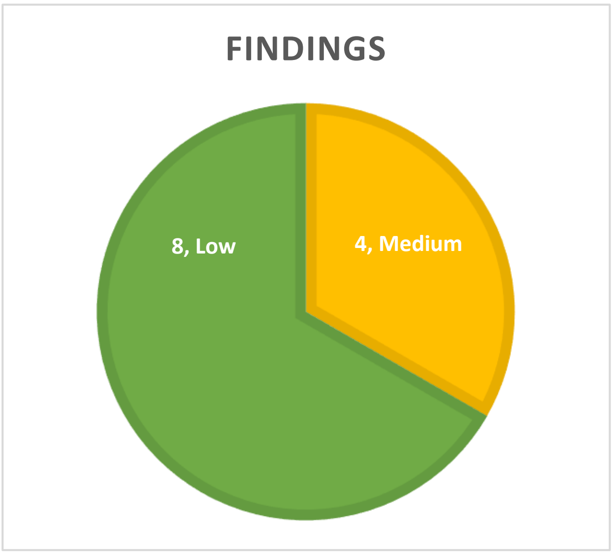
Trust Security

Smart Contract Audit

HoneyJar – SF-contracts-V2

18/12/2025

Executive summary

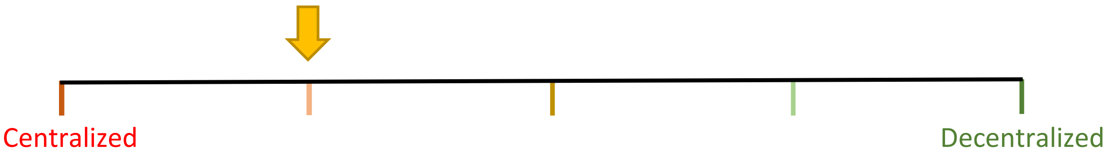


Category	Vaults
Audited file count	7
Lines of Code	891
Auditor	HollaDieWaldfee
Time period	08/12/2025 - 16/12/2025

Findings

Severity	Total	Fixed
High	0	0
Medium	4	4
Low	8	8

Centralization score



SIGNATURE

Digitally signed by Trust
Date: 2025.12.18 11:34:23 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	16/12/2025	Client report
0.2	18/12/2025	Mitigation review

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- src/MultiRewards.sol
- src/SFVault.sol
- src/strategies/BeradromeStrategy.sol
- src/strategies/base/BaseStrategy.sol
- src/strategies/modules/MultiRewardsModule.sol
- src/strategies/modules/BeradromeModule.sol
- src/strategies/modules/BgtWrapperModule.sol

Repository details

- **Repository URL:** https://github.com/0xHoneyJar/SF-contracts-V2
- **Commit hash:** 96961b07b6f198b2c3638cec0719fa94dcdd7444
- **Mitigation hash:** f53b36bac970976b126e75db3085d51764dca894

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

About the Auditors

HollaDieWaldfee is a distinguished security expert with a track record of multiple first places in competitive audits. He is a Lead Auditor at Trust Security and Senior Watson at Sherlock.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	Code is mostly simple and well structured.
Documentation	**Moderate**	Project is only documented with inline comments.
Best practices	**Good**	Project mostly adheres to best practices.
Centralization risks	**Severe**	Protocol admin is fully trusted.

Findings

Medium severity findings

TRST-M-1: WhenNotPaused modifier is missing from *SFVault.harvest()* and *SFVault.setStrategy()*

- **Category:** Logical issues
- **Source:** SFVault.sol
- **Status:** Fixed

Description

When *SFVault* is paused, *mint()* and *deposit()* are unavailable. The paused state is intended to block deposits into the strategy, and its external integration, in case of emergencies. However, *harvest()* and *setStrategy()*,which deposit all idle funds into the strategy, can still be accessed.

This inconsistency allows the restricted **KEEPER_ROLE** to escalate their privileges beyond the reward tokens and cause losses to user deposits. Since **STRATEGY_ROLE** is fully trusted, the lack of the modifier in *setStrategy()* does not lead to a privilege escalation.

Recommended mitigation

It is recommended to add the **whenNotPaused** modifier to *setStrategy()* and *harvest()*.

Team response

Fixed in commit 253dd67.

Mitigation review

Verified, the recommendation has been implemented.

