STADER

Stader SD Utility Pool

Smart Contract Security Review

Version: 2.1

January, 2024

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Stader Utility pool smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Stader smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Stader smart contracts in scope.

Overview

The Stader Utility Pool (SD Utility Pool) is a feature designed to enhance the utility and stability of `SD` tokens within the Stader ecosystem.

The SD Utility Pool was introduced to eliminate the barrier for node operators who needed `SD` tokens to operate `ETHx` nodes. By allowing node operators to utilize `SD` tokens from the Utility Pool for a fee, thereby removing their need to hold `SD` tokens directly.

For `SD` Holders, it provides an opportunity for holders to earn delegation fees. Additionally, it helps reduce selling pressure on `SD` tokens, increases demand, and contributes to price stability.

Security Assessment Summary

This review was conducted on the files hosted on the ETHx repository. Scope of this review was strictly limited to changes introduced in PR 212.

Retesting was performed on commit 21ba418.

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

The manual code review section of the report is focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [?, ?].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 19 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.
- High: 4 issues.
- Medium: 5 issues.
- Low: 2 issues.
- Informational: 7 issues.

Note: considering the large number of critical/high severity issues identified during this time-boxed engagement, Sigma Prime recommends further security testing on the code base in scope prior to any deployment.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Stader smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
SDP-01	Precision Issue In Health Factor Calculation	Critical	Resolved
SDP-02	Operator Can Continue To Utilize The Protocol After Being Liquidated	High	Resolved
SDP-03	Persistent Debt Post Liquidation	High	Resolved
SDP-04	Malicious Reward Address Blocking Liquidation	High	Resolved
SDP-05	WETH Stuck In `OperatorRewardsCollector` During Liquidation Process	High	Resolved
SDP-06	Fee Accounting Rounding Favours Users Over Protocol	Medium	Resolved
SDP-07	Price Inflation Of `cTokenShare` When Supply Is Zero	Medium	Resolved
SDP-08	Non-Reentrant Modifier Conflict	Medium	Resolved
SDP-09	Lost `SD` Rewards	Medium	Resolved
SDP-10	Rounding Error Causing Loss Of Funds	Medium	Resolved
SDP-11	`claim()` Function Always Reverts If Liquidation Occurred	Low	Resolved
SDP-12	Modifications To Rewards And Fees Can Apply Retroactively	Low	Resolved
SDP-13	Missing Price Staleness Checks For `SD/ETH` Oracle	Informational	Closed
SDP-14	Lack Of Slippage Parameter During Withdrawals	Informational	Closed
SDP-15	Precision Loss In Reward Calculation	Informational	Closed
SDP-16	Potentially Excessive `SD/ETH` TWAP Time Window	Informational	Closed
SDP-17	Small Precision Loss In `requestWithdraw()`	Informational	Closed
SDP-18	Operator Can Grief Liquidations	Informational	Closed
SDP-19	Miscellaneous General Comments	Informational	Resolved

SDP-01	Precision Issue In Health Factor Calculation		
Asset	SDUtilityPool.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

All healthy positions can be liquidated due to precision issues in calculating `userData.healthFactor`.

Inside `liquidationCall()`, the function expects `userData.healthFactor` to be expressed in 18 decimals:

```
if (userData.healthFactor > DECIMAL) {
    revert NotLiquidatable();
}
```

However, the calculation of `healthFactor` inside `getUserData()` omits any decimals:

```
uint256 healthFactor = (totalInterestSD == 0)
    ? type(uint256).max
    : (totalCollateralInSD * riskConfig.liquidationThreshold) / (totalInterestSD * 100);
```

All health factor values returned by `getUserData()` will be less than `DECIMAL` (`1e18`), and hence, all positions can be liquidated even if they are healthy.

Recommendations

Scale the calculation of `healthFactor` up by 18 decimals.

```
uint256 healthFactor = (totalInterestSD == 0)
    ? type(uint256).max
    : (totalCollateralInSD * riskConfig.liquidationThreshold * DECIMALS) / (totalInterestSD * 100);
```

Resolution

Decimal scaling was added to `getUserData()`.

This issue has been addressed in commit 21ba418.

SDP-02	Operator Can Continue To Utilize The Protocol After Being Liquidated		
Asset	SDUtilityPool.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

In the `SDUtilityPool` the functions `utilize()` and `utilizeWhileAddingKeys()` lack necessary checks to verify the operator's current status. As a result, operators who have already been liquidated, or those with an unhealthy health factor, are still able to call these functions to further utilize from the pool.

A significant concern arises when an operator, having already undergone liquidation, continues to operate even if their health factor deteriorates to an unhealthy level. The system's current logic prevents an account from being liquidated more than once, preventing operators from subsequent liquidations, regardless of their health factor status. This could potentially lead to protocol insolvency.

Additionally, this issue has a cascading effect on the `OperatorRewardsCollector.withdrawableInEth()` function. An unhealthy health factor could trigger a revert due to an underflow issue on line [71] of `OperatorRewardsCollector`. This underflow results in funds getting stuck in the contract during the claim process.

