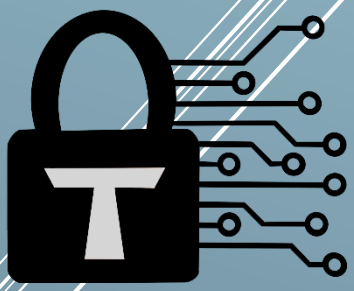
Trust Security

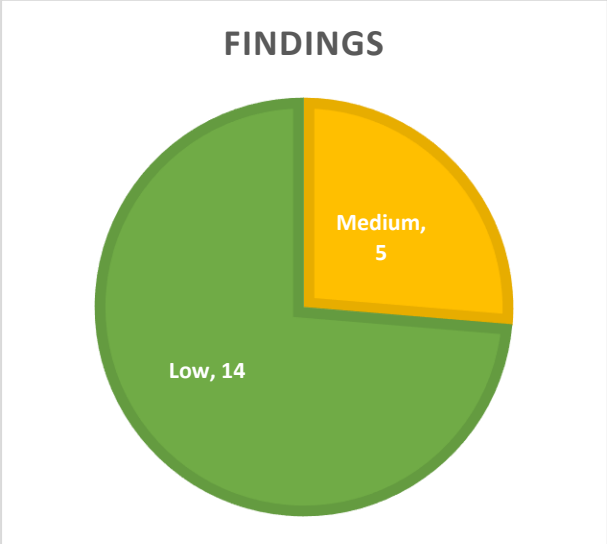


Smart Contract Audit

StakeDAO VLSDT

10/04/2026

Executive summary

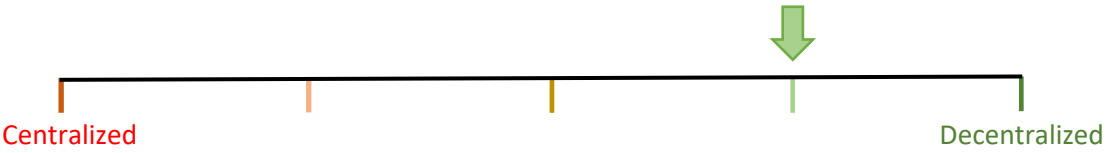


Category	Staking, Migration
Audited file count	11
Lines of Code	1516
Auditor	Trust cccz
Time period	16/03/2026- 26/03/2026

Findings

Severity	Total	Fixed	Open	Acknowledged
High	-	-	-	-
Medium	5	4	-	1
Low	14	11	-	3

Centralization score



Signature

Digitally signed by Trust
Date: 2026.04.10 23:19:35 +03'00'

EXECUTIVE SUMMARY 1

DOCUMENT PROPERTIES 4

Versioning **4**

Contact **4**

INTRODUCTION 5

Scope **5**

Repository details **5**

About Trust Security **5**

About the Auditors **5**

Disclaimer **6**

Methodology **6**

QUALITATIVE ANALYSIS 7

FINDINGS 8

Medium severity findings **8**

TRST-M-1 Premature checkpoint advancement locks rewards in unclaimable epochs 8
TRST-M-2 Precision loss in penalty calculation allows zero-penalty early exit 10
TRST-M-3 MigrateVESDT script fails due to TransparentUpgradeableProxy admin routing 11
TRST-M-4 Composite slippage guard fails to protect against epoch-boundary duration changes 12
TRST-M-5 Front-running checkpointToken() skews epoch reward allocation 13

Low severity findings **15**

TRST-L-1 Missing deadline parameter on stake and unstake exposes users to epoch-boundary timing risk 15
TRST-L-2 Early exit penalties bypass FeeDistributor checkpoint, misallocating reward epochs 16
TRST-L-3 Dust staking creates early checkpoint that causes display of wrong claimable amount 17
TRST-L-4 withdrawEarly() does not auto-apply pending penalty update, enabling rate selection 18
TRST-L-5 Migrate event logs recipient instead of lock owner for migrateTo() 19
TRST-L-6 DeployVLSDT script fails on Ownable2Step two-phase ownership transfer 20
TRST-L-7 Deployment script reverts when block.timestamp falls exactly on epoch boundary 21
TRST-L-8 KillFeeDistributor script blocks repeat calls needed for post-kill token recovery 22
TRST-L-9 Round-up in _calculateEndtime() allows effective duration to exceed maxDuration 22
TRST-L-10 Zero amount listings can be bypassed using updateListing() 23
TRST-L-11 Listings lack minimum duration, limiting seller control over commitment length 24
TRST-L-12 Partial fills can leave unfillable remainder below minFillAmount 25
TRST-L-13 Attacker-planted checkpoint forces victim to iterate empty epochs on claim() 25
TRST-L-14 Governance can extend unstakeDelay between unstake submission and execution 26

Additional recommendations **27**

TRST-R-1 FeeDistributor removes checkpoint rate limiting present in legacy version 27
TRST-R-2 Listing and offer IDs in BoostMarketplace share the same counter space 27
TRST-R-3 Fully filled orders in BoostMarketplace can be reused via updateListing() 27
TRST-R-4 Redundant code and unimplemented interface function 27
TRST-R-5 Documentation references outdated escrow model and incorrect epoch rounding 28
TRST-R-6 Preserve original initialize() logic alongside v2 migration in veSDT 28

Centralization risks 30
TRST-CR-1 Operators can redirect any user's unclaimed FeeDistributor rewards 30
TRST-CR-2 Upgradeable veSDT proxy can mint unbounded vlSDT via migrateFrom() 30

Systemic risks 31
TRST-SR-1 Low vlSDT supply during early migration enables governance capture 31
TRST-SR-2 Migration leaves stale veBoost delegations causing balance underflows 31
TRST-SR-3 Non-standard ERC20 tokens break FeeDistributor and BoostMarketplace accounting 31

Document properties

Versioning

Version	Date	Description
0.1	25/03/2026	Client report
0.2	02/04/2026	Mitigation review
0.3	10/04/2026	Updated responses

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

The following files are in scope of the audit:

- packages/vlsdt/src/BoostMarketplace.sol
- packages/vlsdt/src/FeeDistributor.sol
- packages/vlsdt/src/veSDT.vy
- packages/vlsdt/src/vlBoost.sol
- packages/vlsdt/src/vlSDT.sol
- packages/vlsdt/src/interfaces/IBoostMarketplace.sol
- packages/vlsdt/src/interfaces/IBoostRegistry.sol
- packages/vlsdt/src/interfaces/IVeSDT.sol
- packages/vlsdt/src/interfaces/IVlFeeDistributor.sol
- packages/vlsdt/src/interfaces/IVlSDT.sol
- packages/vlsdt/src/interfaces/IvlBoost.sol

Repository details

- **Repository URL:** https://github.com/stake-dao/contracts-monorepo
- **Commit hash:** bd948dd4ba0e17f58c108bbfca6df0d04fe9c4a7
- **Mitigation commit hash:** 045f1d4cc1b5a022308ec3b8c77f2f9d12bfe777

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

Trust has established a dominating presence in the smart contract security ecosystem since 2022. He is a resident on the Immunefi, Sherlock and C4 leaderboards and is now focused in auditing and managing audit teams under Trust Security. When taking time o ff auditing & bug hunting, he enjoys assessing bounty contests in C4 as a Supreme Court judge.

