Stader Labs
Findings & Analysis Report

2023-07-19

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Stader Labs smart contract system written in Solidity. The audit took place between June 2 - June 9 2023.

Wardens

77 Wardens contributed reports to the Stader Labs:

1. 0x70C9
2. 0xSmartContract
3. 0xWaitress
4. 0xackermann
5. 0xhacksmithh
6. Aymen0909
7. Bauchibred
8. Breeje

9. ChrisTina
10. CoOnan
11. DadeKuma
12. DavidGiladi
13. Deps
14. Hama
15. JCN
16. JGcarv
17. Josiah
18. K42
19. LaScaloneta (nicobevi, juancito and 0x4non)
20. Madalad
21. MohammedRizwan
22. NoamYakov
23. Rageur
24. Raihan
25. RaymondFam
26. Rolezn
27. SAAJ
28. SAQ
29. SM3_SS
30. Sathish9098
31. SovaSlava
32. T1MOH
33. Tomio
34. bigtone
35. bin2chen
36. broccolirob
37. btk

38. c3phas
39. catellatech
40. djxploit
41. dontonka
42. dwward3n
43. erictee
44. ernestognw
45. etherhood
46. fatherOfBlocks
47. hake
48. hals
49. hunter_w3b
50. jaraxxus
51. josephdara
52. koxuan
53. ksk2345
54. kutugu
55. martin
56. matrix_0wl
57. mgf15
58. naman1778
59. niser93
60. peanuts
61. petrichor
62. piyushshukla
63. ravikiranweb3
64. rvierdiiev
65. saneryee
66. sces60107

67. sebghatullah
68. shamsulhaq123
69. silviaxyz
70. solsaver
71. tallo
72. trustOne
73. tsvetanovv
74. turvy_fuzz
75. whimints

This audit was judged by Picodes.

Final report assembled by thebrittfactor.

Summary

The C4 analysis yielded an aggregated total of 15 unique vulnerabilities. Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity and 14 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 24 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 28 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Stader Labs repository, and is composed of 23 smart contracts written in the Solidity programming language and includes 4343 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (1)

[H-01] `VaultProxy` implementation can be initialized by anyone and self-destructed

Submitted by broccolirob*, also found by* dwward3n, hals, bin2chen*, and* 0x70C9

https://github.com/code-423n4/2023-06-stader/blob/7566b5a35f32ebd55d3578b8bd05c038feb7d9cc/contracts/VaultProxy.sol#L20-L36
https://github.com/code-423n4/2023-06-stader/blob/7566b5a35f32ebd55d3578b8bd05c038feb7d9cc/contracts/VaultProxy.sol#L41-L50

When the `VaultFactory` contract is deployed and initialized, the `initialise` method on the newly created `VaultProxy` implementation contract is never called. As such, anyone can call that method and pass in whatever values they want as arguments. One important argument is the `_staderConfig` address, which controls where the `fallback` function will direct `delegatecall` operations. If an attacker passes in a contract that calls `selfdestruct`, it will be run in the context of the `VaultProxy` implementation contract and will erase all code from that address. Since the clones from the `VaultProxy` contract merely delegate calls to the implementation address, all subsequent calls for all created vaults from that implementation will be treated like an EOA and return `true`, even though calls to functions on that proxy were never executed.

Proof of Concept

- First, an attacker deploys a contract called `AttackContract` that calls `selfdestruct` in its `fallback` function.

```
contract AttackContract {
    function getValidatorWithdrawalVaultImplementation() public
        return address(this);
    }
    function getNodeELRewardVaultImplementation() public view re
        return address(this);
    }
    fallback(bytes calldata _input) external payable returns(byt
        selfdestruct(address(0));
    }
}
```

- The attacker calls the `initialise` method on the `VaultProxy` implementation contract. That address is stored in the `vaultProxyImplementation` variable on the `VaultFactory` contract. The attacker passes in the address of `AttackContract` as the `_staderConfig` argument for the `initialise` function.
- The attacker then calls a non-existent function on the `VaultProxy` implementation contract, which triggers it's `fallback` function. The `fallback` function calls `staderConfig.getNodeELRewardVaultImplementation()`, and since `staderConfig` is set the `AttackContract` address, it returns the address of the `AttackContract`. `delegatecall` runs the fallback function of `AttackContract` in its own execution environment. `selfdestruct` is called in the execution environment of the `VaultProxy` implementation, which erases the code at that address.
- All cloned copies of the `VaultProxy` implementation contract are now forwarding calls to an implementation address that has no code stored at it. These calls will be treated like calls to an EOA and return `true` for `success`.

