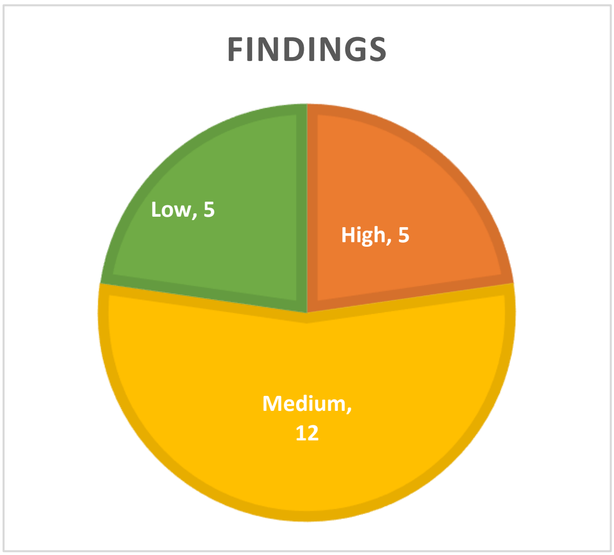
Trust Security

Smart Contract Audit

StakeDAO OnlyBoost V2

11/04/25

Executive summary

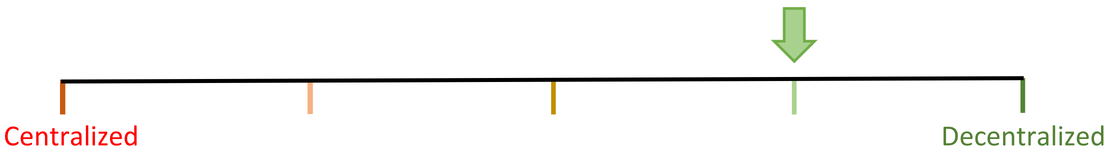


Category	Boost Yield Aggregator
Audited file count	15
Lines of Code	1656
Auditor	Trust cccz
Time period	31/03/2025-11/04/2025

Findings

Severity	Total	Fixed	Open	Acknowledged
High	5	-	-	-
Medium	12	-	-	-
Low	5	-	-	-

Centralization score



Signature

Digitally signed by Trust
Date: 2025.04.11 17:59:46 +03'00'

Document properties

Versioning

Version	Date	Description
0.1	11/04/2025	Client report

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/strategies/src/RewardVault.sol
- packages/strategies/src/Accountant.sol
- packages/strategies/src/Strategy.sol
- packages/strategies/src/ProtocolController.sol
- packages/strategies/src/Factory.sol
- packages/strategies/src/Allocator.sol
- packages/strategies/src/Sidecar.sol
- packages/strategies/src/ProtocolContext.sol
- packages/strategies/src/RewardReceiver.sol
- packages/strategies/src/SidecarFactory.sol
- packages/strategies/src/integrations/curve/ConvexSidecarFactory.sol
- packages/strategies/src/integrations/curve/CurveStrategy.sol
- packages/strategies/src/integrations/curve/ConvexSidecar.sol
- packages/strategies/src/integrations/curve/CurveFactory.sol
- packages/strategies/src/integrations/curve/CurveAllocator.sol

Repository details

- **Repository URL:** https://github.com/stake-dao/contracts-monorepo
- **Commit hash:** e56978ec5f34d82bfb65558d0c3bbd0a944a6a95

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

High severity findings

TRST-H-1 Incorrect update of netCredited results in reward loss

- **Category:** Accounting issues
- **Source:** Accountant.sol
- **Status:** Open

Description

When the Vault uses sync mode to update rewards, in *Accountant*, synced but unharvested rewards are stored in the **VaultData** structure. These rewards have been used to increase **integral**, so they will not be counted again during harvest or next sync. Also, since *harvest()* claims all rewards, the protocol needs to reset the **VaultData** to indicate that there are no unclaimed rewards.

The problem here is that **VaultData.netCredited** is not reset, it indicates the part of unclaimed rewards that have been used to increase **integral**, and since it is not reset, then the new rewards are treated as already used.

Consider pending rewards of 1000 CRV, all in LOCKER, **protocolFee** is 2%, **harvesterFee** is 1%, **TRIGGER_HARVEST** is false (sync mode).

1. User deposits, in *checkpoint()*, **totalAmount** == **feeSubjectAmount** == 1000, **protocolFee** == 20, **harvesterFee** == 10, **netCredited** = 970.
2. User calls *harvest()*, **totalAmount** == **feeSubjectAmount** == 1000, **protocolFee** == 20, **harvesterFee** == 10, **newNet** == 970, **_vault.netCredited** is still 970.
3. After that 30 CRV rewards are generated.
4. User calls *harvest()*, **totalAmount** == **feeSubjectAmount** == 30, **protocolFee** == 0.6, **harvesterFee** == 0.3, **newNet** == 29.1.
5. **newNet** < **oldNet**(970) does not update **integral**, in fact the **integral** should increase because there are new rewards.

Recommended mitigation

It is recommended to reset **netCredited** in *_harvest()*, and also take into account the case where **oldNet** is greater than **newNet** (due to rounding in TRST-L-4).

```
            uint256 newNet = pendingRewards.totalAmount - protocolFee - harvesterFee;
            uint256 oldNet = _vault.netCredited;

            if (newNet > oldNet) {
                uint256 netDelta = newNet - oldNet;
                // Add only that delta to the integral
                _vault.integral += netDelta.mulDiv(SCALING_FACTOR, _vault.supply);
            }

            // Update the net credited so far
-           _vault.netCredited = newNet.toUint128();
+           _vault.netCredited = newNet >= oldNet ? 0 : oldNet - newNet;

            // Always clear pending rewards after harvesting
            _vault.feeSubjectAmount = 0;
            _vault.totalAmount = 0;
```

Team response

TBD

TRST-H-2 Precision loss in calculating rewardRate results in reward loss

- **Category:** Precision loss issues
- **Source:** RewardVault.sol
- **Status:** Open

Description

The extra rewards generated by staking will be claimed to *RewardReceiver*, and then anyone can distribute the rewards to users through *RewardReceiver.distributeRewardToken()*. The problem is in *RewardVault.depositRewards()*, the **rewardRate** is calculated by direct division of the amount and length (7 days).

```
        if (currentTime >= periodFinish) {
            newRewardRate = _amount / DEFAULT_REWARDS_DURATION;
```

Due to precision loss, each distribution will result in a maximum loss of 604799 wei. When the extra reward token is WBTC (8 decimals), each distribution will result in a maximum loss of 0.0064 WBTC, about 500 USD.

When rewards are accrued, it will multiply by the time elapsed. So, in essence there is a division before multiply causing precision loss.

```
        if (timeDelta > 0 && _totalSupply > 0) {
            // Calculate additional rewards per token since last update
            rewardRatePerToken = (timeDelta * reward.rewardRate * 1e18) /
_totalSupply;
        }
```

Recommended mitigation

It is recommended to scale by 1e18 when calculating **rewardRate** to reduce precision loss.

```
        if (currentTime >= periodFinish) {
-            newRewardRate = _amount / DEFAULT_REWARDS_DURATION;
+            newRewardRate = _amount * 1e18 / DEFAULT_REWARDS_DURATION;
        } else {
            uint32 remainingTime = periodFinish - currentTime;
-            uint128 remainingRewards = remainingTime * reward.rewardRate;
-            newRewardRate = (_amount + remainingRewards) / DEFAULT_REWARDS_DURATION;
+            uint128 remainingRewards = remainingTime * reward.rewardRate / 1e18;
+            newRewardRate = (_amount + remainingRewards) * 1e18 /
DEFAULT_REWARDS_DURATION;
        }
...
        if (timeDelta > 0 && _totalSupply > 0) {
            // Calculate additional rewards per token since last update
-            rewardRatePerToken = (timeDelta * reward.rewardRate * 1e18) /
_totalSupply;
+           rewardRatePerToken = (timeDelta * reward.rewardRate) / _totalSupply;
        }
```

Team response

TBD

TRST-H-3 Share transfer does not consider the latest rewards resulting in loss of rewards

- **Category:** Logical flaws
- **Source:** RewardVault.sol
- **Status:** Open

Description

When minting or burning shares, the protocol syncs or harvests the pending rewards and updates the accountant.

However when transferring shares, the protocol doesn't do that, it simply considers the pending rewards to be 0 and updates the accountant. Once any pending reward exists at transfer, this will cause the **address(from)** to lose the reward and the **address(to)** to gain the reward.

```
    function _update(address from, address to, uint256 amount) internal override {
        // 1. Update Balances via Accountant
        ACCOUNTANT.checkpoint(
            gauge(), from, to, uint128(amount),
IStrategy.PendingRewards({feeSubjectAmount: 0, totalAmount: 0}), false
        );
```

Recommended mitigation

It is recommended to withdraw 0 shares in *_update()* to get the pending rewards and call *ACCOUNTANT.checkpoint()* accordingly.

```
    function _update(address from, address to, uint256 amount) internal override {
+       IAllocator.Allocation memory allocation = IAllocator.Allocation({asset:
asset(), gauge: gauge()});
+       IStrategy.PendingRewards memory pendingRewards =
strategy().withdraw(allocation, TRIGGER_HARVEST, receiver);
        // 1. Update Balances via Accountant
        ACCOUNTANT.checkpoint(
-           gauge(), from, to, uint128(amount),
IStrategy.PendingRewards({feeSubjectAmount: 0, totalAmount: 0}), false
+           gauge(), from, to, uint128(amount), pendingRewards, TRIGGER_HARVEST
        );
```

Team response

TBD

TRST-H-4 Checkpoint after balance change results in loss of rewards

- **Category:** Logical flaws
- **Source:** RewardVault.sol
- **Status:** Open

Description

When transferring shares, in *_update()*, *ACCOUNTANT.checkpoint()* is called to checkpoint the rewards in *ACCOUNTANT* and update the user's balance, and then *_checkpoint()* is called to checkpoint the extra rewards.

```
    function _update(address from, address to, uint256 amount) internal override {
        // 1. Update Balances via Accountant
        ACCOUNTANT.checkpoint(
            gauge(), from, to, uint128(amount),
IStrategy.PendingRewards({feeSubjectAmount: 0, totalAmount: 0}), false
        );

        // 2. Update Reward State
        _checkpoint(from, to);

        // 3. Emit Transfer event
        emit Transfer(from, to, amount);
    }
```

Since the extra rewards are checkpointed after the balance changes, this will result in the extra rewards for **address(from)** being lost and **address(to)** being eligible for those rewards.

Recommended mitigation

It is recommended to checkpoint the extra rewards before the balance changes.

```
    function _update(address from, address to, uint256 amount) internal override {
+       _checkpoint(from, to);
        // 1. Update Balances via Accountant
        ACCOUNTANT.checkpoint(
            gauge(), from, to, uint128(amount),
IStrategy.PendingRewards({feeSubjectAmount: 0, totalAmount: 0}), false
        );

        // 2. Update Reward State
-       _checkpoint(from, to);

        // 3. Emit Transfer event
        emit Transfer(from, to, amount);
    }
```

Team response

TBD

TRST-H-5 Malicious users can drain all CRV rewards by exploiting a shutdown Gauge

- **Category:** Logical flaws
- **Source:** Strategy.sol
- **Status:** Open

Description

After shutdown, *Strategy.withdraw()* will force to sync even if **TRIGGER_HARVEST** is true.

```
    function withdraw(IAllocator.Allocation memory allocation, bool doHarvest, address
receiver)
        external
        override
        onlyVault(allocation.gauge)
        returns (PendingRewards memory pendingRewards)
    {
        /// If the pool is shutdown, return the pending rewards.
        /// Use the shutdown function to withdraw the funds.
        if (PROTOCOL_CONTROLLER.isShutdown(allocation.gauge)) {
            return _sync(allocation.gauge);
        }
```

Note that in *RewardVault._burn()*, it will call *ACCOUNTANT.checkpoint()* with **harvested** == true when **TRIGGER_HARVEST** is true.

```
        IStrategy.PendingRewards memory pendingRewards =
strategy().withdraw(allocation, TRIGGER_HARVEST, receiver);

        // Burn the shares by calling the endpoint function of the accountant contract
        _burn(owner, shares, pendingRewards, TRIGGER_HARVEST);
...
    function _burn(address from, uint256 amount, IStrategy.PendingRewards memory
pendingRewards, bool harvested)
        internal
    {
        ACCOUNTANT.checkpoint(gauge(), from, address(0), uint128(amount),
pendingRewards, harvested);
    }
```

These rewards are always considered harvested in *ACCOUNTANT.checkpoint()* but actually they are not. This causes the **integral** to always grow each time the user calls *RewardVault.withdraw()*.

```
            if (harvested && newRewards > 0) {
                // Calculate total fees in one operation
                // We charge only protocol fee on the harvested rewards.
                if (newFeeSubjectAmount > 0) {
                    totalFees = newFeeSubjectAmount.mulDiv(getProtocolFeePercent(),
1e18).toUint128();
                    // Update protocol fees accrued.
                    protocolFeesAccrued += totalFees;
                }

                // Update integral with new rewards per token
                integral += (newRewards - totalFees).mulDiv(SCALING_FACTOR, supply);
            }
```

Note that since all gauge rewards are in *Accountant*, if one of the gauges is shutdown, the attacker can exploit it to drain all CRV rewards.

Also, even if the reward for that gauge is up to date, the attacker can send CRV to the sidecar to "forge" rewards, and then use those sent CRV as rewards in *_sync()*. These rewards will not be actually claimed to *Accountant*, and the user can repeatedly withdraw 0 shares to inflate **integral**.

```
    function _sync(address gauge) internal view override returns (PendingRewards
memory pendingRewards) {
        ...
            } else {
                // For sidecar contracts, use their getPendingRewards() function
                pendingRewardsAmount = ISidecar(target).getPendingRewards();
            }

            pendingRewards.totalAmount += pendingRewardsAmount.toUint128();
        }
...
    function getPendingRewards() public view override returns (uint256) {
        return baseRewardPool().earned(address(this)) +
REWARD_TOKEN.balanceOf(address(this));
    }
```

Recommended mitigation

It is recommended to choose to call *_harvest()* or *_sync()* according to **doHarvest** in *Strategy.withdraw()* even after the gauge shutdown.

```
        if (PROTOCOL_CONTROLLER.isShutdown(allocation.gauge)) {
-            return _sync(allocation.gauge);
+            return doHarvest ? _harvest(allocation.gauge, "", false) :
_sync(allocation.gauge);
        }
```

Team response

TBD

Medium severity findings

TRST-M-1 The syncRewardTokens() function will never work, making users miss out on new rewards

- **Category:** Logical flaws
- **Source:** Factory.sol
- **Status:** Open

Description

When creating the Vault, *_setupRewardTokens()* will be called to set the reward tokens. In addition, when the reward tokens in the gauge are updated, *syncRewardTokens()* will also call *_setupRewardTokens()* to synchronize the latest reward tokens.

But the problem here is *_setupRewardTokens()* will add CVX and the previous reward tokens to the reward tokens again, which will cause the **RewardAlreadyExists** error to be thrown. Due to inability to sync new reward tokens, this will result in new reward tokens not being distributed.

```
    function _setupRewardTokens(address _vault, address _gauge, address
_rewardReceiver) internal virtual override {
        /// Then we add the extra reward token to the reward distributor through the
strategy.
        IRewardVault(_vault).addRewardToken(CVX, _rewardReceiver);
...
    function addRewardToken(address rewardsToken, address distributor) external
onlyRegistrar {
        require(distributor != address(0), ZeroAddress());
        require(rewardTokens.length < MAX_REWARD_TOKEN_COUNT,
MaxRewardTokensExceeded());

        RewardData storage reward = rewardData[rewardsToken];
        require(_isRewardToken(reward) == false, RewardAlreadyExists());
```

Recommended mitigation

It is recommended to skip adding already existing reward tokens in *_setupRewardTokens()*.

```
    function _setupRewardTokens(address _vault, address _gauge, address
_rewardReceiver) internal virtual override {
        /// Then we add the extra reward token to the reward distributor through the
strategy.
+       if(!IRewardVault(_vault).isRewardToken(CVX))
        IRewardVault(_vault).addRewardToken(CVX, _rewardReceiver);

        /// Check if the gauge supports extra rewards.
        /// This function is not supported on all gauges, depending on when they were
deployed.
        bytes memory data = abi.encodeWithSignature("reward_tokens(uint256)", 0);

        (bool success,) = _gauge.call(data);
        if (!success) return;

        /// Loop through the extra reward tokens.
        /// 8 is the maximum number of extra reward tokens supported by the gauges.
        for (uint8 i = 0; i < 8; i++) {
            /// Get the extra reward token address.
            address _extraRewardToken = ILiquidityGauge(_gauge).reward_tokens(i);
            /// If the address is 0, it means there are no more extra reward tokens.
            if (_extraRewardToken == address(0)) break;
+           if(IRewardVault(_vault).isRewardToken(_extraRewardToken)) continue;
            /// Performs checks on the extra reward token.
            /// Checks like if the token is also an lp token that can be staked in the
locker, these tokens are not supported.
            if (_isValidToken(_extraRewardToken)) {
                /// Then we add the extra reward token to the reward distributor
through the strategy.
                IRewardVault(_vault).addRewardToken(_extraRewardToken,
_rewardReceiver);
            }
        }

        /// Set RewardReceiver as RewardReceiver on Gauge.
        data = abi.encodeWithSignature("set_rewards_receiver(address)",
_rewardReceiver);
        require(_executeTransaction(_gauge, data), SetRewardReceiverFailed());
    }
```

Note that for gauge, except for V2, it is not allowed to overwrite previous reward tokens, so *syncRewardTokens()* does not need to delete previous reward tokens. To support V2 in the future, the code would need to claim the pending rewards first, then delete the previous reward tokens and add new reward tokens.

Team response

TBD

TRST-M-2 CurveAllocator.getAllocationTargets() makes the protocol unavailable when no sidecar is available

- **Category:** Logical flaws
- **Source:** CurveAllocator.sol
- **Status:** Open

Description

When sidecar is **address(0)**, *CurveAllocator.getDepositAllocation()* and *getWithdrawalAllocation()* both return the result with only one valid target.

```
    function getDepositAllocation(address asset, address gauge, uint256 amount)
        public
        view
        override
        returns (Allocation memory)
    {
        /// 1. Get the sidecar for the gauge.
        address sidecar = CONVEX_SIDECAR_FACTORY.sidecar(gauge);

        /// 2. If the sidecar is not set, use the default allocation.
        if (sidecar == address(0)) {
            return super.getDepositAllocation(asset, gauge, amount);
        }
```

However, *getAllocationTargets()* does not take this case into account. This results in when the sidecar is **address(0)**, *getAllocationTargets()* returning a target with **address(0)**, which prevents functions such as *deposit()/withdraw()* from working.

```
    function getAllocationTargets(address gauge) public view override returns
(address[] memory) {
        address sidecar = CONVEX_SIDECAR_FACTORY.sidecar(gauge);

        address[] memory targets = new address[](2);
        targets[0] = sidecar;
        targets[1] = LOCKER;

        return targets;
    }
```

This could happen when a Curve gauge is deployed without a matching Convex pool.

Recommended mitigation

```
    function getAllocationTargets(address gauge) public view override returns
(address[] memory) {
        address sidecar = CONVEX_SIDECAR_FACTORY.sidecar(gauge);
+       if (sidecar == address(0)) {
+           return super.getAllocationTargets(gauge);
+       }
        address[] memory targets = new address[](2);
        targets[0] = sidecar;
        targets[1] = LOCKER;

        return targets;
    }
```

Team response

TBD

TRST-M-3 When extra Reward tokens include CRV there will be double-counting of rewards

- **Category:** Accounting issues
- **Source:** CurveFactory.sol
- **Status:** Open

Description

When the vault is created, the extra reward tokens in the gauge are synced as the extra reward tokens of the Vault.

```
        for (uint8 i = 0; i < 8; i++) {
            /// Get the extra reward token address.
            address _extraRewardToken = ILiquidityGauge(_gauge).reward_tokens(i);

            /// If the address is 0, it means there are no more extra reward tokens.
            if (_extraRewardToken == address(0)) break;

            /// Performs checks on the extra reward token.
            /// Checks like if the token is also an lp token that can be staked in the
locker, these tokens are not supported.
```

```
            if (_isValidToken(_extraRewardToken)) {
                /// Then we add the extra reward token to the reward distributor 
through the strategy.
                IRewardVault(_vault).addRewardToken(_extraRewardToken, 
_rewardReceiver);
            }
        }
```

In *_isValidToken()* it only filters the case where the reward token is CVX because CVX will always be added to the reward token, but does not consider the case where the reward token is CRV.

```
    function _isValidToken(address _token) internal view virtual override returns 
(bool) {
        /// We already add CVX to the vault by default.
        if (_token == CVX) return false;

        /// If the token is available as an inflation receiver, it's not valid.
        try GAUGE_CONTROLLER.gauge_types(_token) {
            return false;
        } catch {
            return true;
        }
    }
```

Since gauge does not restrict CRV from being added as reward token, once CRV is used as extra reward token, it will be confused with **REWARD_TOKEN**, especially in Sidecar, where *claimExtraRewards()* will send the **REWARD_TOKEN** tokens to *rewardReceiver* and distribute to users. However, these CRV may have been used in sync mode to increase the integral.

```
    function claimExtraRewards() external {
        address[] memory extraRewardTokens = getRewardTokens();

        /// We can save gas by not claiming extra rewards if we don't need them, 
there's no extra rewards, or not enough rewards worth to claim.
        if (extraRewardTokens.length > 0) {
            baseRewardPool().getReward(address(this), true);
        }

        /// It'll claim rewardToken but we'll leave it here for clarity until the 
claim() function is called by the strategy.
        baseRewardPool().getReward(address(this), true);

        /// Send the reward token to the reward receiver.
        CVX.safeTransfer(rewardReceiver(), CVX.balanceOf(address(this)));

        /// Handle the extra reward tokens.
        for (uint256 i = 0; i < extraRewardTokens.length;) {
            uint256 _balance = IERC20(extraRewardTokens[i]).balanceOf(address(this));
            if (_balance > 0) {
                /// Send the whole balance to the strategy.
                IERC20(extraRewardTokens[i]).safeTransfer(rewardReceiver(), _balance);
            }

            unchecked {
                ++i;
            }
        }
    }
```

Recommended mitigation

It is recommended to skip adding CRV as extra reward token in *CurveFactory._isValidToken()*. It can use the *isValidToken()* implementation of the Factory parent contract which already performs the check.

Team response

TBD

TRST-M-4 Rebalance may fail due to zero-sized deposit

- **Category:** Logical flaws
- **Source:** Strategy.sol
- **Status:** Open

Description

When depositing into the sidecar, it will eventually call *BaseRewardPool.stakeFor(),* which does not allow 0 amount deposits.

```
    function _deposit(uint256 amount) internal override {
        /// Deposit the LP token into Convex and stake it (true) to receive rewards.
        BOOSTER.deposit(pid(), amount, true);
    }
...
    function deposit(uint256 _pid, uint256 _amount, bool _stake) public returns(bool){
        ...
        address token = pool.token;
        if(_stake){
            //mint here and send to rewards on user behalf
            ITokenMinter(token).mint(address(this),_amount);
            address rewardContract = pool.crvRewards;
            IERC20(token).safeApprove(rewardContract,0);
            IERC20(token).safeApprove(rewardContract,_amount);
            IRewards(rewardContract).stakeFor(msg.sender,_amount);
...
    function stakeFor(address _for, uint256 _amount)
        public
        updateReward(_for)
        returns(bool)
    {
        require(_amount > 0, 'RewardPool : Cannot stake 0');
```

This case is handled in *Strategy.deposit()*, where the *Sidecar.deposit()* is skipped when the allocated amount is 0.

```
        for (uint256 i = 0; i < allocation.targets.length; i++) {
            if (allocation.amounts[i] > 0) {
                if (allocation.targets[i] == LOCKER) {
                    _deposit(allocation.asset, allocation.gauge,
allocation.amounts[i]);
                } else {
                    ISidecar(allocation.targets[i]).deposit(allocation.amounts[i]);
                }
            }
        }
```

However, *Strategy.rebalance()* doesn't handle it, which causes *Strategy.rebalance()* to fail due to depositing 0 amount to the sidecar.

```
        for (uint256 i = 0; i < allocation.targets.length; i++) {
            address target = allocation.targets[i];
            uint256 amount = allocation.amounts[i];

            asset.safeTransfer(target, amount);

            if (target == LOCKER) {
                _deposit(address(asset), gauge, amount);
            } else {
                ISidecar(target).deposit(amount);
            }
        }
```

Recommended mitigation

It is recommended to skip 0 amount deposits in *Strategy.rebalance()*.

```
        /// 7. Deposit the amounts into the gauge with new allocations
        for (uint256 i = 0; i < allocation.targets.length; i++) {
            address target = allocation.targets[i];
            uint256 amount = allocation.amounts[i];
+           if(amount == 0) continue;
            asset.safeTransfer(target, amount);

            if (target == LOCKER) {
                _deposit(address(asset), gauge, amount);
            } else {
                ISidecar(target).deposit(amount);
            }
        }
```

Team response

TBD

TRST-M-5 Malicious user can block setting Locker as allocation target for gauge

- **Category:** Logical flaws
- **Source:** ConvexSidecarFactory.sol
- **Status:** Open

Description

CurveFactory.create() is used to create Vault and Sidecar, and set the allocation target for gauge. It will first call *createVault()* to create the Vault.

```
    function create(uint256 _pid) external returns (address vault, address
rewardReceiver, address sidecar) {
        (,, address gauge,,,) = IBooster(BOOSTER).poolInfo(_pid);

        /// 1. Create the vault.
        (vault, rewardReceiver) = createVault(gauge);

        /// 2. Attach the sidecar.
        sidecar = ISidecarFactory(CONVEX_SIDECAR_FACTORY).create(gauge,
abi.encode(_pid));

        /// 3. Set the valid allocation target.
        PROTOCOL_CONTROLLER.setValidAllocationTarget(gauge, LOCKER);
        PROTOCOL_CONTROLLER.setValidAllocationTarget(gauge, sidecar);
        emit VaultDeployed(gauge, vault, rewardReceiver, sidecar);
    }
```

The problem here is that *createVault()* is a public function, and for one gauge, it can only be called once, otherwise it will throw an **AlreadyDeployed** error.

```
    function createVault(address gauge) public virtual returns (address vault, address
rewardReceiver) {
        /// Perform checks on the gauge to make sure it's valid and can be used
        require(_isValidGauge(gauge), InvalidGauge());
        require(_isValidDeployment(gauge), InvalidDeployment());
        require(PROTOCOL_CONTROLLER.vaults(gauge) == address(0), AlreadyDeployed());
...
        _registerVault(gauge, vault, asset, rewardReceiver);
...
    function _registerVault(address gauge, address vault, address asset, address
rewardReceiver) internal {
        PROTOCOL_CONTROLLER.registerVault(gauge, vault, asset, rewardReceiver,
PROTOCOL_ID);
    }
```

Once a malicious user calls *createVault()*, *SidecarFactory.create()* can be called separately to create the sidecar (and it will set the sidecar as the allocation target), but LOCKER cannot be set as the allocation target.

Recommended mitigation

It is recommended to set Locker as the allocation target in *SidecarFactory.create()* and to add the LOCKER immutable in *ConvexSidecarFactory* or *SidecarFactory*. Also remove the code that sets the allocation target in *CurveFactory.create()*.

```
        ConvexSidecar(sidecarAddress).initialize();

+       PROTOCOL_CONTROLLER.setValidAllocationTarget(gauge, Locker);
        // Set the valid allocation target
        PROTOCOL_CONTROLLER.setValidAllocationTarget(gauge, sidecarAddress);

        return sidecarAddress;
```

```
…
    function create(uint256 _pid) external returns (address vault, address
rewardReceiver, address sidecar) {
        (,, address gauge,,,) = IBooster(BOOSTER).poolInfo(_pid);

        /// 1. Create the vault.
        (vault, rewardReceiver) = createVault(gauge);

        /// 2. Attach the sidecar.
        sidecar = ISidecarFactory(CONVEX_SIDECAR_FACTORY).create(gauge,
abi.encode(_pid));

        /// 3. Set the valid allocation target.
-       PROTOCOL_CONTROLLER.setValidAllocationTarget(gauge, LOCKER);
-       PROTOCOL_CONTROLLER.setValidAllocationTarget(gauge, sidecar);

        emit VaultDeployed(gauge, vault, rewardReceiver, sidecar);
    }
```

Team response

TBD

TRST-M-6 The new gauge cannot be deployed on the new strategy

- **Category:** Integration issues
- **Source:** CurveFactory.sol
- **Status:** Open

Description

When creating the Vault, *_isValidDeployment()* requires that the gauge has been shutdown on the old strategy. This results in new gauges that were not deployed on the old strategy not being deployed on the new strategy.

```
    function createVault(address gauge) public virtual returns (address vault, address
rewardReceiver) {
        /// Perform checks on the gauge to make sure it's valid and can be used
        require(_isValidGauge(gauge), InvalidGauge());
        require(_isValidDeployment(gauge), InvalidDeployment());
…
    function _isValidDeployment(address _gauge) internal view virtual override returns
(bool) {
        return IStrategy(OLD_STRATEGY).isShutdown(_gauge);
    }
```

Recommended mitigation

It is recommended to add the *exists()* function on the old strategy to allow deployment when the gauge does not exist on the old strategy.

Team response

TBD

TRST-M-7 CurveAllocator.getWithdrawalAllocation() should not prioritize withdrawing the entire amount from Locker

- **Category:** Logical flaws
- **Source:** CurveAllocator.sol
- **Status:** Open

Description

When **balanceOfLocker** is greater than **optimalBalanceOfLocker**, *CurveAllocator.getWithdrawalAllocation()* will prioritize withdrawals from Locker, but in fact only the excess part should be prioritized, and the rest should still be prioritized from sidecar.

```
        } else {
            /// 7c. If Stake DAO balance is above optimal, prioritize withdrawing from
Stake DAO
            amounts[1] = Math.min(amount, balanceOfLocker);
            amounts[0] = amount > amounts[1] ? amount - amounts[1] : 0;
        }
```

Recommended mitigation

It is recommended to prioritize only the excess part.

```
        } else {
            /// 7c. If Stake DAO balance is above optimal, prioritize withdrawing from
Stake DAO
-           amounts[1] = Math.min(amount, balanceOfLocker);
-           amounts[0] = amount > amounts[1] ? amount - amounts[1] : 0;
+           amounts[1] = Math.min(amount, balanceOfLocker - optimalBalanceOfLocker);
+           amounts[0] = amount > amounts[1] ? Math.min(amount - amounts[1],
balanceOfSidecar) : 0;
+           if(amount > amounts[0] + amounts[1]) {
+               amounts[1] += amount - amounts[0] - amounts[1];
+           }
        }
```

In addition, since both deposits and withdrawals affect TVL, this will cause the result of *getOptimalLockerBalance()* obtained before deposits and withdrawals to be different from the actual **OptimalLockerBalance** after deposits and withdrawals. One option is to deposit the part exceeding the **OptimalLockerBalance** into Locker and Sidecar separately and to maintain correct proportion following the equilibrium formula.

Team response

TBD

TRST-M-8 Vault is not compliant with EIP4626

- **Category:** Compliance issues
- **Source:** RewardVault.sol
- **Status:** Open

Description

RewardVault intends to be EIP4626 compliant, but its *maxDeposit()* and *maxMint()* functions return **type(unit256).max** directly.

```
function maxDeposit(address) public pure returns (uint256) {
    return type(uint256).max;
}
/// @notice Returns the maximum amount of shares that can be minted.
/// @dev Due to the 1:1 relationship between assets and shares, the max mint
///      is the same as the max deposit.
/// @param __ This parameter is not used and is included to satisfy the interface.
Pass whatever you want to.
/// @return _ The maximum amount of shares that can be minted.
function maxMint(address __) external pure returns (uint256) {
    return maxDeposit(__);
}
```

The EIP4626 specs states:

> Maximum amount of the underlying asset that can be deposited into the Vault for the receiver, through a deposit call.

MUST return the maximum amount of assets deposit would allow to be deposited for receiver and not cause a revert, which MUST NOT be higher than the actual maximum that would be accepted (it should underestimate if necessary). This assumes that the user has infinite assets, i.e. MUST NOT rely on balanceOf of asset.

MUST factor in both global and user-specific limits, like if deposits are entirely disabled (even temporarily) it MUST return 0.

The specs are not followed, the max deposit of **type(uint256).max** would cause a revert.

Recommended mitigation

It is recommended to change the *maxDeposit()* function to be compliant with the specs.

Team response

TBD

TRST-M-9 Depositing rewards will lower the previous reward rate

- **Category:** DoS attacks
- **Source:** RewardVault.sol
- **Status:** Open

Description

The behavior in *depositRewards()* is, when new rewards come in, they are added to the current unpaid rewards and re-stretched to the **DEFAULT_REWARDS_DURATION.**

```
        if (currentTime >= periodFinish) {
            newRewardRate = _amount / DEFAULT_REWARDS_DURATION;
        } else {
            uint32 remainingTime = periodFinish - currentTime;
            uint128 remainingRewards = remainingTime * reward.rewardRate;
            newRewardRate = (_amount + remainingRewards) / DEFAULT_REWARDS_DURATION;
        }
```

Consider *depositRewards()* is called to deposit 7000 reward tokens, the **rewardRate** is 1000/day.

3 days later, *depositRewards()* is called to deposit new 1000 reward tokens, now the remaining reward tokens are 4000 and the new **rewardRate** is 5000/7 = 714/day, reducing the reward rate from 1000/day to 714/day. The method can be repeated to delay an ever-decreasing subset of the rewards indefinitely.

Recommended mitigation

One option is to maintain a list of reward rates that change based on timestamps, so that when new rewards are deposited, it only affects reward rate afterward. For example, in the above example, day0 through day7 have a reward rate of 1000, and when 1000 new reward tokens are deposited three days later, the reward rate for day4 through day7 would become 1143, and the reward rate for day8 through day10 would be 143.

Team response

TBD

TRST-M-10 Strategy.rebalance() allocates incorrect amount

- **Category:** Logical flaws
- **Source:** Strategy.sol
- **Status:** Open

Description

In *Strategy.rebalance()*, *getRebalancedAllocation()* is called before *_withdrawFromAllTargets()*.

```
        IAllocator.Allocation memory allocation =
            IAllocator(allocator).getRebalancedAllocation(address(asset), gauge,
currentBalance);

        /// 5. Ensure the allocation has more than one target.
        require(allocation.targets.length > 1, RebalanceNotNeeded());

        /// 6. Withdraw all assets from all targets to this contract
        _withdrawFromAllTargets(address(asset), gauge, allocation.targets,
address(this));
```

The *getRebalancedAllocation()* function directly calls *getDepositAllocation()*, in which the allocation amount is adjusted according to the balance of Locker. In fact, during rebalance, the balance of Locker should be considered to be 0, but since *_withdrawFromAllTargets()* is called later, the balance of Locker is inaccurate at this time.

```
        uint256 balanceOfLocker = IBalanceProvider(gauge).balanceOf(LOCKER);

        /// 5. Get the optimal amount of lps that must be held by the locker.
        uint256 optimalBalanceOfLocker = getOptimalLockerBalance(gauge);

        /// 6. Calculate the amount of lps to deposit into the locker.
        amounts[1] =
            optimalBalanceOfLocker > balanceOfLocker ? Math.min(optimalBalanceOfLocker
- balanceOfLocker, amount) : 0;

        /// 7. Calculate the amount of lps to deposit into the sidecar.
        amounts[0] = amount - amounts[1];

        /// 8. Return the allocation.
        return Allocation({asset: asset, gauge: gauge, targets: targets, amounts:
amounts});
    }
```

Recommended mitigation

It is recommended to call *getRebalancedAllocation()* after calling *_withdrawFromAllTargets()*. The change should be made in tandem with the TRST-M-7 fixes.

```
    function rebalance(address gauge) external {
        /// 1. Get the allocator.
        address allocator = PROTOCOL_CONTROLLER.allocator(PROTOCOL_ID);

        /// 2. Get the asset.
        IERC20 asset = IERC20(PROTOCOL_CONTROLLER.asset(gauge));
        /// 3. Snapshot the current balance.
        uint256 currentBalance = balanceOf(gauge);

+       address[] memory targets = IAllocator(allocator).getAllocationTargets(gauge);
+       _withdrawFromAllTargets(address(asset), gauge, targets, address(this));

        /// 4. Get the allocation amounts for the gauge.
        IAllocator.Allocation memory allocation =
            IAllocator(allocator).getRebalancedAllocation(address(asset), gauge, 
currentBalance);

        /// 5. Ensure the allocation has more than one target.
        require(allocation.targets.length > 1, RebalanceNotNeeded());

        /// 6. Withdraw all assets from all targets to this contract
-       _withdrawFromAllTargets(address(asset), gauge, allocation.targets, 
address(this));
```

Team response

TBD

TRST-M-11 Convex-only extra reward tokens inwill be stuck in RewardReceiver

- **Category:** Integration issues
- **Source:** RewardReceiver.sol, ConvexSidecar.sol
- **Status:** Open

Description

ConvexSidecar.claimExtraRewards() will send all extra reward tokens in the Convex reward pool to *RewardReceiver*, but *RewardReceiver* will only distribute Curve *gauge.reward_tokens()* and CVX to users.

```
    function claimExtraRewards() external {
        address[] memory extraRewardTokens = getRewardTokens();

        /// We can save gas by not claiming extra rewards if we don't need them, 
there's no extra rewards, or not enough rewards worth to claim.
        if (extraRewardTokens.length > 0) {
            baseRewardPool().getReward(address(this), true);
        }

        /// It'll claim rewardToken but we'll leave it here for clarity until the 
claim() function is called by the strategy.
        baseRewardPool().getReward(address(this), true);

        /// Send the reward token to the reward receiver.
        CVX.safeTransfer(rewardReceiver(), CVX.balanceOf(address(this)));

        /// Handle the extra reward tokens.
        for (uint256 i = 0; i < extraRewardTokens.length;) {
            uint256 _balance = IERC20(extraRewardTokens[i]).balanceOf(address(this));
            if (_balance > 0) {
                /// Send the whole balance to the strategy.
                IERC20(extraRewardTokens[i]).safeTransfer(rewardReceiver(), _balance);
            }

            unchecked {
                ++i;
            }
        }
    }
```

Considering that the Convex admin may add other extra rewards to the Convex reward pool, this will cause these reward tokens in the Convex reward pool but not in the gauge to be stuck in *RewardReceiver*.

Recommended mitigation

It is recommended to send these reward tokens that are in the Convex reward pool but not in the gauge to other recipients.

Team response

TBD

TRST-M-12 Changes of the harvest fee will cause accounting inaccuracies

- **Category:** Configuration issues
- **Source:** Accountant.sol
- **Status:** Open

Description

In sync mode, **HarvestFee** is reserved for each checkpoint, but it waits until *harvest()* is called to collect them. This brings up the following issues:

1. If the owner changes the **harvestFeePercent** between syncing and harvesting, the fees would be too large or too small.
2. In *harvest()*, the effective **harvesterFee** which depends on the *Accountant* balance, would be lower. Therefore, too many rewards are reserved.

Recommended mitigation

For 1, one option is to document that the owner should harvest all gauges before changing **harvestFeePercent**. It is difficult to ensure this has been done for all gauges on-chain.

For 2, it is recommended to record the reserved **harvestFee** when syncing and collect them in *harvest()*.

Team response

TBD

Low severity findings

TRST-L-1 The default receiver of RewardVault.withdraw() should be msg.sender

- **Category:** Logical flaws
- **Source:** RewardVault.sol
- **Status:** Open

Description

The default receiver for *RewardVault.withdraw()* is the **owner**, which may cause the **msg.sender**'s allowance to be spent, but the withdrawal will be given to the **owner**.

Recommended mitigation

It is recommended to set the default receiver to **msg.sender** in *RewardVault.withdraw()*.

```
    function withdraw(uint256 assets, address receiver, address owner) public returns
(uint256) {
-       if (receiver == address(0)) receiver = owner;
+       if (receiver == address(0)) receiver = msg.sender;
        if (msg.sender != owner) {
            uint256 allowed = allowance(owner, msg.sender);
            if (assets > allowed) revert NotApproved();
            if (allowed != type(uint256).max) _spendAllowance(owner, msg.sender,
assets);
```

Team response

TBD

TRST-L-2 ProtocolController.onlyRegistrar() should allow owner access

- **Category:** Logical flaws
- **Source:** ProtocolController.sol
- **Status:** Open

Description

ProtocolController documents *onlyRegistrar()* should handle owner, but it doesn't.

```
    /// @notice Modifier to restrict function access to registrars or owner
    /// @custom:reverts OnlyRegistrar if the caller is not a registrar
    modifier onlyRegistrar() {
        require(registrar[msg.sender], OnlyRegistrar());
        _;
    }
```

It should be same as *onlyPermissionSetter()*.

```
    /// @notice Modifier to restrict function access to permission setters or owner
    /// @custom:reverts NotPermissionSetter if the caller is not a permission setter
    modifier onlyPermissionSetter() {
        require(permissionSetters[msg.sender] || msg.sender == owner(),
NotPermissionSetter());
        _;
    }
```

Recommended mitigation

It is recommended to change as follows.

```
    modifier onlyRegistrar() {
-       require(registrar[msg.sender], OnlyRegistrar());
+       require(registrar[msg.sender] || msg.sender == owner(), OnlyRegistrar());
        _;
    }
```

Team response

TBD

TRST-L-3 Pending extra rewards are not considered when depositing and withdrawing

- **Category:** Logical flaws
- **Source:** RewardVault.sol
- **Status:** Open

Description

Until *distributeRewardToken()* is called to deposit extra reward tokens, they are not tracked when depositing and withdrawing. A whale can call *deposit()* followed by *distributeRewardToken()* to receive rewards earned before their deposit. Note that the protocol will distribute the rewards linearly over 7 days so attackers are not as incentivized to perform the attack as otherwise.

Recommended mitigation

One option is to track pending rewards in *RewardVault* to prevent users from manipulating reward rate.

Team response

TBD

TRST-L-4 Rounding error in HarvestFee causes dust insolvency

- **Category:** Logical flaws
- **Source:** Accountant.sol
- **Status:** Open

Description

In *Accountant.checkpoint()* with sync mode, it decrements the credited rewards by the future harvest fee.

```
                // Get harvest fee for the unclaimed rewards.
                totalFees += newRewards.mulDiv(getHarvestFeePercent(),
1e18).toUint128();
```

```
            // The net rewards we are *actually crediting* now
            uint128 netIncrement = newRewards - totalFees;
```

This would behave correctly if *harvest()* is called immediately after, but if not, there is a risk that rounding would cause it to increment too much.

Consider at time T, the **totalFees** = 100.6, rounded to 100. Then at time T+1, **totalFees** = 100.6 again, rounded to 100. Then it decrements rewards by 200 total.

Now imagine that *harvest()* only occurs at time T+1. At that point, the **totalFees** in *harvest()* will be 201.2, rounded to 201. The code has deducted 200 but paid 201, so its balance is 1 wei short.

Recommended mitigation

The recommendation for TRST-H-1 could also fix it, and another option is to consider rounding up the **HarvestFee**.

Team response

TBD

TRST-L-5 Claiming in sync mode may fail due to insufficient balance

- **Category:** Logical flaws
- **Source:** Accountant.sol
- **Status:** Open

Description

When updating the *Accountant* in sync mode, if the reward is more than **MIN_MEANINGFUL_REWARDS**, the **integral** is updated. At this point users are eligible to claim them, but they have not yet been harvested into *Accountant*. Then, when users call *Accountant.claim()*, they would not be able to claim their rightful rewards.

Note that there is an option to pass **harvestData** to call *claim()*, but even if the user correctly harvests from their own gauge, the reward may still not be enough, because others could have claimed and taken the rewards from their gauge, while their gauges were unharvested. The only way is ensuring reward is to enumerate on all gauges and force a harvest.

Recommended mitigation

It is recommended to maintain a list for each gauge's harvested rewards so that users are guaranteed to claim only their own gauge's harvested rewards.

Team response

TBD

Additional recommendations

TRST-R-1 Check if the Gauge is shutdown in Strategy.rebalance()

Strategy.rebalance() can still be called after shutdown, although it won't actually rebalance because **currentBalance** is 0, it will emit the *Rebalance* event. It is recommended to check if the gauge is shutdown in *Strategy.rebalance()*.

```
    function rebalance(address gauge) external {

+       require(!PROTOCOL_CONTROLLER.isShutdown(gauge), GaugeShutdown());
        /// 1. Get the allocator.
        address allocator = PROTOCOL_CONTROLLER.allocator(PROTOCOL_ID);
```

TRST-R-2 ProtocolController.setStrategy() should perform gateway toggle_approve_mint() call on the new strategy

In the deployment script, the deployer calls *_enableModule()* to allow gateway calls from *curveStrategy* and *curveFactory*. They will manually call *toggle_approve_mint()* to allow *curveStrategy* to call *mint_for()* to claim CRV.

In *ProtocolController.setStrategy()*, the new strategy is set. At this time, it needs to first call *toggle_approve_mint()* on the old strategy to turn off approval, and call *toggle_approve_mint()* on the new strategy to turn on approval.

More specifically, it needs to call *disableModule()* to turn off approval of the old strategy to execute gateway calls and call *enableModule()* to turn on approval of new strategy to execute gateway calls.

```
        /// 7. Deploy Allocator.
        curveAllocator = new CurveAllocator(address(protocolController), locker,
address(convexSidecarFactory));

        /// 6. Setup the allocator in the protocol controller.
        protocolController.setAllocator(PROTOCOL_ID, address(curveAllocator));

+       _enableModule(address(protocolController));

        /// 7. Setup the strategy in the protocol controller.
        protocolController.setStrategy(PROTOCOL_ID, address(curveStrategy));
...
    function setStrategy(bytes4 protocolId, address _strategy) external onlyOwner {
        require(_strategy != address(0), ZeroAddress());
+       bytes memory data = abi.encodeWithSignature("toggle_approve_mint(address)",
address(_protocolComponents[protocolId].strategy));
+       if(_protocolComponents[protocolId].strategy != address(0))
_executeTransaction(address(MINTER), data);
        _protocolComponents[protocolId].strategy = _strategy;
+       data = abi.encodeWithSignature("toggle_approve_mint(address)",
address(_strategy));
+       _executeTransaction(address(MINTER), data);
        emit ProtocolComponentSet(protocolId, COMPONENT_ID_STRATEGY, _strategy);
    }
```

TRST-R-3 Add a check when creating sidecar in SidecarFactory.create()

Currently, when *SidecarFactory.create()* is called multiple times, it will revert because the sidecar has already been created.

```
    function create(address gauge, bytes memory args) public virtual override returns
(address sidecarAddress) {
        // Validate the gauge and args
        _isValidGauge(gauge, args);

        // Create the sidecar
        sidecarAddress = _create(gauge, args);

        // Store the sidecar address
        sidecar[gauge] = sidecarAddress;

        emit SidecarCreated(gauge, sidecarAddress, args);
    }
```

It would be dangerous if it was allowed to be called repeatedly, as it would overwrite **sidecar[gauge]**, causing loss of assets in the sidecar.

```
        bytes memory data = abi.encodePacked(lpToken, rewardReceiver, baseRewardPool,
pid);

        // Create a deterministic salt based on the token and gauge
        bytes32 salt = keccak256(data);

        // Clone the implementation contract
        sidecarAddress = Clones.cloneDeterministicWithImmutableArgs(IMPLEMENTATION,
data, salt);
```

Currently this is not a problem as the sidecar creation parameters are fixed, but this may be broken in future updates. It is recommended to require the sidecar to be not yet created in *SidecarFactory.create()*.

```
    function create(address gauge, bytes memory args) public virtual override returns
(address sidecarAddress) {
+       require(sidecar[gauge] == address(0));
        // Validate the gauge and args
        _isValidGauge(gauge, args);

        // Create the sidecar
        sidecarAddress = _create(gauge, args);

        // Store the sidecar address
        sidecar[gauge] = sidecarAddress;
```

TRST-R-4 Should check rewardsDistributor before depositing rewards into RewardVault

The *RewardReceiver.distributeRewards()* will look over the vault's reward tokens and call *depositRewards()* on it for every reward token that it has a positive balance for.

However, only the registered distributor may deposit rewards for a particular token.

```
    function depositRewards(address _rewardsToken, uint128 _amount) external {
        // Ensure all reward states are current
        _checkpoint(address(0), address(0));

        RewardData storage reward = rewardData[_rewardsToken];
        if (reward.rewardsDistributor != msg.sender) revert
UnauthorizedRewardsDistributor();
```

Therefore, if the vault has different distributors, an attacker could donate 1 wei of an inappropriate asset for each distributor to make the whole *distributeRewards()* fail. Currently that is not a problem since there can only be one distributor per vault.

It is recommended to check the **rewardsDistributor** of *rewardVault.rewardData(token)* in *RewardReceiver._depositRewards()*.

```
    function _depositRewards(IERC20 token, uint128 amount) internal {

+       if(rewardVault().rewardData(token).rewardsDistributor != address(this))
return;
        /// Approve the reward vault to spend the reward token.
        token.safeIncreaseAllowance(address(rewardVault()), amount);

        // Deposit rewards to the vault
        rewardVault().depositRewards(address(token), amount);
    }
```

TRST-R-5 Unsafe transfer is used in claimProtocolFee()

There is an unsafe transfer in *claimProtocolFees()*. Other usages of REWARD_TOKEN use *safeTransfer()*, so it is recommended to also use it to support non-conforming tokens.

```
    function claimProtocolFees() external nonReentrant {
        // get the fee receiver from the protocol controller and check that it is
valid
        address feeReceiver = PROTOCOL_CONTROLLER.feeReceiver(PROTOCOL_ID);
        require(feeReceiver != address(0), NoFeeReceiver());

        // get the protocol fees accrued until now and reset the stored value
        uint256 currentAccruedProtocolFees = protocolFeesAccrued;
        protocolFeesAccrued = 0;

        // transfer the accrued protocol fees to the fee receiver and emit the claim
event
-       IERC20(REWARD_TOKEN).transfer(feeReceiver, currentAccruedProtocolFees);
+       IERC20(REWARD_TOKEN).safeTransfer(feeReceiver, currentAccruedProtocolFees);
        emit ProtocolFeesClaimed(currentAccruedProtocolFees);
    }
```

TRST-R-6 The getWithdrawalAllocation() function should revert when the balance is insufficient

When the *RewardVault* calls *getWithdrawalAllocation()* in *_withdraw()*, it assumes that the assets amount is withdrawn into the Vault.

In the case of *CurveAllocator*, if **totalBalance** less than **amount**, it just withdraws all balance.

```
if (totalBalance <= amount) {
    /// 7a. If the total balance is less than or equal to the withdrawal amount,
withdraw everything
    amounts[0] = balanceOfSidecar;
    amounts[1] = balanceOfLocker;
```

In case of *Allocator*, it will assume amount is available.

```
address[] memory targets = new address[](1);
targets[0] = LOCKER;
uint256[] memory amounts = new uint256[](1);
amounts[0] = amount;
return Allocation({asset: asset, gauge: gauge, targets: targets, amounts: amounts});
```

For robustness, it is recommended to revert in both allocators when amount exceeds the balance, and add shutdown handling.

```
+       if(PROTOCOL_CONTROLLER.isShutdown(gauge)
+           return Allocation({asset: asset, gauge: gauge, targets: targets, amounts:
amounts});
        if (totalBalance <= amount) {
            /// 7a. If the total balance is less than or equal to the withdrawal
amount, withdraw everything
-           amounts[0] = balanceOfSidecar;
-           amounts[1] = balanceOfLocker;
+           revert;
        } else if (optimalBalanceOfLocker >= balanceOfLocker) {
            /// 7b. If Stake DAO balance is below optimal, prioritize withdrawing from
Convex
            amounts[0] = Math.min(amount, balanceOfSidecar);
            amounts[1] = amount > amounts[0] ? amount - amounts[0] : 0;
        } else {
            /// 7c. If Stake DAO balance is above optimal, prioritize withdrawing from
Stake DAO
            amounts[1] = Math.min(amount, balanceOfLocker);
            amounts[0] = amount > amounts[1] ? amount - amounts[1] : 0;
        }
```

TRST-R-7 Redundant claim behavior

In *RewardVault._claim()*, it first updates all the rewards.

```
    function _claim(address accountAddress, address[] calldata tokens, address
receiver)
        internal
        returns (uint256[] memory amounts)
    {
        if (receiver == address(0)) receiver = accountAddress;

        // Update all reward states before processing claims
        _checkpoint(accountAddress, address(0));
```

Then, it ignores the values of **rewardPerTokenPaid** and **claimable** which are already fresh, and runs the calculation again.

```
            // Calculate earned rewards since last claim
            AccountData storage account = accountData[accountAddress][rewardToken];
            uint256 accountEarned = _earned(accountAddress, rewardToken,
account.claimable, account.rewardPerTokenPaid);
```

```
            if (accountEarned == 0) continue;

            // Reset claimable amount and update reward checkpoint
            account.rewardPerTokenPaid = rewardPerToken(rewardToken);
            account.claimable = 0;
```

It is recommended to delete the redundant code in *RewardVault._claim()*.

```
            AccountData storage account = accountData[accountAddress][rewardToken];
-           uint256 accountEarned = _earned(accountAddress, rewardToken,
account.claimable, account.rewardPerTokenPaid);
+           uint256 accountEarned = account.claimable;
            if (accountEarned == 0) continue;

            // Reset claimable amount and update reward checkpoint
-           account.rewardPerTokenPaid = rewardPerToken(rewardToken);
            account.claimable = 0;
```

TRST-R-8 Allowing active claiming all extra rewards on strategy

Sidecar.claimExtraRewards() allows active claiming of extra rewards.

For Locker, the extra rewards will be accrued to the *rewardReceiver* when deposit/withdraw is made, and users will need to call *claim_rewards(Locker)* in *gauge* to claim the extra rewards.

It is recommended to implement a function in the *strategy* that calls *gauge.claim_rewards(Locker)* and *Sidecar.claimExtraRewards()* to claim all the extra rewards.

```
@external
@nonreentrant('lock')
def claim_rewards(_addr: address = msg.sender, _receiver: address = ZERO_ADDRESS):
    """
    @notice Claim available reward tokens for `_addr`
    @param _addr Address to claim for
    @param _receiver Address to transfer rewards to - if set to
                     ZERO_ADDRESS, uses the default reward receiver
                     for the caller
    """
    if _receiver != ZERO_ADDRESS:
        assert _addr == msg.sender  # dev: cannot redirect when claiming for another
user
    self._checkpoint_rewards(_addr, self.totalSupply, True, _receiver)
```

TRST-R-9 Potential division by zero in getOptimalLockerBalance()

In *getOptimalLockerBalance()*, the case where **veBoostOfConvex** is 0 needs to be handled to avoid a divide-by-zero error.

```
    function getOptimalLockerBalance(address gauge) public view returns (uint256
balanceOfLocker) {
        // 1. Get the balance of veBoost on Stake DAO and Convex
        uint256 veBoostOfLocker =
IBalanceProvider(BOOST_DELEGATION_V3).balanceOf(LOCKER);
        uint256 veBoostOfConvex =
IBalanceProvider(BOOST_DELEGATION_V3).balanceOf(CONVEX_BOOST_HOLDER);

        // 2. Get the balance of the liquidity gauge on Convex
        uint256 balanceOfConvex =
IBalanceProvider(gauge).balanceOf(CONVEX_BOOST_HOLDER);

        // 3. If there is no balance of Convex, return 0
        if (balanceOfConvex == 0) {
            return 0;
        }
+       if (veBoostOfConvex == 0) {
+           return type(uint256).max;
+       }
        // 4. Compute the optimal balance for Stake DAO
        balanceOfLocker = balanceOfConvex.mulDiv(veBoostOfLocker, veBoostOfConvex);
    }
```

TRST-R-10 Future compiler dirty bytes may cause revert

The contract reads bytes of *cloneWithImmutableArgs()* in assembly.

```
    /// @notice Staking token address.
    function asset() public view override returns (IERC20 _asset) {
        bytes memory args = Clones.fetchCloneArgs(address(this));
        assembly {
            _asset := mload(add(args, 20))
        }
    }
```

Note that it read the last 12 bytes of the bytes size followed by 20 packed bytes of address. It is read into an address type, meaning there is cleaning of upper bits.

The behavior is described in the Solidity docs. As it states, higher bits are currently zeroed, but it may cause an exception in the future.

It is recommended to load it into a uint256 and to trim the bytes by hand before storing in **_asset**.

TRST-R-11 Add check for discontinued gauges without is_killed() interface

Currently when the gauge has the *is_killed()* interface, it requires *is_killed()* to be false, but when the gauge does not have the *is_killed()* interface, no related check is implemented.

When the gauge does not have the *is_killed()* interface, consider checking the gauge's weight to determine whether the gauge is live or discontinued.

TRST-R-12 Repeated initialization in Factory

The *Factory* initializes the *ProtocolContext* variables.

```
        PROTOCOL_CONTROLLER = IProtocolController(_protocolController);

        ACCOUNTANT = PROTOCOL_CONTROLLER.accountant(PROTOCOL_ID);
        REWARD_TOKEN = IAccountant(ACCOUNTANT).REWARD_TOKEN();
```

But they are already initialized to the same values in the parent *ProtocolContext* constructor.

```
    constructor(bytes4 _protocolId, address _protocolController, address _locker,
address _gateway) {
        require(_protocolController != address(0) && _gateway != address(0),
ZeroAddress());

        GATEWAY = _gateway;
        PROTOCOL_ID = _protocolId;
        ACCOUNTANT = IProtocolController(_protocolController).accountant(_protocolId);
        REWARD_TOKEN = IAccountant(ACCOUNTANT).REWARD_TOKEN();
        PROTOCOL_CONTROLLER = IProtocolController(_protocolController);

        // In some cases (L2s), the locker is the same as the gateway.
        if (_locker == address(0)) {
            LOCKER = GATEWAY;
        } else {
            LOCKER = _locker;
        }
    }
```

It is recommended to delete the repeated initialization code in *Factory*.

```
    constructor(
        address _protocolController,
        address _vaultImplementation,
        address _rewardReceiverImplementation,
        bytes4 _protocolId,
        address _locker,
        address _gateway
    ) ProtocolContext(_protocolId, _protocolController, _locker, _gateway) {
        require(
            _protocolController != address(0) && _vaultImplementation != address(0)
                && _rewardReceiverImplementation != address(0),
            ZeroAddress()
        );

        REWARD_VAULT_IMPLEMENTATION = _vaultImplementation;
        REWARD_RECEIVER_IMPLEMENTATION = _rewardReceiverImplementation;
-       PROTOCOL_CONTROLLER = IProtocolController(_protocolController);

-       ACCOUNTANT = PROTOCOL_CONTROLLER.accountant(PROTOCOL_ID);
-       REWARD_TOKEN = IAccountant(ACCOUNTANT).REWARD_TOKEN();
    }
```

TRST-R-13 Skip zero transfer in RewardVault._deposit()

In *RewardVault._deposit()*, it loops over allocation targets and performs the transfer. It should skip 0 amount to save gas.

```
    function _deposit(address account, address receiver, uint256 assets, uint256
shares) internal {
        // Update the reward state for the receiver
        _checkpoint(receiver, address(0));

        // Get the address of the allocator contract from the protocol controller
        // then fetch the recommended deposit allocation from the allocator
        IAllocator.Allocation memory allocation =
allocator().getDepositAllocation(asset(), gauge(), assets);
```

```
        // Get the address of the asset from the clone's immutable args then for each
target recommended by
        // the allocator, transfer the amount from the account to the target
        IERC20 _asset = IERC20(asset());
        for (uint256 i; i < allocation.targets.length; i++) {
+           if(allocation.amounts[i] == 0) continue;
            require(PROTOCOL_CONTROLLER.isValidAllocationTarget(gauge(),
allocation.targets[i]), TargetNotApproved());
            SafeERC20.safeTransferFrom(_asset, account, allocation.targets[i],
allocation.amounts[i]);
        }
```

TRST-R-14 Strategy.shutdown() can be called multiple times

Strategy.shutdown() in theory should only be called once, to clean up the balance in the targets.

However, a user could donate 1 wei to the Locker to make the balance positive. It will cause the donated balance to be withdrawn to vault, and the Shutdown() event will be emitted.

It is recommended to enforce one shutdown per gauge through an **isShutdown** mapping inside the *Strategy*.

TRST-R-15 Improved state checks

It is recommended to require the new state to be different from the old state in the following functions.

```
function setValidAllocationTarget(address _gauge, address _target) external
onlyRegistrar {
    _isValidAllocationTargets[_gauge][_target] = true;
}
…
function removeValidAllocationTarget(address _gauge, address _target) external
onlyRegistrar {
    _isValidAllocationTargets[_gauge][_target] = false;
}
…
    function shutdown(address _gauge) external onlyOwner {
        gauge[_gauge].isShutdown = true;
        emit GaugeShutdown(_gauge);
    }
…
    function shutdownProtocol(bytes4 protocolId) external onlyOwner {
        _protocolComponents[protocolId].isShutdown = true;
        emit ProtocolShutdown(protocolId);
    }
```

TRST-R-16 ProtocolController.setAccountant() does not enable the new Accountant

The **ACCOUNTANT** is hard coded in construction of *ProtocolContext* and *RewardVault*, *Sidecar*. If **_protocolComponents[protocolId].accountant** is updated in *ProtocolController.setAccountant()*, by intention the contracts like RewardVault and Strategy will still use the old accountant.

However this means that also newly created vaults will use the old accountant, because they are cloned with the old immutable accountant inside.

If the protocol intends newly created Vaults to use the new accountant, it is recommended to store **_protocolComponents[protocolId].accountant** when cloning and use it instead of the hard-coded **ACCOUNTANT**. If that isn't done, to use a new accountant for new vaults will require deployment of almost all contracts, including the RewardVault, Factory, SidecarFactory, and CurveAllocator.

Also, each **protocolId** has only one corresponding *Strategy*. The *harvest()* and *flush()* functions in *Strategy* have the *onlyAccountant* modifier and *Strategy* assumes that there is only one accountant, so it is recommended to change the *Strategy* to support multiple accountants.

TRST-R-17 Sync relies on deposit and withdraw to update rewards

In sync mode, for pending rewards in Locker, the protocol always assumes that **integrate_fraction** is up to date. This assumption relies on deposit and withdraw always calling *gauge._checkpoint()* to update **integrate_fraction**. In future iterations of the code, this assumption may be broken and rewards may be lost.

It is recommended to execute *gauge.user_checkpoint(address(this))* in sync mode to update **integrate_fraction**.

```
            if (target == LOCKER) {
                // Calculate pending rewards for the locker by comparing  total earned
by gauge with already minted tokens
+               ILiquidityGauge(gauge).user_checkpoint(address(this))
                pendingRewardsAmount =
                    ILiquidityGauge(gauge).integrate_fraction(LOCKER) -
IMinter(MINTER).minted(LOCKER, gauge);

                pendingRewards.feeSubjectAmount += pendingRewardsAmount.toUint128();
```

TRST-R-18 Mixing harvest() and sync() will break accountant

The protocol currently uses the **TRIGGER_HARVEST** immutable to indicate that the accountant is always in harvest mode or always in sync mode. Once the protocol mixes harvest and sync in the future, it will cause core problems in *Accountant.checkpoint()*.

1. **Underflow issue:**
   Consider the pending rewards are 1000 CRV and the fee is 0.
   1. The first *checkpoint()* is called in sync mode, **_vault.totalAmount** is updated to 1000.
   2. The second *checkpoint()* is called in harvest mode, 1000 CRV rewards are collected, and **_vault.totalAmount** is still 1000.
   3. Then 1 CRV of rewards is generated.
   4. On the third *checkpoint()* call, **pendingRewards.totalAmount** is 1, which is less than **_vault.totalAmount**, and the subtraction underflow occurs when calculating **newRewards**.

```
        if (pendingRewards.totalAmount > 0 && supply > 0) {
            // Calculate the new rewards to be added to the vault.
            uint128 newRewards = pendingRewards.totalAmount - _vault.totalAmount;
```

2. **Protocol fees loss:**
   Consider the pending rewards are 1000 CRV and the protocol fee is 1%.
   1. The first *checkpoint()* is called in sync mode, **_vault.totalAmount** is updated to 1000, at which point the protocol fee of 10 CRV is reserved and not charged.
   2. The second *checkpoint()* is called in harvest mode, the 1000 CRV reward is claimed, but since **newRewards** is 0, no protocol fee will be charged.
   3. After that, even when *harvest()* is called, no protocol fee is charged because the reward has already been claimed.

```
            uint128 newRewards = pendingRewards.totalAmount - _vault.totalAmount;
            uint128 newFeeSubjectAmount = pendingRewards.feeSubjectAmount -
_vault.feeSubjectAmount;

            uint128 totalFees;
            if (harvested && newRewards > 0) {
                // Calculate total fees in one operation
                // We charge only protocol fee on the harvested rewards.
                if (newFeeSubjectAmount > 0) {
                    totalFees =
newFeeSubjectAmount.mulDiv(getProtocolFeePercent(), 1e18).toUint128();
                    // Update protocol fees accrued.
                    protocolFeesAccrued += totalFees;
                }

                // Update integral with new rewards per token
                integral += (newRewards - totalFees).mulDiv(SCALING_FACTOR,
supply);
            }
```

TRST-R-19 Rewards need to always keep increasing

In sync mode, the protocol does not claim pending rewards, but only synchronizes them. *Accountant.checkpoint()* assumes that rewards are always increasing, otherwise there will be a subtraction underflow when calculating **newRewards**.

```
        if (pendingRewards.totalAmount > 0 && supply > 0) {
            // Calculate the new rewards to be added to the vault.
            uint128 newRewards = pendingRewards.totalAmount - _vault.totalAmount;
            uint128 newFeeSubjectAmount = pendingRewards.feeSubjectAmount -
_vault.feeSubjectAmount;
```

One possible case is that for Locker, if *mint_for()* or *mint()* is called outside of *_harvestLocker()* to claim the CRV reward, then **pendingReward** will be reduced, so that **pendingRewards.totalAmount** is less than **_vault.totalAmount**.

```
            if (target == LOCKER) {
                // Calculate pending rewards for the locker by comparing  total earned
by gauge with already minted tokens
                pendingRewardsAmount =
                    ILiquidityGauge(gauge).integrate_fraction(LOCKER) -
IMinter(MINTER).minted(LOCKER, gauge);

                pendingRewards.feeSubjectAmount += pendingRewardsAmount.toUint128();
```

Additionally, in *Accountant.harvest()*, if in a future strategy's *_harvestLocker()* function, for example, due to an inappropriate **extraData** parameter, the harvested rewards are smaller than the already synced rewards. Since *Accountant.harvest()* always resets the _vault amount, this results in those rewards being credited twice when the rewards are later harvested correctly.

TRST-R-20 Typo in Accountant

In Accountant, **harvestTreshold** should be corrected to **harvestThreshold**.

TRST-R-21 Code quality improvements

1. The docs mention this struct is supposed to store in a single storage slot, but in total this is 0x81 bytes, it wouldn't fit in one. Should change docs or refactor the struct.

```
  /// @notice Struct to store protocol components in a single storage slot
    struct ProtocolComponents {
        address strategy;
        address allocator;
        address accountant;
        address feeReceiver;
        bool isShutdown;
    }
```

2. There is a redundant call to *getReward()*.

```
        /// We can save gas by not claiming extra rewards if we don't need them,
there's no extra rewards, or not enough rewards worth to claim.
        if (extraRewardTokens.length > 0) {
            baseRewardPool().getReward(address(this), true);
        }

        /// It'll claim rewardToken but we'll leave it here for clarity until the
claim() function is called by the strategy.
        baseRewardPool().getReward(address(this), true);
```

3. The **totalSupply > 0** check is redundant, as the code exits earlier if that not the case.

```
        if (_totalSupply == 0) return reward.rewardPerTokenStored;

        uint256 timeDelta = _lastTimeRewardApplicable(reward.periodFinish) -
reward.lastUpdateTime;
        uint256 rewardRatePerToken = 0;

        if (timeDelta > 0 && _totalSupply > 0) {
```

4. The calculation in *getRewardForDuration()* is only the correct amount of rewards for the next **DEFAULT_REWARDS_DURATION** if rewards were set in the current block. Otherwise, the finish time would be earlier, and the function is counting rewards which will never be released.

```
    function getRewardForDuration(address token) external view returns (uint256) {
        RewardData storage reward = rewardData[token];
        return reward.rewardRate * DEFAULT_REWARDS_DURATION;
    }
```

5. In most contracts, there is no validation that **PROTOCOL_ID** is not 0x00000000. However it does exist in *Accountant* constructor, which is inconsistent.
   Also, the docs list the possible exceptions but do not list **InvalidProtocolId**.

```
require(_protocolId != bytes4(0), InvalidProtocolId());
```

6. In *Strategy._harvest()*, in case of Locker it sets the **feeSubjectAmount** directly. The **feeSubjectAmount** is currently only used when **target == LOCKER**, but it is still good practice to use the **+=** operator to avoid overriding a non-zero **feeSubjectAmount**. For example, if some strategies in the future will have fees from sidecar, this could lead to loss of fees.

```
if (target == LOCKER) {
    pendingRewardsAmount = _harvestLocker(gauge, extraData);
    pendingRewards.feeSubjectAmount = pendingRewardsAmount.toUint128();
```

7. The *checkpoint()* function in *RewardVault* is in the list of view/pure functions, but it changes state so should be moved higher up in the contract.

```
    ///////////////////////////////////////////////////////////////
    /// ~ VIEW / PURE METHODS ~
    ///////////////////////////////////////////////////////////////
    ...
    /// @notice Updates reward state for a single account
    /// @dev Wrapper around _checkpoint for single account updates
    /// @param account Account to update rewards for
    function checkpoint(address account) external {
        _checkpoint(account, address(0));
    }
```

TRST-R-22 Factory._isValidGauge() needs to ensure that the gauge is valid

Anyone can call *Factory.createVault()* to create the vault, and it will call *_initializeVault()* to approve the gauge to spend the Locker's assets.

```
    function _initializeVault(address, address _asset, address _gauge) internal
override {
        /// Initialize the vault.
        /// We need to approve the asset to the gauge using the Locker.
        bytes memory data = abi.encodeWithSignature("approve(address,uint256)",
_gauge, type(uint256).max);

        /// Execute the transaction.
        require(_executeTransaction(_asset, data), ApproveFailed());
    }
```

Throughout the process, *_isValidGauge()* is used to check whether the gauge is valid.

```
    function createVault(address gauge) public virtual returns (address vault, address
rewardReceiver) {
        /// Perform checks on the gauge to make sure it's valid and can be used
        require(_isValidGauge(gauge), InvalidGauge());
        require(_isValidDeployment(gauge), InvalidDeployment());
        require(PROTOCOL_CONTROLLER.vaults(gauge) == address(0), AlreadyDeployed());
```

In the implementation of *CurveFactory._isValidGauge()*, it checks the validity of the gauge according to *GAUGE_CONTROLLER.gauge_types()*, which prevents users from forging malicious gauges to steal Locker's assets.

```
    function _isValidGauge(address _gauge) internal view virtual override returns
(bool) {
        bool isValid;
        /// Check if the gauge is a valid candidate and available as an inflation
receiver.
        /// This call always reverts if the gauge is not valid.
        try GAUGE_CONTROLLER.gauge_types(_gauge) {
            isValid = true;
        } catch {
            return false;
        }

        /// Check if the gauge is not killed.
        /// Not all the pools, but most of them, have this function.
        try ILiquidityGauge(_gauge).is_killed() returns (bool isKilled) {
            if (isKilled) return false;
        } catch {}

        /// If the gauge doesn't support the is_killed function, but is unofficially
killed, it can be deployed.
        return isValid;
    }
```

It is necessary to ensure that other Factory implementations in the future and *gauge_types()* functions on other chains cannot bypass the *_isValidGauge()* check.

TRST-R-23 The initialize() function of the Sidecar implementation can be called

In the deployment script, when the *Sidecar* implementation is created, its *initialize()* function is not called, which leads to anyone being able to call it later.

```
    function initialize() external {
        if (_initialized) revert AlreadyInitialized();
        _initialized = true;
        _initialize();
    }
```

It is recommended to set **_initialized** to true in the *Sidecar*'s constructor to prevent the *Sidecar* implementation's *initialize()* function from being called.

```
    constructor(bytes4 _protocolId, address _accountant, address _protocolController)
{
        PROTOCOL_ID = _protocolId;
        ACCOUNTANT = _accountant;
        PROTOCOL_CONTROLLER = IProtocolController(_protocolController);
        REWARD_TOKEN = IERC20(IAccountant(_accountant).REWARD_TOKEN());
+       _initialized = true;
    }
```

Centralization risks

TRST-CR-1 Authenticated user can claim other users’ rewards

Authenticated user can claim other users' rewards and specify the receiver.

```
    function claim(address account, address[] calldata tokens, address receiver)
        public
        onlyAllowed
        returns (uint256[] memory amounts)
    {
        return _claim(account, tokens, receiver);
    }
    function claim(address[] calldata _gauges, address account, bytes[] calldata
harvestData, address receiver)
        public
        onlyAllowed
    {
        require(harvestData.length == 0 || harvestData.length == _gauges.length,
InvalidHarvestDataLength());

        if (harvestData.length != 0) {
            _harvest(_gauges, harvestData, receiver);
        }

        _claim({_gauges: _gauges, accountAddress: account, receiver: receiver});
    }
```

TRST-CR-2 Owner of the ProtocolController contract is trusted

The owner is able to modify core addresses which are used in the flow of funds from contract to contract. It should be well understood that compromise of a permissioned account could lead to loss of assets held by the contracts.

Systemic risks

TRST-SR-1 Integration Risks

The contract interacts with several integration partners, such as Curve, Convex, Gnosis Safe among others. A bug or compromise of any trusted integration could lead to adverse effects of the OnlyBoost platform, including loss of funds. A responsible admin of any partner integration has been assumed.

TRST-SR-2 Mathematical assumptions

The correctness of the OnlyBoost allocation algorithm has not been rigorously tested during the engagement. Furthermore, its assumptions regarding the incentive and fee structures of different yield platforms may not hold in the future.