A top competitor in audit contests, cccz has achieved superstar status in the security space. He is a Black Hat / DEFCON speaker with rich experience in both traditional and blockchain security.

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. Fuzz tests and unit tests have also been used as needed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	Moderate	The integration with the Convex ecosystem introduces complexity.
Documentation	Excellent	Project is very well documented.
Best practices	Excellent	Project consistently adheres to industry standards.
Centralization risks	Good	The Project introduces several centralization risks.

Findings

Medium severity findings

TRST-M-1 Premature checkpoint advancement locks rewards in unclaimable epochs

- **Category:** Logical flaws
- **Source:** FeeDistributor.sol
- **Status:** Fixed

Description

The *FeeDistributor* constructor rounds **START_TIME** up to the next epoch boundary and initializes **lastTokenTime** to the same value. This design intends to ensure reward distribution begins cleanly from the first full epoch.

```
    constructor(address _vlSDT, address _rewardToken, address _owner, address
_emergencyAddress) Ownable(_owner) {
        require(_vlSDT != address(0), ZeroAddress());
        require(_rewardToken != address(0), ZeroAddress());
        require(_emergencyAddress != address(0), ZeroAddress());

        uint256 startTime = _roundUpToEpoch(block.timestamp);
        lastTokenTime = startTime;
        emergencyAddress = _emergencyAddress;

        START_TIME = startTime;
        VLSDT = IVlSDT(_vlSDT);
        REWARD_TOKEN = IERC20(_rewardToken);
    }
```

However, between deployment and **START_TIME**, anyone can call *checkpointToken()*. When the **REWARD_TOKEN** balance is zero, the function advances **lastTokenTime** to **block.timestamp** without distributing anything. This effectively moves the checkpoint cursor behind the intended start boundary.

```
    function checkpointToken() external whenNotKilled {
        _checkpointToken();
    }
...
    function _checkpointToken() internal {
        // Calculate tokens received since last checkpoint
        uint256 tokenBalance = REWARD_TOKEN.balanceOf(address(this));
        uint256 toDistribute = tokenBalance - tokenLastBalance;

        if (toDistribute == 0) {
            lastTokenTime = block.timestamp;
            return;
        }
```

Once rewards are later deposited via *deposit()*, the subsequent *_checkpointToken()* call distributes them proportionally across time since **lastTokenTime**. Because **lastTokenTime** was advanced to a pre-**START_TIME** timestamp, a portion of the rewards gets allocated to **tokensPerEpoch** entries for epochs before **START_TIME**.

```
        // Distribute across epochs
        uint256 time = lastTokenTime;
        uint256 sinceLast = block.timestamp - time;
        uint256 thisEpoch = _roundDownToEpoch(time);
        uint256 distributed;

        for (uint256 i; i < MAX_CHECKPOINT_EPOCHS; i++) {
            uint256 nextEpoch = thisEpoch + EPOCH_DURATION;
            uint256 epochShare;

            if (block.timestamp < nextEpoch) {
                // Current epoch (partial): allocate remaining time's share
                if (sinceLast == 0 && block.timestamp == time) {
                    // Edge case: called twice in same block
                    epochShare = toDistribute;
                } else {
                    epochShare = (toDistribute * (block.timestamp - time)) /
sinceLast;
                }
                tokensPerEpoch[thisEpoch] += epochShare;
                distributed += epochShare;
```

The *_claimable()* loop starts from **START_TIME** (or the user's first checkpoint, whichever is later), so tokens assigned to pre-**START_TIME** epochs are permanently unclaimable. The amount locked depends on how far **lastTokenTime** was advanced before the first real deposit.

```
    function _getStartEpoch(address user) internal view returns (uint256 startEpoch) {
        uint256 lastClaimed = lastClaimedEpoch[user];

        if (lastClaimed == 0) {
            // First-time claimer: find their first checkpoint in vlSDT
            uint256 numCheckpoints = VLSDT.numCheckpoints(user);
            if (numCheckpoints == 0) return 0; // User has never staked

            // Round UP timestamp of user's first checkpoint to next epoch boundary
            // (user earns from first full epoch after staking)
            (uint48 firstCheckpointTime,) = VLSDT.checkpoints(user, 0);
            startEpoch = _roundUpToEpoch(uint256(firstCheckpointTime));

            // Clamp to START_TIME (can't claim before distribution started)
            if (startEpoch < START_TIME) startEpoch = START_TIME;
```

Recommended mitigation

Require **block.timestamp** >= **START_TIME** at the beginning of *_checkpointToken()*. This prevents any checkpoint activity before the distribution period begins, ensuring **lastTokenTime** cannot be advanced prematurely. Alternatively, skip distribution to epochs before **START_TIME** within the checkpoint loop.

Team response

Fixed.

Mitigation review

The code now skips any checkpoint state changes in case it is called before the start time.

TRST-M-2 Precision loss in penalty calculation allows zero-penalty early exit

- **Category:** Precision loss issues
- **Source:** vlSDT.sol
- **Status:** Fixed

Description

vlSDT allows users who have requested an unstake to exit early by paying a time-proportional penalty via *withdrawEarly()*. The penalty is intended to decrease linearly from **maxPenaltyBps** (default 25%) at the moment of the unstake request down to 0% at the deadline.

```
    function withdrawEarly(uint256 id, address recipient) external nonReentrant {
        require(recipient != address(0), RecipientCannotBeZeroAddress());
        PenaltyConfig memory config = penaltyConfig;
        require(config.penaltyReceiver != address(0), EarlyExitDisabled());

        // Extract owner from ID = (nonce << 160) | owner
        address owner = address(uint160(id));
        require(msg.sender == owner, Unauthorized());

        UnstakeRequest memory request = unstakeRequests[id];
        require(request.amount > 0, UnstakeRequestNotFound());
        require(block.timestamp < uint256(request.deadline),
UnstakeRequestAlreadyExpired());
        (uint256 penaltyAmount,, uint256 userReceives) = _calculatePenalty(request,
config.maxPenaltyBps);

        delete unstakeRequests[id];
```

The *_calculatePenalty()* function computes the penalty in two steps: first it calculates **penaltyBps** as (**maxPenaltyBps** * **remainingTime**) / **totalDelay**, then it computes **penaltyAmount** as (**amount** * **penaltyBps**) / 10_000. The division-then-multiplication sequence introduces a precision loss window.

```
    function _calculatePenalty(UnstakeRequest memory request, uint24 maxPenaltyBps)
        internal
        view
        returns (uint256 penaltyAmount, uint256 penaltyBps, uint256 userReceives)
    {
        uint256 elapsed = block.timestamp - uint256(request.createdAt);
        uint256 totalDelay = uint256(request.deadline) - uint256(request.createdAt);
        uint256 remainingTime = totalDelay - elapsed;

        penaltyBps = (uint256(maxPenaltyBps) * remainingTime) / totalDelay;

        uint256 amount = uint256(request.amount);
        penaltyAmount = (amount * penaltyBps) / 10_000;
        userReceives = amount - penaltyAmount;
    }
```

With the default 8-week **unstakeDelay** (4,838,400 seconds) and **maxPenaltyBps** of 2500, **penaltyBps** rounds to zero when **remainingTime** < 4,838,400 / 2500 = 1,935 seconds (approximately 32 minutes). During this final 32-minute window, any user can call *withdrawEarly()* and receive their full unstaked amount with zero penalty, regardless of the amount being withdrawn. For large positions, this represents a meaningful value extraction, and regardless of whether it is in the final window or not, the issue presents a value leak.