Recommended Mitigation Steps

Prevent the `initialise` function from being called on the `VaultProxy` implementation contract by inheriting from OpenZeppelin's `Initializable` contract, like the system is doing in other contracts. Call the `_disableInitializers` function in the constructor and protect `initialise` with the `initializer` modifier. Alternatively, the `initialise` function can be called from the `initialize` function of the `VaultFactory` contract when the `VaultProxy` contract is instantiated.

Assessed type

Access Control

[Picodes (judge) commented]:

> Keeping High severity. This seems exploitable to lock funds with no cost, as the fallback function is payable.

[sanjay-staderlabs (Stader) confirmed]

[sanjay-staderlabs (Stader) commented]:

> This is fixed in the code.

Medium Risk Findings (14)

[M-01] Risk of losing admin access if `updateAdmin` set with same current admin address

Submitted by [ksk2345], *also found by* [ChrisTina] *and* [NoamYakov]

Current admin will lose `DEFAULT_ADMIN_ROLE` role if `updateAdmin` issued with same address.

There is a possibility of loss of protocol admin access to the critical `StaderConfig.sol` contract, if `updateAdmin()` is set with same current admin address by mistake.

Proof of Concept

Contract : StaderConfig.sol.

Function : function `updateAdmin(address _admin)`.

Using Brownie python automation framework commands in below examples:

- Step #1 After initialization, admin-A is the admin which has the `DEFAULT_ADMIN_ROLE`.
- Step #2 update new Admin:
  `StaderConfig.updateAdmin(admin-B, {'from':admin-A})`.
  The value of `StaderConfig.getAdmin()` is admin-B.
- Step #3 admin-B updates admin to itself again:
  `StaderConfig.updateAdmin(admin-B, {'from':admin-B})`.
  The value of `StaderConfig.getAdmin()` is admin-B, but the `DEFAULT_ADMIN_ROLE` is revoked due to `_revokeRole(DEFAULT_ADMIN_ROLE, oldAdmin)`.
  Now the protocol admin control is lost for StaderConfig contract.

Recommended Mitigation Steps

Reference: https://github.com/code-423n4/2023-06-stader/blob/7566b5a35f32ebd55d3578b8bd05c038feb7d9cc/contracts/StaderConfig.sol#L177

In the `updateAdmin()` function, add a check for `oldAdmin != _admin`, like below:

```
        address oldAdmin = accountsMap[ADMIN];
+       require(oldAdmin != _admin, "Already set to admin");
```

Assessed type

Access Control

[Picodes (judge) decreased severity to Medium]()

manoj9april (Stader) confirmed and commented:

> Sure, we will fix this.

rvierdiyev (warden) commented:

> Isn't this same as just transferring roles to the `address 0` or any other address? Why would the protocol need to change roles to same address? Isn't this an informative issue?

CoOnan (warden) commented:

> I believe this falls under the "Admin Privilege" category, as such an issue should be marked as QA based on C4 docs and how similar issues got judged.

Picodes (judge) commented:

> @rvierdiyev and @CoOnan - I respectfully disagree:

- This is not an occurrence of "Admin Privilege", which are issues where a privileged role uses their position to grief the protocol. Here, there is clearly a slight bug in the code.
- We could argue that transferring the role to the same address is very unlikely but it is not an error in itself. The function clearly does not behave as intended in this case.
- If you combine this with the fact that in `initialize` there is no call to `setAccount(ADMIN, _admin);` see here. It becomes actually likely that the admin calls this function for themselves.

Picodes (judge) commented:

> @CoOnan - for information, "Admin Privilege" aren't always QA, it depends on the context and is up to the judge. See here and here for the ongoing discussion about this.

CoOnan (warden) commented:

> Here there is clearly a slight bug in the code.

> @Picodes - This bug occurs from `Admin` as they pass an address twice. I have to stand with @rvierdiyev. This is likely due to missing `Zero address Check` on `onlyOwner` functions.

> However, it's up to you as the Judge and I respect your final conclusion.

sanjay-staderlabs (Stader) commented:

> This is fixed in the code.

