zero before proceeding. Consequently, when the balance is zero, the entire operation becomes a no-op, wasting computational resources and gas.

Problem Scenarios

- When the `totalUnboostedBalance` is zero, `queueUnboostedBalance` still loops through all validators.
- The function calls `getValidatorTargetBoost`, which returns zero boost amounts for all validators.
- The `queueBoost` function is then invoked with zero values, which results in no changes but still consumes gas due to the unnecessary computation and function calls.
- This problem is further propagated to `optimizeBoostCycle`, where `queueUnboostedBalance` is called without first checking if there is any unboosted balance to queue.

No functional errors or security risks are introduced. However, this issue increases gas usage unnecessarily resulting in **low** impact. This function will often be called as part of a regular boost cycle, and in cases where there is no unboosted balance, this inefficiency will occur frequently - **medium** exploitability.

Recommendation

- Add a conditional check in `queueUnboostedBalance` to return early if `totalUnboostedBalance == 0`. This prevents the function from running through unnecessary iterations.

- Alternatively, add this check in `optimizeBoostCycle` before calling `queueUnboostedBalance`, ensuring that the boost queuing process only occurs when there is an actual balance to queue.

```
function queueUnboostedBalance() internal {
    uint256 totalUnboostedBalance = bgt.unboostedBalanceOf(address(this));
    if (totalUnboostedBalance == 0) {
        return; // Avoid unnecessary iterations and computations
    }
    for (uint16 i; i < validatorPubKeys.length; i++) {
        bytes memory pubKey = validatorPubKeys[i];
        uint128 targetBoostAmount = getValidatorTargetBoost(pubKey,
totalUnboostedBalance);
        bgt.queueBoost(pubKey, targetBoostAmount);
    }
    lastQueuedBoostBlock = block.number;
}
```

Alternatively, the following could be added to `optimizeBoostCycle`:

- ```
uint256 totalUnboostedBalance = bgt.unboostedBalanceOf(address(this));
if (totalUnboostedBalance > 0) {
 queueUnboostedBalance();
}
```

## Status Update

The previous code [has been refactored](#) with the implementation of `processBoostCycle()` function, which replaces the two-loop structure previously found in `optimizeBoostCycle()`. This new function streamlines the boost management process by consolidating queuing and activation into a single iteration over validators, effectively reducing redundant loops and optimizing gas efficiency.



# Unnecessary Validator Weight Validation Loop in assertValidatorWeights

| Project        | Stride 2024 Q4 |
|----------------|----------------|
| Type           | IMPLEMENTATION |
| Severity       | LOW            |
| Impact         | LOW            |
| Exploitability | MEDIUM         |
| Status         | RESOLVED       |

## Involved artifacts

[src/StrideManager.sol](#)

## Description

The `assertValidatorWeights` function performs a loop to ensure that all validator weights are non-zero and sum to the expected total. This validation, while important, introduces additional gas overhead due to the extra loop. However, this validation can be integrated directly into the initial loop inside the `setValidators` function, where validator weights are already being set. This would eliminate the need for a second loop and reduce computational redundancy.

## Problem Scenarios

- When `setValidators` is called, the function loops over all validators to set their weights.
- After this, the `assertValidatorWeights` function runs an additional loop over the same set of validators to check two things:
  - Validator weights are non-zero.
  - Validator weights sum to 100%.
- This leads to a duplication of effort since the weights are already processed in the first loop and could easily be validated during that same loop, avoiding the need for an extra gas-consuming operation.

This issue doesn't affect functionality but adds unnecessary gas consumption. By removing the redundant loop, the function can become more efficient - **low** impact. Every time `setValidators` is called, the extra loop will execute - **medium** likelihood.

## Recommendation

Integrate the weight validation logic into the initial loop inside the `setValidators` function. This avoids the need for the additional `assertValidatorWeights` function and ensures that the weight validation occurs as the weights are being assigned.

Specifically:

- Check that each validator's weight is non-zero during the first loop.
- Track the total weight in the first loop and confirm it matches the expected `TOTAL_VALIDATOR_WEIGHTS` after the loop completes.