Recommended mitigation

Compute the penalty in a single expression to avoid intermediate rounding: **penaltyAmount** = (**amount** * **maxPenaltyBps** * **remainingTime**) / (**totalDelay** * 10_000). This eliminates the precision loss entirely and ensures the penalty remains non-zero until **remainingTime** reaches zero.

Team response

Fixed.

Mitigation review

Issue has been addressed using the *mulDiv()* utility from OpenZeppelin.

TRST-M-3 MigrateVESDT script fails due to TransparentUpgradeableProxy admin routing

- **Category:** Logical flaws
- **Source:** 10_KillFeeDistributor.s.sol
- **Status:** Fixed

Description

veSDT is deployed behind a *TransparentUpgradeableProxy*. This proxy pattern intercepts calls from the **proxyAdmin** address and routes them to proxy management functions rather than the implementation. Any call from a non-admin address passes through to the implementation normally.

The *MigrateVESDT* deployment script encounters two distinct failures due to this proxy behavior:

1. Calling *IVeSDT(veSDTProxy).admin()* before *vm.startBroadcast()* returns **veSDT.admin** (i.e., **governance**) because the caller is the *MigrateVESDT* contract rather than **proxyAdmin**. This fails the **msg.sender == proxyAdmin** check.
2. Calling *IVeSDT(veSDTProxy).vlSDT()* before *vm.stopBroadcast()*, since the caller is **proxyAdmin**, it does not call the function in the implementation, so the call fails due to the function not being found.

```
    modifier ifAdmin() {
        if (msg.sender == _getAdmin()) {
            _;
        } else {
            _fallback();
        }
    }
```

Recommended mitigation

Restructure the script to avoid calling implementation functions through the proxy when the caller is the proxy admin.

```
contract MigrateVESDT is Script {
    address internal constant VE_SDT = Core.VESDT;

    function run() public returns (address veSDTProxy, address veSDTImplementation) {
        address vlsdt = vm.envAddress("VLSDT");
```

```
        veSDTProxy = VE_SDT;
+       vm.startBroadcast();
        address proxyAdmin = IVeSDT(veSDTProxy).admin();

-       vm.startBroadcast();
        require(msg.sender == proxyAdmin, "Only the admin of veSDT can migrate");

        // Migrate veSDT to the v2 implementation
        veSDTImplementation = deployCode("src/veSDT.json");
        ITransparentUpgradeableProxy(veSDTProxy)
            .upgradeToAndCall(veSDTImplementation,
abi.encodeWithSelector(IVeSDT.initialize.selector, vlsdt));

+       vm.stopBroadcast();
        require(IVeSDT(veSDTProxy).vlSDT() == vlsdt, "veSDT not upgraded to vlSDT");
-       vm.stopBroadcast();
    }
}
```

Team response

Fixed.

Mitigation review

The two issues have been resolved.

TRST-M-4 Composite slippage guard fails to protect against epoch-boundary duration changes

- **Category:** Slippage issues
- **Source:** BoostMarketplace.sol
- **Status:** Fixed

Description

In both *fillListing()* and *acceptOffer()*, the total payment is computed as **fillAmount** * **pricePerWeek** * **effectiveDuration / WEEK / PRECISION**. The **effectiveDuration** is derived from *_calculateEndtime()*, which rounds up to the next week boundary. Users provide slippage protection via **maxTotalPayment** or **minTotalPayment**, which guard the final computed amount.

This slippage design is flawed because it guards the product of two independent variables (**effectiveDuration** and **pricePerWeek**) rather than each individually. The **effectiveDuration** can change materially between transaction submission and execution — particularly if the transaction lands in a different epoch than expected.

Consider a seller calling *acceptOffer()* just before an epoch change, while the transaction executes after an epoch boundary. The duration will be 1 week more than anticipated when calculating **minTotalPayment**, selling boost for longer but price guarding for shorter. The offeror can then extract MEV by calling *updateOffer()* with a lower price per week which would still be accepted.

Recommended mitigation

Add explicit guards for **effectiveDuration** in both *fillListing()* and *acceptOffer()*. A **maxEffectiveDuration** parameter would allow users to bound the duration independently of price. This prevents epoch-boundary timing attacks while preserving the existing price slippage protection.

Team response

Fixed.

Mitigation

The **maxEffectiveDuration** flag was introduced as suggested.

TRST-M-5 Front-running checkpointToken() skews epoch reward allocation

- **Category:** Time sensitivity issues
- **Source:** FeeDistributor.sol
- **Status:** Acknowledged

Description

The *FeeDistributor*'s *_checkpointToken()* distributes newly received tokens proportionally across elapsed time since the last checkpoint. If 3 weeks have passed since the last checkpoint, a deposit is spread across all 3 epochs based on time weighting. However, *checkpointToken()* is permissionless — anyone can call it at any time to advance the checkpoint cursor.

This creates an exploitable asymmetry. A user who holds a disproportionately large share of vlSDT in the current epoch (but not in prior epochs) is incentivized to call *checkpointToken()* just before *deposit()* is called. This forces the deposit to be attributed only to the current epoch, maximizing their share. Without this front-run, the same deposit would be spread across prior epochs where other users may have held larger shares.

The inconsistency is fundamental: the contract attempts to distribute rewards retroactively across past epochs, but this behavior can be trivially circumvented by any participant calling a public function. The retroactive distribution is unreliable as a mechanism.

Recommended mitigation

Make the distribution behavior deterministic and front-run resistant. One approach: always attribute deposited tokens only to the current epoch at the time of *deposit()*, eliminating the retroactive distribution entirely. Furthermore, ensure every state changing action in the *FeeDistributor* first performs a checkpoint so that new rewards don't apply retroactively.

Team response

Fixed.

Mitigation review

The new code skips updating the **lastTokenTime** to current time in case there are no tokens to distribute. This change is not adequate as the exact same attack can still be performed, by performing a prior donation attack of 1 wei reward tokens. The single wei would be split to previous epochs, but the new deposits will only reward the current one.

