Stader Labs

ETHx – Phase 2

Smart Contract Security Assessment Report

Version: 2.2

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Stader Labs smart contracts (Phase 2). 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 Labs 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 Labs smart contracts.

Overview

ETHx staking is a liquid ETH staking protocol that is permissionless and non-custodial. It enables users to participate in proof-of-stake validation with amounts smaller than the 32 ETH validator requirement by staking alongside other users.

The protocol is controlled by the Stader DAO which has a governance token called `SD`.

The validators are split between *permissioned* and *permissionless* operators, meaning anyone can elect to run an operator for ETHx staking. Permissionless operators must stake `SD tokens` and offer an `ETH` deposit to disincentivise theft of accrued MEV fees. Permissioned operators are added by Stader and do not have to provide an `ETH` deposit as a result. Earned fees are split between the users, operators and Stader DAO in a yet to be decided manner.

Users receive the `ETHx` token when staking. `ETHx` is a fungible `ERC20` that can be used in the wider DeFi ecosystem to earn further yield.

Security Assessment Summary

This review was conducted on the files hosted on the ethx repository and were assessed at commit eb9140b.

Retesting activities targeted commit 141dcaa and lead to the identification of the following additional findings: `ETHX2-18`, `ETHX2-19`, `ETHX2-20`, `ETHX2-21`, and `ETHX2-22`.

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 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 [1, 2].

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 22 issues during this assessment. Categorised by their severity:

- High: 3 issues.
- Medium: 6 issues.
- Low: 3 issues.
- Informational: 10 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Stader Labs 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
ETHX2-01	Malicious Node Operators Can Cause a DoS through `DistributeRewards()` And `SettleFunds()`, Locking User Funds	High	Resolved
ETHX2-02	DoS In `StakeUserETHToBeaconChain()` Due To Forced ETH Transfer Through `SelfDestruct`	High	Resolved
ETHX2-03	Duplicate `PoolIds` Can Cause Loss Of Funds For Users	Medium	Resolved
ETHX2-04	`StaderConfig` Cannot Be Updated	Medium	Resolved
ETHX2-05	`HasEnoughSDCollateral()` Check Is Performed Only Once During Onboarding	Medium	Closed
ETHX2-06	Submit Functions Are Susceptible To Front Running When Trusted Nodes Are Removed	Medium	Closed
ETHX2-07	No Minimum Trusted Node Validation	Medium	Resolved
ETHX2-08	Uninitialized `WithdrawDelay`	Medium	Closed
ETHX2-09	Operator Reward Address Modification Using Hot Wallet	Low	Closed
ETHX2-10	Partial DoS Possible For User Withdrawals	Low	Resolved
ETHX2-11	Centralisation Risk Due To Extensive System Upgradability	Informational	Closed
ETHX2-12	Phantom Overflow In `CalculateRewardShare()`	Informational	Closed
ETHX2-13	Potential Incorrect ETH Distribution In `ValidatorWithdrawalVault`	Informational	Closed
ETHX2-14	`BidIncrement` Can Be Changed Using `UpdateBidIncrement()` Which Affects Ongoing Auctions	Informational	Closed
ETHX2-15	Centralisation Risk In `MaxApproveSD()`	Informational	Resolved
ETHX2-16	`TrustedNode` Cannot Update Incorrectly Submitted Oracle Details	Informational	Closed
ETHX2-17	Miscellaneous General Comments	Informational	Resolved
ETHX2-18	OperatorRewardsCollector Missing Call To OZ `_disableInitializers`	High	Resolved
ETHX2-19	Lack Of External Contract Existence Check On DelegateCall	Low	Closed
ETHX2-20	`VaultProxy` Does Not Follow Standard Proxy Pattern Usage	Informational	Closed
ETHX2-21	`Owner` In `VaultProxy` Does Not Adhere To The Single Source Of Truth Principle	Informational	Resolved
ETHX2-22	Miscellaneous General Comments — Round 2	Informational	Closed

ETHX2-01	Malicious Node Operators Can Cause a DoS through `DistributeRewards()` And `SettleFunds()`, Locking User Funds		
Asset	`ValidatorWithdrawalVault.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

During the onboarding process, node operators - irrespective of whether they are permissioned or permissionless - can define their unique `operatorRewardAddress` when invoking the `onboardNodeOperator()` function. Presently, the system only checks to ensure that the reward address does not equal to the zero address.

This current system, however, leaves room for abuse. A malicious node operator can, during onboarding, deliberately set their `operatorRewardAddress` to a contract address. When the `distributeRewards()` or `settleFunds()` functions are triggered, the contract can cause a reversion at line [75] and line [100] respectively in `ValidatorWithdrawalVault`:

```
sendValue(getNodeRecipient(), operatorShare);
```

Moreover, node operators have can alter their `operatorRewardAddress` at will through the `updateOperatorDetails()` function. This allows malicious operators to manipulate when to trigger a reversion in `distributeRewards()` and `settleFunds()` functions.