Example of optimized code:

```
function setValidators(Validator[] calldata validators) external onlyOwner
stBgtCollateralized {
 bytes[] memory oldValidatorPubKeys = validatorPubKeys;
 delete validatorPubKeys;

 uint256 totalWeight = 0;
 for (uint16 i = 0; i < validators.length; i++) {
 Validator memory validator = validators[i];
 require(validator.weight != 0, "Validator weight cannot be zero");

 validatorPubKeys.push(validator.pubKey);
 validatorWeights[validator.pubKey] = validator.weight;
 _validatorInNewSet[validator.pubKey] = true;
 totalWeight += validator.weight;
 }

 require(totalWeight == TOTAL_VALIDATOR_WEIGHTS, "Validator weights must sum to
10000 (100%)");

 ...
}
```

## Status Update

Resolved in [\[PR\]](#).

## Inefficient Gas Usage Due to Uncached validatorPubKeys.length in Loops

| Project        | Stride 2024 Q4 |
|----------------|----------------|
| Type           | IMPLEMENTATION |
| Severity       | LOW            |
| Impact         | LOW            |
| Exploitability | MEDIUM         |
| Status         | RESOLVED       |

### Involved artifacts

[src/StrideManager.sol](#)

### Description

In the `queueUnboostedBalance` function, the length of the `validatorPubKeys` array (a state variable) is accessed in every iteration of the for loop. Since `validatorPubKeys` is a state variable stored in the blockchain, each access to its `.length` is more costly in terms of gas. Repeatedly querying the length of a state variable inside a loop increases gas consumption unnecessarily. This can be avoided by caching the length in a local variable before the loop starts, which significantly reduces gas usage. This pattern is present not only in this function but in other parts of the code as well.

### Problem Scenarios

`validatorPubKeys.length` is accessed in each iteration of the for loop in `queueUnboostedBalance` :

```
for (uint16 i = 0; i < validatorPubKeys.length; i++) {
 bytes memory pubKey = validatorPubKeys[i];
 uint128 targetBoostAmount = getValidatorTargetBoost(pubKey,
totalUnboostedBalance);
 bgt.queueBoost(pubKey, targetBoostAmount);
}
```

Since `validatorPubKeys` is a state variable, each access to `.length` involves additional gas costs.

This is not only an issue in `queueUnboostedBalance` but also in other functions where state variables are used in for loops without caching their length:

- <https://github.com/Stride-Labs/stBGT/blob/264f6e3107154f113dd985b67f491e27c5a867a1/src/StrideManager.sol#L402>
- <https://github.com/Stride-Labs/stBGT/blob/264f6e3107154f113dd985b67f491e27c5a867a1/src/StrideManager.sol#L536>

- <https://github.com/Stride-Labs/stBGT/blob/264f6e3107154f113dd985b67f491e27c5a867a1/src/StrideManager.sol#L671>
- <https://github.com/Stride-Labs/stBGT/blob/264f6e3107154f113dd985b67f491e27c5a867a1/src/StrideManager.sol#L697>
- <https://github.com/Stride-Labs/stBGT/blob/264f6e3107154f113dd985b67f491e27c5a867a1/src/StrideManager.sol#L807>

While this issue doesn't impact the functionality or security of the contract, it increases gas costs, especially when the length of the `validatorPubKeys` array is large and when the loop runs frequently. Impact: **low**, exploitability: **medium**.

## Recommendation

Cache the length of `validatorPubKeys` in a local variable before entering the loop. This reduces the number of state variable reads and optimizes gas consumption.

Optimized example:

```
function queueUnboostedBalance() internal {
 uint256 totalUnboostedBalance = bgt.unboostedBalanceOf(address(this));
 uint256 validatorCount = validatorPubKeys.length; // Cache the length here

 for (uint16 i = 0; i < validatorCount; i++) {
 bytes memory pubKey = validatorPubKeys[i];
 uint128 targetBoostAmount = getValidatorTargetBoost(pubKey,
totalUnboostedBalance);
 bgt.queueBoost(pubKey, targetBoostAmount);
 }
 lastQueuedBoostBlock = block.number;
}
```

Review other similar functions and apply the same caching technique where state variable lengths are accessed in loops to ensure gas optimization across the contract.

## Status Update

Resolved in [\[PR\]](#).