Team response

It will be operationally mitigated under normal operations and self-healing by design. We accept the tradeoff that the guarantee is operational rather than at code-level.

Low severity findings

TRST-L-1 Missing deadline parameter on stake and unstake exposes users to epoch-boundary timing risk

- **Category:** Time sensitivity issues
- **Source:** vlSDT.sol
- **Status:** Acknowledged

Description

The *stake()* and *unstake()* functions in vlSDT accept no deadline parameter. These functions create checkpoints that the *FeeDistributor* uses to determine reward eligibility at epoch boundaries.

Because *FeeDistributor* snapshots balances at epoch-start timestamps (Thursday 00:00 UTC), a transaction that executes even one second after a boundary creates its checkpoint in the new epoch rather than the old one. A user intending to stake before the boundary to earn the current epoch's rewards could lose an entire week of rewards if their transaction is delayed by mempool congestion, gas price spikes, or sequencer delays on L2s. The same risk applies to *unstake()* — a user may unintentionally lose rewards for the current epoch. Had they known the situation, they may have preferred to keep hold of their vlSDT.

```
    function stake(uint256 amount, address recipient) external {
        require(amount > 0, AmountMustBeGreaterThanZero());
        require(recipient != address(0), RecipientCannotBeZeroAddress());

        IERC20(SDT).transferFrom(msg.sender, address(this), amount);
        _mint(recipient, amount);

        _checkpoint(recipient);
        emit Staked(msg.sender, recipient, amount);
    }

    /**
     * @notice Request to unstake vlSDT
     * @dev Burns vlSDT immediately. SDT becomes withdrawable after unstakeDelay.
     *      Cannot unstake more than available (balance - delegatedOut).
     *      Creates checkpoints for caller and totalSupply.
     * @param amount Amount to unstake
     * @return id The unstake request ID for later withdrawal
     */
    function unstake(uint256 amount) external returns (uint256 id) {
        require(amount > 0, AmountMustBeGreaterThanZero());
        uint256 balance = balanceOf(msg.sender);
        uint256 delegatedOut =
IBoostRegistry(BOOST_REGISTRY).delegatedOut(msg.sender);
        require(amount + delegatedOut <= balance, AmountExceedsUsableBalance());
```

Recommended mitigation

Add a **deadline** parameter to *stake()* and *unstake()* that reverts if **block.timestamp** exceeds the user-specified **deadline**. This is standard practice in time-sensitive DeFi operations and allows users to bound the timing risk.

Team response

The risk requires a transaction to land within seconds of an epoch boundary due to mempool/sequencer delay, which is highly unlikely in practice. Adding a deadline parameter to *stake()* and *unstake()* would add calldata cost to every call and integration friction for a scenario that almost never materializes. Users should avoid submitting these transactions in the final seconds before Thursday 00:00 UTC.

TRST-L-2 Early exit penalties bypass FeeDistributor checkpoint, misallocating reward epochs

- **Category:** Logical flaws
- **Source:** vlSDT.sol
- **Status:** Acknowledged

Description

When a user calls *withdrawEarly()* on vlSDT, the penalty **SDT** is transferred directly to the **penaltyReceiver** (intended to be a *FeeDistributor*) via a raw *IERC20.transfer()*. The *FeeDistributor* documentation states that early-exit penalties are redistributed to remaining vlSDT holders.

```
**Early exit** is an optional path for users who don't want to wait the full 8 weeks.
If governance has enabled it (by setting a penalty receiver address), a user can
withdraw early by paying a linear time-decaying penalty. The penalty is at its maximum
(initially 25%, adjustable by governance) immediately after requesting unstake and
decays to zero at the original deadline. The penalty SDT is sent to the penalty
receiver (typically the FeeDistributor), redistributing it to remaining stakers.
Changes to the maximum penalty rate require a 7-day timelock, so users in the queue
cannot be surprised by a governance vote raising the exit cost.
```

However, the *FeeDistributor* only becomes aware of new tokens when *_checkpointToken()* is called. A raw transfer does not trigger this checkpoint. The penalty tokens sit in the *FeeDistributor*'s balance unaccounted until the next *checkpointToken()* call (either manual or via the 24-hour auto-checkpoint in *claim()*). When the checkpoint finally runs, it distributes those tokens proportionally across all elapsed time since the last checkpoint — potentially attributing them to epochs where the penalty didn't exist yet, or to a future epoch where the staker composition has changed.

```
    function deposit(uint256 amount) external nonReentrant whenNotKilled {
        require(amount != 0, ZeroAmount());
        require(VLSDT.totalSupply() > 0, NoStakers());

        REWARD_TOKEN.safeTransferFrom(msg.sender, address(this), amount);
        _checkpointToken();
        emit Deposited(msg.sender, amount);
    }
```

Recommended mitigation

The fix should be either:

1. Force the penalty receiver to implement *deposit().*
2. Add an **isFeeDistributor** bool to the penalty structure. If true, call *deposit()* instead of transferring the funds.

Team response

The penalty tokens are accounted for at the next checkpoint, which happens automatically within 24h via *claim()* auto-checkpoint, or immediately if anyone calls the permissionless *checkpointToken().* A bot watching **WithdrawnEarly** events is trivial to set up. The retroactive time-weighted spreading is the same behavior as any other token arrival, not specific to penalties.

Using *transfer()* instead of *deposit()* keeps the **penaltyReceiver** implementation-agnostic: governance can route penalties to a treasury, a different distributor, or any address without requiring it to implement the FeeDistributor interface.

TRST-L-3 Dust staking creates early checkpoint that causes display of wrong claimable amount

- **Category:** View-related issues
- **Source:** FeeDistributor.sol
- **Status:** Fixed

Description

The *_claimable()* function iterates through unclaimed epochs starting from the user's first vlSDT checkpoint, processing a maximum of **MAX_CLAIM_EPOCHS** (50) per call.

An attacker can grief a victim by calling *vlSDT.stake(1, victim)* in an early epoch, creating a checkpoint for the victim long before they actually participate. When the victim later stakes a meaningful amount (e.g., 200 epochs later), *claimable()* starts iterating from the dust-deposit epoch and exhausts the 50-epoch limit before reaching epochs with real rewards. The view function returns 0, causing frontends to display no claimable rewards.

```
    function _claimable(address user) internal view returns (uint256 amount, uint256
lastProcessedEpoch) {
        uint256 startEpoch = _getStartEpoch(user);
        if (startEpoch == 0) return (0, 0); // User has no checkpoints in vlSDT

        // Determine end epoch: lastTokenTime rounded down to epoch
        // We can only claim for epochs where tokens have been fully checkpointed
        uint256 endEpoch = _roundDownToEpoch(lastTokenTime);
        if (startEpoch >= endEpoch) return (0, 0);

        // Iterate through epochs
        // Only distribute if both user has balance and total supply > 0
        uint256 epochCursor = startEpoch;
        uint256 epochsProcessed;
        while (epochCursor < endEpoch && epochsProcessed < MAX_CLAIM_EPOCHS) {
```

Recommended mitigation

Document the **MAX_CLAIM_EPOCHS** limitation prominently under *claimable()*. Consider adding a *claimableWithLoop()* function that allows users (or frontends) to fetch the full amount offline in one call.