TRST-M-2: Protocol keepers can break *MultiRewards* accounting via *notifyRewardAmount()* reentrancy

- **Category:** Reentrancy issues, Privilege escalation issues
- **Source:** MultiRewards.sol
- **Status:** Fixed

Description

In *MultiRewards*, keepers can call *getRebate()*, which executes a swap with the **kxRouter**, allowing the keeper to receive a callback (e.g., by swapping through a malicious pool). The token balances of **_feeToken** and **henloToken** before and after the swap are compared to calculate how many fee tokens were consumed and HENLO tokens received. By reentering *notifyRewardAmount()* from within the swap, token balances can be manipulated to include new rewards.

If HENLO is configured as a reward token, the HENLO rewards that are sent to *MultiRewards* within the swap callback are paid out to the **account**. Meanwhile, if *MultiRewards* receives additional **_feeToken** within the swap callback, they are registered as an **unswapped** amount and refunded to **account**.

As a requirement for the attack, the *MultiRewards* keeper must also have keeper privileges in a strategy such that *notifyRewardAmount()* can be accessed. Furthermore, keepers are trusted not to extract rewards, which means the issue lies in the broken *MultiRewards* accounting, rather than the loss of funds itself. As a result of the issue, token balances of *MultiRewards* are less than the reward liabilities that must be paid to users, creating bank-run dynamics, unless the correct reward token balance is restored via a donation.

Recommended mitigation

MultiRewards.notifyRewardAmount() should be protected with a reentrancy guard.

Team response

Fixed in commit 4b42f34.

Mitigation review

Verified, the recommendation has been implemented.

TRST-M-3: Donations to *BeradromeStrategy* can grief *MultiRewards* rewards rate

- **Category:** Griefing issues
- **Source:** BeradromeStrategy.sol, MultiRewardsModule.sol
- **Status:** Fixed

Description

Each call to *BeradromeStrategy.harvest()* calls *_distributeRewardsToMultiRewards()*, which loops over all configured reward tokens and notifies *MultiRewards* of any reward token balance that is greater than zero. In *MultiRewards*, the new rewards, including any pending previous rewards, are then paid out over **rewardsDuration** seconds.

Not each call to *harvest()* may swap into all available reward tokens. For example, the configured reward tokens may include HENLO and WBERA, but the keeper may intend to only swap into HENLO and not send any WBERA rewards into *MultiRewards*. By front-running the call to *harvest()*, an attacker can send 1 wei, such that *MultiRewards* is notified about the new WBERA rewards and dilutes the existing **rewardRate** over another full **rewardsDuration** period.

Recommended mitigation

The issue may be addressed by adopting a safe policy for *harvest()* calls where only the specific reward tokens are configured that are currently used. A more robust approach is to allow the keeper to specify the reward tokens, and to check in *_distributeRewardsToMultiRewards()* that these are valid reward tokens. This solution does not introduce new trust assumptions, since the keeper is already trusted not to extract value via swaps, and to harvest at a reasonable frequency.

Team response

Fixed in commit f641009.

Mitigation review

Verified, the finding has been addressed by requiring the keeper to specify the reward tokens.

TRST-M-4: *SFVault.emergencyWithdrawFromStrategy()* does not prevent subsequent strategy deposits

- **Category:** Logical issues
- **Source:** SFVault.sol
- **Status:** Fixed

Description

It is documented that *emergencyWithdrawFromStrategy()* should be used "in emergencies when immediate withdrawal is needed". But the function neither sets **strategy** to **address(0)**, nor pauses the contract, thus allowing further deposits into the same strategy, unless the admin has previously paused the contract, or disables deposits atomically after making the emergency withdrawal. This leads to unnecessary monetary risks for user assets.

Recommended mitigation

It is recommended to either call *_pause()* or set **strategy=ISFStrategy(address(0))**.

```
diff --git a/src/SFVault.sol b/src/SFVault.sol
index d9a087c..aab56a2 100644
--- a/src/SFVault.sol
+++ b/src/SFVault.sol
@@ -151,6 +151,7 @@ contract SFVault is
     function emergencyWithdrawFromStrategy() external onlyRole(DEFAULT_ADMIN_ROLE) {
         if (address(strategy) == address(0)) revert StrategyNotSet();
         strategy.emergencyWithdraw();
+        strategy = ISFStrategy(address(0));
         emit EmergencyWithdrawExecuted(msg.sender);
     }
```

Team response

Fixed in commit 6972509.

Mitigation review

Verified, the recommendation has been implemented.