# Inefficient Use of State Variable `_validatorTargetBoost` in rebalance Function

| Project        | Stride 2024 Q4 |
|----------------|----------------|
| Type           | IMPLEMENTATION |
| Severity       | LOW            |
| Impact         | LOW            |
| Exploitability | MEDIUM         |
| Status         | RESOLVED       |

## Involved artifacts

[src/StrideManager.sol](#)

## Description

In the `rebalance` function, the `_validatorTargetBoost` state variable is used to temporarily store each validator's target boost between the two for loops. However, this is inefficient because `_validatorTargetBoost` is a state variable, and repeatedly writing to and reading from state variables is more gas-intensive compared to using local memory. Since `_validatorTargetBoost` is only needed within the context of the function and not across contract calls, it can be replaced by a local memory array, which would significantly reduce gas usage.

## Problem Scenarios

- In the `first loop`, `_validatorTargetBoost[pubKey]` is used to store the target boost for each validator after calculating it based on their current and active boosts.
- This state variable is used solely for the purpose of the `second loop`, where the saved target boost values are read again.
- Writing and reading from state variables like `_validatorTargetBoost` is more costly compared to storing this data in memory, which is only needed during the execution of this function.

This issue doesn't affect the functionality of the contract, but it introduces unnecessary gas consumption. Impact: **low**, likelihood: **medium**.

## Recommendation

Instead of using a state variable, use a local memory structure (such as a mapping or an array) to store the target boosts during the rebalancing process. This will reduce gas costs as memory is much cheaper than state storage.

```
function rebalance() external onlyOwner stBgtCollateralized {
```

```

...
// Local memory mapping for storing target boosts
uint128[] memory targetBoosts = new uint128[](validatorPubKeys.length);

// First loop: unqueue boosts and drop surplus
for (uint16 i = 0; i < validatorPubKeys.length; i++) {
 ...
 // Store the target boost in memory for the next loop
 targetBoosts[i] = targetBoost;
}

// Second loop: queue boosts for validators with a deficit
for (uint16 i = 0; i < validatorPubKeys.length; i++) {
 bytes memory pubKey = validatorPubKeys[i];

 uint128 activeBoost = bgt.boosted(address(this), pubKey);
 uint128 targetBoost = targetBoosts[i];
 ...
}
}

```

This approach uses a memory array `targetBoosts` to store the target boost values between loops, which significantly reduces the gas costs compared to using the state variable `_validatorTargetBoost`.

## Status Update

Resolved in [\[PR\]](#).

## Redundant Use of `stBgtCollateralized` Modifier in `setBoostActivationDelay` and `addVaultReceiptPair`

| Project        | Stride 2024 Q4 |
|----------------|----------------|
| Type           | IMPLEMENTATION |
| Severity       | LOW            |
| Impact         | LOW            |
| Exploitability | MEDIUM         |
| Status         | RESOLVED       |

### Involved artifacts

[src/StrideManager.sol](#)

### Description

The `stBgtCollateralized` modifier is applied to the `setBoostActivationDelay` and `addVaultReceiptPair` functions, even though these functions do not depend on or interact with any collateralization logic. This modifier adds unnecessary gas overhead and complexity, as it serves no functional purpose for these operations.

### Recommendation

Remove the `stBgtCollateralized` modifier from both `setBoostActivationDelay` and `addVaultReceiptPair` functions to reduce gas consumption and simplify the code.

### Status Update

Resolved in [\[PR\]](#).

## Inefficient Execution Order in deposit Function

| Project        | Stride 2024 Q4 |
|----------------|----------------|
| Type           | IMPLEMENTATION |
| Severity       | LOW            |
| Impact         | LOW            |
| Exploitability | MEDIUM         |
| Status         | RESOLVED       |

### Involved artifacts

[src/StrideManager.sol](#)

### Description

In the `deposit` function, the logic first checks and potentially creates a new agent contract for the user, updates mappings, and then performs the `transferFrom` operation for receipt tokens. If the `transferFrom` call fails due to insufficient allowance, balance, or other reasons, the transaction reverts after unnecessary computation and state updates have already occurred. This violates the "fail early" principle, which aims to handle potential errors (such as failed transfers) as early as possible to save on unnecessary gas costs and reduce complexity.

