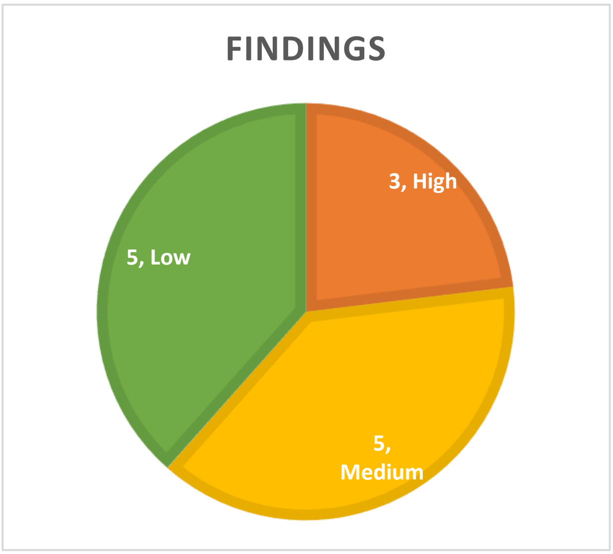
Trust Security

Smart Contract Audit

LinkPool SDLPool

02/07/2023

Executive summary

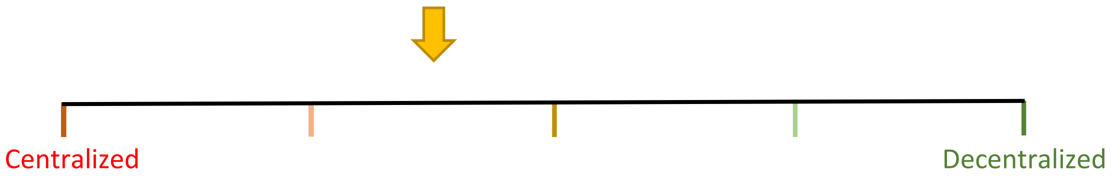


Category	Liquid Staking
Audited file count	5
Lines of Code	809
Auditor	Trust
Time period	25/06-02/07/23

Findings

Severity	Total	Open	Fixed	Acknowledged
High	3	-	2	1
Medium	5	-	3	2
Low	5	-	1	4

Centralization score



Signature

Digitally signed by Trust
Date: 2023-08-06 15:40:55

Document properties

Versioning

Version	Date	Description
0.1	07/02/2023	Client report
0.2	06/08/2023	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

- sdlPool/SDLPool.sol
- sdlPool/LinearBoostController.sol
- linkStaking/OperatorVCS.sol
- linkStaking/OperatorVault.sol
- core/test/deprecated/DelegatorPool.sol

Repository details

- **Repository URL:** https://github.com/stakedotlink/contracts
- **Commit hash:** 049f1928f9c938ae058e6b6e032484350585cc1e
- **Mitigation commit hash:** 99798c4fa18f3e3d9abfaf40675648894b33c8e5

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.

Disclaimer

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

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

Methodology

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

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	Code is modularized well, reducing overall complexity.
Documentation	**Good**	Project source code is well documented. Better user-facing docs are encouraged.
Best practices	**Good**	Project usually adheres to industry standards.
Centralization risks	**Moderate**	Certain addresses have concerning privileges that allow to potentially harm the users or the protocol.

Findings

High severity findings

TRST-H-1 Attacker could block user's access to staked SDL in DelegatorPool

- **Category:** Time-sensitivity flaws
- **Source:** DelegatorPool.sol
- **Status:** Fixed

Description

Migration to the new pool takes place by calling *retireDelegatorPool()* with a list of all addresses with locked tokens. The function will:

1. Burn the stSDL for each user with locked tokens.
2. Migrate their vested balance to the SDLPool.
3. Burn all locked SDL.

```
function retireDelegatorPool(address[] calldata _lockedAddresses,
address _sdlPool) external onlyOwner {
    require(_sdlPool != address(0), "Invalid address");
    allowanceToken.approve(_sdlPool, type(uint256).max);
    IRewardsPool rewardsPool = tokenPools[tokens[0]];
    for (uint256 i = 0; i < _lockedAddresses.length; ++i) {
        address account = _lockedAddresses[i];
        uint256 unlockedBalance = availableBalanceOf(account);
        rewardsPool.withdraw(account);
        _burn(account, balanceOf(account));
        delete lockedBalances[account];
        delete lockedApprovals[account];
        if (unlockedBalance != 0) {
            ISDLPool(_sdlPool).migrate(account, unlockedBalance, 0);
            emit Migration(account, unlockedBalance);
        }
    }
    IStakingAllowance(address(allowanceToken)).burn(totalLocked);
    totalLocked = 0;
    sdlPool = _sdlPool;
}
```