Low severity findings

TRST-L-1: *BaseStrategy.sweepTokens()* does not account for staked assets

- **Category:** Logical issues
- **Source:** BaseStrategy.sol
- **Status:** Fixed

Description

When *sweepTokens()* is called with **token == _getAsset()**, the function protects the managed assets such that only excess assets can be swept. However, the logic fails to account for the assets that have been staked, e.g. in Beradrome. Since **this.totalManagedAssets()** includes assets that have been staked and are no longer held by the strategy, the excess asset balance is underestimated. Effectively, the admin won't be able to sweep any excess assets. This can be mitigated by an upgrade, and excess assets are already swept automatically as part of withdrawals.

Recommended mitigation

It is recommended that *sweepTokens()* matches the implementation of *getSweepableAmount()*. This means the **asset** balance that must be protected is not **this.totalManagedAssets()** but **this.totalManagedAssets() – this.stakedBalance()**.

Team response

Fixed in commit a434faa.

Mitigation review

Verified, the recommendation has been implemented.

TRST-L-2: *BaseStrategy.sweepTokens()* fails to protect receipt tokens and permissionlessly claimed reward tokens

- **Category:** Logical issues
- **Source:** BaseStrategy.sol
- **Status:** Fixed

Description

The only asset which is protected inside *sweepTokens()* is the vault's **asset**. It is stated that any other token balances are either donations or leftover rewards, which happens to be incorrect. *BaseStrategy* is an abstract contract which intends to support strategy implementations where staking assets in an external protocol mints a receipt token. This receipt token is not protected, and can be swept by the admin. Consequently, the protection for the **asset** is nullified.

Additionally, many protocols, including Beradrome, allow permissionless reward claiming. The notion of leftover rewards is not well defined, since it is unknown which rewards are left over from a previous harvesting and which have been permissionlessly claimed.

Given that the vault admin can also upgrade the strategy implementation, the checks in *sweepTokens()* do not prevent malicious behavior. Instead, the issue is in the lack of the sanity check and the permissionless reward claiming which break the function's assumptions, so that the admin may inadvertently transfer out rewards or staked tokens.

For example, by calling *sweepTokens()* with **amount=type(uint256).max** the admin may expect that only leftover (i.e., dust) rewards are swept, and the **amount** clamped to **maxSweep**. By permissionlessly claiming rewards, the swept **amount** can be higher than expected, thereby griefing stakers that should have earned the rewards.

Recommended mitigation

Since the admin is fully trusted, and *sweepTokens()* accepts **token** and **amount** parameters by which the swept tokens can be specified, the issue may be resolved by updating the documentation. Another complementary option is to introduce a mapping of protected tokens, which strategy implementations can access to manage receipt tokens and reward tokens. This functionality can be exposed to the vault admin. Finally, *sweepTokens()* and *getSweepableAmount()* can be defined as virtual functions so that their implementation can be adjusted depending on the strategy.

Team response

Fixed in commit f53b36b.

Mitigation review

Verified, the recommendation has been implemented. *BeradromeStrategy* implements *_isStrategyProtectedToken()*, which protects oBERO and reward tokens by default. The vault admin can use *setProtectedToken()* and *clearProtectedToken()* to override the standard protection.

TRST-L-3: Deposit cap can be exceeded due to donations

- **Category:** Logical issues
- **Source:** SFVault.sol
- **Status:** Fixed

Description

SFVault.deposit() and *SFVault.mint()* check that the new total assets don't break the **depositCap**, which is documented as a "hard cap on total managed assets". However, the **depositCap** can be bypassed via donations to *SFVault*, *BeradromeStrategy*, or a deposit in Beradrome on behalf of *BeradromeStrategy*.

Since a donation is shared among all users, increasing the total assets in this way causes a loss to the user making the donation, except for the unlikely case where there is just one depositor.

Recommended mitigation

Given that donations are shared pro-rata, the risk can likely be acknowledged. Still, it must be noted, and should be documented, that **depositCap** fails to impose a hard limit on the assets under management.

A strict **depositCap** can be implemented by not recognizing any assets beyond those that enter the protocol via *deposit()* or *mint()*. Any additional assets can be swept by the admin.

Team response

Fixed in commit cae5b26.

Mitigation review

The documentation has been updated to clarify that **depositCap** only restricts deposits via *mint()* and *deposit()*, but that *totalManagedAssets()* can grow above the **depositCap** due to donations.