### Problem Scenarios

- When a user calls `deposit`, the function first checks if an agent exists and, if not, creates one and updates mappings.
- Only after these steps does the function attempt to transfer the receipt tokens from the user to the contract via `transferFrom`.
- If the `transferFrom` fails (e.g., due to insufficient balance or allowance), the transaction reverts, wasting gas on the preceding operations that did not need to happen in the first place.

The issue doesn't affect core functionality but results in unnecessary gas consumption if the `transferFrom` call fails. Impact: **low**, exploitability: **medium**.

### Recommendation

Reorder the logic to perform the `transferFrom` operation first. If the transfer fails, the transaction will revert immediately, avoiding unnecessary computations and storage updates related to agent creation and mapping updates.

Optimized code snippet:



```
function deposit(address receipt, uint256 amount, string memory referralCode)
external stBgtCollateralized {
 require(amount > 0, "Deposit amount must be greater than 0");

 IBerachainRewardsVault vault = receiptVault[receipt];
 require(address(vault) != address(0), "Receipt not in allowlist");

 // Transfer the receipt from the user to the manager
 IERC20(receipt).transferFrom(msg.sender, address(this), amount);

 address agent = receiptUserAgent[receipt][msg.sender];
 if (agent == address(0)) {
 // Create the agent contract
 agent = _createVaultAgent(address(vault), receipt);

 // Store agent
 receiptUserAgent[receipt][msg.sender] = agent;

 // Update the indexers
 _receiptsByUser[msg.sender].push(receipt);
 _usersByReceipt[receipt].push(msg.sender);
 }

 // Additional logic...
}
```

## Status Update

Resolved in [\[PR\]](#).

## Informational Miscellaneous Findings

### Involved artifacts

[src/StrideManager.sol](#)

### Description

#### 1. Missing Check for Zero Address in `initialize` Function:

- In the `initialize` function, while there are checks to ensure that `_bgt` and `_stBgt` are not zero addresses, there is no such check for `_owner`. If `_owner` is set to the zero address, the contract's ownership may be inadvertently assigned to an invalid address.
- Recommendation:** Add a `require` statement to ensure `_owner != address(0)` to prevent incorrect initialization.

```
require(_owner != address(0), "Invalid owner address");
```

#### 2. Optimization Opportunity in `claimAndDistributeRewards` Function:

- In the `claimAndDistributeRewards` function, the line `userClaimedRewards[msg.sender] = 0;` resets the claimed rewards to zero after transferring the rewards. This operation is performed unconditionally, even if no rewards were claimed (i.e., `claimedRewards == 0`).
- Recommendation:** Move the reset operation into the `if` block that checks whether `claimedRewards > 0`, to avoid unnecessary state changes when no rewards are claimed.

```
if (claimedRewards > 0) {
 stBgt.transfer(msg.sender, claimedRewards);
 userClaimedRewards[msg.sender] = 0; // Reset only if rewards were claimed
}
```

#### 3. Redundant Check in `dropBoostForRedemption` Function:

- In the `dropBoostForRedemption` function, the line `require(remainingDropAmount == 0, "Failed to drop entire boost amount");` could be redundant. Since the function iterates until `remainingDropAmount` is exhausted, this check may no longer be necessary, as the logic ensures the remaining amount will be zero by the end.
- Recommendation:** Consider removing this `require` statement to simplify the code, as the condition is guaranteed by the logic.

#### 4. Clarification of Error Message in `dropBoostForRedemption` Function:

- The error message `require(totalBoostedAmount >= totalDropAmount, "Insufficient boosted balance to fulfill redemption");` could be improved for clarity. The current message does not make it clear that only the **active boosted balance** (not the queued boost) is considered during redemption.
- Recommendation:** Modify the error message to specify that the "active" boosted balance is insufficient.

```
require(totalBoostedAmount >= totalDropAmount, "Insufficient active boosted
balance to fulfill redemption");
```

These are informational recommendations aimed at improving code clarity, efficiency, and maintainability without affecting core functionality.

#### 5. Missing Event Emission for `boostActivationDelay` Changes

- The `setBoostActivationDelay` function allows the contract owner to modify the `boostActivationDelay`, a state parameter that controls how long a boosted stake must wait before activation. However, there is no event emitted when this state-changing action occurs. Emitting an event when a state variable is updated is a [best practice](#) in smart contract development, as it provides transparency and traceability for off-chain services and users monitoring the contract's state.