Later on, any user can call *migrate()*, which will transfer their entire vested balance to the SDLPool.

```
function migrate(uint64 _lockingDuration) external {
    require(sdlPool != address(0), "Cannot migrate until contract is
retired");
    uint256 amount = balanceOf(msg.sender);
    require(amount != 0, "Nothing to migrate");
    tokenPools[tokens[0]].withdraw(msg.sender);
    _burn(msg.sender, amount);
    ISDLPool(sdlPool).migrate(msg.sender, amount, _lockingDuration);
    emit Migration(msg.sender, amount);
}
```

The root of the issue is that **lockedAddresses** is assumed to include all addresses with locked tokens. If that is not the case, *retire()* will burn the SDL for that user, yet it will be withdrawable using *migrate()*, since it assumes all stSDL are unlocked. This means locked SDL are *double-burnt*. Indeed this could be the case, as any user can front-run the *retireDelegatorPool()* call with a large SDL transfer to the pool. Generally, due to the nature of blockchains the pool can never guarantee the list to be up-to-date.

The effect is very dangerous. The attacker can immediately call *migrate()* after the attack and leave the DelegatorPool without enough SDL tokens to honor migrations. Meanwhile, the attacker could profit by selling SDL while its supply has deflated due to the attack.

Recommended mitigation

One solution would be to pause token transfers to the DelegatorPool before generating the **lockedAddresses** list.

Team response

Resolved.

Mitigation review

The code can no longer perform double-burning of tokens as *retireDelegatorPool()* will only deduct the locked amount from the passed account array. As a result, insolvency cannot occur.

TRST-H-2 Operators will not be able to withdraw all their rewards

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

Description

The *updateDeposits()* function will update **totalDeposits** and calculate rewards to beneficiaries.

```
function updateDeposits()
    external
    override
    onlyStakingPool
    returns (address[] memory receivers, uint256[] memory amounts)
{
    int256 balanceChange = depositChange();
    if (balanceChange > 0) {
        totalDeposits += uint256(balanceChange);
        receivers = new address[](fees.length + 1);
        amounts = new uint256[](fees.length + 1);
        receivers[0] = address(this);
        amounts[0] = (uint256(balanceChange) *
operatorRewardPercentage) / 10000;
        for (uint256 i = 1; i < receivers.length; ++i) {
            receivers[i] = fees[i - 1].receiver;
            amounts[i] = (uint256(balanceChange) * fees[i -
1].basisPoints) / 10000;
```

```
        }
    } else if (balanceChange < 0) {
        totalDeposits -= uint256(balanceChange * -1);
    }
}
```

The code reserves rewards for the operators in the first array entry. It uses the difference in total deposits of all vaults.

```
function depositChange() public view returns (int) {
    uint256 totalBalance = token.balanceOf(address(this));
    for (uint256 i = 0; i < vaults.length; ++i) {
        totalBalance += vaults[i].getTotalDeposits();
    }
    return int(totalBalance) - int(totalDeposits);
}
```

The issue with this calculation is that it will only be correct if all vaults appreciated in value. Suppose one vault lost 100 LINK and another gained 150 LINK. The function will issue awards based on **50 LINK** of value. However, the operators collectively will expect rewards totaling **150 LINK** of value (0 from Operator #1 and 150 from Operator #2). This means not enough awards are reserved in the OperatorVCS contract to satisfy operators when cashing out.

Recommended mitigation

Sum the individual expected rewards of operators instead of totaling them together.

Team response

Resolved.