TRST-L-4: Missing reentrancy guards in *SFVault.setStrategy()* and *SFVault.emergencyWithdrawFromStrategy()* allow protocol interaction in inconsistent state

- **Category:** Reentrancy issues
- **Source:** SFVault.sol
- **Status:** Fixed

Description

The root cause for the vulnerability is the lack of reentrancy guards in *SFVault.setStrategy()* and *SFVault.emergencyWithdrawFromStrategy()*. Suppose that the call to **strategy.emergencyWithdraw()** in *SFVault.setStrategy()* issues a callback to an untrusted address. Calls to *SFVault.deposit()* from within this callback make the deposit into the old strategy instead of the new one. When execution is resumed inside *SFVault.setStrategy()*, the **strategy** variable is set to the new **_strategy**, thus making deposits into the old strategy inaccessible. This loss is shared among all depositors. Even worse, if a keeper escalates their privileges and uses this callback to call *SFVault.harvest()*, all assets, not just new deposited ones, are deposited into the old strategy and become inaccessible.

Similar attack vectors exist in *SFVault.emergencyWithdrawFromStrategy()*. As an example, the emergency withdrawal may lead to an observed drop in *totalAssets()*, allowing users to deposit and receive more shares than they should be able to. This could be caused by an external protocol integration that first resets the balance owned by the HoneyJar strategy, then performs a callback, and then transfers the withdrawn tokens to HoneyJar.

The finding is reported as Low severity as there is no intention to support strategies with such callbacks for now, so the attack vector remains theoretical.

Recommended mitigation

It is recommended to apply reentrancy guards to *SFVault.setStrategy()* and *SFVault.emergencyWithdrawFromStrategy()*. Furthermore, in TRST-R-8, more reentrancy guards are recommended to reduce the risk of reentrancy attacks.

Team response

Fixed in commit cf59d64.

Mitigation review

Verified, the recommendation has been implemented.

TRST-L-5: *MultiRewards* rebates are unavailable if HENLO is configured as a fee token

- **Category:** Logical issues
- **Source:** MultiRewards.sol
- **Status:** Fixed

Description

MultiRewards is not compatible with HENLO as a fee token, since that implies the input and output tokens in the rebate swap are equal, but this is not allowed in the **kxRouter**. While in the deployment scripts only BGT wrappers and oBERO are configured as fee tokens, HENLO is a regular token, so it should be possible to use it as a fee token.

If HENLO is configured as a fee token, rebates are temporarily stuck, and HENLO must be removed again as a fee token, causing a loss of fees for the treasury and delayed reward payouts for users.

Recommended mitigation

It is recommended to make the contract compatible with HENLO as a fee token.

```
diff --git a/src/MultiRewards.sol b/src/MultiRewards.sol
index 27b88b2..cf9031a 100644
--- a/src/MultiRewards.sol
+++ b/src/MultiRewards.sol
@@ -422,7 +422,7 @@ contract MultiRewards is MultiRewardsBase {
         }

         // Swap rebate to HENLO
-        if (rebateAmount > 0) {
+        if (rebateAmount > 0 && _feeToken != henloToken) {
             // Validate swap parameters
             if (swap.input.amount == 0) revert ZeroSwapAmount();
             if (swap.input.token != _feeToken) revert InvalidSwapInputToken();
@@ -456,6 +456,13 @@ contract MultiRewards is MultiRewardsBase {
                 rewards[account][_feeToken] += unswapped;
             }

+            emit RewardPaid(account, henloToken, henloReceived);
+        } else if (rebateAmount > 0 && _feeToken == henloToken) {
+            henloReceived = rebateAmount;
+
+            // Transfer HENLO to user
+            ERC20(henloToken).safeTransfer(account, henloReceived);
+
             emit RewardPaid(account, henloToken, henloReceived);
         }
     }
```

Team response

Fixed in commit 5197a89.

Mitigation review

Verified, the recommendation has been implemented.

TRST-L-6: Re-adding reward token in *MultiRewards* leads to corrupted state

- **Category:** Logical issues
- **Source:** MultiRewards.sol
- **Status:** Fixed

Description

If a token is added as a reward token in *MultiRewards*, then removed and re-added, reward accounting is corrupted. For the re-added reward, **rewardPerTokenStored** starts at zero, while **userRewardPerTokenPaid** maintains its old values. This can cause an underflow in *earned()*, where the latter is subtracted from the former. As a result, users are unable to claim their rewards.