Team response

Fixed.

Mitigation review

The issue has been addressed by changing the *claimable()* method to account for possible dust attacks.

TRST-L-4 withdrawEarly() does not auto-apply pending penalty update, enabling rate selection

- **Category:** Time-sensitivity issues
- **Source:** vlBoost.sol
- **Status:** Acknowledged

Description

Governance can schedule a **maxPenaltyBps** update via *scheduleMaxPenaltyUpdate()*, which sets a **pendingMaxPenaltyBps** and an **effectiveTimestamp** (current time + **PENALTY_UPDATE_DELAY** of 7 days). The update only takes effect when someone calls *applyMaxPenaltyUpdate()* after the **effectiveTimestamp**.

```
    function scheduleMaxPenaltyUpdate(uint256 _newMaxPenaltyBps) external onlyOwner {
        require(_newMaxPenaltyBps <= 10_000, InvalidPenalty());

        uint48 effectiveAt = uint48(block.timestamp + PENALTY_UPDATE_DELAY);
        PenaltyConfig storage config = penaltyConfig;
        penaltyConfig.pendingMaxPenaltyBps = uint24(_newMaxPenaltyBps);
        penaltyConfig.effectiveTimestamp = effectiveAt;

        emit MaxPenaltyUpdateScheduled(uint256(config.maxPenaltyBps),
_newMaxPenaltyBps, uint256(effectiveAt));
    }
...
    function applyMaxPenaltyUpdate() external {
        PenaltyConfig memory config = penaltyConfig;

        uint256 effectiveTimestamp = uint256(config.effectiveTimestamp);
        require(effectiveTimestamp != 0, NoPendingUpdate());
        require(block.timestamp >= effectiveTimestamp, UpdateNotEffectiveYet());

        uint24 newMax = config.pendingMaxPenaltyBps;
        penaltyConfig = PenaltyConfig({
            penaltyReceiver: config.penaltyReceiver,
            maxPenaltyBps: newMax,
            pendingMaxPenaltyBps: 0,
            effectiveTimestamp: 0
        });

        emit MaxPenaltyUpdateApplied(uint256(config.maxPenaltyBps), uint256(newMax));
    }
```

The *withdrawEarly()* function reads **maxPenaltyBps** from the current **penaltyConfig** but does not check whether a pending update has become effective. This creates a race condition window after **effectiveTimestamp** where two different penalty rates coexist depending on whether *applyMaxPenaltyUpdate()* has been called.

```
    function withdrawEarly(uint256 id, address recipient) external nonReentrant {
        require(recipient != address(0), RecipientCannotBeZeroAddress());
```

```
        PenaltyConfig memory config = penaltyConfig;
```

A sophisticated user monitoring the mempool can exploit this: if the pending rate is lower, they call *applyMaxPenaltyUpdate()* before *withdrawEarly()*. If the current rate is lower, they call *withdrawEarly()* before anyone applies the update. This creates an information asymmetry that favors technically sophisticated users over regular participants.

Recommended mitigation

In *withdrawEarly()*, check if **block.timestamp** >= **effectiveTimestamp** and auto-apply the pending update before computing the penalty. This ensures all users see the same penalty rate and eliminates the race condition.

Team response

The race condition only exists if nobody calls *applyMaxPenaltyUpdate()* after **effectiveTimestamp**. Governance can eliminate it by applying the update promptly. Auto-applying in *withdrawEarly()* would add gas cost to every early withdrawal to cover a scenario governance can trivially prevent. We decide to pass on this finding.

TRST-L-5 Migrate event logs recipient instead of lock owner for migrateTo()

- **Category:** Event errors
- **Source:** veSDT.vy
- **Status:** Fixed

Description

The Migrate event is defined as **Migrate(sender: indexed(address), user: indexed(address), amount, ts)**. The second indexed parameter is intended to identify the user whose lock is being consumed. However, in *_migrate()*, the event logs **_recipient** as the second parameter: **log Migrate(_caller, _recipient, value, block.timestamp)**. For *migrateTo(recipient)* where **_recipient != _user**, this means the event identifies the recipient of the vlSDT position, not the veSDT lock owner whose funds were consumed. Off-chain indexers tracking migration status per user will fail to attribute the migration correctly.

```
    /// @notice Emitted when a user migrates to vlSDT
    /// @param sender The address that initiated the migration (user or admin)
    /// @param user The user whose lock was migrated
    /// @param amount The amount of SDT migrated
    /// @param ts The timestamp of the migration
    event Migrate(address indexed sender, address indexed user, uint256 amount,
uint256 ts);
...
def _migrate(_caller: address, _user: address, _recipient: address):
    ...
    log Migrate(_caller, _recipient, value, block.timestamp)
```

Recommended mitigation

Change the event to **log Migrate(_caller, _user, _recipient, value, block.timestamp)**, or emit a separate event for recipient-directed migrations that includes both addresses.

```
-   log Migrate(_caller, _recipient, value, block.timestamp)
+   log Migrate(_caller, _user, value, block.timestamp)
```

Team response

Fixed.

Mitigation review

Fixed as recommended.

TRST-L-6 DeployVLSDT script fails on Ownable2Step two-phase ownership transfer

- **Category:** Logical flaws
- **Source:** 03_MigrateVESDT.s.sol
- **Status:** Fixed

Description

vlSDT inherits *Ownable2Step* from OpenZeppelin, which requires a two-phase ownership transfer: *transferOwnership()* sets the pending owner, and *acceptOwnership()* must be called by the new owner to complete the transfer.

```
contract vlSDT is ERC20, Ownable2Step, ReentrancyGuard {
```

The deployment script calls *transferOwnership(governance)* and then asserts *vlsdt.owner()* == governance. This assertion will always fail because *owner()* still returns the deployer until governance calls *acceptOwnership()*. The script will revert, blocking the deployment pipeline.

```
        // 4. Transfer vlSDT ownership to DAO
        vlsdt.transferOwnership(governance);

        feeDistributorStartTime = feeDistributor.START_TIME();

        vm.stopBroadcast();

        require(vlsdt.owner() == governance, "vlSDT ownership not transferred to
DAO");
```

Recommended mitigation

Remove the **vlsdt.owner() == governance** assertion from the script. Instead, assert that **vlsdt.pendingOwner() == governance** after calling *transferOwnership()*. The actual ownership acceptance should be a separate governance action.

Team response

Fixed.

Mitigation review

Fixed as recommended.