Team response

The award distribution logic has been rewritten. Rewards are now accumulated correctly.

```
for (uint256 i = 0; i < vaultCount; ++i) {
    (uint256 deposits, uint256 rewards) =
IOperatorVault(address(vaults[i])).updateDeposits();
    vaultDeposits += deposits;
    operatorRewards += rewards;
}
```

TRST-H-3 DelegatorPool is susceptible to double-spending attacks

- **Category:** MEV attacks
- **Source:** DelegatorPool.sol
- **Status:** Acknowledged

Description

The DelegatorPool allows the owner to unlock access to additional tokens in the user's balance.

```
function setLockedApproval(address _account, uint256 _amount)
external onlyOwner {
    require(lockedBalances[_account] >= _amount, "Cannot approve more
than locked balance");
    lockedApprovals[_account] = _amount;
}
```

Users can withdraw from the approved amount using *withdrawAllowance().*

```
function withdrawAllowance(uint256 _amount) external
updateRewards(msg.sender) {
    require(!poolRouter.isReservedMode(), "Allowance cannot be
withdrawn when pools are reserved");
    require(availableBalanceOf(msg.sender) >= _amount, "Withdrawal
amount exceeds available balance");
    uint256 unlockedBalance = balanceOf(msg.sender) -
lockedBalances[msg.sender];
    if (_amount > unlockedBalance) {
        uint256 approvedAmountToUnlock = _amount - unlockedBalance;
        lockedApprovals[msg.sender] -= approvedAmountToUnlock;
        lockedBalances[msg.sender] -= approvedAmountToUnlock;
        totalLocked -= approvedAmountToUnlock;
    }
    _burn(msg.sender, _amount);
    emit AllowanceWithdrawn(msg.sender, _amount);
    allowanceToken.safeTransfer(msg.sender, _amount);
}
```

Suppose a user was allocated 100 SDL, with 50 SDL unlocked after 6 months and 50 SDL unlocked after 1 year. The expected call sequence would be:

1. *setLockedApproval(UserA, 50)*
2. *setLockedApproval(UserA, 100)*
3. *withdrawAllowance(100)*

Or, if user decides to spend their allowance:

1. *setLockedApproval(UserA, 50)*
2. *withdrawAllowance(50)*
3. *setLockedApproval(UserA, 50)*
4. *withdrawAllowance(50)*

However, the flow can be attacked by a frontrunning attack:

1. *setLockedApproval(UserA, 50)*
2. *withdrawAllowance(50)* – Only submitted to the mempool after next TX!
3. *setLockedApproval(UserA, **100**)*
4. *withdrawAllowance(**100**)*

This grants the user access to 150 tokens instead of 100, or in the general case, to X+Y instead of max(X,Y). The issue is similar to the approval double-spend in the OpenZeppelin ERC20 library, which has been mitigated using *increaseAllowance()* and *decreaseAllowance()* functions.

Recommended mitigation

Only change the allowance amounts in deltas to the previous amount.

Team response

Acknowledged, *setLockedApproval()* will never be called.

Medium severity findings

TRST-M-1 Users cannot directly call withdraw() when lock has not been locked

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

Description

When funds are migrated to the SDLPool, a user can choose to not lock their tokens and not receive boost awards. This will create a **Lock** object with **expiry=0**.

```
function _createLock(
    address _sender,
    uint256 _amount,
    uint64 _lockingDuration
) private updateRewards(_sender) {
    uint256 boostAmount = boostController.getBoostAmount(_amount,
_lockingDuration);
    uint256 totalAmount = _amount + boostAmount;
    uint64 startTime = _lockingDuration != 0 ?
uint64(block.timestamp) : 0;
    uint256 lockId = lastLockId + 1;
    locks[lockId] = Lock(_amount, boostAmount, startTime,
_lockingDuration, 0);
```

Then, if user would like to withdraw directly, they would call the *withdraw()* function.

```
function withdraw(uint256 _lockId, uint256 _amount)
    external
    onlyLockOwner(_lockId, msg.sender)
    updateRewards(msg.sender)
{
    uint64 expiry = locks[_lockId].expiry;
    if (expiry == 0) revert UnlockNotInitiated();
    if (expiry > block.timestamp) revert TotalDurationNotElapsed();
```

Since expiry is zero, it will not allow immediate withdrawals. A user can circumvent the issue by calling *initiateUnlock()* first, which should set expiry to the current **block.timestamp**.

Recommended mitigation

If the user does not intend to lock tokens, allow *withdraw()*, by checking for **duration == 0** for example.

Team response

Resolved.