Recommended mitigation

To make the requirement that reward tokens can't be re-added explicit, it is recommended introduce a mapping of existing and removed reward tokens, and to check the mapping in *addReward()*.

```
diff --git a/src/MultiRewards.sol b/src/MultiRewards.sol
index 27b88b2..8a51cda 100644
--- a/src/MultiRewards.sol
+++ b/src/MultiRewards.sol
@@ -109,6 +109,8 @@ contract MultiRewards is MultiRewardsBase {
     /// @notice Tracks which reward tokens are fee tokens (claimed via getRebate)
     mapping(address => bool) public isFeeToken;

+    mapping(address => bool) public usedRewardTokens;
+
     /*//////////////////////////////////////////////////////////////
                             CONSTRUCTOR
     //////////////////////////////////////////////////////////////*/
@@ -173,6 +175,8 @@ contract MultiRewards is MultiRewardsBase {
         external
         onlyAdmin
     {
+        require(!usedRewardTokens[_rewardsToken], "invalid reward token");
+        usedRewardTokens[_rewardsToken] = true;
         _addReward(_rewardsToken, _rewardsDistributor, _rewardsDuration);
     }
```

Team response

Fixed in commit 8da9271.

Mitigation review

Verified, the recommendation has been implemented.

TRST-L-7: *BeradromeStrategy.emergencyWithdraw()* does not send idle funds to vault

- **Category:** Logical issues
- **Source:** BeradromeStrategy.sol
- **Status:** Fixed

Description

While idle funds in *BeradromeStrategy* don't count towards *SFVault.totalAssets()*, *beforeWithdraw()* and *harvest()* monetize the idle funds. Similarly, *emergencyWithdraw()* should not only unstake all funds from Beradrome, but also send idle funds to the vault. While idle funds can be swept by the admin, they are temporarily lost from *SFVault*'s tracking and not re-deposited into the new strategy.

Recommended mitigation

Idle funds should be sent to the vault as part of the emergency withdrawal.

```
diff --git a/src/strategies/BeradromeStrategy.sol
b/src/strategies/BeradromeStrategy.sol
index 0b348d0..1e613bc 100644
--- a/src/strategies/BeradromeStrategy.sol
+++ b/src/strategies/BeradromeStrategy.sol
@@ -280,6 +280,10 @@ contract BeradromeStrategy is BaseStrategy, BeradromeModule,
MultiRewardsModule,
         if (balance > 0) {
             _beradromeUnstakeToVault(balance);
         }
+        uint256 idle = IERC20(_getAsset()).balanceOf(address(this));
+        if (idle > 0) {
+            IERC20(_getAsset()).safeTransfer(vault, idle);
+        }
     }
```

Team response

Fixed in commit 7857c2e.

Mitigation review

Verified, the recommendation has been implemented.

TRST-L-8: Immediate release of idle strategy assets allows for arbitrage

- **Category:** Sandwiching issues
- **Source:** SFVault.sol, BeradromeStrategy.sol
- **Status:** Fixed

Description

SFVault.harvest() immediately stakes any underlying assets that have been returned by the call to *ISFStrategy.harvest()*. Since *BeradromeStrategy* does not include idle assets in *totalManagedAssets()*, *SFVault.totalAssets()* is immediately increased by the amount of idle assets that have been harvested.

This increase in *totalAssets()* can be profitably sandwiched since deposits and withdrawals can be made immediately and at zero cost.

For *BeradromeStrategy*, which is the only strategy implemented thus far, the risk may be accepted. Idle funds in *BeradromeStrategy* can only exist due to donations because *Beradrome* deposits don't generate any rewards in the underlying asset, and reward tokens should be sent to *MultiRewards*.

Recommended mitigation

The issue can be mitigated by implementing a mechanism to slowly release underlying assets before they are recognized in *totalAssets()*. This requires separate tracking of released and unreleased assets in *SFVault*. Another option is to implement deposit fees that make the cost of a deposit higher than the profit gained in the arbitrage. A third option is to only monetize such donations by sending them to *MultiRewards* instead of *SFVault* since *MultiRewards* has a timed release by default.

Team response

Fixed in commit 3928d21.

Mitigation review

The issue has been mitigated with a profit locking mechanism. Each time *harvest()* is called, the harvested **asset** balance is recognized as profit and released linearly over **profitUnlockTime**.

Additional recommendations

TRST-R-1: Local variable names should not shadow storage variables

It is error-prone to declare local variables with the same names as storage variables. The affected variable names are **_beradromePlugin**, **_vTokenRewarder**, **_oberoToken**, **_multiRewards**, **_rewardTokens**, **_rewardVault** and **_bgtConverter** in *BeradromeStrategy.initialize()*. Also affected are **_bgtConverter** and **_multiRewards** in their respective setters.