TRST-L-7 Deployment script reverts when block.timestamp falls exactly on epoch boundary

- **Category:** Logical flaws
- **Source:** 03_MigrateVESDT.s.sol
- **Status:** Fixed

Description

The DeployVLSDT script computes **feeDistributorStartTime** from the *FeeDistributor*'s **START_TIME** (which is *_roundUpToEpoch(block.timestamp)*) and then requires **feeDistributorStartTime** > **block.timestamp**. The *_roundUpToEpoch()* function uses ceiling division: **((timestamp + EPOCH_DURATION - 1) / EPOCH_DURATION) * EPOCH_DURATION**. When **block.timestamp** is exactly on an epoch boundary (e.g., Thursday 00:00:00 UTC), this returns **block.timestamp** itself, causing **feeDistributorStartTime** == **block.timestamp** and failing the strict inequality check.

```
        feeDistributorStartTime = feeDistributor.START_TIME();

        vm.stopBroadcast();

        require(vlsdt.owner() == governance, "vlSDT ownership not transferred to
DAO");
        require(feeDistributor.owner() == governance, "FeeDistributor ownership not
transferred to DAO");
        require(feeDistributor.emergencyAddress() == emergencyAddress, "FeeDistributor
emergency address not set");
        require(address(feeDistributor.REWARD_TOKEN()) == Core.SDT, "FeeDistributor
reward token not set");
        require(feeDistributorStartTime > block.timestamp, "FeeDistributor start time
not in the future");
```

Recommended mitigation

Change the assertion to **feeDistributorStartTime** >= **block.timestamp**. The boundary case (deploying exactly at an epoch start), although unlikely, is a valid deployment scenario and should not cause a revert.

```
    constructor(address _owner, address _boostRegistry) ERC20("Vote Locked SDT",
"vlSDT") Ownable(_owner) {
        require(_boostRegistry != address(0), BoostRegistryCannotBeZeroAddress());
        BOOST_REGISTRY = _boostRegistry;
        _setUnstakeDelay(8 weeks);
        penaltyConfig = PenaltyConfig({
            penaltyReceiver: address(0), maxPenaltyBps: 2500, pendingMaxPenaltyBps: 0,
effectiveTimestamp: 0
        });
    }
```

Team response

Fixed.

Mitigation review

Fixed as recommended.

TRST-L-8 KillFeeDistributor script blocks repeat calls needed for post-kill token recovery

- **Category:** Logical flaws
- **Source:** 10_KillFeeDistributor.s.sol
- **Status:** Fixed

Description

The *KillFeeDistributor* script checks *isKilled()* and only calls *kill()* if the contract is not yet killed. However, the *FeeDistributor*'s *kill()* function does not revert on repeated calls — it sets **isKilled** = true (idempotent) and transfers the current **REWARD_TOKEN** balance to the emergency address. This is a deliberate design that allows recovery of tokens accidentally sent to an already-killed distributor.

```
        for (uint256 i; i < feeDistributors.length; i++) {
            FeeDistributor feeDistributor = FeeDistributor(feeDistributors[i]);
            require(!feeDistributor.isKilled(), "FeeDistributor already killed");

            feeDistributor.kill();
        }
```

Recommended mitigation

Remove the *isKilled()* guard from the script so it can be called multiple times to recover tokens sent to a killed *FeeDistributor*. The contract already handles the idempotent case correctly.

```
        for (uint256 i; i < feeDistributors.length; i++) {
            FeeDistributor feeDistributor = FeeDistributor(feeDistributors[i]);
-           require(!feeDistributor.isKilled(), "FeeDistributor already killed");

            feeDistributor.kill();
        }
```

Team response

Fixed.

Mitigation review

Fixed as recommended.

TRST-L-9 Round-up in _calculateEndtime() allows effective duration to exceed maxDuration

- **Category:** Logical flaws
- **Source:** BoostMarketplace.sol
- **Status:** Fixed

Description

The *_calculateEndtime()* function computes endtime as **((block.timestamp + durationWeeks * WEEK + WEEK - 1) / WEEK) * WEEK**, which rounds up to the next week boundary. This ensures the buyer receives at least durationWeeks of full boost coverage.

In *fillListing()*, the buyer specifies duration which is validated against **listing.maxDuration**. However, *_calculateEndtime()* can produce an effective duration up to nearly one week longer than the nominal duration. A buyer requesting duration = 2 at the start of an epoch receives approximately 3 weeks of effective boost — exceeding **listing.maxDuration**.

In *acceptOffer()*, the same applies in reverse: a seller accepting an offer with **offer.duration** = 2 may be forced to delegate boost for nearly 3 weeks, locking their voting power for longer than the offer specified.

```
    function _calculateEndtime(uint256 durationWeeks) internal view returns (uint256
endtime) {
        // 1. Compute the minimum acceptable endtime (now + requested duration).
        uint256 target = block.timestamp + durationWeeks * WEEK;
        // 2. Round up to next week boundary so buyer gets at least durationWeeks full
weeks.
        endtime = ((target + WEEK - 1) / WEEK) * WEEK;
    }
```

For sellers, this means their boost is locked longer than **listing.maxDuration** was designed to cap. For buyers filling offers, they receive more delegation than intended and pay proportionally more, although **maxTotalPayment** provides some protection.

This asymmetry is inherent to the round-up design and affects both sides of the market.

Recommended mitigation

Either cap the **endtime** to not exceed the listing's **maxDuration** in absolute terms, or document that the actual duration may be up to 1 week longer than the nominal value.

Team response

Fixed.

Mitigation review

Issue has been addressed as suggested.

TRST-L-10 Zero amount listings can be bypassed using updateListing()

- **Category:** Validation flaws
- **Source:** BoostMarketplace.sol
- **Status:** Fixed

Description

createListing() requires **amount != 0** as a dedicated check before calling *_validateAmountAndPrice()*. However, *updateListing()* omits this check — it only calls *_validateAmountAndPrice()*, which validates **amount** >= **minOrderAmount**.

```
    function createListing(
        uint256 amount,
        uint256 pricePerWeek,
        uint256 maxDuration,
        address paymentToken,
        uint256 expiry
    ) external nonReentrant returns (uint256 listingId) {
```

```
        // 1. Validate listing parameters.
        require(amount != 0, ZERO_AMOUNT());
```

If **minOrderAmount** is set to 0, a seller could call *updateListing()* with **newAmount** = 0, creating a zero-amount listing that bypasses the *createListing()* guard. The listing would be unfillable but would exist in storage.

```
    function updateListing(uint256 listingId, uint256 newAmount, uint256
newPricePerWeek) external nonReentrant {
        SellListing storage listing = _listings[listingId];

        // 1. Validate ownership and new listing parameters.
        require(listing.seller == msg.sender, NOT_OWNER());
        _validateAmountAndPrice(newAmount, newPricePerWeek);
```

In practice, **minOrderAmount** should ideally be non-zero in production, making this a theoretical rather than practical concern. However, the inconsistency between create and update validation is a code quality issue.

Recommended mitigation

Add an explicit **amount != 0** check to *updateListing()*.