Recommendations

The Pull over Push pattern could be used to ensure that user and protocol shares can be distributed even if the `operatorRewardAddress` is assigned to a malicious contract.

One viable solution could involve maintaining a record of the `nodeRefundBalance`.

Resolution

Implemented pull payment over push. Rewards will be sent to the `OperatorRewardsCollector`, and will require the operator to call the `claim()` function to withdraw the rewards to the `operatorRewardsAddr`.

The recommendation has been implemented in commit 141dcaa.

ETHX2-02	DoS In `StakeUserETHToBeaconChain()` Due To Forced ETH Transfer Through `SelfDestruct`		
Asset	`PermissionlessPool.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

The function `stakeUserETHToBeaconChain()` incorporates an assert statement - `assert(address(this).balance == 0)`, at its termination on line [155], to verify that all ETH sent from the pool manager has been transferred to the beacon chain. However, the calculation of ETH is dependent on the `msg.value` received from the pool manager, which makes the assumption that the ETH balance in `PermissionlessPool` is zero prior to the function call.

While `PermissionlessPool` would typically revert in the `fallback()` function with `UnsupportedOperation()` if any ETH is transferred to the contract, there's an unhandled case where ETH is forcibly sent to the contract through a self-destruct operation. This undermines the assumption that `PermissionlessPool` will retain no funds prior to invoking `stakeUserETHToBeaconChain()`.

In a scenario where an attacker forcibly sends a small amount (1 wei) of ETH to `PermissionlessPool` via self destruct, the function `stakeUserETHToBeaconChain()` would always revert due to the assert statement at its end.

Recommendations

The testing team recommends removing this assert statement.

Resolution

The recommendation has been implemented in commit 141dcaa.

ETHX2-03	Duplicate `PoolIds` Can Cause Loss Of Funds For Users		
Asset	`VaultFactory.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Withdrawal vaults handle the distribution of rewards to users, node operators and protocol fee recipients. Due to the use of `CREATE2` opcode and contract architectural decisions, accidental or malicious administration can generate the same withdrawal address over two pool registries with identical `poolIds` .

`VaultFactory.deployWithdrawVault()` and `VaultFactory.computeWithdrawVaultAddress()` use the OpenZeppelin Clones Upgradeable contracts to manage the creation of new withdrawal vaults. In turn these use `CREATE2` with a salted value that relies on `_poolId, _operatorId, _validatorCount` . These values are entirely determined by the `PoolRegistry` contracts, which means that salt hash collisions can occur, resulting in the same withdrawal address being calculated. If two users share the same withdrawal address across two different pool registry contracts, it is possible for the earlier user to steal funds of the latter user.

Currently `PermissionlessNodeRegistry` and `PermissionedNodeRegistry` use the `VaultFactory.deployWithdrawVault()` which will revert if the same address is calculated twice (since CREATE2 reverts if an address has non-zero nonce, or has `extcodesize>0`). However, `PermissionlessPool.preDepositOnBeaconChain()` uses `VaultFactory.computeWithdrawVaultAddress` which will not be able to differentiate between whether vault is being calculated that matches a different pool with the same `poolId` .

Recommendations

Instead of `poolId` , use registry address (`msg.sender`) and pass this information to the pool contracts as well, this will protect against accidental `poolId` collisions from potentially causing loss of funds. It will not protect against malicious action, but the current access control (ie. `NODE_REGISTRY_CONTRACT`) here will restrict the attack surface to administrative accounts only. An alternative is moving `deployWithdrawVault` (ie `CREATE2`) logic to the pool registry contracts (though this would defeat the purpose of the VaultFactory).

Resolution

The `onboardNodeOperator()` function in `PermissionlessNodeRegistry` and `PermissionedNodeRegistry` now validates the `poolId` by matching it against the `poolId` of `PermissionlessPool` and `permissionedPool` respectively.

Since `addNewPool()` in `PoolUtils` validates that pools cannot be added with a duplicate `poolId` , the validation in `onboardNodeOperator()` guarantees that operators cannot be onboarded in the registry contract if a duplicate poolId exists.

Further, the protocol team has indicated that pools would not be onboarded without manual verification and further audits decreasing the chance of conflicting IDs.

The change has been implemented in commit 141dcaa.

ETHX2-04	`StaderConfig` Cannot Be Updated		
Asset	`ETHx.sol, OperatorRewardsCollector.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

`StaderConfig` serves as a central contract that maintains all protocol settings and access control, crucial for the protocol's regular operation. Every key contract within the codebase holds the current `StaderConfig` address as a reference and provides a method for altering the `StaderConfig` address, namely `updateStaderConfig()`.