TRST-R-2: Remove unused code

- **CannotSweepAsset** error is never used.
- **SwapFailed** error is never used.
- **AlreadyInitialized** error is never used.

TRST-R-3: *BaseStrategy* does not define a storage gap

Since *BaseStrategy* is inherited by *BeradromeStrategy*, upgrading *BaseStrategy* with more storage variables would corrupt the storage layout. Therefore, a **__gap** with **49** slots should be defined. Additionally, although not necessary, *BeradromeStrategy* and *SFVault* may implement a **__gap** so that they can be safely inherited from.

TRST-R-4: Document that only oBERO is supported as a reward token

In *BeradromeModule._beradromeClaimObero()*, all rewards in **_vTokenRewarder** are claimed by calling *getReward()*. However, only oBERO is accounted for inside *_beradromeClaimObero()* and the upstream *BeradromeStrategy.harvest()* function. This is intentional as it is assumed that **_vTokenRewarder** will not support any additional reward tokens. It is recommended to add documentation for this assumption. Additional rewards could be swept by *BaseStrategy.sweepTokens()*.

TRST-R-5: Update documentation for protection against inflation attacks

It is stated that **minDeposit**, as well as how donations are treated, are mechanisms to protect against inflation attacks. However, both protections can be bypassed. A deposit with the minimum amount can be followed by a withdrawal that leaves just 1 wei shares in the vault. Meanwhile, a donation into *SFVault* is immediately recognized as part of the *totalAssets()*.

Since OpenZeppelin's ERC4626 implementation protects against inflation attacks by default with virtual shares, there is no security risk. The comments should be updated to specify that these mechanisms are not used for inflation protection. Instead, a comment may be added referring to OpenZeppelin's inflation protection. Since it is not needed, and can be bypassed, it is recommended that **minDeposit** is removed.

TRST-R-6: Additional swap validation in *BeradromeStrategy._executeKXSwaps()* and *MultiRewards._processRebate()*

It is recommended to add the following validations in *_executeKXSwaps()*:

- **swap.input.wrap == false**. Native input tokens are not supported.
- **swap.output.unwrap == false**. Native output tokens are not supported.
- **swap.output.token** is a configured reward token.
- **swap.feeData.surplusFeeBps** and **swap.feeData.referrerFeeBps** are zero so that no optional fees are paid.

Also, the following validations should be included in *_processRebate()*:

- **swap.input.wrap == false**. Native input tokens are not supported.
- **swap.output.unwrap == false**. Native output tokens are not supported.
- **swap.feeData.surplusFeeBps** and **swap.feeData.referrerFeeBps** are zero so that no optional fees are paid.

Given the keeper's trust assumptions, the above suggestions are optional.

TRST-R-7: *SFVault* should inherit its interface

As a best practice, *SFVault* should inherit *ISFVault* to avoid discrepancies.

TRST-R-8: Implement additional reentrancy guards

It is recommended to add reentrancy guards to the following functions, as an additional layer of safety, even though no vulnerabilities could be identified:

- *BeradromeStrategy.emergencyWithdraw()*
- *SFVault.transfer() / SFVault.transferFrom()* (override from *ERC20Upgradeable*)

TRST-R-9: Avoid code duplication in *SFVault.deposit()* and *SFVault.mint()*

The overridden *deposit()* and *mint()* implementations perform a check that the deposited assets do not break the **depositCap** and then the parent implementations are called. However, this logic is redundant with the parent implementations which already check *maxDeposit()* and *maxMint()*. The following simplification is recommended:

```
diff --git a/src/SFVault.sol b/src/SFVault.sol
index d9a087c..d05b3f8 100644
--- a/src/SFVault.sol
+++ b/src/SFVault.sol
@@ -226,11 +226,6 @@ contract SFVault is
         // Prevent inflation attacks with minimum deposit
         require(assets >= minDeposit, "Deposit below minimum");

-        if (depositCap != 0) {
-            uint256 newTotalAssets = totalAssets() + assets;
-            if (newTotalAssets > depositCap) revert
DepositCapExceeded(newTotalAssets, depositCap);
-        }
-
         // Standard 4626 mint logic first
         shares = super.deposit(assets, receiver);

@@ -253,12 +248,6 @@ contract SFVault is
         uint256 assetsNeeded = previewMint(shares);
         require(assetsNeeded >= minDeposit, "Deposit below minimum");

-        // Check deposit cap BEFORE minting (consistent with deposit())
-        if (depositCap != 0) {
-            uint256 newTotalAssets = totalAssets() + assetsNeeded;
-            if (newTotalAssets > depositCap) revert
DepositCapExceeded(newTotalAssets, depositCap);
-        }
-
         assetsIn = super.mint(shares, receiver);
```

TRST-R-10: Incorrect documentation for keeper privileges

- Validation of **receiver** does not prevent a draining of rewards. Rewards can still be drained via malicious swap parameters. The comment should say that tokens can't be swapped to an incorrect **receiver**.
- Underlying fee tokens can be lost by providing bad swap parameters. The documentation states they can't be lost.