Recommendations

Implement checks within `utilize()` and `utilizeWhileAddingKeys()` functions, to ensure that the operator has not been already liquidated and ensure the health factor is above the liquidation threshold.

Resolution

Validation was added to `_utilize()` to check that the operator has not been already liquidated and has a good health factor.

`OperatorRewardsCollector.withdrawableInEth()` now returns `0` if there isn't enough collateral to cover total `SD` interest and open liquidations.

This issue has been addressed in commit 21ba418.

SDP-03	Persistent Debt Post Liquidation		
Asset	OperatorRewardsCollector.sol, SDUtilityPool.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Pending and total interests are not correctly addressed in the liquidation logic.

In `SDUtilityPool.liquidationCall()`, the liquidator pays off all of the utilizer's interest. To reset the utilizer's tracked total interest back to 0, their `utilizeIndex` is updated to the global `utilizeIndex` value.

```
utilizerData[account].utilizeIndex = utilizeIndex
```

However, if the utilizer has updated their `UtilizerStruct` by calling `_utilize()` or `_repay()` after their initial utilization, then updating the utilizer's `utilizeIndex` does not reset the utilizer's tracked total interest back to 0. This is because `_utilize()` and `_repay()` compound the utilizer's interest by adding any pending interest back to their tracked `principal` amount and updating their `utilizeIndex`.

In `_utilize()`:

```
uint256 accountUtilizedPrev = _utilizerBalanceStoredInternal(utilizer);

utilizerData[utilizer].principal = accountUtilizedPrev + utilizeAmount;
utilizerData[utilizer].utilizeIndex = utilizeIndex;
totalUtilizedSD += utilizeAmount;
```

In `_repay()`:

```
uint256 feeAccrued = accountUtilizedPrev -
    ISDCollateral(staderConfig.getSDCollateral()).operatorUtilizedSDBalance(utilizer);
if (!staderConfig.onlyStaderContract(msg.sender, staderConfig.SD_COLLATERAL())) {
    if (repayAmountFinal > feeAccrued) {
        ISDCollateral(staderConfig.getSDCollateral()).reduceUtilizedSDPosition(
            utilizer,
            repayAmountFinal - feeAccrued
        );
    }
}
feePaid = Math.min(repayAmountFinal, feeAccrued);
utilizerData[utilizer].principal = accountUtilizedPrev - repayAmountFinal;
utilizerData[utilizer].utilizeIndex = utilizeIndex;
```

This can cause several issues:

1. During liquidation call, the liquidator pays for the utilizer's entire `totalInterestSD`, but not all or even none of the utilizer's debt is cleared.
2. Since the liquidation process does not adequately address the total interest due, the `OperatorRewardsCollector.claimFor()` function can potentially revert if:
   (a) the utilizer has no remaining active keys and needs to withdraw their utilized SD balance. The utilizer will not have enough SD balance and revert on line [132] of `SDCollateral.withdrawOnBehalf()`.

(b) the operator's health factor falls below `1e18`. An underflow issue on line [71] of `OperatorRewardsCollector.withdrawableInEth()` will cause the call to revert.

The issue can be exploited as follows:

1. *Set Up:* Initialize variables and deposit amounts for the liquidation scenario.
2. *Operator Action:* The operator (Bob) utilizes `SD` from the `SDUtilityPool`, leading to the accrual of fees.
3. *Interest Accrual:* Allow significant fees to accrue over time, simulating long-term use of the pool.
4. *Repay Zero Amount:* Bob attempts to repay a zero amount, which updates his `utilizeIndex` but does not affect his total interest due.
5. *Liquidation Call:* Alice initiates a liquidation call against Bob.
6. *Post-Liquidation Check:* Despite the liquidation process, Bob's total interest remains unchanged, demonstrating that the liquidation did not clear Bob's debt.

Recommendations

Ensure that both pending and total interests are fully addressed in the liquidation logic of `SDUtilityPool` by calling `_repay()` to handle the clearing of debt instead of updating the utilizer's `utilizeIndex`.

This change will ensure the operator's financial obligations are completely resolved post-liquidation, thereby restoring the health of their position.

Resolution

`liquidationCall()` was modified to call `repay()`.

`OperatorRewardsCollector.withdrawableInEth()` now returns `0` if there isn't enough collateral to cover total `SD` interest and open liquidations.

This issue has been addressed in commit 21ba418.

SDP-04	Malicious Reward Address Blocking Liquidation		
Asset	`OperatorRewardsCollector.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

In `OperatorRewardsCollector._claim()`, the function responsible for finalizing liquidations and transferring `ETH` to the operator reward address is vulnerable to Denial-of-Service (DoS) attacks. An operator can set a malicious reward address that causes the transaction to revert, effectively preventing the liquidation process.