Mitigation review

Issue has been fixed by skipping expiry check when **startTime** is 0. For locks of duration zero, **startTime** will always be zero, from the behavior in *_updateLock()* and *_createLock()*.

```
if (locks[_lockId].startTime != 0) {
    uint64 expiry = locks[_lockId].expiry;
    if (expiry == 0) revert UnlockNotInitiated();
    if (expiry > block.timestamp) revert TotalDurationNotElapsed();
}
```

TRST-M-2 Users cannot lock tokens for a shorter period then previously, after unlock

- **Category:** Time-related flaws
- **Source:** SDLPool.sol
- **Status:** Fixed

Description

The *_updateLock()* function allows users to reconfigure the locked token amount and lock duration. It validates that the new duration is not smaller than the current duration.

```
uint64 curLockingDuration = locks[_lockId].duration;
if (_lockingDuration < curLockingDuration) revert
InvalidLockingDuration();
```

The intention is that users cannot commit to a lower lock lifespan than previously set. However, there are cases where the requirement is too strict:

1. If the expiry has passed, a user should be able to set any duration
2. If the time left to expiry is below the duration, the duration should be reducible to the point that a future expiry will always be later than the current expiry.

Recommended mitigation

Change the validations performed in *_updateLock().*

Team response

Resolved. First part of issue was found and fixed during audit, second is by design as otherwise a user could bypass the 0 boost unlock period by relocking for half the original time and receiving boost.

Mitigation review

The validation has been fixed correctly.

TRST-M-3 Unvested tokens will be silently dropped during migration

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

Description

The DelegatorPool contract has two balance functions, a standard *balanceOf()* and *availableBalanceOf()*, which takes into account the amount of locked tokens vested.

```
function availableBalanceOf(address _account) public view returns
(uint256) {
    return balanceOf(_account) - lockedBalances[_account] +
lockedApprovals[_account];
}
```

The new SDLPool will only receive the vested amount, rather than the intended future amount. The rest will be silently dropped, without an event specifying the amount of locked tokens lost.

Recommended mitigation

Consider implementing the migration of unvested tokens to the SDLPool. At a minimum, log the amount of tokens not migrated.

Team response

Acknowledged. Only whitelisted addresses have locked balances and the burning of unvested tokens is intended.

TRST-M-4 Slashing of Operator affects stakers but not the Operator

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

Description

In OperatorVault, the operator's rewards payout is determined by the difference between the previous total LINK value and the current value.

```
function getRewards() public view returns (uint256) {
    uint256 curTotalDeposits = getTotalDeposits();
    uint256 totalRewards = rewards;
    if (curTotalDeposits > totalDeposits) {
        totalRewards +=
            ((curTotalDeposits - totalDeposits) *
IOperatorVCS(vaultController).operatorRewardPercentage()) /
            10000;
    }
    return totalRewards;
}
```

An issue with the logic is that Operators are not directly incentivized to behave well. After getting slashed, they could call *updateRewards()* to reset the baseline **totalDeposits**, after which they resume getting rewards despite overall causing losses in LINK for stakers.

Recommended mitigation

Keep track of the performance of the operator and only credit them when staking rewards have passed the top point (ignoring user deposits).

Team response

Resolved.

Mitigation review

Reward accounting has been rewritten. The **trackedTotalDeposits** variable in the vault will only increase through *deposit()* and *updateDeposits()*. When staking losses occur, the operator will not be eligible for gains until they pass the **trackedTotalDeposits** again.

TRST-M-5 OperatorVCS awards too many reward tokens

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

Description

The *updateDeposits()* function calculates the rewards to hand out to creditors following an appreciation in the total value of the strategies. The issue is similar to M-4, where fluctuations in the value of the strategy will be continuously rewarded to beneficiaries. This causes a dilution of the fair supply cap of the reward token.

```
int256 balanceChange = depositChange();
if (balanceChange > 0) {
    totalDeposits += uint256(balanceChange);
    receivers = new address[](fees.length + 1);
    amounts = new uint256[](fees.length + 1);
    receivers[0] = address(this);
    amounts[0] = (uint256(balanceChange) * operatorRewardPercentage)
/ 10000;
    for (uint256 i = 1; i < receivers.length; ++i) {
        receivers[i] = fees[i - 1].receiver;
        amounts[i] = (uint256(balanceChange) * fees[i -
1].basisPoints) / 10000;
    }
} else if (balanceChange < 0) {
    // @audit - reset totalDeposits expectations
    totalDeposits -= uint256(balanceChange * -1);
}
```

Recommended mitigation

Keep account of the running track record of the vault and only give tokens when the strategy has appreciated in value since its high point.

Team response

Acknowledged. This is intended.

Low severity findings

TRST-L-1 DelegatorPool assumes tokens array will not be changed

- **Category:** Validation issues
- **Source:** DelegatorPool.sol
- **Status:** Acknowledged

Description

The migration to the SDLPool will move funds from the DelegatorPool and withdraw any user rewards. In *retireDelegatorPool()* and *migrate()*, the code assumes **tokens[0]** is the SDL token. However, the owner can change the token array freely using *addToken()* and *removeToken()* functions. If at migration-time **tokens[0]** will be different, the user's rewards will not be collected.

```
IRewardsPool rewardsPool = tokenPools[tokens[0]];
for (uint256 i = 0; i < _lockedAddresses.length; ++i) {
    address account = _lockedAddresses[i];
    uint256 unlockedBalance = availableBalanceOf(account);
    rewardsPool.withdraw(account);
```

Recommended mitigation

Consider implementing a configuration freeze, where tokens can no longer be changed. This can be called before migration to be on the safe side.

Team response

Acknowledged

TRST-L-2 Migration requires the reward pool to satisfy entire requests

- **Category:** Coupling flaws
- **Source:** DelegatorPool.sol
- **Status:** Acknowledged

Description

When migrating, the DelegatorPool first withdraws all the user's rewards, and then moves their funds to the SDLPool. However, it is not guaranteed that the RewardPool will have sufficient funds to accommodate user's rewards. In that case, users will not be able to migrate until the RewardPool is topped up.

```
rewardsPool.withdraw(account);
```

```
function _withdraw(address _account) internal virtual {
    uint256 toWithdraw = withdrawableRewards(_account);
    if (toWithdraw > 0) {
        updateReward(_account);
        userRewards[_account] -= toWithdraw;
        totalRewards -= toWithdraw;
        token.safeTransfer(_account, toWithdraw);
        emit Withdraw(_account, toWithdraw);
    }
}
```

As seen above, RewardPool *withdraw()* does not support partial withdrawals.

Recommended mitigation

Consider storing an **owedAmount** mapping per user. If migration can't reward the full amount, store the amount owed to be collected later, and continue with the migration process.

Team response

Acknowledged.

TRST-L-3 A reentrancy attack allows migration of more tokens than in the user's balance

- **Category:** Reentrancy attacks
- **Source:** DelegatorPool.sol
- **Status:** Acknowledged

Description

The body of *retireDelegatorPool()* does not follow the Check-Effects-Interactions design pattern.

```
for (uint256 i = 0; i < _lockedAddresses.length; ++i) {
    // @audit - CHECK
    address account = _lockedAddresses[i];
    uint256 unlockedBalance = availableBalanceOf(account);
    // @audit - INTERACTION
    rewardsPool.withdraw(account);
    // @audit - EFFECT
    _burn(account, balanceOf(account));
    delete lockedBalances[account];
    delete lockedApprovals[account];
    // @audit - INTERACTION
    if (unlockedBalance != 0) {
        ISDLPool(_sdlPool).migrate(account, unlockedBalance, 0);
        emit Migration(account, unlockedBalance);
    }
}
```

Suppose an attacker gets control of execution in *rewardsPool.withdraw()* and reenters *retireDelegatorPool()*. At this point, the account's balance is still the same, so the 2nd interaction will migrate the balance for the account. Since the check is already passed in the first call's stack frame, it will migrate the balance once more.

Note that this is a low likelihood reentrancy as it can only be triggered by the owner.

Recommended mitigation

A good way to structure code is to ensure the effect is mapped one-to-one with the checks. If the *balanceOf(account)* part is calculated before the interaction, it will commit the function to burn the correct amount of tokens for the migration. After the reentrancy, the second burn will overflow the account balance and revert the transaction.

Team response

Acknowledged.

TRST-L-4 If a user calls initiateUnlock() more than once, the timer will reset

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

Description

In the SDLPool, the *initiateUnlock()* function starts a timer until funds are withdrawable for the user. Notably, the function is missing a validation that lock is currently locked. If it isn't, there's no reason to reset the timer and the function should revert.

Recommended mitigation

Validate that the expiry is currently not set.

Team response

Resolved.

Mitigation review

Additional validation was added successfully.

TRST-L-5 OperatorVault does not support deregistration of operators

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

Description

Operators are nominated when adding a vault in OperatorVCS. It is reasonable at some point, operators would want to retire, or that the VCS would need to dispose of them due to misbehavior. The code does not facilitate this important lifecycle change.

Recommended mitigation

Consider adding the recommended functionality.

Team response

Acknowledged

Additional recommendations

TRST-R-1 Use of licensed code without credit

The SDLPool makes use of OpenZeppelin's ERC721.sol code. The OZ code is BSD-licensed, therefore copying it without including the original license and crediting OZ is a breach of intellectual property law.

```
function _checkOnERC721Received(
    address _from,
    address _to,
    uint256 _lockId,
    bytes memory _data
) private returns (bool) {
    if (_to.code.length > 0) {
        try IERC721Receiver(_to).onERC721Received(msg.sender, _from,
_lockId, _data) returns (bytes4 retval) {
            return retval ==
IERC721Receiver.onERC721Received.selector;
        } catch (bytes memory reason) {
            if (reason.length == 0) {
                revert TransferToNonERC721Implementer();
            } else {
                assembly {
                    revert(add(32, reason), mload(reason))
                }
            }
        }
    } else {
        return true;
    }
}
```

TRST-R-2 getLockIdsByOwner() does not validate all locks were found

The *getLockIdsByOwner()* function will iterate on all locks. It keeps track of the number of locks belong to the owner found.

```
function getLockIdsByOwner(address _owner) external view returns
(uint256[] memory) {
    uint256 maxLockId = lastLockId;
    uint256 lockCount = balanceOf(_owner);
    uint256 lockIdsFound;
    uint256[] memory lockIds = new uint256[](lockCount);
    for (uint256 i = 1; i <= maxLockId; ++i) {
        if (lockOwners[i] == _owner) {
            lockIds[lockIdsFound] = i;
            lockIdsFound++;
            if (lockIdsFound == lockCount) break;
        }
    }
    return lockIds;
}
```

It would be good invariant validation to require that **lockIdsFound == lockCount**.

TRST-R-3 Waste of gas due to passing 0x00 as calldata

In numerous places in the codebase, when passing calldata to *transferAndCall()*, the contract passes **"0x00".** In fact, this would correspond to 4 non-zero bytes which cost gas as they are part of the calldata. It is recommended to pass **""** unless there is a specific reason to do so.

TRST-R-4 Upgradeable contract gaps are smaller than expected

Gaps are used for abstract, upgradeable contracts. However, the industry standard of reserving 50 slots for future expansion of the contract is not kept.

```
uint256[10] private __gap;
```

TRST-R-5 Lack of validation in getLocks()

The *getLocks()* function is assumed to revert when one of the inputs in **lockIds** is invalid.

```
function getLocks(uint256[] calldata _lockIds) external view returns
(Lock[] memory) {
    uint256 maxLockId = lastLockId;
    Lock[] memory retLocks = new Lock[](_lockIds.length);
    for (uint256 i = 0; i < _lockIds.length; ++i) {
        uint256 lockId = _lockIds[i];
        if (lockId == 0 || lockId > maxLockId) revert
InvalidLockId();
        retLocks[i] = locks[lockId];
    }
    return retLocks;
}
```

However, it does not check the lockId does not refer to a consumed and empty lock. Consider checking that it has an owner through the **lockOwners** mapping.

TRST-R-6 Unsafe casting to uint128

After the initial audit, the OperatorVault introduced some uint128 state variables. It should be noted that they are used in many instances with unsafe casting from uint256 to uint128. An example is shown below.

```
function updateDeposits() external onlyVaultController returns
(uint256, uint256) {
    uint256 totalDeposits = getTotalDeposits();
    int256 depositChange = int256(totalDeposits) -
int256(uint256(trackedTotalDeposits));
```

```
    if (depositChange > 0) {
        uint256 rewards = (uint256(depositChange) *
IOperatorVCS(vaultController).operatorRewardPercentage()) / 10000;
        unclaimedRewards += uint128(rewards);
        trackedTotalDeposits = uint128(totalDeposits);
        return (totalDeposits, rewards);
    }
    return (totalDeposits, 0);
}
```

It is recommended to verify the values do not exceed the target size before casting. Usage of the SafeCast.sol library is encouraged.

Centralization risks

TRST-CR-1 Vault deployment and upgrades introduce draining risks

OperatorVCS can deploy new vaults for Operators and is authorized to upgrade them through *upgradeVaults()*.

```
function upgradeVaults(
    uint256 _startIndex,
    uint256 _numVaults,
    bytes memory _data
) external onlyOwner {
    for (uint256 i = _startIndex; i < _startIndex + _numVaults; ++i)
{
        _upgradeVault(i, _data);
    }
    emit UpgradedVaults(_startIndex, _numVaults, _data);
}
```

This ability represents several centralization risks, if the owner account were to be hacked:

1. A malicious vault could request to withdraw all the rewards available on the VCS, as it trusts the *withdrawVaultRewards()* call.
2. In the future, it could unstake from Chainlink and rug all associated funds.
3. New malicious vaults could be created to consume all available tokens in the StakingPool and then steal them.

TRST-CR-2 Migration to SDLPool introduces several risks

The migration process trusts the incoming parameters, **lockedAddresses** and **sdlPool**.

1. If **lockedAddresses** is supplied incorrectly, it could lead to issues like H-1.
2. If **sdlPool** is malicious, it could steal all SDL tokens in the contract.

TRST-CR-3 SDL owner can freeze user's assets in the SDLPool

ERC677 tokens allow the owner to call *onTokenTransfer()* on the recipient with an artificial message sender, using the call flow below.

```
function mintToContract(
    address _contract,
    address _account,
    uint256 _amount,
    bytes calldata _calldata
) public onlyOwner {
    _mint(msg.sender, _amount);
    transferAndCallWithSender(_account, _contract, _amount,
_calldata);
}
```

```
function transferAndCallWithSender(
    address _sender,
    address _to,
    uint256 _value,
    bytes calldata _data
) private returns (bool) {
    require(isContract(_to), "to address has to be a contract");
    super.transfer(_to, _value);
    contractFallback(_sender, _to, _value, _data);
```

```
function contractFallback(
    address _sender,
    address _to,
    uint256 _value,
    bytes memory _data
) internal {
    IERC677Receiver receiver = IERC677Receiver(_to);
    receiver.onTokenTransfer(_sender, _value, _data);
}
```

So, they can call SDLPool's *onTokenTransfer()* on behalf of an owner of an existing lock.

```
function onTokenTransfer(
    address _sender,
    uint256 _value,
    bytes calldata _calldata
) external override {
    if (msg.sender != address(sdlToken) &&
!isTokenSupported(msg.sender)) revert UnauthorizedToken();
    if (_value == 0) revert InvalidValue();
    if (msg.sender == address(sdlToken)) {
        (uint256 lockId, uint64 lockingDuration) =
abi.decode(_calldata, (uint256, uint64));
        if (lockId > 0) {
            _updateLock(_sender, lockId, _value, lockingDuration);
        } else {
            _createLock(_sender, _value, lockingDuration);
        }
    } else {
        distributeToken(msg.sender);
    }
}
```

They could specify a maximum **lockingDuration**, which would make **existing** tokens in that **lockId** stuck until the specified date. This could be repeated to effectively freeze a user's asset.

Systemic risks

TRST-SR-1 Stakers can lose value when Operator does not behave well

The OperatorVCS and OperatorVault reward allocation depend on the Operator to behave well and attain good yield. If the Operator was to misbehave, they could up to all funds staked in the specific OperatorVault contract.

TRST-SR-2 Rewards depend on balance of RewardPool

Even if Operators behave well, their rewards depend on the replenishing of the rewards in the VCS contract with stLINK tokens from the staking pool. It is not guaranteed that tokens will be available.

TRST-SR-3 Chainlink integration risks

The OperatorVault contract integrates with Chainlink's staking controller contract. It is important to note that Chainlink does not currently support withdrawing of staked LINK and its behavior is expected to change significantly in the iterations that will follow.