Team response

Fixed.

Mitigation review

Fixed as recommended.

TRST-L-11 Listings lack minimum duration, limiting seller control over commitment length

- **Category:** Logical flaws
- **Source:** BoostMarketplace.sol
- **Status:** Fixed

Description

A **SellListing** specifies **maxDuration** (the longest delegation a buyer can request) but has no **minDuration** field. A seller who wants to delegate boost for a minimum of 4 weeks cannot enforce this — any buyer can fill the listing with **duration** = 1. This forces sellers who want longer-term commitments to repeatedly manage listings and rely on off-chain coordination, reducing marketplace efficiency.

Recommended mitigation

Add a **minDuration** field to the **SellListing** struct and validate **duration** >= **listing.minDuration** in *fillListing()*. This gives sellers control over the minimum commitment they accept.

Team response

Fixed.

Mitigation review

Fixed as recommended.

TRST-L-12 Partial fills can leave unfillable remainder below minFillAmount

- **Category:** Logical flaws
- **Source:** BoostMarketplace.sol
- **Status:** Fixed

Description

Every fill of a listing or offer is validated against **minFillAmount**. However, after multiple partial fills, the remaining unfilled amount may drop below **minFillAmount**. At that point, no one can fill the remainder (it fails the **minFillAmount** check), and the order is effectively stuck with a permanently unfillable dust tail (unless the item is manually updated).

Recommended mitigation

Enforce one of two options:

1. When fulfilling, do not allow a small remainder to be left unfilled.
2. When fulfilling, allow below the **minFillAmount** in case it fully fills the offer.

The second option is more user-friendly and is the standard approach in order book implementations. It preserves the minimum fill protection for large orders while allowing the final fill to clear the order completely.

Team response

Fixed.

Mitigation review

The second option has been taken and implemented correctly.

TRST-L-13 Attacker-planted checkpoint forces victim to iterate empty epochs on claim()

- **Category:** Logical flaws
- **Source:** FeeDistributor.sol
- **Status:** Fixed

Description

The *FeeDistributor*'s *claim()* iterates from the user's first vlSDT checkpoint through unclaimed epochs. The *_getStartEpoch()* function finds the user's first checkpoint via *VLSDT.numCheckpoints()* and *VLSDT.checkpoints(user, 0)*, then rounds up to the next epoch. An attacker can call *vlSDT.stake(1, victim)* in epoch 1, creating a checkpoint for the victim at minimal cost. When the victim joins in epoch 200 and calls *claim()*, the loop starts from epoch 1 and must iterate through 199 empty epochs (capped at 50 per call). The victim needs 4 separate *claim()* transactions, each consuming gas for 50 iterations of *vlSDT.balanceOfAt()* and *vlSDT.totalSupplyAt()* external calls, before reaching epochs with real rewards.

Recommended mitigation

Allow users to pass an optional **startEpoch** parameter to *claim()* that overrides the checkpoint-derived start. The function should validate that **startEpoch** >= *_getStartEpoch(user)* to prevent skipping earned rewards, but allow jumping past known-empty epochs. Frontends can compute the optimal start epoch off-chain.

Team response

Fixed.

Mitigation review

The issue has been addressed as recommended.

TRST-L-14 Governance can extend unstakeDelay between unstake submission and execution

- **Category:** Logical flaws
- **Source:** vlSDT.sol
- **Status:** Fixed

Description

When a user calls *unstake()*, the function reads the current **unstakeDelay** and computes a deadline as **block.timestamp** + **unstakeDelay**. Between the time the user submits the transaction and when it executes, the owner can call *setUnstakeDelay()* to increase the delay. The user's vlSDT is burned immediately, but they are now locked into a longer withdrawal period than anticipated. Since *setUnstakeDelay()* has no upper bound or timelock, governance could theoretically set an arbitrarily long delay. This flow does not need to be malicious, but a simple act of misfortune where the user did not expect a change at that moment.

Recommended mitigation

Either add a timelock mechanism to *setUnstakeDelay()* (similar to **PENALTY_UPDATE_DELAY** for penalty changes), or add a **maxDeadline** parameter to *unstake()* that reverts if the computed deadline exceeds the user's expectation. The latter approach is simpler and gives users direct control.

Team response

Fixed.

Mitigation review

Issue has been addressed as recommended. A minor note is that the same **PENALTY_UPDATE_DELAY** constant is used for the unstaking cooldown, so it should be renamed to a more general name like **UPDATE_DELAY** to be semantically correct.

Additional recommendations

TRST-R-1 FeeDistributor removes checkpoint rate limiting present in legacy version

In *FeeDistributor.sol*, anyone can call *checkpointToken()* at any time with no minimum interval between calls. This is a behavioral change from the legacy Curve-style *FeeDistributor.vy*, where the admin must first enable a **can_checkpoint_token** flag and calls have a minimum 1-day interval.

The removal of access control and rate limiting is intentional (simplifies operations), but it changes the security model: it enables the front-running behavior described in **TRST-M-5** and allows frequent checkpoint calls that fragment token distribution across tiny time slices.

Document this behavioral change prominently, particularly for integrators who may assume the legacy rate-limiting behavior.

TRST-R-2 Listing and offer IDs in BoostMarketplace share the same counter space

In *BoostMarketplace*, IDs are not unique. one ID can correspond to both a listing and an offer.

It is recommended to make IDs unique in *BoostMarketplace*.

TRST-R-3 Fully filled orders in BoostMarketplace can be reused via updateListing()

Orders in *BoostMarketplace* are not deleted even if fully fulfilled, the order owner can reuse the order by updating the amount and price.

The concern is that if the frontend prioritizes earlier orders when prices are the same, users could always gain priority by reusing orders.

It is recommended to delete the order once it is fully fulfilled.

TRST-R-4 Redundant code and unimplemented interface function

1. In the else branch below, **block.timestamp >= nextEpoch**, which is unreachable because **nextEpoch > time** (**time** belongs to the previous epoch).

```
            if (block.timestamp < nextEpoch) {
                // Current epoch (partial): allocate remaining time's share
                if (sinceLast == 0 && block.timestamp == time) {
                    // Edge case: called twice in same block
                    epochShare = toDistribute;
                } else {
                    epochShare = (toDistribute * (block.timestamp - time)) / sinceLast;
                }
                tokensPerEpoch[thisEpoch] += epochShare;
```

```
                distributed += epochShare;

                time = block.timestamp; // Mark that we reached current time
                break;
            } else {
                // Past epoch: allocate that epoch's time share
-               if (sinceLast == 0 && nextEpoch == time) {
-                   // Edge case: called twice in same block exactly at epoch boundary
-                   epochShare = toDistribute;
-               } else {
                    epochShare = (toDistribute * (nextEpoch - time)) / sinceLast;
                }
                tokensPerEpoch[thisEpoch] += epochShare;
                distributed += epochShare;
            }
```