However, `ETHx.sol` and `OperatorRewardsCollector.sol` contracts lack a function to update the `StaderConfig` in case of future changes to this address.

Recommendations

The testing team recommends adding the `updateStaderConfig()` function, similar to the setter function present in all other key contracts in the ETHx codebase.

Resolution

The recommendation has been implemented in commit 141dcaa.

ETHX2-05	`HasEnoughSDCollateral()` Check Is Performed Only Once During Onboarding		
Asset	`SDCollateral.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

The validation of SD token quantity currently only happens during the onboarding of new validators. This process specifically involves the call to `addValidatorKeys()`, which further invokes `checkInputKeysCountAndCollateral()`, ultimately calling `hasEnoughSDCollateral()`. However, this value is only contingent upon the price value during onboarding, allowing users to exploit price volatility and potentially maintain fewer SD tokens than required.

This issue could adversely affect the security offered by SD tokens to users against fraudulent or negligent validators. There is no assurance that the market value of SD collaterals will suffice over the entire staking duration. A depreciation in the market value of SD collateral could alter the staker's incentives, making malicious activities potentially more profitable.

Recommendations

To address this issue, the testing team recommends that stakers should cease accruing rewards until they increase the SD collateral amount to pass the `hasEnoughSDCollateral()` check.

Resolution

The development team closed the issue with the following comments.

When the min requirement of SD is not met, operators stop getting SD rewards. We believe it is a sufficient motivating factor for operators to continue maintaining that threshold. In the future, we could direct their ETH commissions to meet the min SD target. This is however not part of the scope for now.

ETHX2-06	Submit Functions Are Susceptible To Front Running When Trusted Nodes Are Removed		
Asset	`StaderOracle.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

When a trusted `StaderOracle` node is removed using the `removeTrustedNode()` they should not be able to vote on balances, withdrawals, or `beaconStateRoots`. However, since the current voting process allows for submissions if the reporting block is >= to the current `block.number` with no delay period, it is possible for `remoteTrustedNode()` to be front run with a call to (for instance) `submitBalances()`.

Since both transactions can happen in the same block, and submissions do not allow for delay, there is no way to protect against malicious node removal.

Recommendations

The testing team advises providing a delay before voting begins to ensure malicious entities can't vote prior to their own removal

Resolution

The development team closed the issue with the following comments.

We do not think this issue is applicable as only approved trusted nodes are allowed to join, trusted nodes also have to deposit some USDC as collateral which is done off-chain, and trusted nodes are changed in a time-spaced manner to ensure backwards compatibility

ETHX2-07	No Minimum Trusted Node Validation		
Asset	`StaderOracle.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

When nodes submit information to `StaderOracle`, there is no validation of a minimum number of `trustedNodeOnly` roles. Thus as new trusted nodes are being added to the system, an existing node can take advantage and vote for a malicious exchange rate. When early in the setup of `StaderOracle`, byzantine protection relying on the following can be passed with even just 1 malicious vote;

```
if (
    submissionCount == trustedNodesCount / 2 + 1 &&
    _exchangeRate.reportingBlockNumber > exchangeRate.reportingBlockNumber
)
```

Recommendations

Testing team recommends validating if a set minimum trusted nodes has been reached before allowing voting to begin.

Resolution

A minimum of 5 trusted nodes is enforced before any oracle data submission.

The recommendation has been implemented in commit 141dcaa.

ETHX2-08	Uninitialized `WithdrawDelay`		
Asset	`SDCollateral.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `withdrawDelay` value is used in the function `claimWithdraw()` to calculate the waiting period before operators are able to withdraw their `SDCollateral` tokens.

However `withdrawDelay` is uninitialized, so that prior to `setWithdrawDelay()` being called `withdrawDelay` will default to zero which results in operators being able to immediately claim their sd collateral tokens.

Recommendations

The testing team recommends initializing `withdrawDelay` to a reasonable default value in the initializer.

Resolution

The development team closed the issue with the following comments.

SD withdraw has been changed to immediate withdraw, rather than a two step process.

ETHX2-09	Operator Reward Address Modification Using Hot Wallet		
Asset	`PermissionlessNodeRegistry.sol`, `PermissionedNodeRegistry.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Onboarding a node operator involves passing the `operatorRewardAddress` as a parameter to the `onboardNodeOperator()` function. Post-onboarding, the node operator has the capacity to change this address at any time through the `updateOperatorDetails()` function.

In a typical setup, the operator account is a hot wallet controlled by node software, while the reward address is set to a more secure account such as a cold wallet or a contract wallet.