**Recommendation:** Emit an event whenever `boostActivationDelay` is modified to ensure proper logging and allow external systems to track changes.

```
event BoostActivationDelayUpdated(uint256 newDelay);

function setBoostActivationDelay(uint256 newDelay) external onlyOwner
stBgtCollateralized {
 boostActivationDelay = newDelay;
 emit BoostActivationDelayUpdated(newDelay);
}
```

#### 6. Redundant Comparison to Boolean Constant in `setValidators` Function

- In the `setValidators` function, the code compares the boolean value `_validatorInNewSet[validatorPubKey]` to `true`, which is unnecessary. Boolean values can be used directly in conditional statements without the need for comparison to `true` or `false`.
- Recommendation:** Simplify the condition by using the boolean value directly.

```
if (_validatorInNewSet[validatorPubKey]) {
 continue;
}
```

## 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 A: 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                                                                                                                                                                                                                                                                                                                            |
|---------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  <b>High</b>   | 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                                                                                                                                            |
|  <b>Medium</b> | 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) |
|  <b>Low</b>    | 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)                                                               |
|  <b>None</b>   | 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:

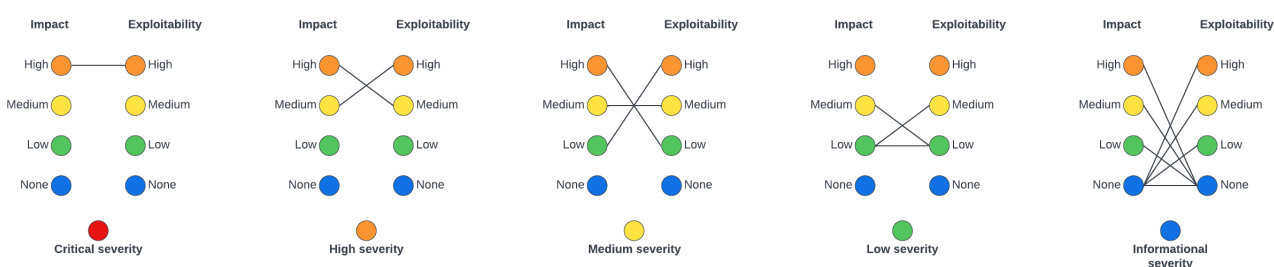
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

- *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                                                                                                                              |
|-------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
|  <b>High</b>   | illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors |
|  <b>Medium</b> | illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors |
|  <b>Low</b>    | illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors               |
|  <b>None</b>  | 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                                                             |
|-----------------------------------------------------------------------------------------------------|----------------------------------------------------------------------|
|  <b>Critical</b> | Halting of chain via a submission of a specially crafted transaction |

| Severity Score                                                                                         | Examples                                                                                                                                                                                                |
|--------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  <b>High</b>          | 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                                           |
|  <b>Medium</b>        | 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 |
|  <b>Low</b>           | 2x increase in node computational requirements via coordinated withdrawal of all user tokens                                                                                                            |
|  <b>Informational</b> | 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         |

## Appendix B: Slither Findings

- `StrideManager.deposit(address,uint256,string)` (src/StrideManager.sol#241-276) ignores return value by `IERC20(receipt).transferFrom(msg.sender,address(this),amount)` (src/StrideManager.sol#261)
- `StrideManager.withdraw(address,uint256)` (src/StrideManager.sol#284-300) ignores return value by `IERC20(receipt).transfer(msg.sender,amount)` (src/StrideManager.sol#296)
- `StrideManager.claimAndDistributeRewards(address)` (src/StrideManager.sol#351-369) ignores return value by `stBgt.transfer(msg.sender,claimedRewards)` (src/StrideManager.sol#359)
- `StrideManager.adminWithdraw(address,uint256)` (src/StrideManager.sol#519-528) ignores return value by `tokenContract.transfer(owner(),amount)` (src/StrideManager.sol#527)