TRST-R-11: Restricting gas for calls to badges contract is error-prone

By restricting the gas with which *badgesPercentageOfUser()* is called, it is possible that the call runs out of gas but function execution in *computeFees()* continues with the reserved gas. Since *badgesPercentageOfUser()* only receives 100k gas, the reserved gas is insufficient to execute the subsequent ERC20 transfer. Therefore, an OOG revert can't be abused by a malicious **keeper**. Still, there is no reason why an OOG failure should be accepted since the admin can replace a reverting **badges** contract, so it is recommended to make a regular Solidity function call.

```
diff --git a/src/MultiRewards.sol b/src/MultiRewards.sol
index 27b88b2..ab4ec09 100644
--- a/src/MultiRewards.sol
+++ b/src/MultiRewards.sol
```

```
@@ -479,15 +479,7 @@ contract MultiRewards is MultiRewardsBase {
             return amount; // No badges contract = all to treasury
         }

-        // Gas-limited external call to prevent DOS from malicious badges contract
-        (bool success, bytes memory data) =
-            address(badgesContract).staticcall{gas:
100_000}(abi.encodeCall(IBadges.badgesPercentageOfUser, (user)));
-
-        if (!success || data.length < 32) {
-            return amount; // On failure, all goes to treasury (safe default)
-        }
-
-        uint256 badgePercentageBps = abi.decode(data, (uint256));
+        uint256 badgePercentageBps = badges.badgesPercentageOfUser(user);
         if (badgePercentageBps == 0) {
             return amount; // No badge = all to treasury
         }
```

TRST-R-12: Setters and *notifyRewardAmount()* should check that reward token is valid

MultiRewards allows the admin to access setters for tokens that have not been added as reward tokens. Similarly, *notifyRewardAmount()* can be called for any token, though it would be an admin error to configure a **rewardsDistributor** for an invalid token and downstream division by **rewardsDuration** would cause a revert. It is recommended to add explicit checks that only configurations for valid reward tokens can be changed. Valid reward tokens can be identified with an invariant that **rewardsDuration > 0 ⇔ token is a reward token**.

```
--- a/src/MultiRewards.sol
+++ b/src/MultiRewards.sol
@@ -173,6 +173,7 @@ contract MultiRewards is MultiRewardsBase {
         external
         onlyAdmin
     {
+        require(_rewardsDuration > 0, "invalid rewards duration");
         _addReward(_rewardsToken, _rewardsDistributor, _rewardsDuration);
     }

@@ -181,6 +182,7 @@ contract MultiRewards is MultiRewardsBase {
      * @param _rewardsToken Reward token address to remove
      */
     function removeReward(address _rewardsToken) external onlyAdmin {
+        require(rewardData[_rewardsToken].rewardsDuration != 0, "invalid reward
token");
         _removeReward(_rewardsToken);
     }

@@ -255,6 +257,7 @@ contract MultiRewards is MultiRewardsBase {
      * @param _rewardsDistributor New distributor address
      */
     function setRewardsDistributor(address _rewardsToken, address
_rewardsDistributor) external onlyAdmin {
+        require(rewardData[_rewardsToken].rewardsDuration != 0, "invalid reward
token");
         rewardData[_rewardsToken].rewardsDistributor = _rewardsDistributor;
         emit RewardsDistributorUpdated(_rewardsToken, _rewardsDistributor);
     }
@@ -287,6 +290,7 @@ contract MultiRewards is MultiRewardsBase {
      * @param _rewardsDuration New duration in seconds
      */
     function setRewardsDuration(address _rewardsToken, uint256 _rewardsDuration)
external onlyAdmin {
```

```
+        require(rewardData[_rewardsToken].rewardsDuration != 0, "invalid reward 
token");
         _setRewardsDuration(_rewardsToken, _rewardsDuration);
     }

@@ -301,6 +305,7 @@ contract MultiRewards is MultiRewardsBase {
      * @dev Only callable by the designated rewards distributor
      */
     function notifyRewardAmount(address _rewardsToken, uint256 reward) external {
+        require(rewardData[_rewardsToken].rewardsDuration != 0, "invalid reward 
token");
         if (msg.sender != rewardData[_rewardsToken].rewardsDistributor) {
             revert NotRewardDistributor();
         }
```