A security concern arises if the operator hot wallet is compromised. The attacker would then have the ability to alter the reward address and potentially steal the funds allocated to that address.

Recommendations

The testing team recommends that only the reward address itself should have the permission to alter the reward address.

This change can potentially introduce a tradeoff between security and availability - the risk of unintentionally setting the reward address to an erroneous account increases. Therefore, it would be beneficial to incorporate a safe ownership transfer pattern. This pattern would require the new reward address to acknowledge the change before it is enacted, thereby minimising the risk of errors.

Resolution

The development team closed the issue with the following comments.

We will keep the same as its up to the Node Operators to maintain their security. Our user research indicates the same preference.

ETHX2-10	Partial DoS Possible For User Withdrawals		
Asset	`UserWithdrawalManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Users request withdrawals by calling `UserWithdrawalManager.requestWithdraw()`, however this function allows a user to specify a different address as the `owner`. line [**101**] validates if the user has exceeded a `maxNonRedeemedUserRequestCount`, by default this value is set to `1000` in `StaderConfig`. On line [**107**] this value is incremented for the `owner`, which means if the owner attempted to request a withdrawal for their own address the check on line [**101**] will revert with `MaxLimitOnWIthdrawRequestCoutReached`.

There is a a check on line [**98**] that validates the amount withdrawn with `staderConfig.getMinWithdrawAmout`, however this is set to a default of `100`. This means the minimum amount to cause a revert in `UserWithdrawalManager.requestWithdraw()` is `100,000` wei worth of ETHx.

It is important to note that ETHx's value will increase as ETH staking rewards accumulate, so the cost of the attack may increase marginally over the course of the protocol's lifespan. Furthermore, it is important to note that the affected user cannot nominate a separate withdrawal address (effectively copying the technique used by the attacker) to withdraw to an address that has not exceeded the `maxNonRedeemedUserRequestCount`. The transaction will revert before incrementing the address of the withdrawal address `maxNonRedeemedUserRequestCount`.

While requests can be removed, this process requires them to first be finalised at a max batch limit of 50 withdrawals per transaction. Subsequently the user will need to claim the fund, which at certain gas prices may cost more than they are receiving from the claim. Claiming is capped at 1 claim per transaction.

Recommendations

The testing team recommends setting healthier limits on per user withdrawal requests (ie increasing the cost of a minimum withdrawal such that this attack is no longer feasible), or allow a user to quickly remove all pending withdrawal requests.

Resolution

The protocol team has implemented the following:

1. Increased `minWithdraw` amount to `10**14`
2. Fixed if condition to check for `owner` request count instead of `msg.sender`, therefore anyone can get around this by directing withdrawals to a new wallet.

The change has been implemented in commit 141dcaa.

ETHX2-11	Centralisation Risk Due To Extensive System Upgradability
Asset	`StaderConfig.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

`StaderConfig` is a centralised contract that is used for protocol governance and keeps track of all key values and contracts in the system. All contract addresses can be modified as well as values which are critical to the normal operation of the protocol.

The `ADMIN` role has the ability to change values in `staderConfig` as well as the `staderConfig` address values in all key contracts in the system through the `updateStaderConfig()` function.

The protocol team has indicated that the `ADMIN` role will be a large multi-sig where Stader would hold one signature while the rest will be prominent members of the Ethereum community.

However despite this, the extensive authority held by the `ADMIN` account, including potential fund misappropriation from users and operators, combined with the system's extensive upgradability, may make this safeguard insufficient. This could potentially expose the system to catastrophic failure, should there be malevolent actors or a compromise in the related security keys.

Recommendations

The testing team advises that essential updates, such as those related to critical contract address modifications and system upgrades, should only be implemented post-approval by the oracle DAO via an on-chain public voting mechanism. This would essentially form a large multi-signature contract with public voting.

Furthermore, we recommend implementing a mechanism to allow node operators to veto or rollback vault updates that pose a risk to the theft of funds.

Resolution

The development team closed the issue with the following comments.

Will migrate ADMIN power to a DAO vote mechanism in the future.

ETHX2-12	Phantom Overflow In `CalculateRewardShare()`
Asset	`PoolUtils.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

A phantom overflow is when final calculation result would fit into the result data type, but an intermediate operation overflows.

This can occur in `calculateRewardShare()` on line [**258**] if `_totalRewards * usersETH` is greater than `MAX_UINT256`. Although this is unlikely to occur in practice due to the size of `usersETH`.

Recommendations

A potential solution is to divide before multiply which will result in the user receiving a larger share due to division, and `protocolShare` and `operatorShare` being be rounded down.

Resolution

The development team closed the issue with the following comments.

These scenarios won't be possible in practice. The current implementation is capable of handling very big numbers (max uint128).

ETHX2-13	Potential Incorrect ETH Distribution In `ValidatorWithdrawalVault`
Asset	`ValidatorWithdrawalVault.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The issue relates to `validatorWithdrawVault.sol`, where two functions, `distributeRewards()` and `settleFunds()`, handle the distribution of ETH held in the contract.

The function `settleFunds()`, which disburses ETH after the validator is withdrawn, accounts for both user-deposited and operator-deposited ETH. It first reimburses the users and operators, and then allocates the remaining funds as rewards.

The `distributeRewards()` function, responsible for periodic reward distribution, operates differently. It distributes the full ETH balance held in the contract as rewards without factoring in the portions deposited by users and operators.

The problem arises when `distributeRewards()` is called post the validator's withdrawal, while the complete ETH balance is retained within the `ValidatorWithdrawalVault` contract. This scenario results in an incorrect ETH distribution as the `distributeRewards()` function misinterprets the total staked ETH balance as rewards, failing to return the shares of users and operators first.

To prevent a bad actor from calling `distributeRewards()` when the validator has withdrawn, the function will revert if `totalRewards` exceed `staderConfig.getRewardsThreshold()`.

Recommendations

It is critical to set `REWARD_THRESHOLD` to an amount that will be exceeded once the full amount of staked ETH is withdrawn to the vault contract. The protocol team has indicated that a value of 4-8 ETH will be used to ensure this condition is met even in the event of significant slashing. This value should be carefully monitored to ensure the accuracy of security assumptions.

Resolution

The development team closed the issue with the following comments.

We will be setting the rewardThreshold to 8 ETH at time of deployment.

ETHX2-14	`BidIncrement` Can Be Changed Using `UpdateBidIncrement()` Which Affects Ongoing Auctions
Asset	`Auction.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `bidIncrement` is used to calculate the minimum amount a bid must be above the current bid for the bidder to become the new highest bidder. This value can be updated anytime by the `MANAGER` using the `updateBidIncrement()` function.

However, `bidIncrement` used to calculate the minimum value depends on the current `bidIncrement` value rather than the original value that was set when the lot was created during `createLot()`.

Recommendations

The testing team recommends saving the current `bidIncrement` value as part of the lot settings during `createLot()` so that if the bid increment value is changed it will not retroactively affect existing auctions and cause unexpected reverts during bidding.

Resolution

The development team closed the issue with the following comments.

User can check the then current bidIncrement before placing any bid.

ETHX2-15	Centralisation Risk In `MaxApproveSD()`
Asset	`SDCollateral.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The function `maxApproveSD()` in `SDCollateral.sol` approves any arbitrary address the control of `uint256.max` of Stader tokens. This function can be called by the `MANAGER` role.

This is a centralisation risk if the `MANAGER` role is compromised, as the SD tokens held in this contract can be drained.

Recommendations

The testing team recommends limiting the addresses which can be approved using the `maxApproveSD()`, such as only to the Auction contract for the purposes of slashing.

Resolution

The recommendation has been implemented in commit 141dcaa.

ETHX2-16	`TrustedNode` Cannot Update Incorrectly Submitted Oracle Details
Asset	`StaderOracle.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

`TrustedNode` cannot update incorrectly submitted oracle details before consensus is completed, due to the following check on line [**578-580**]:

```
if (nodeSubmissionKeys[_nodeSubmissionKey]) {
            revert DuplicateSubmissionFromNode();
        }
```

An address granted `trustedNodeOnly` is unable to correct mistakes. Double voting is not a problem in this case as the original votes are overwritten, instead the risk is that incorrect votes cannot be corrected.

Recommendations

Testing team advises providing mechanisms for updating incorrect votes.

Resolution

The development team closed the issue with the following comments.

Oracle clients are expected to be proficient in their operations.

ETHX2-17	Miscellaneous General Comments
Asset	`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. **`createLot()` potentially has unexpected order of operations** When using shorthand addition, developers must be cautious of Solidity compilers optimisation and expected order of operations. In the case of `createLot()` the iteration to `nextLot++` is done inside of an event loop. In Solidity, the event will fire prior to the increment of the `nextLot` id. This behaviour may be counter intuitive. This could lead to confusion when a developer or user reads the code / event, expecting the next ID to be something other than what it is.
2. **Cannot change protocol fee independently of operator fee** If `staderConfig.onlyManagerRole()` decides to update the protocol or operator fee of any pool, currently they will have to change both values which limits usability of those functions.
3. **Ambiguous Role Name** The role identified as `OPERATOR`, which is enforced through `OnlyOperatorRole()`, is designed to denote whitelisted EOAs/contracts that are under Stader's control. Nevertheless, the current naming system may lead to ambiguity as it could be mistaken for Node Operators.

Recommendations

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

Resolution

The development team closed the issue with the following comments.

Improved readability.

ETHX2-18	OperatorRewardsCollector Missing Call To OZ `_disableInitializers`		
Asset	`OperatorRewardsCollector.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

The following passage can be found in the OpenZeppelin documentation for writing upgradable contracts.

```
Do not leave an implementation contract uninitialized.  An uninitialized implementation contract can be
taken over by an attacker, which may impact the proxy.  To prevent the implementation contract from being
used, you should invoke the _disableInitializers function in the constructor to automatically lock it when
it is deployed
```

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

`OperatorRewardsCollector` is missing a constructor which calls `_disableInitializers()`.

Recommendations

Add a constructor which invokes `_disableInitializers()` as described in the OZ documentation.

Resolution

The recommendation has been implemented in commit 141dcaa.

ETHX2-19	Lack Of External Contract Existence Check On DelegateCall		
Asset	`VaultProxy.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The low-level functions `call`, `delegatecall` and `staticcall` return `TRUE` as their first return value if the account called is non-existent or if `extcodesize == 0`. Therefore account existence must be checked prior to making a low level call.

The implementation contracts for `NodeELRewardVault` and `ValidatorWithdrawalVault` is set by the admin in `StaderConfig` using the functions `updateNodeELRewardImplementation()` and `updateValidatorWithdrawalVaultImplementation()` respectively.

However there is no validation that the implementation contract has been set in `StaderConfig`. Prior to the admin setting the implementation contracts, `getValidatorWithdrawalVaultImplementation()` and `getNodeELRewardVaultImplementation()` will return address(0).

As there is the possibility of the `StaderConfig` changing in the future, there is risk that `delegatecall` could be performed before the implementation contracts are set.

Recommendations

Validate that `vaultImplementation` does not equal `address(0)` prior to performing `delegatecall`.

Resolution

The development team closed the issue with the following comments.

The issues have been acknowledged

ETHX2-20	`VaultProxy` Does Not Follow Standard Proxy Pattern Usage
Asset	`VaultProxy.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `VaultProxy` contract is using a delegate forwarding call, a pattern often used when trying to avoid exceeding size limits. Though upgradeable proxy contracts also use `DELEGATECALL`, strictly using a forwarding pattern isn't recommend for systems designed with upgradeability in mind. Proxy patterns have added security features which minimise the risk of compromises.

There are several drawbacks with the current structure that impact upgradeability:

1. The logic contract and implementation contract ideally should be separate. This makes changes to storage across different versions clearer to follow for both developer and auditor. In the case of `VaultProxy` contract, it has some storage allocation specific to that contract, which is inaccessible to the current implementation contract. These variables could be defined in the implementation contract (ie poolId, isInitialised, owner etc) at a later stage, or overwritten by other storage requirements of the implementation contract. This presents a significant risk of storage collisions occuring during subsequent upgrades.
2. Proxy upgrades rely on a third contract (StaderConfig), this pattern is typically associated with Beacon Proxy contracts. Beacon proxy contracts implement EIP-1967 for storing proxy/logic contract details into a specific slot. This is to prevent the possibility of a storage collision with the implementation address (in this case the the staderConfig address which fetches the implementation contract address). In the case of VaultProxy it does not implement EIP-1967 which further increases the possibility of difficult to detect storage collisions due to potential quirks at compilation time.

Recommendations

The testing team advises the following changes:

1. Avoid splitting logic between the proxy and implementation contract, if administrative logic is required on the proxy level, consider using a transparent proxy with an OpenZeppelin ProxyAdmin contract to segregate proxy admin logic and general user implementation logic.
2. Implement suitable Proxy Storage patterns to reduce risk of collision (or use a proxy pattern that already has this functionality and is vetted to avoid collisions). Commonly used patterns include Transparent and UUPS.
3. Alternatively, if you would like to maintain upgradeability of contracts with a third party contract deciding the implementation of all contracts, consider using the BeaconProxy pattern.
4. Future upgrades should be validated that they do not overwrite existing variables.

Resolution

The development team closed the issue with the following comments.

The issues have been acknowledged

ETHX2-21	`Owner` In `VaultProxy` Does Not Adhere To The Single Source Of Truth Principle
Asset	`VaultProxy.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

When `VaultProxy` is initialised the owner is set to `staderConfig.getAdmin()`. `StaderConfig` is used as a centralised contract which serves as a single source of truth for the ETHx protocol.

However the owner in `Vaultproxy` can be updated independently using the `updateOwner()` function, this can accidentally result in different owner addresses being set if `StaderConfig` changes in the future.

Recommendations

Modify the `updateOwner()` function to get the current admin by calling `staderConfig.getAdmin()`.

Resolution

The recommendation has been implemented in commit 46c0988.

ETHX2-22	Miscellaneous General Comments — Round 2
Asset	`contracts/*`
Status	**Closed:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team in the second round of review (retesting), that do not have direct security implications:

1. **OpenZeppelin upgrade files are not committed to the repository:**
   The OpenZeppelin hardhat plugin recommends committing to source control the upgrade data for all but the dev network.

   > *It is advised that you commit to source control the files for all networks except the development ones (you may see them as .openzeppelin/unknown-*.json).* (`https://docs.openzeppelin.com/upgrades-plugins/1.x/network-files#configuration-files-in-version-control`)

   The current repository's `.gitignore` ignores the entire `.openzeppelin` folder and therefore makes it difficult for ETHx contributors to consistently interact with and upgrade contracts on persistent chains. The OpenZeppelin plugin relies on this to allow maintainers to reuse an implementation contract with multiple proxy deployments. Consider following the documentation's recommendations here.

2. **OpenZeppelin deployment scripts do not wait for deployment transaction to be mined:**
   Waiting on `upgrades.deployProxy()` returns the contract address but does not wait for the contract deployment to be mined.

   Consider the following excerpt from `scripts/deploy/ETHx.ts`

```
const ethXFactory = await ethers.getContractFactory('ETHX')
const ethX = await upgrades.deployProxy(ethXFactory, [owner.address, staderConfigAddr])
console.log('ethX Token deployed to: ', ethX.address)
```

   According to the upgrades plugin documentation, this should be followed by a `await ethX.deployed();` statement to ensure the contract actually exists on-chain.

Recommendations

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

Resolution

The development team closed the issue with the following comments.