Reference: <https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer>

- Reentrancy in `StrideManager.claimAndDistributeRewards(address)` (src/StrideManager.sol#351-369):  
External calls:
  - `_claimAndMint(msg.sender,receipt)` (src/StrideManager.sol#354)
  - `bgtRewards = vault.getReward(address(agent),address(this))` (src/StrideManager.sol#319)
  - `stBgt.mint(address(this),bgtRewards)` (src/StrideManager.sol#325)
  - `stBgt.transfer(msg.sender,claimedRewards)` (src/StrideManager.sol#359)
 State variables written after the call(s):
  - `userClaimedRewards[msg.sender] = 0` (src/StrideManager.sol#366)`StrideManager.userClaimedRewards` (src/StrideManager.sol#52) can be used in cross function reentrancies:
  - `StrideManager._claimAndMint(address,address)` (src/StrideManager.sol#308-332)
  - `StrideManager.claimAndDistributeRewards(address)` (src/StrideManager.sol#351-369)
  - `StrideManager.pendingRewards(address,address)` (src/StrideManager.sol#727-733)
  - `StrideManager.userClaimedRewards` (src/StrideManager.sol#52)
- Reentrancy in `StrideManager.deposit(address,uint256,string)` (src/StrideManager.sol#241-276):  
External calls:
  - `agent = _createVaultAgent(address(vault),receipt)` (src/StrideManager.sol#250)
  - `agent = new BeaconProxy(address(agentBeacon),agentInitializer)` (src/StrideManager.sol#229)
 State variables written after the call(s):
  - `receiptUserAgent[receipt][msg.sender] = agent` (src/StrideManager.sol#253)`StrideManager.receiptUserAgent` (src/StrideManager.sol#48) can be used in cross function reentrancies:
  - `StrideManager._claimAndMint(address,address)` (src/StrideManager.sol#308-332)
  - `StrideManager.deposit(address,uint256,string)` (src/StrideManager.sol#241-276)
  - `StrideManager.pendingRewards(address,address)` (src/StrideManager.sol#727-733)
  - `StrideManager.receiptUserAgent` (src/StrideManager.sol#48)
  - `StrideManager.withdraw(address,uint256)` (src/StrideManager.sol#284-300)
- Reentrancy in `StrideManager.setValidators(StrideManager.Validator[])` (src/StrideManager.sol#563-614):  
External calls:
  - `bgt.cancelBoost(validatorPubKey,queueBoostAmount)` (src/StrideManager.sol#597)
  - `bgt.dropBoost(validatorPubKey,activeBoostAmount)` (src/StrideManager.sol#600)
 State variables written after the call(s):
  - `_validatorInNewSet[validator_scope_2.pubKey] = false` (src/StrideManager.sol#610)`StrideManager._validatorInNewSet` (src/StrideManager.sol#82) can be used in cross function reentrancies:
  - `StrideManager.setValidators(StrideManager.Validator[])` (src/StrideManager.sol#563-614)
 Reference: <https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1>
- `StrideManager.deposit(address,uint256,string)` (src/StrideManager.sol#241-276) ignores return value by `IERC20(receipt).approve(address(vault),amount)` (src/StrideManager.sol#265)
- `StrideManager.setValidators(StrideManager.Validator[])` (src/StrideManager.sol#563-614) ignores return value by `(None,queueBoostAmount) = bgt.boostedQueue(address(this),validatorPubKey)` (src/StrideManager.sol#593)
- `StrideManager.rebalance()` (src/StrideManager.sol#622-672) ignores return value by `(None,queuedBoost) = bgt.boostedQueue(address(this),pubKey)` (src/StrideManager.sol#638)

Reference: <https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return>



- `StrideManager.setBoostActivationDelay(uint256) (src/StrideManager.sol#537-539)` should emit an event for:  
- `boostActivationDelay = newDelay (src/StrideManager.sol#538)`

Reference: <https://github.com/crytic/slither/wiki/Detector-Documentation#missing-events-arithmetic>