The process can be outlined as such:

1. *Initial Setup:* Alice delegates a specified amount of `SD` as collateral.
2. *Validator Preparation:* Bob adds a new validator using `SD` from `SDUtilityPool`.
3. *Fees Accrual:* Accrue fees for several blocks so that Bob's health factor becoming unhealthy.
4. *Bob's Liquidation:* Alice liquidates Bob's position by calling `liquidationCall()`.
5. *Operator Reward Address Change:* Bob maliciously changes his reward address to a contract designed to revert transactions when receiving `ETH`.
6. *Finalize Liquidation:* Alice calls `claimFor()` on Bob to complete the liquidation. The `amount` specified when calling `claimFor()` will be non-zero (since zero represents Bob's full balance).
7. *Transaction Failure:* Since the `amount` is non-zero, in the function `_claim()`, there will be an attempt to transfer the `amount` to the malicious reward address, which will revert, preventing the completion of the liquidation process.

Recommendations

Convert the `ETH` balance to `WETH`, similar to how the liquidator is paid, to prevent a malicious reward address from blocking the liquidation process.

Resolution

A new function `claimLiquidation()` was added for the liquidator to claim their portion separately.

This issue has been addressed in commit 21ba418.

SDP-05	WETH Stuck In `OperatorRewardsCollector` During Liquidation Process		
Asset	`OperatorRewardsCollector.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

In `OperatorRewardsCollector.sol`, an incorrect amount of `ETH` is converted into `WETH` in the `_completeLiquidationIfExists()` function, resulting in excess `WETH` being stuck in `OperatorRewardsCollector` contract.

Specifically, the contract does not account for the `operatorLiquidation.totalFeeInEth` when depositing into `WETH`, converting more than necessary to pay the liquidator, which leaves an amount of `WETH` stuck in the contract.

This issue leads to operators being unable to withdraw their full remaining balance, gradually making the protocol insolvent as stuck funds accumulate.

Recommendations

Adjust the deposit logic in `OperatorRewardsCollector` to ensure that the `operatorLiquidation.totalFeeInEth` is excluded from the `WETH` deposit. This change will prevent funds from being locked in the contract and allow operators to fully withdraw their balances post-liquidation.

Resolution

`_completeLiquidationIfExists()` was modified according to the recommendations.

This issue has been addressed in commit 21ba418.

SDP-06	Fee Accounting Rounding Favours Users Over Protocol		
Asset	`SDUtilityPool.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

Rounding errors in `accrueFee()` may result in accrual of bad debt over time.

Inside the `accrueFee()` function, fees are accrued by scaling both `totalUtilizedSD` and `utilizeSD` up by the `simpleFeeFactor`.

```
/*
 * Calculate the fee accumulated into utilized and totalProtocolFee and the new index:
 *  simpleFeeFactor = utilizationRate * blockDelta
 *  feeAccumulated = simpleFeeFactor * totalUtilizedSD
 *  totalUtilizedSDNew = feeAccumulated + totalUtilizedSD
 *  totalProtocolFeeNew = feeAccumulated * protocolFeeFactor + totalProtocolFee
 *  utilizeIndexNew = simpleFeeFactor * utilizeIndex + utilizeIndex
 */

uint256 simpleFeeFactor = utilizationRatePerBlock * blockDelta;
uint256 feeAccumulated = (simpleFeeFactor * totalUtilizedSD) / DECIMAL;
totalUtilizedSD += feeAccumulated;
accumulatedProtocolFee += (protocolFee * feeAccumulated) / DECIMAL;
utilizeIndex += (simpleFeeFactor * utilizeIndex) / DECIMAL;
```

Since `utilizeIndex < totalUtilized`, there is a small discrepancy between the total amount of fees that accrue to `totalUtilizedSD` and the amount of fees that accrue to each utilizer via `utilizeIndex`, which results in rounding errors from integer division.

Due to this behaviour, utilizers end up paying less interest than recorded (and claimable by delegators), potentially resulting in bad debt.

Recommendations

Although the rounding error is minimal, it is still preferable for any rounding to favour the protocol over users.

Consider rounding up the accruing of fees to `utilizeIndex` and calculations of utilizer balances. If this is done, then the `_repay()` function needs to be adjusted to account for potential underflow scenarios.

An example of what the `_repay()` adjustment may look like:

```
utilizerData[utilizer].principal = accountUtilizedPrev - repayAmountFinal;
utilizerData[utilizer].utilizeIndex = utilizeIndex;
totalUtilizedSD = totalUtilizedSD > repayAmountFinal ? totalUtilizedSD - repayAmountFinal : 0;
emit Repaid(utilizer, repayAmountFinal);
```

Resolution

Calculation of `utilizeIndex` was modified to round up and `_repay()` modified according to the recommendations.

This issue has been addressed in commit 21ba418.

SDP-07	Price Inflation Of `cTokenShare` When Supply Is Zero		
Asset	`SDUtilityPool.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

If the current supply is zero, an attacker can perform a share inflation attack during delegation.

This can be illustrated through the following steps:

1. Alice wants to delegate 1 `SD` token (which has 18 decimals) to the utility pool calling `delegate()`.
2. The pool is empty. The exchange rate is the default 1 `SD` per `cTokenShare`.
3. Bob sees Alice's transaction in the mempool and decides to sandwich it.
4. Bob delegates 1 `wei` of `SD` and receives 1 `wei` of `cTokenShare` in exchange, The exchange rate is now 1 `SD` per `cTokenShare`.
5. Bob transfers 1 `SD` (`1e18 wei`) to the vault using an `ERC-20` transfer. No new `cTokenShares` are created. Hence, the exchange rate is now `1e18 + 1 SD` per `cTokenShare`, or `1e18 + 1 wei` of `SD` per `wei` of `cTokenShare`.
6. Alice's deposit is executed. Her `1e18 wei` of `SD` tokens are worth less than `1 wei` of `cTokenShare`. Therefore, the contract takes the assets, but does not add shares. Alice has effectively *"donated"* her tokens.

Recommendations

Consider implementing a decimal offset virtual shares and assets to the pool.

See the following for more details: Addressing Inflation Attacks With Virtual Shares And Assets

Resolution

The Stader Team has elected to resolve this issue using an initial delegate of 1 SD during the initialization.

This issue has been addressed in commit 21ba418.

SDP-08	Non-Reentrant Modifier Conflict		
Asset	`PermissionlessNodeRegistry.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

In the `PermissionlessNodeRegistry` contract, the function `addValidatorKeys()` is marked with two `nonReentrant` modifiers.

This double application of the `nonReentrant` modifier can lead to unexpected behaviour, causing the function to revert due to the reentrancy guard.

Recommendations

Remove the redundant modifier to prevent unwarranted reverts and align with standard smart contract practices.

Resolution

The redundant modifier was removed on `addValidatorKeys()`.

This issue has been addressed in commit 21ba418.

SDP-09	Lost SD Rewards		
Asset	SDIncentiveController.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

SD rewards are lost if no one delegates or requests withdrawals before `rewardEndBlock`.

In `SDIncentiveController`, the `rewardPerToken()` function calculates the current reward-per-token value based on the current `block.number`. The following lines determine the reward-per-token once rewards have ended:

```
if (block.number >= rewardEndBlock) {
    return rewardPerTokenStored;
}
```

Calling `updateReward()` assigns `rewardPerTokenStored = rewardPerToken()`. If `updateReward()` has not been called at `block.number == rewardEndBlock - 1`, then rewards in the range `[lastUpdateBlockNumber, rewardEndBlock - 1]` do not get accrued and are lost, since `rewardPerToken()` will always return the outdated `rewardPerTokenStored`.

`updateReward()` is only called if a delegator delegates or requests the withdrawal of SD. Hence, if there are no delegations or withdrawal requests right before rewards inside `SDIncentiveController` end, then delegators lose out on SD incentive rewards.

Furthermore, since `>=` is used in the comparison, rewards only accrue up to block number `rewardEndBlock - 1`. SD that is delegated inside block number `rewardEndBlock - 1` will not accrue any rewards.

Recommendations

Use `>` instead of `>=` in the comparison. This will ensure that rewards are accrued up to block number `rewardEndBlock`.

Consider adding logic into `rewardPerToken()` to account for the case where `block.number > rewardEndBlock` but `rewardPerTokenStored` is outdated. Here's an example:

```
if (block.number > rewardEndBlock) {
    // If the last update block is before the end of the reward period,
    // calculate the reward per token at the end of the reward period
    if (lastUpdateBlockNumber < rewardEndBlock) {
        return rewardPerTokenStored +
            (((rewardEndBlock - lastUpdateBlockNumber) * emissionPerBlock * DECIMAL) /
                ISDUtilityPool(staderConfig.getSDUtilityPool()).cTokenTotalSupply());
    }
    return rewardPerTokenStored;
}
```

Resolution

`block.number` was replaced with `_lastRewardTime()` which takes into account the rewards up to the `rewardEndBlock`.

This issue has been addressed in commit 21ba418.

SDP-10	Rounding Error Causing Loss Of Funds		
Asset	OperatorRewardsCollector.sol, SDUtilityPool.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

There is a rounding error in the `OperatorRewardsCollector.withdrawableInEth()` function, such that it can cause a scenario where an operator can lose a small amount of ETH even when not utilizing the SD Utility Pool.

When calculating total collateral in `withdrawableInEth()`, the collateral value is converted from ETH to SD, then back to ETH. This conversion process, which occur in the `getUserData()` and `withdrawableInEth()` functions, will cause `withdrawableInEth()` to ultimately round down the amount of ETH the operator is entitled to withdraw.

The conversion process is as follows:

In `SDUtilityPool.getUserData()`:

```
uint256 totalCollateralInSD = ISDCollateral(staderConfig.getSDCollateral()).convertETHToSD(
    totalCollateralInEth
);
```

In `OperatorRewardsCollector.withdrawableInEth()`:

```
uint256 availableBalance = ISDCollateral(staderConfig.getSDCollateral()).convertSDToETH(withdrawableInSd);
```

This rounding down effect can lead to scenarios where operators are unable to withdraw their full entitled ETH balance, despite not having any outstanding interest or open liquidations.

Recommendations

Consider calculating `withdrawableInEth` instead of `withdrawableInSd` by changing `userData.totalInterestSD` into ETH.

This will result in a small rounding down in favour of the user in `withdrawableInEth()`, so we recommend that the total interest be rounded up.

Resolution

The `UserData` struct was modified to return `totalCollateralInEth` and calculation of `totalInterestAdjustedInEth` rounds up in `withdrawableInEth()`.

This issue has been addressed in commit 73514c3.

SDP-11	`claim()` Function Always Reverts If Liquidation Occurred		
Asset	`OperatorRewardsCollector.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

In the `OperatorRewardsCollector` contract, the `claim()` function will always revert when there is an existing liquidation for the operator. This is due to the `claim()` function attempting to claim the operator's full balance without taking into account the amount that will be paid to settle any liquidations.

Consequently, `claim()` does not work as intended and operators are forced to use the alternative `claimFor()` function, specifying an amount less than their total rewards balance, to successfully claim their rewards.

The issue can be illustrated through the following steps:

1. *Liquidation Setup:* Initiate a liquidation scenario for an operator.
2. *Attempted Claim Process:* The operator (Bob), after making a deposit (e.g. 5 ETH), tries to claim rewards using the `claim()` function.
3. *Revert on Claim Attempt:* The call to `claim()` reverts due to the ongoing liquidation, even though the operator is entitled to a certain amount of rewards.

Recommendations

The testing team recommends revising the `claim()` function to take into account the amount paid for liquidations, so that the remainder operator balance can be successfully withdrawn.

Resolution

The `claim()` function was modified to pay for any liquidations first before calculating the withdrawable amount.

This issue has been addressed in commit 21ba418.

SDP-12	Modifications To Rewards And Fees Can Apply Retroactively		
Asset	SDUtilityPool.sol, SDIncentiveController.sol		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

In the `SDIncentiveController` contract, the `updateEmissionRate()` function allows for modifying the emission rate even during an ongoing reward period. This could impact the reward calculation for blocks that have not yet been updated using the old emission rate.

Specifically, if `updateReward()` has not been called before updating the emission rate, the new emission rate could retroactively affect the accrual of rewards for past blocks. This situation creates inconsistency in the reward calculations, potentially benefiting some users while disadvantaging others.

The same issue exists inside the `SDUtilityPool` contract with the `updateProtocolFee()` and `updateUtilizationRatePerBlock()` functions. Changing the protocol fee or utilization rates when `accrueFee()` has not been called in the same block impacts the fee calculation for blocks that have not been updated yet using the old rates.

Recommendations

In `SDIncentiveController`, call `updateReward(address(0))` at the start of the `updateEmissionRate()` function.

In `SDUtilityPool`, call `accrueFee()` at the start of the `updateProtocolFee()` and `updateUtilizationRatePerBlock()` functions.

These changes ensure that old rewards and fees have been accrued with the current rates before any changes are made, maintaining consistency in reward and fee calculations across all blocks.

Resolution

`SDIncentiveController` was modified so that it is not possible to change the emission rate during a reward block.

`SDUtilityPool` was modified according to the recommendations.

This issue has been addressed in commit 21ba418.

SDP-13	Missing Price Staleness Checks For `SD/ETH` Oracle
Asset	`SDCollateral.sol, StaderOracle.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

`convertETHToSD()` and `convertSDToETH()` use the `StaderOracle` to grab the `SD/ETH` price. However, there are no price staleness checks performed.

If the oracle was to halt such as in the case where not enough trusted nodes submit a price, an outdated and incorrect price would be used.

Recommendations

Consider modifying the `StaderOracle.getSDPriceInETH()` function to also return the `reportingBlockNumber`.

This addition would allow for an evaluation of the price information. By establishing and applying a block threshold, it would be possible to determine whether the current price is stale.

Resolution

The issue was acknowledged by the project team with the following comment:

> *"Stader ETHx is supported by 7 Oracles (Stader guardians), and they have worked efficiently in the 7 months of our operation. The correct SD price is provided every 24 hours, and there has never been an incident where the price was not updated."*

SDP-14	Lack Of Slippage Parameter During Withdrawals
Asset	`SDUtilityPool.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

`SDUtilityPool` enforces a mandatory delay period, termed as `minBlockDelayToFinalizeRequest`, which spans 7 days between invoking the `requestWithdraw()` function and the subsequent `finalizeDelegatorWithdrawalRequest()`.

During this period, potential fluctuations in the exchange rate can occur.

This is particularly relevant as the `finalizeDelegatorWithdrawalRequest()` function computes `minSDRequiredToFinalizeRequest` based on the prevailing exchange rate at the time of finalization.

Recommendations

Consider allowing the user to specify a slippage parameter to enable users to specify the degree of variation in SD they are willing to accept at the time of withdrawal completion.

Resolution

The issue was acknowledged by the project team with the following comment:

> *"The Utility Pool ER will go down in an extremely rare scenario where most of the ETHx node operators are slashed. Even in such rare cases, the ETH deposited by the node operators will take precedence to cover the slashing penalty, followed by their self-bonded SD collateral. Only after exhausting these options will the Utilized SD collateral be used to address any remaining deficiencies. Additionally, every time a user delegates SD to the Utility Pool, we display a disclaimer explaining the slashing risk and obtain confirmation from them before SD delegation."*

SDP-15	Precision Loss In Reward Calculation
Asset	`SDIncentiveController.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

In `SDIncentiveController`, there is a potential, albeit unlikely, risk of precision loss in reward calculations.

This scenario may occur when the `totalSupply` of `cTokens` significantly outweighs the `emissionPerBlock`. Precision loss in reward distribution can lead to users receiving slightly fewer rewards than expected.

Recommendations

Although no immediate fix is necessary, the Stader team should be aware of this potential issue and set reasonable emission rates relative to the total `cToken` supply.

A guideline to prevent significant precision loss is to ensure that the product of `emissionPerBlock` and `1e18` is greater than the `totalSupply`.

Resolution

The issue was acknowledged by the project team with the following comment:

> *"Minimum reward is 1e18."*

SDP-16	Potentially Excessive `SD/ETH` TWAP Time Window
Asset	`StaderOracle.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

According to Stader Labs' documentation, the `SD` price oracle is a 24 hour `TWAP`.

During times of high volatility and price movement, the `SD` price oracle may return a very outdated price. This could result in delayed or omitted liquidations that could lead to bad debt for `SDUtilityPool`.

Recommendations

Consider reducing the time-window of the `TWAP` oracle to a shorter period, such as 1 hour.

Resolution

The issue was acknowledged by the project team with the following comment:

> *"The SD price is updated every 24 hours considering the average price for the day. This is done to offset price fluctuations and for the simplicity of operations."*

SDP-17	Small Precision Loss In `requestWithdraw()`
Asset	`SDUtilityPool.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

There is a small precision loss in `requestWithdraw()` due to division operation occurring before multiplication.

`requestWithdraw()` takes in the `_cTokenAmount` to withdraw and calculates the `sdRequested` using the `_exchangeRateStoredInternal()` function.

```
uint256 exchangeRate = _exchangeRateStoredInternal();
ISDIncentiveController(staderConfig.getSDIncentiveController()).claim(msg.sender);
delegatorCTokenBalance[msg.sender] -= _cTokenAmount;
delegatorWithdrawRequestedCTokenCount[msg.sender] += _cTokenAmount;
uint256 sdRequested = (exchangeRate * _cTokenAmount) / DECIMAL;
```

The exchange rate is calculated by dividing the adjusted pool balance by `cTokenTotalSupply`.

```
/*
 * Otherwise:
 *  exchangeRate = (totalCash + totalUtilizedSD - totalFee) / totalSupply
 */
uint256 poolBalancePlusUtilizedSDMinusReserves = getPoolAvailableSDBalance() +
    totalUtilizedSD -
    accumulatedProtocolFee;
uint256 exchangeRate = (poolBalancePlusUtilizedSDMinusReserves * DECIMAL) / cTokenTotalSupply;
```

This means that a division operation occurs before multiplication, which leads to precision loss in the amount of `sdRequested`. Due to the amount of decimals used in the calculations, the rounding error is small.

Recommendations

Consider allowing the exchange rate function to take in `_cTokenAmount` and calculating the `sdAmountOut`, instead of getting the exchange rate for 1 `cToken` and then multiplying that to get `sdRequested`.

The same fix could be applied to `requestWithdrawWithSDAmount()` by adding another function that calculates the `cTokenIn` amount based on `sdAmountOut`.

Resolution

The issue was acknowledged by the project team with the following comment:

> *"The difference is very minimal and favours protocol over users."*

SDP-18	Operator Can Grief Liquidations
Asset	`OperatorRewardsCollector.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The operator can prevent liquidations by conducting just-in-time ETH transfers to `OperatorRewardsCollector.sol`.

When an operator's health factor becomes unhealthy, indicating potential liquidation, they can utilize the `depositFor()` function in `OperatorRewardsCollector` to temporarily increase their health factor due to increasing their ETH balances.

This increase in health factor can cause any impending liquidation call to revert. Subsequently, the operator can immediately reclaim their balances using the `claim()` function. This enables the operator to grief any liquidations with little cost.

Recommendations

Consider implementing access control on the `depositFor()` function. The access should be restricted such that it can only be invoked from certain contracts: reward vaults and `PermissionlessNodeRegistry`.

Resolution

The issue was acknowledged by the project team with the following comment:

> *"It is a normal form of depositing more collateral to avoid liquidation."*

SDP-19	Miscellaneous General Comments
Asset	All contracts
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Return Value For `finalizeDelegatorWithdrawalRequest()`**

   Related Asset(s): SDUtilityPool.sol

   It is currently not possible to extract the `nextRequestIdToFinalize` variable without listening to emitted events. However, returning the `nextRequestIdToFinalize` variable could be beneficial for testing purposes.

   The following modification could be considered:

```
function finalizeDelegatorWithdrawalRequest() external override whenNotPaused returns (uint256) {
    accrueFee();
    uint256 exchangeRate = _exchangeRateStoredInternal();
    ...
    nextRequestIdToFinalize = requestId;
    sdReservedForClaim += sdToReserveToFinalizeRequests;
    emit FinalizedWithdrawRequest(nextRequestIdToFinalize);

    return nextRequestIdToFinalize;
}
```

2. **`riskConfig` Is Not Initialised In The `initialize()` Function**

   Related Asset(s): SDUtilityPool.sol

   `riskConfig` parameters are not initialised inside the `initialize()` function. They have to be initialised by calling the `updateRiskConfig` function after deployment.

   Consider initializing `riskConfig` in the `initialize()` function.

3. **Simplify The `SDAsCollateral()` Function**

   Related Asset(s): SDCollateral.sol

   The `depositSDAsCollateral()` function can be simplified by calling the `depositSDAsCollateralOnBehalf()` function and following the same pattern as the `withdraw()` and `withdrawOnBehalf()` functions.

   Consider implementing the following refactor of the code:

```
    function depositSDAsCollateral(uint256 _sdAmount) external override {
    depositSDAsCollateralOnBehalf(msg.sender, _sdAmount)
}
```

4. **Naming Convention**

   Related Asset(s): SDIncentiveController.sol, SDCollateral.sol

   (a) `SDIncentiveController.updateReward()`

   (b) `SDCollateral.slashSD()`

Ensure that internal functions have the `_` prefix, for consistency.

5. **`delegatorWithdrawRequestedCTokenCount` mapping is unnecessary**

   Related Asset(s): SDUtilityPool.sol

   The `delegatorWithdrawRequestedCTokenCount` mapping stores the current total amount of cTokens that the delegator has requested to withdraw. However, the mapping is not used in any business logic in any functions and contracts and can be removed to save gas.

   Consider removing the `delegatorWithdrawRequestedCTokenCount` mapping from `SDUtilityPool` to save gas on `SSTORE` operations.

6. **Delegation Limit**

   Related Asset(s): SDUtilityPool.sol

   The documentation for SDUtilityPool mentions there is a 1 SD minimum delegation limit.

   However the minimum is not enforced in the `delegate()` function.

   Ensure that delegate limits are enforced where appropriate.

7. **Reward Update**

   Related Asset(s): SDUtilityPool.sol

   Calling `updateRewardForAccount()` on line [**147**] is redundant as `claim()` was already called on line [**141**], which just updated the reward.

   Additionally, rewards will not be updated anymore from this point on during the withdraw process, since it will take 7 days to finalize.

   Make sure to call `updateRewardForAccount()` during `finalizeDelegatorWithdrawalRequest()` and remove the TODO comment.

8. **Reward Withdraw**

   Related Asset(s): SDUtilityPool.sol

   User will be transferred the full amount of their reward balance immediately when calling `requestWithdrawWithSDAmount()`, even if the withdraw request was just for 1 wei

   Ensure that rewards are sent after the withdraw has been finalized and claimed in `SDUtilityPool.claim()`.

9. **Redundant Check**

   Related Asset(s): SDUtilityPool.sol

   The check `accrualBlockNumber != block.number` is redundant on line `740` as we are setting the `accrualBlockNumber = block.number` inside the `accrueFee()` function that in turn is called by the external `delegate()`

10. **Missing Address Validation**

    Related Asset(s): SDCollateral.sol

    The `depositSDAsCollateralOnBehalf()` function in the `SDCollateral` contract lacks a check for the validity of the `_operator` address. It is important to validate that `_operator` is a non-zero address to avoid potential losses.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The relevant issues have been addressed in commit 21ba418.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `forge` framework was used to perform these tests and the output is given below.

```
Running 2 tests for test/PermissionedNodeRegistry.t.sol:PermissionedNodeRegistryTests
[PASS] test_addValidatorKeys() (gas: 1616462)
[PASS] test_addValidatorKeys_noCollateral() (gas: 892349)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 14.51ms

Running 2 tests for test/SocializingPool.t.sol:SocializingPoolTests
[PASS] test_claim() (gas: 5546580)
[PASS] test_claimAndDepositSD() (gas: 5727432)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 18.87ms

Running 2 tests for test/PermissionlessNodeRegistry.t.sol:PermissionlessNodeRegistryTests
[PASS] test_addValidatorKeys() (gas: 1885894)
[PASS] test_addValidatorKeysWithUtilizeSD() (gas: 2339573)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 24.05ms

Running 4 tests for test/StaderConfig.t.sol:StaderConfigTests
[PASS] test_updateSDIncentiveController_notOwner() (gas: 80122)
[PASS] test_updateSDIncentiveController_proper() (gas: 29849)
[PASS] test_updateSDUtilityPool_notOwner() (gas: 80156)
[PASS] test_updateSDUtilityPool_proper() (gas: 29958)
Test result: ok. 4 passed; 0 failed; 0 skipped; finished in 9.51ms

Running 1 test for test/PoolUtils.t.sol:PoolUtilsTests
[PASS] test_processOperatorExit(address,uint256) (runs: 10000, u: 40051, ~: 40051)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.03s

Running 28 tests for test/SDUtilityPool.t.sol:SDUtilityPoolTests
[PASS] test_Delegate() (gas: 275956)
[PASS] test_FinalizeDelegatorWithdrawalRequest() (gas: 755367)
[PASS] test_FinalizeDelegatorWithdrawalRequestBatch() (gas: 126506756)
[PASS] test_accrueFee_accountingFavoursProtocol() (gas: 153344113)
[PASS] test_accrueFee_precisionLoss() (gas: 733123498)
[PASS] test_deployerSDRewards() (gas: 421702)
[PASS] test_exchangeRateStored() (gas: 19702097)
[PASS] test_firstDelegatorPriceManip() (gas: 499861)
[PASS] test_getUserData_healthFactorDecimals() (gas: 2417467)
[PASS] test_liquidation() (gas: 3145477)
[PASS] test_liquidation_WETHStuck() (gas: 3145735)
[PASS] test_liquidation_claimRevert() (gas: 3145499)
[PASS] test_liquidation_frontrun() (gas: 3016027)
[PASS] test_liquidation_liquidateAnyone() (gas: 2191892)
[PASS] test_liquidation_repayZeroThenLiquidate() (gas: 3387770)
[PASS] test_liquidation_rewardAddressDOS() (gas: 3195935)
[PASS] test_liquidation_underflow() (gas: 3195892)
[PASS] test_repay() (gas: 2660515)
[PASS] test_repayAfterliquidation() (gas: 3381840)
[PASS] test_requestWithdraw() (gas: 437408)
[PASS] test_requestWithdrawWithSDAmount() (gas: 437463)
[PASS] test_requestWithdraw_precisionLoss() (gas: 19421141)
[PASS] test_updateProtocolFee_affectsPastBlocks() (gas: 2441183)
[PASS] test_updateUtilizationRatePerBlock_affectsPastBlocks() (gas: 2382920)
[PASS] test_utilizeThenUtilizeWithZeroAmount() (gas: 2609266)
[PASS] test_utilize_after_liquidation() (gas: 3059778)
[PASS] test_withdrawProtocolFee() (gas: 2670222)
[PASS] test_withdrawProtocolFee_roundDown() (gas: 2593288)
Test result: ok. 28 passed; 0 failed; 0 skipped; finished in 2.53s

Running 3 tests for test/SDIncentiveController.t.sol:SDIncentiveControllerTests
[PASS] test_multipleRewardPeriods() (gas: 903413)
[PASS] test_rewardPerToken_precisionLoss(uint256,uint256) (runs: 100, u: 24579679, ~: 24579679)
[PASS] test_updateReward_losePendingRewards() (gas: 1096168)
Test result: ok. 3 passed; 0 failed; 0 skipped; finished in 9.89s
```

```
Running 6 tests for test/SDCollateral.t.sol:SDCollateralTests
[PASS] test_depositSDAsCollateralOnBehalf(address,uint256) (runs: 10000, u: 235909, ~: 237912)
[PASS] test_depositSDFromUtilityPool(address,uint256) (runs: 10000, u: 224035, ~: 225340)
[PASS] test_getOperatorInfo(address) (runs: 10000, u: 594762, ~: 594763)
[PASS] test_reduceUtilizedSDPosition(address,uint256) (runs: 10000, u: 262908, ~: 262906)
[PASS] test_withdraw(uint256,uint256) (runs: 10000, u: 1261811, ~: 1265154)
[PASS] test_withdrawOnBehalf(uint256,uint256,address) (runs: 10000, u: 1264807, ~: 1267854)
Test result: ok. 6 passed; 0 failed; 0 skipped; finished in 13.04s

Running 4 tests for test/OperatorRewardsCollector.t.sol:OperatorRewardsCollectorTests
[PASS] test_claim(uint256) (runs: 10000, u: 1307357, ~: 1307357)
[PASS] test_claimFor(uint256) (runs: 10000, u: 1306848, ~: 1306848)
[PASS] test_updateWethAddress() (gas: 31571)
[PASS] test_withdrawableInEth_rounding() (gas: 1995917)
Test result: ok. 4 passed; 0 failed; 0 skipped; finished in 16.05s
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact \ Likelihood	Low	Medium	High
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Low	Low	Medium

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

σ'