The issues have been acknowledged

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 `Foundry` framework was used to perform these tests and the output is given below.

```
Running 2 tests for test/PoolSelector.t.sol:PoolSelectorTests
[PASS] test_computePoolAllocationForDeposit() (gas: 1936172)
[PASS] test_initializePoolSelector() (gas: 22794)
Test result: ok. 2 passed; 0 failed; finished in 13.01ms

Running 4 tests for test/Penalty.t.sol:PenaltyTests
[PASS] test_constructor_Vuln() (gas: 22351)
[PASS] test_initialize() (gas: 35962)
[PASS] test_setAdditionalPenaltyAmount() (gas: 102231)
[PASS] test_setPenaltyOracleAddress() (gas: 94295)
Test result: ok. 4 passed; 0 failed; finished in 10.79ms

Running 6 tests for test/StaderStakePoolsManager.t.sol:StaderStakePoolsManagerTests
[PASS] test_completeDepositCyclePermissionless() (gas: 1962757)
[PASS] test_deposit() (gas: 287983)
[PASS] test_initialize() (gas: 17488)
[PASS] test_newPermissionlessValidator_Vuln() (gas: 1713316)
[PASS] test_validatorBatchDeposit() (gas: 1846064)
[PASS] test_validatorBatchDeposit_Vuln() (gas: 1623896)
Test result: ok. 6 passed; 0 failed; finished in 19.54ms

Running 3 tests for test/PermissionlessPool.t.sol:PermissionlessPoolTests
[PASS] test_initialize() (gas: 17511)
[PASS] test_setProtocolFeePercent() (gas: 136499)
[FAIL. Reason: Assertion violated] test_stakeUserETHToBeaconChain_vuln() (gas: 2125190)
Test result: FAILED. 2 passed; 1 failed; finished in 15.93ms

Running 4 tests for test/ETHx.t.sol:ETHxTests
[FAIL. Reason: Call did not revert as expected] test_cannotChangeStaderConfig_vuln() (gas: 3518227)
[PASS] test_initialize() (gas: 31035)
[PASS] test_mintAndBurn() (gas: 180301)
[PASS] test_pause() (gas: 104713)
Test result: FAILED. 3 passed; 1 failed; finished in 16.47ms

Running 11 tests for test/PermissionedNodeRegistry.t.sol:PermissionedNodeRegistryTests
[PASS] test_addValidatorKeys() (gas: 1304136)
[PASS] test_addValidatorKeys_Vuln() (gas: 370855)
[PASS] test_initialize() (gas: 17534)
[PASS] test_onboardNodeOperator() (gas: 475745)
[PASS] test_reportFrontRunValidator() (gas: 1579179)
[PASS] test_reportInvalidSignatureValidator() (gas: 1477421)
[PASS] test_updateBatchKeyDepositLimit() (gas: 64844)
[PASS] test_updateMaxKeyPerOperator() (gas: 65112)
[PASS] test_updateOperatorDetailsP() (gas: 459840)
[PASS] test_updateQueuedValidatorIndex() (gas: 87080)
[PASS] test_whitelistPermissionedNOs() (gas: 116079)
Test result: ok. 11 passed; 0 failed; finished in 16.92ms

Running 4 tests for test/SDCollateral.t.sol:SDCollateralTests
[PASS] test_conversionZeroAmounts() (gas: 219697)
[PASS] test_maxApproveRug_vuln() (gas: 215372)
[PASS] test_updatePoolThreshold() (gas: 111559)
[FAIL. Reason: withdrawDelay is uninitialized] test_withdrawDelayUninitialized_vuln() (gas: 14992)
Test result: FAILED. 3 passed; 1 failed; finished in 18.38ms

Running 5 tests for test/Auction.t.sol:AuctionTests
[PASS] test_addBid() (gas: 373118)
[PASS] test_createLot(uint256) (runs: 256, μ: 281083, ~: 282797)
[PASS] test_createZeroLot() (gas: 110562)
[FAIL. Reason: InSufficientBid()] test_increaseIncrement_vuln() (gas: 451293)
[PASS] test_initialize() (gas: 31175)
```

```
Test result: FAILED. 4 passed; 1 failed; finished in 94.48ms

Running 2 tests for test/PermissionedPool.t.sol:PermissionedPoolTests
[PASS] test_initialize() (gas: 17533)
[PASS] test_setComissionFees() (gas: 133819)
Test result: ok. 2 passed; 0 failed; finished in 10.32ms

Running 8 tests for test/PermissionlessNodeRegistry.t.sol:PermissionlessNodeRegistryTests
[PASS] test_OnboardNodeOperator() (gas: 567103)
[PASS] test_addValidatorKeys() (gas: 1436510)
[PASS] test_addValidatorKeys_Vuln() (gas: 543719)
[PASS] test_changeSocializingPoolState() (gas: 494106)
[PASS] test_initialize() (gas: 17643)
[PASS] test_markValidatorReadyToDeposit() (gas: 1487923)
[PASS] test_updateNextQueuedValidatorIndex() (gas: 86645)
[PASS] test_updateOperatorDetails() (gas: 612860)
Test result: ok. 8 passed; 0 failed; finished in 13.03ms

Running 9 tests for test/StaderOracle.t.sol:StaderOracleTests
[PASS] test_addTrustedNode() (gas: 144565)
[FAIL. Reason: DuplicateSubmissionFromNode()] test_cannotUpdatePrice_vuln() (gas: 215743)
[FAIL. Reason: Price should not update: only 2 out of 4 have submitted] test_frontunRemoveTrustedNode_vuln() (gas: 461816)
[PASS] test_getLatestReportableBlock_Vuln() (gas: 22178)
[PASS] test_initialize() (gas: 25526)
[PASS] test_noMinTrustedNode_vuln() (gas: 295797)
[PASS] test_removeTrustedNode() (gas: 133364)
[PASS] test_setUpdateFrequency() (gas: 218198)
[PASS] test_submitBalances() (gas: 326187)
Test result: FAILED. 7 passed; 2 failed; finished in 15.99ms

Running 6 tests for test/VaultFactory.t.sol:VaultFactoryTests
[PASS] test_addNewPool() (gas: 106335)
[FAIL. Reason: users received less than their stake of 28 eth] test_distributeRewardsFrontrun_vuln() (gas: 2701077)
[FAIL. Reason: users received less than their stake of 28 eth] test_distributeRewardsHighRewardsThreshold_vuln() (gas: 2732727)
[PASS] test_initialize() (gas: 17473)
[FAIL. Reason: ETHTransferFailed(0x31256Cb3D8Cb35671F13b5B1680B2CF4FE55fC4F, 6290625000000000000)]
      ↪  test_maliciousNodeOperatorDoS_vuln() (gas: 2502805)
[PASS] test_settleFunds() (gas: 2507354)
Test result: FAILED. 3 passed; 3 failed; finished in 15.83ms

Running 2 tests for test/PoolUtils.t.sol:PoolUtilsTests
[FAIL. Reason: Arithmetic over/underflow Counterexample:
      ↪  calldata=0x58dde8e10000000000000000a8a7ec10a6fcfc9a8c915debde04245a833ff172bc175c8b,
      ↪  args=[4135431758475578407984678036024568137616785166630020144267]] test_calculateRewardShare_vuln(uint256) (runs: 103, μ:
      ↪  77404, ~: 77404)
[PASS] test_initialize() (gas: 31177)
Test result: FAILED. 1 passed; 1 failed; finished in 34.57ms
```

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				
	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

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

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: `https://blog.sigmaprime.io/solidity-security.html`. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: `http://www.dasp.co/`. [Accessed 2018].