TRST-R-13: Incorrect documentation for CEI pattern in *MultiRewards._processRebate()*

In *_processRebate()*, it is documented that the CEI (Checks-Effects-Interactions) pattern is followed. This is incorrect due to the previous swap and ERC20 transfer. In fact, the function can't follow the CEI pattern since the swap has to occur before the accounting of the refund. Thus, the comment is incorrect and should be removed. Reentrancy issues are addressed in separate findings.

TRST-R-14: Limit number of reward tokens in *MultiRewards*

To prevent issues arising due to excessive gas consumption of reward calculations, it is recommended to implement a constant that specifies the maximum number of reward tokens. A reasonable number is 10, to match the original *InfraredVault* implementation.

```
diff --git a/src/MultiRewards.sol b/src/MultiRewards.sol
index 27b88b2..49a0650 100644
--- a/src/MultiRewards.sol
+++ b/src/MultiRewards.sol
@@ -82,6 +82,10 @@ contract MultiRewards is MultiRewardsBase {
     /// @notice Basis points denominator (100% = 10000)
     uint256 private constant BPS_DENOMINATOR = 10_000;

+    /// @notice Maximum number of reward tokens that can be supported
+    /// @dev Limited to prevent gas issues with reward calculations
+    uint256 public constant MAX_NUM_REWARD_TOKENS = 10;
+
     /*//////////////////////////////////////////////////////////////
                                STATE
     //////////////////////////////////////////////////////////////*/
@@ -173,6 +177,7 @@ contract MultiRewards is MultiRewardsBase {
         external
         onlyAdmin
     {
+        require(rewardTokens.length < MAX_NUM_REWARD_TOKENS, "too many reward 
tokens");
         _addReward(_rewardsToken, _rewardsDistributor, _rewardsDuration);
     }
```

TRST-R-15: Add token checks in *BeradromeStrategy* to protect against misconfigurations

BeradromeStrategy can hold balances of different tokens. These are the **asset**, **oBERO**, **BGT** wrappers and reward tokens. To avoid misconfigurations, it is recommended to check in the initializer that **asset** is not **oBERO** and not any of the **BGT** wrappers, and in *addRewardToken()* that the new reward token is not **asset**.

The checks in the initializer are needed to prevent a scenario where the keeper can swap the **asset**, and the checks in *addRewardToken()* prevent the **asset** from being sent to *MultiRewards*.

Centralization risks

TRST-CR-1: Protocol admin is fully trusted

The protocol admin, i.e., the **DEFAULT_ADMIN_ROLE** in *SFVault* and **ADMIN** in *MultiRewards*, is fully trusted. *SFVault* and strategies are upgradeable, allowing the **DEFAULT_ADMIN_ROLE** to access all funds. In *MultiRewards*, the **ADMIN** can use *recoverERC20()* to recover all tokens, including staking and reward tokens.

TRST-CR-2: *SFVault* keeper, strategy and pauser roles

KEEPER_ROLE can call *harvest()* and is trusted to manage rewards and provide swap parameters. It cannot access strategy deposits.

STRATEGY_ROLE is fully trusted inside *SFVault*, since by providing a malicious **_strategy**, user funds can be stolen.

PAUSER_ROLE can call *pause()* which pauses strategy deposits, while *unpause()* must be called by the admin.

TRST-CR-3: *MultiRewards* keeper role

The **keeper** in *MultiRewards()* can call *getRebate()* which claims rebates for fee tokens. It is trusted to provide swap parameters, and therefore a malicious **keeper** can cause a loss of all rebates.

TRST-CR-4: *MultiRewards* rewards distributors

The **rewardsDistributor** for a reward token can call *notifyRewardAmount()*. By timing the call, certain users can be benefitted, and the **rewardRate** can be diluted by frequent calls. In the protocol's setup, **rewardsDistributors** are set to strategies, which in turn are harvested by strategy keepers.

Systemic risks

TRST-SR-1: Integrations with external protocols

The project integrates with Kodiak for swaps, Beradrome for strategy deposits, and Infrared, Miso, Bearn and BeraPaw for wrapping BGT. External integrations are considered trusted, and any issue in an external protocol can lead to a loss of users' assets and / or reward tokens.