- `StrideManager.processBoostCycle() (src/StrideManager.sol#476-507)` has external calls inside a loop:  
(`lastQueuedBoostBlock,queuedBoostAmount`) = `bgt.boostedQueue(address(this),pubKey) (src/StrideManager.sol#485)`  
`StrideManager.processBoostCycle() (src/StrideManager.sol#476-507)` has external calls inside a loop:  
`bgt.activateBoost(address(this),pubKey) (src/StrideManager.sol#497)`  
`StrideManager.processBoostCycle() (src/StrideManager.sol#476-507)` has external calls inside a loop:  
`bgt.queueBoost(pubKey,targetBoostAmount) (src/StrideManager.sol#504)`  
`StrideManager.setValidators(StrideManager.Validator[]) (src/StrideManager.sol#563-614)` has external calls inside a loop: (`None,queueBoostAmount`) = `bgt.boostedQueue(address(this),validatorPubKey) (src/StrideManager.sol#593)`  
`StrideManager.setValidators(StrideManager.Validator[]) (src/StrideManager.sol#563-614)` has external calls inside a loop: `activeBoostAmount = bgt.boosted(address(this),validatorPubKey) (src/StrideManager.sol#594)`  
`StrideManager.setValidators(StrideManager.Validator[]) (src/StrideManager.sol#563-614)` has external calls inside a loop: `bgt.cancelBoost(validatorPubKey,queueBoostAmount) (src/StrideManager.sol#597)`  
`StrideManager.setValidators(StrideManager.Validator[]) (src/StrideManager.sol#563-614)` has external calls inside a loop: `bgt.dropBoost(validatorPubKey,activeBoostAmount) (src/StrideManager.sol#600)`  
`StrideManager.rebalance() (src/StrideManager.sol#622-672)` has external calls inside a loop: (`None,queuedBoost`) = `bgt.boostedQueue(address(this),pubKey) (src/StrideManager.sol#638)`  
`StrideManager.rebalance() (src/StrideManager.sol#622-672)` has external calls inside a loop: `activeBoost = bgt.boosted(address(this),pubKey) (src/StrideManager.sol#639)`  
`StrideManager.rebalance() (src/StrideManager.sol#622-672)` has external calls inside a loop:  
`bgt.cancelBoost(pubKey,queuedBoost) (src/StrideManager.sol#644)`  
`StrideManager.rebalance() (src/StrideManager.sol#622-672)` has external calls inside a loop:  
`bgt.dropBoost(pubKey,surplus) (src/StrideManager.sol#650)`  
`StrideManager.rebalance() (src/StrideManager.sol#622-672)` has external calls inside a loop:  
`activeBoost_scope_2 = bgt.boosted(address(this),pubKey_scope_1) (src/StrideManager.sol#663)`  
`StrideManager.rebalance() (src/StrideManager.sol#622-672)` has external calls inside a loop:  
`bgt.queueBoost(pubKey_scope_1,deficit) (src/StrideManager.sol#669)`

Reference: <https://github.com/crytic/slither/wiki/Detector-Documentation/#calls-inside-a-loop>

- `StrideManager.setValidators(StrideManager.Validator[]) (src/StrideManager.sol#563-614)` compares to a boolean constant:  
- `_validatorInNewSet[validatorPubKey] == true (src/StrideManager.sol#587)`

Reference: <https://github.com/crytic/slither/wiki/Detector-Documentation#boolean-equality>

- Parameter `StrideManager.initialize(address,address,address,uint256)._owner (src/StrideManager.sol#197)` is not in mixedCase  
Parameter `StrideManager.initialize(address,address,address,uint256)._bgt (src/StrideManager.sol#197)` is not in mixedCase  
Parameter `StrideManager.initialize(address,address,address,uint256)._stBgt (src/StrideManager.sol#197)` is not in mixedCase  
Parameter `StrideManager.initialize(address,address,address,uint256)._boostActivationDelay (src/StrideManager.sol#197)` is not in mixedCase
- Parameter `StrideVaultAgent.initialize(address,address,address,address)._user (src/StrideVaultAgent.sol#30)` is not in mixedCase  
Parameter `StrideVaultAgent.initialize(address,address,address,address)._vault (src/StrideVaultAgent.sol#30)` is not in mixedCase  
Parameter `StrideVaultAgent.initialize(address,address,address,address)._receipt (src/StrideVaultAgent.sol#30)` is not in mixedCase  
Parameter `StrideVaultAgent.initialize(address,address,address,address)._manager (src/StrideVaultAgent.sol#30)` is not in mixedCase

Reference: <https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions>