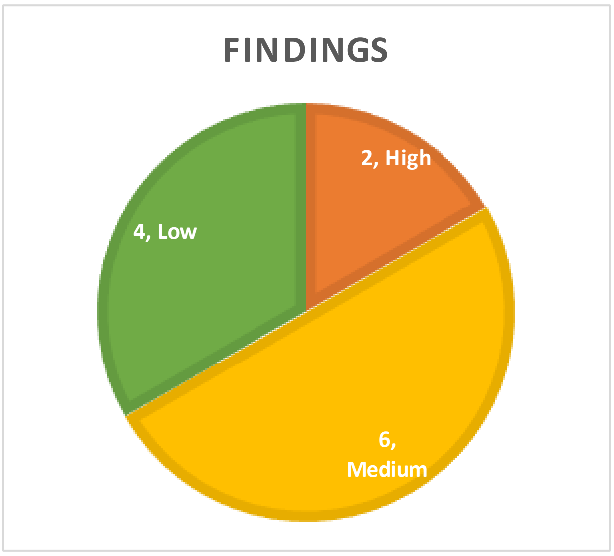
Trust Security

Smart Contract Audit

StakeDAO OnlyBoost

12/02/24

Executive summary

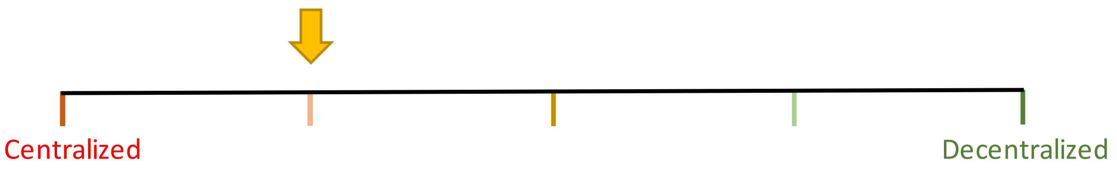


Category	Boost Yield Aggregator
Audited file count	10
Lines of Code	1010
Auditor	Trust
Time period	20-29/11/23

Findings

Severity	Total	Open	Fixed	Acknowledged
High	2	-	1	1
Medium	6	2	2	2
Low	4	-	3	1

Centralization score



Signature

Digitally signed by Trust
Date: 2024.02.12 15:31:35 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	29/11/23	Client report
0.2	12/02/24	Mitigation review

Contact

Trust

trust@trust-security.xyz

Introduction

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

Scope

- strategy/Strategy.sol
- strategy/only-boost/OnlyBoost.sol
- CRVStrategy.sol
- optimizer/Optimizer.sol
- fallbacks/ConvexImplementation.sol
- fallbacks/ConvexMinimalProxyFactory.sol
- factory/PoolFactory.sol
- factory/curve/CRVPoolFactory.sol
- staking/Vault.sol
- libraries/SafeExecute.sol

Repository details

- **Repository URL:** https://github.com/stake-dao/only-boost
- **Commit hash:** 81fd37d09968f6ce7f6379f8125705ece270adab

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Since its inception it has safeguarded over 30 clients through private services and over 30 additional projects through bug bounty submissions.

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 off auditing & bug hunting, he enjoys assessing bounty contests in C4 as a Supreme Court judge.

Audit Notes

This report represents a part of a joint effort between Trust and Zach Obront. It should be read in conjunction with the complementary report by Zach.

Disclaimer

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

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

Methodology

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

Qualitative analysis

Metric	**Rating**	**Comments**
Code complexity	Moderate	The integration with the Convex ecosystem introduces notable complexity.
Documentation	Excellent	Project is very well documented.
Best practices	Excellent	Project consistently adheres to industry standards.
Centralization risks	Moderate	The current design has significant centralization concerns.

Findings

High severity findings

TRST-H-1 Anyone can prevent the strategy from gaining extra rewards

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

Description

The *_claimExtraRewards()* function in Strategy should pull any matured rewards from the StakeDAO locker, and deposit them in the RewardDistributor. To pull the tokens, it first tries to use *_claimRewards()* which receives a **recipient**.

```
(bool isRewardReceived,) = locker.execute(
    gauge, 0,
abi.encodeWithSignature("claim_rewards(address,address)",
address(locker), address(this))
);
```

If that doesn't work, it will claim into the locker and then transfer it to the strategy.

```
if (!isRewardReceived) {
    ILiquidityGauge(gauge).claim_rewards(address(locker));
}
for (i = 0; i < 8;) {
    extraRewardToken = extraRewardTokens[i];
    if (extraRewardToken == address(0)) break;
    uint256 claimed;
    if (!isRewardReceived) {
        claimed = ERC20(extraRewardToken).balanceOf(address(locker))
- snapshotLockerRewardBalances[i];
        if (claimed != 0) {
            // Transfer the freshly rewards from the locker to this
contract.
            _transferFromLocker(extraRewardToken, address(this),
claimed);
        }
    }
```

It's important to note that in the backup flow, the amount transferred from the locker is only the difference from the snapshot, created just before the *claim()* call. Also, note that in Curve gauges, *claim_reward()* is a permissionless function, which is why the Strategy is able to call it to send the locker its rewards.

These properties enable the attacker to abuse the rewarding mechanism by simply calling *claim_rewards(**locker**)* before the Strategy harvests the rewards. Now, regardless of the reward claim path, the rewards in the locker balance will not be withdrawn into the strategy.

- In the **recipient** flow, there are no more rewards to pull from the Gauge, so *claim_rewards()* does nothing.
- In the backup flow, the snapshot will include the reward tokens, so it would not detect any difference after the call.

As a result, the tokens would be impermanently stuck in the locker. The locker admin still has access to it, but the rewarding mechanism is deeply flawed and damage has already been done (users that withdraw their OnlyBoost stake after *harvest()* will not enjoy the rewards.)

Recommended mitigation

Consider refactoring the snapshotting logic to protect against this attack.

Team response

Acknowledged. We plan to recover it manually if it happens using the execute function and an allowed external module.

TRST-H-2 Persistent halt of rewarding by making the tokens stuck in the Strategy

- **Category:** Gas-assumption issues
- **Source:** Strategy.sol,OnlyBoost.sol
- **Status:** Fixed

Description

In Strategy, several functions use the locker's *execute()* function without the SafeExecute wrapper. This means if the execution reverts, it doesn't bubble upwards. This happens in:

- *_depositIntoLocker()*
- *_withdrawFromLocker()*
- *_claimNativeRewards()*
- *_claimRewardToken()*

It is important to be familiar with EIP150, which states that each call by default will pass 63/64 of the remaining gas forwards, and store the remainder in the current execution frame. This means that even if a called function should never revert, an attacker could engineer execution with less than the required gas, which would revert with out-of-gas. Then, if the remaining 1/64 is enough to complete the transaction, it will finalize.

It turns out there is one flow in which one of the functions above is exploitable. In *rebalance()*, it withdraws all funds from the underlying and then re-deposits them optimally, shown below.

```
for (uint256 i; i < fundsManagers.length; ++i) {
    // Skip if the allocation amount is 0.
    if (allocations[i] == 0) continue;
    /// Deposit into the locker if the recipient is the locker.
    if (fundsManagers[i] == address(locker)) {
        _depositIntoLocker(asset, gauge, allocations[i]);
    } else {
        /// Else, transfer the asset to the fallback recipient and
call deposit.
        SafeTransferLib.safeTransfer(asset, fundsManagers[i],
allocations[i]);
        IFallback(fundsManagers[i]).deposit(asset, allocations[i]);
    }
}
_currentBalance = balanceOf(asset);
/// This invariant should never be broken.
if (_currentBalance < _snapshotBalance) revert REBALANCE_FAILED();
```

The external call stack during the *deposit()* will be:

OnlyBoost::rebalance() -> Locker::execute() -> LiquidityGauge::deposit()

deposit() is a reasonably expensive function, it updates many state variables and performs a token transfer call. Suppose that it consumes **X** gas. Then attacker can provide an initial gas amount to the transaction so that at the moment of *deposit()* call, there will be **X-1** gas sent. Therefore, execution will revert and the *execute()* will complete with **X/63** – **EXE_RET_COST**. This amount will return to *rebalance()*, which left 1/63 of (**X-1 + EXE_BEFORE_CALL_COST**), which is the amount available to *execute()*. The 1/63 is the ratio between remaining gas (1/64) and sent gas (63/64).

So, when coming back into *rebalance()*, it'd have roughly 2/64 = 1/32 of the gas sent to *deposit()*, to consume. Since the remaining code is extremely short (external read-only *balanceOf()* calls and a couple of SLOADs), it is fair to assume the remaining gas would suffice. This means an attacker can call *rebalance()* with a specific gas amount and make the final *depositIntoLocker()* revert, keeping the funds in the strategy. Rebalance can be called later with sufficient gas to redistribute the funds correctly, however that can always be undone again by an attacker.

The main impact is that after a malicious *rebalance()*, the deposits will not be earning any rewards.

Recommended mitigation

Replace all uses of *locker.execute()* with *locker.safeExecute()* to reduce exposure to gas attacks.

Team response

Fixed.

Mitigation review

Fixed as suggested.

Medium severity findings

TRST-M-1 Reward processing of extra rewards would fail when one of the tokens is skipped in construction

- **Category:** Logical flaws

- **Source:** Strategy.sol
- **Status:** Open

Description

When a Vault is constructed for a new gauge, the PoolFactory must register the reward tokens for the matching gauge in the reward distributor. It is done in *_addExtraRewards()*:

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

Which will add the reward on the distributor:

```
function addRewardToken(address gauge, address extraRewardToken)
external onlyGovernanceOrAllowed {
    if (gauge == address(0) || extraRewardToken == address(0)) revert
ADDRESS_NULL();
    /// Get the rewardDistributor address to add the reward token to.
    address _rewardDistributor = rewardDistributors[gauge];
    /// Approve the rewardDistributor to spend token.
    SafeTransferLib.safeApproveWithRetry(extraRewardToken,
_rewardDistributor, type(uint256).max);
    /// Add it to the Gauge with Distributor as this contract.
    ILiquidityGauge(_rewardDistributor).add_reward(extraRewardToken,
address(this));
}
```

Then, in *_claimExtraRewards(),* those rewards are deposited:

```
} else {
    claimed = ERC20(extraRewardToken).balanceOf(address(this));
    if (claimed != 0) {
        // Distribute the extra reward token.

ILiquidityGauge(rewardDistributor).deposit_reward_token(extraRewardTo
ken, claimed);
    }
}
```

An issue occurs when a token is not valid as determined by *_isValidToken()*. In this case, it is skipped at construction stage, but not when claiming rewards. This causes the entire transaction to fail, as the distributor is not registered on the RewardDistributor:

```
def deposit_reward_token(_reward_token: address, _amount: uint256):
    assert msg.sender == self.reward_data[_reward_token].distributor
```

That would normally occur in *add_reward()*:

```
def add_reward(_reward_token: address, _distributor: address):
    """
    @notice Set the active reward contract
    """
    assert msg.sender == self.admin  # dev: only owner
    reward_count: uint256 = self.reward_count
    assert reward_count < MAX_REWARDS
    assert self.reward_data[_reward_token].distributor ==
ZERO_ADDRESS

    self.reward_data[_reward_token].distributor = _distributor
```

Therefore, all rewarding would fail if one token is deemed to be invalid.

Recommended mitigation

If some tokens are skipped during construction, make sure to also skip them in the processing of the rewards.

Team response

Fixed.

Mitigation review

The follow check was added:

```
/// Check if the rewardDistributor is valid.
/// Else, there'll be some extra rewards that are not valid to
distribute left in the strategy.
if
(ILiquidityGauge(rewardDistributor).reward_data(extraRewardToken)
.distributor != address(this)) break;
```

Although it does prevent the rewarding loop from reverting, it should use **continue;** instead of **break.** If valid tokens come after invalid tokens in the gauge reward token list, those will be skipped.

TRST-M-2 Withdrawal logic is unsatisfactory

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

Description

The goal of the withdrawal algorithm is to keep as many tokens deposited in an optimized way, to maintain the whitepaper formula for boosts. However, the withdraw operates as following:

- Claim as much as possible from the larger allocation
- Claim the rest from the smaller allocation

The logic does not take into account the boost power, or the formula at all. On large withdrawals, it would also create a larger imbalance. Consider that when the difference between allocations is just 1 wei, the de-allocation would create an imbalance in an arbitrary direction.

Recommended mitigation

Refactor the withdrawal function to account for the boost formula.

Team response

Fixed.

Mitigation review

The fix involves allocating based on the optimal deposit amount.

```
// Calculate optimal boost for both pools
uint256 optimalSD = cachedOptimizations[gauge].value;
uint256 totalBalance = balanceSD + balanceConvex;
// Adjust the withdrawal based on the optimal amount for Stake
DAO
if (totalBalance <= amount) {
    // If the total balance is less than or equal to the
withdrawal amount, withdraw everything
    _allocations[0] = balanceConvex;
    _allocations[1] = balanceSD;
} else if (optimalSD >= balanceSD) {
    // If Stake DAO balance is below optimal, prioritize
withdrawing from Convex
    _allocations[0] = FixedPointMathLib.min(amount,
balanceConvex);
    _allocations[1] = amount > _allocations[0] ? amount -
_allocations[0] : 0;
} else {
    // If Stake DAO balance is above optimal, prioritize
withdrawing from Stake DAO
    _allocations[1] = FixedPointMathLib.min(amount, balanceSD);
    _allocations[0] = amount > _allocations[1] ? amount -
_allocations[1] : 0;
}
```

It prioritizes withdrawing from convex when SD has not reach optimal allocation. Notice that the result would still not be formula-precise because if **balanceSD** is just 1 wei above **optimalSD**, everything is taken from StakeDAO, and when they are equal all is taken from Convex. However that can be treated as acceptable behavior.

Another issue is that it uses the **cachedOptimizations** to determine the optimal SD amount, however it is never checked if the value is stale. Therefore the optimizer could be very inaccurate if no deposits were made for a while.

TRST-M-3 claimNativeRewards() can be abused to lose rewards

- **Category:** Validation issues
- **Source:** Strategy.sol
- **Status:** Fixed

Description

In *claimNativeRewards()*, it claims rewards from the feeDistributor and sends it to the **accumulator**.

```
function _claimNativeRewards() internal virtual {
    locker.execute(feeDistributor, 0,
abi.encodeWithSignature("claim()"));
    /// Check if there is something to send.
    uint256 _claimed =
ERC20(feeRewardToken).balanceOf(address(locker));
    if (_claimed == 0) return;
    /// Transfer the rewards to the Accumulator contract.
    _transferFromLocker(feeRewardToken, accumulator, _claimed);
}
```

Note that the **feeDistributor**, **feeRewardToken** and **accumulator** are all set by different functions, and are zero-initialized. Therefore, if **accumulator** is set last, an attacker can claim *claimNativeRewards()* to transfer the fee tokens claimed so far to address zero.

Recommended mitigation

Validate that **accumulator** is not zero in *claimNativeRewards()*.

Team response

Fixed.

Mitigation review

Fixed as suggested.

TRST-M-4 Max approvals would lead Gauges emitting certain rewards to be unusable

- **Category:** ERC20-compatibility issues
- **Source:** Strategy.sol
- **Status:** Acknowledged

Description

There is a certain class of ERC20 tokens, like UNI and COMP, which revert when approving or transferring with a value larger than **2^96**. Since these tokens could be used as extra reward tokens for Curve gauges, it would make deployment of vaults for them revert.

```
function addRewardToken(address gauge, address extraRewardToken)
external onlyGovernanceOrAllowed {
    if (gauge == address(0) || extraRewardToken == address(0)) revert
ADDRESS_NULL();
    /// Get the rewardDistributor address to add the reward token to.
    address _rewardDistributor = rewardDistributors[gauge];
    /// Approve the rewardDistributor to spend token.
    SafeTransferLib.safeApproveWithRetry(extraRewardToken,
_rewardDistributor, type(uint256).max);
    /// Add it to the Gauge with Distributor as this contract.
    ILiquidityGauge(_rewardDistributor).add_reward(extraRewardToken,
address(this));
}
```

Recommended mitigation

The safest way to interact with tokens is to approve the actual amount that is transferred, at transfer-time.

Team response

Acknowledged.

TRST-M-5 Locker transfers could misinterpret the transfer() result, potentially leading to losses

- **Category:** ERC20-compatibility issues
- **Source:** Strategy.sol
- **Status:** Fixed

Description

The *_transferFromLocker()* utility in Strategy will use *safeExecute()* to transfer funds from the locker.

```
function _transferFromLocker(address asset, address recipient,
uint256 amount) internal {
    locker.safeExecute(asset, 0,
abi.encodeWithSignature("transfer(address,uint256)", recipient,
amount));
}
```

Note that the ERC20 *transfer()* function returns a Boolean indicating the success status. However, the code assumes that any transfer that did not revert is successful.

```
function safeExecute(ILocker locker, address to, uint256 value, bytes
memory data)
    internal
    returns (bool success)
{
    (success,) = locker.execute(to, value, data);
    if (!success) revert CALL_FAILED();
}
```

This is typically handled by using *safeTransfer()* variants, but due to the custom low-level calling it is missing in the library. In the worst-case scenario, when interacting with tokens that return false instead of revert, the transferring will be marked as successful. This could make some reward tokens become stuck in the locker contract, for example in *_claimExtraRewards()*:

```
for (i = 0; i < 8;) {
    extraRewardToken = extraRewardTokens[i];
    if (extraRewardToken == address(0)) break;
    uint256 claimed;
    if (!isRewardReceived) {
        claimed = ERC20(extraRewardToken).balanceOf(address(locker))
- snapshotLockerRewardBalances[i];
        if (claimed != 0) {
            // Transfer the freshly rewards from the locker to this
contract.
            _transferFromLocker(extraRewardToken, address(this),
claimed);
        }
    }
```

Recommended mitigation

Create a *safeExecuteTransfer()* function in the SafeExecute library to check the return code.

Team response

Fixed.

Mitigation review

Fixed as suggested.

TRST-M-6 When the Convex reward pool has rewards that don't exist in the Gauge, they will be stuck in the strategy

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

Description

Harvesting in the OnlyBoost can claim extra rewards from the Convex fallback.

```
baseRewardPool().getReward(address(this), _claimExtraRewards);
```

If **claimExtraRewards** is true, later on a loop will deposit all the Curve gauge reward tokens into the RewardDistributor. However, the gauge rewards may not line up with the **baseRewardPool** rewards. An admin can add additional rewards through this function.

When the two lists are mismatched, rewards would be stuck in the strategy.

Recommended mitigation

Take into consideration the list of rewards in the **baseRewardPool** when using a strategy that interacts with Convex.

Team response

Acknowledged.

Low severity findings

TRST-L-1 The effective fees are somewhat different than expected

- **Category:** Arithmetic flaws
- **Source:** OnlyBoost.sol,Strategy.sol
- **Status:** Fixed

Description

In the Strategy, claimer fees and protocol fees are charged upon harvest.

```
/// 4. Take Fees from _claimed amount.
claimed = _chargeProtocolFees(claimed, claimedFromFallbacks,
protocolFeesFromFallbacks);
/// 5. Distribute Claim Incentive
claimed = _distributeClaimIncentive(claimed);
```

Each takes some percentage of **claimed** as fees. They are set independently.

There is an implicit assumption that the sum of the two percentages is the total percentage charged on rewards (Which is why the total cannot exceed **DENOMINATOR**). However, since they are charged sequentially, the protocol fee will be higher than claimer fee.

For example, suppose claimer **fee % = protocol fee % = 20%**, and **reward = 100**

The calculations are: **protocol fee = 20**, **claimer fee = (100 - 20) * 20% = 16**, **total fee = 36**.

Recommended mitigation

Merge the fee claiming functions so both percentages are calculated on the original amount.

Team response

Fixed.

Mitigation review

The fee calculation code has been refactored to address it.

TRST-L-2 Governance can be accepted repeatedly due to leak of futureGovernance

- **Category:** Logical flaws
- **Source:** Strategy.sol,Optimizer.sol,ConvexMinimalProxyFactory.sol
- **Status:** Fixed

Description

The governance state transition in multiple contracts is handled with the code below:

```
function transferGovernance(address _governance) external
onlyGovernance {
    futureGovernance = _governance;
}
/// @notice Accept the governance transfer.
function acceptGovernance() external {
    if (msg.sender != futureGovernance) revert GOVERNANCE();
    governance = msg.sender;
    emit GovernanceChanged(msg.sender);
}
```

When governance is accepted, **futureGovernance** should be deleted, or else *acceptGovernance()* can be called repeatedly. That is a common safety check in the Ownable2Step pattern of OpenZeppelin.

Recommended mitigation

Delete the **futureGovernance** variable before changing governance.

Team response

Fixed.

Mitigation review

Fixed as suggested.

TRST-L-3 Possible overflow in the Optimizer deposit calculation

- **Category:** Overflow issues
- **Source:** Optimizer.sol
- **Status:** Acknowledged

Description

The Optimizer calculates the boost incentive below:

```
// Additional boost
uint256 boost = adjustmentFactor * (1e26 - cvxTotal) * veCRVConvex /
(1e26 * vlCVXTotal);
```

There is some risk that the above line could overflow. A **uint256** holds values up to **~1e77**. The **cvxTotal** is in the magnitude of **~1e26**. The **veCRVConvex** at this time is around **~1e26**, and **adjustmentFactor** without any changes is **1e18**.

Since these are all multiplied, this leaves just 77 – 26 – 26 – 18 = **7** orders of magnitudes to overcome before an overflow would happen. Typically, that is not a safe margin, in the event the context or adjustment factor would increase considerably.

Recommended mitigation

Perform the calculation using an intermediate step, or use an external library to deal with data types larger than **uint256**.

Team response

We'll consider some times in production run in order to evaluate the efficiency of the optimizer, we might need to redeploy it at a certain point and add a fix for this issue in the same time.

TRST-L-4 Harvesting will fail when Convex fallback is used unless CVX is manually added

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

Description

In the OnlyBoost strategy, when claiming from Convex fallback it will award CVX tokens, which are deposited into the RewardDistributor:

```
if (fallbackRewardToken != address(0) && fallbackRewardTokenAmount >
0) {
    /// Distribute the fallbackRewardToken.

ILiquidityGauge(rewardDistributor).deposit_reward_token(fallbackRewar
dToken, fallbackRewardTokenAmount);
}
```

The issue is that **fallbackRewardToken** is never registered as a reward, which means the action will revert unless it is manually added by governance (see M-1).

Recommended mitigation

Consider overriding *_addRewardToken()* in CRVPoolFactory to approve CVX, so intervention won't be needed.

Team response

Fixed.

Mitigation review

Fixed as suggested.

Additional recommendations

TRST-R-1 Reconsider incentives for calling *harvest()*

There is a caller incentive for *harvest()*, which is a percentage of the CRV rewards accumulated. Harvest accepts parameters **distributeSDT**, **claimExtra**, which claim additional rewards. However, these do not yield anything for the caller. This means a bot would surely always call the function with these set to **false** to cut down on gas costs. Consider re-working the incentive structure so these would be set to **true** (for example, one in every X calls must set these to **true**).

TRST-R-2 Disallow overriding of fallbacks in the Convex Factory

The *create()* function sets **fallbacks[gauges]**, but does not verify this was empty before. If it isn't, it could cause significant damage in accessing funds.

This could theoretically occur when more than one pool is matched to the same gauge, since *create()* accepts a **poolID**. This is the case for poolds 80 and 132, however since they are both shut down, the *create()* call will revert regardless. It is still recommended to protect from a potential override.

TRST-R-3 Assumptions from call failures are dangerous

Take for example the code from the PoolFactory below:

```
function _addExtraRewards(address _gauge) internal virtual {
    /// 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);
    /// Hence the call to the function is wrapped in a try catch.
    (bool success,) = _gauge.call(data);
    if (!success) {
        /// If it fails, we set the LGtype to 1 to indicate that the
gauge doesn't support extra rewards.
        /// So the harvest would skip the extra rewards.
        strategy.setLGtype(_gauge, 1);
        return;
    }
```

Since any user can create a pool, a user could always plant a specific amount of gas so that the *_gauge.call(data)* would revert. When that occurs, the code assumes the gauge does not support extra rewards, even though it does. Fortunately, this attack is only theoretical, because the *setLGtype()* call would be too expensive for the remaining 1/64 gas, but it serves as an example for the danger of extrapolating information from external calls.

TRST-R-4 Dead code paths

In the CRVPoolFactory *_isValidGauge()*, **isValid** will always be true when used, so it is unnecessary.

In *_chargeProtocolFess()*, it decreases a memory variable during a return statement, which doesn't have any use:

```
return amount -= _feeAccrued;
```

The *absDiff()* function in Optimizer is never used.

TRST-R-5 Use of incorrect interface

In *_addExtraRewards()*, gauge is converted to **ISDLiquidityGauge**, but it is native Curve (not StakeDAO) gauge. In this case it is harmless but still bad code hygiene.

TRST-R-6 Handle lack of fallbacks safely

The Optimizer would return a single zero element in an array when there's no fallback. In this case, the *_claimFallbacks()* call will revert as it will try to call *claim()* on address zero. It is recommended to skip the calls instead.

Centralization risks

TRST-CR-1 Governance role has high privileges

The Governance role is highly privileged. It may perform the following actions:

1. It can update the Convex reward fee up to 100% through *updateProtocolFee()*.
2. It can control the allocation of funds through the Optimizer by assigning fake params.
3. It can set a new Optimizer in the OnlyBoost
4. It can arbitrarily upgrade the Strategy logic or perform any call on its behalf. This means **any funds** lying in the strategy **can be lost** through a compromised governance.

Systemic risks

TRST-SR-1 Integration risks with Curve/Convex

The codebase heavily interacts with Curve pools and Convex contracts. It also may place funds in Convex vaults. Naturally, any admin-related risk or security flaws in the Convex dependencies could expose the funds of OnlyBoost to losses.

TRST-SR-2 Rewards could be diluted

The rewards going to Vault shareholders come from the RewardDistributor. Note that rewards are retroactive, which means a user could enter at time T and leave at time T+X, and will get rewards from the time period before T. The larger the harvest potential, the more likely it is for a large whale to enter into a Vault and dilute the earnings up to this point.