2. *operatorAndDelegableBalance()* interface is not implemented.

```
    /// @notice Get operator approval and delegable balance in a single call.
    /// @param delegator The delegator address.
    /// @param operator The operator address.
    /// @return approved True if operator is approved for delegator.
    /// @return available Delegable balance for delegator.
    function operatorAndDelegableBalance(address delegator, address operator)
        external
        view
        returns (bool approved, uint256 available);
```

TRST-R-5 Documentation references outdated escrow model and incorrect epoch rounding

The buyer does not escrow funds, when the offer is accepted, the funds are sent directly from the buyer to the seller.

```
**Buy offers.** A buyer creates an offer specifying an amount, price, duration, payment token, expiry, and an optional recipient for the boost. The buyer escrows the worst-case payment upfront (calculated assuming the maximum effective duration due to week-boundary rounding). When a seller fills the offer, actual payment is computed from the real effective duration, and any excess escrow is refunded when the offer is fully filled.
```

When the timestamp is at the week boundary, the next epoch will not be used, instead, the current epoch will be used as **START_TIME**.

```
### FeeDistributor: Revenue Sharing

The FeeDistributor pays protocol fees to vlSDT holders, proportional to their share of total supply at each epoch's start.

**Epoch structure.** An epoch is 7 days, aligned to Thursday 00:00 UTC (because Unix timestamp 0 was a Thursday). The contract's `START_TIME` is set to the next epoch boundary at deployment, ensuring a clean first epoch.
```

TRST-R-6 Preserve original initialize() logic alongside v2 migration in veSDT

The *veSDT* contract has been upgraded, while the *initialize()* function has been changed to address the migration process. However, it is recommended to not remove old *initialize()* logic to maintain accurate account of contract state and history. Consider introducing the migration logic in an *initializeV2()* function, while keeping the old logic untouched.

Centralization risks

TRST-CR-1 Operators can redirect any user's unclaimed FeeDistributor rewards

The *FeeDistributor* exposes a *claim(address user, address receiver)* function gated by the *onlyOperator* modifier, which checks *VLSDT.isOperator(msg.sender).* An authorized operator can claim any user's accumulated rewards and direct them to any receiver address. This is designed for composition (e.g., the Router can claim on behalf of a user and pipe rewards into another action), but it means a compromised or malicious operator has unrestricted access to redirect all unclaimed rewards across all users. The operator set is managed by governance via *vlSDT.setOperatorAuthorization()*, so the risk is bounded by governance security, but users have no individual opt-out mechanism.

```
    function claim(address user, address receiver)
        external
        onlyOperator
        nonReentrant
        whenNotKilled
        returns (uint256 amount)
    {
        require(user != address(0), ZeroAddress());
        require(receiver != address(0), ZeroAddress());
        amount = _claim(user, receiver);
    }
...
    modifier onlyOperator() {
        require(VLSDT.isOperator(msg.sender), OnlyOperator());
        _;
    }
```

TRST-CR-2 Upgradeable veSDT proxy can mint unbounded vlSDT via migrateFrom()

vlSDT's *migrateFrom()* function mints vlSDT to any address and is gated solely by **require(msg.sender == VE_SDT)**. The veSDT contract is deployed behind a *TransparentUpgradeableProxy*, meaning the proxy admin can deploy a new implementation at any time.

A malicious or compromised proxy admin could deploy a new veSDT implementation that calls *vlSDT.migrateFrom(attacker, arbitraryAmount)* without transferring any SDT. This would mint unbounded vlSDT, diluting all existing holders' voting power and fee shares. The proxy admin is a multisig (**DAO.PROXY_ADMIN**), providing some protection, but this represents the single largest centralization vector in the system — it bypasses all other checks and can drain value from every vlSDT holder.

```
    function migrateFrom(address user, uint256 amount) external {
        require(amount > 0, AmountMustBeGreaterThanZero());
        require(msg.sender == VE_SDT, Unauthorized());

        _mint(user, amount);
        _checkpoint(user);

        emit Staked(VE_SDT, user, amount);
    }
```

Systemic risks

TRST-SR-1 Low vlSDT supply during early migration enables governance capture

During the migration window (Phase 3 of the governance proposal), vlSDT's **totalSupply** grows gradually as individual users migrate from veSDT. In the early stages, **totalSupply** may be very low relative to the total migrateable SDT. An attacker who stakes a relatively small amount of SDT can acquire a dominant share of vlSDT voting power. If Snapshot governance is switched to read vlSDT before sufficient migration volume, the attacker can single-handedly pass proposals. The governance proposal addresses this by specifying force-migration in Phase 4, but the Phase 3 window is inherently vulnerable. The severity depends on when Snapshot voting authority transitions from veSDT to vlSDT.

TRST-SR-2 Migration leaves stale veBoost delegations causing balance underflows

The veSDT migration clears the user's lock data (**locked[user] = {0, 0}**) and reduces veSDT supply, but does not interact with the old *veBoost* delegation registry. Any active delegations in *veBoost* remain in storage with non-zero amounts.

```
**`migrate()` transfers the entire locked SDT to vlSDT** regardless of remaining lock time. A user who locked for 4 years and has 3 years remaining gets their full SDT amount migrated, receiving equivalent vlSDT voting power immediately. The lock is cleared, the veSDT supply is decremented, and the SDT is sent directly to the vlSDT contract, which mints vlSDT to the user.
```

After migration, the delegator's veSDT balance is 0 but their **delegatedTotal** in *veBoost* is non-zero. The *veBoost._balance_of()* function computes **votingPower - delegatedAmount**, which underflows (Vyper's safe math will revert). Similarly, *veBoost*'s **totalSupply**() still delegates to *veSDT.totalSupply()*, which decreases with each migration while boost totals do not, causing aggregate balance to exceed total supply.

```
@view
@external
def delegable_balance(_user: address) -> uint256:
    point: Point = self._checkpoint_read(_user, True)
    return VotingEscrow(VE).balanceOf(_user) - (point.bias - point.slope * (block.timestamp - point.ts))
```

While the SDGP-63 proposal states that governance will transition to *vlSDT*/*vlBoost* and the old *veBoost* will be deprecated, any integration or contract still reading *veBoost* balances (e.g., gauge contracts not yet migrated) will encounter reverts or incorrect values during the transition period. This should be communicated to all integrators before migration begins.

TRST-SR-3 Non-standard ERC20 tokens break FeeDistributor and BoostMarketplace accounting

The *FeeDistributor*'s reward token and the *BoostMarketplace*'s payment tokens are assumed to be standard ERC20 tokens with exact transfer semantics. Fee-on-transfer tokens would break the *FeeDistributor*'s **tokenLastBalance** accounting (the contract would believe it received more tokens than it did), leading to reverts when distributing. Rebasing tokens would cause similar accounting drift. The *BoostMarketplace* explicitly documents this assumption in its NatSpec, but the *FeeDistributor* only notes it in passing. If non-standard tokens are ever used, a dedicated integration audit is required.