[M-02] `pause/unpause` functionalities not implemented in many pausable contracts

Submitted by Aymen0909*, also found by* LaScaloneta, tallo, martin, bin2chen, jaraxxus, josephdara, djxploit, SovaSlava, SovaSlava, SovaSlava, ChrisTina, T1MOH, NoamYakov, NoamYakov*, and* Deps

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/SocializingPool.sol#L21
https://github.com/code-423n4/2023-06-stader/blob/main/contracts/Auction.sol#L14
https://github.com/code-423n4/2023-06-stader/blob/main/contracts/StaderOracle.sol#L17
https://github.com/code-423n4/2023-06-stader/blob/main/contracts/OperatorRewardsCollector.sol#L16

The following contracts: `SocializingPool`, `StaderOracle`, `OperatorRewardsCollector` and `Auction` are supposed to be pausable (as they all inherit from `PausableUpgradeable`), but they don't implement the external `pause/unpause` functionalities which means it will never be possible to pause them.

Proof of Concept

All the following contracts `SocializingPool`, `StaderOracle`, `OperatorRewardsCollector` and `Auction` inherit from the openzeppelin `PausableUpgradeable` extension which means that they contain internal functions `_pause` and `_unpause`.

Because those functions are internal, the contract must implement two other public/external `pause` and `unpause` functions to allow the manager to pause and unpause the contracts when necessary. None of the aforementioned contracts implement those functions, which means even if those contracts are supposed to be pausable (and have the `pause/unpause` functionalities), none of them can be paused.

Recommended Mitigation Steps

Add public/external `pause` and `unpause` functions in the aforementioned contracts to allow them to be pausable, this can be done as in the `UserWithdrawalManager` contract. For example:

```
    /**
     * @dev Triggers stopped state.
     * Contract must not be paused
     */
    function pause() external {
        UtilLib.onlyManagerRole(msg.sender, staderConfig);
        _pause();
    }

    /**
     * @dev Returns to normal state.
     * Contract must be paused
     */
    function unpause() external onlyRole(DEFAULT_ADMIN_ROLE) {
        _unpause();
    }
```

Picodes (judge) decreased severity to Medium

manoj9april (Stader) confirmed and commented:

> Thanks! We will fix this.

sanjay-staderlabs (Stader) commented:

> This is fixed.

[M-03] Stader OPERATOR is a single point of failure

Submitted by JGcarv

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/PermissionlessNodeRegistry.sol#L183
https://github.com/code-423n4/2023-06-stader/blob/main/contracts/PermissionedNodeRegistry.sol#L254

The OPERATOR role holds a lot of power within the system, which can compromise both the system integrity and it's permission-less nature.

Proof of Concept

The OPERATOR key is responsible for confirming the marking of each validator submitted key as either valid or invalid, without any assurance to validators.

1. Arbitrary negation of participation makes permissionless pool permissioned.

The documentation states:

> Any validator in permissionless pool can run a node with 4 ETH + 0.4 ETH worth of SD token.

Which is not strictly true, since any participant in the system must be vetted by the OPERATOR, which can arbitrarily mark as invalid or frontrun the key without the need to provide justification or having an appeal system. Alternatively, the OPERATOR can simply ignore the added key and never mark it as `ready to deposit`.

Therefore, the pool can't be considered permissionless, since participants must rely on the benevolence of the OPERATOR to participate.

2. Authorization of invalid keys

There is no way for the smart contract system to check or confirm that a given public key is really legit. This could generate income to ETHx holders, so the system relies solely on the OPERATOR to make that distinction, rendering the system vulnerable in case of a comprised wallet.

Recommended Mitigation Steps

There is no simple fix for the issue, but at minimum, the protocol shouldn't be advertised as permissionless.

Assessed type

Rug-Pull

Picodes (judge) decreased severity to Medium

manoj9april (Stader) confirmed and commented:

> Thank you for pointing it out. We will move this logic to oracle.

Picodes (judge) commented:

> Keeping Medium severity considering this could be an instance of "function of the protocol or its availability could be impacted".

sanjay-staderlabs (Stader) commented:

> This is fixed.

[M-04] `updatePoolAddress` functions always revert when updating existing `poolId`

Submitted by Aymen0909, also found by trustOne, bin2chen, and T1MOH

The purpose of the `updatePoolAddress` function is to update the pool address associated with an existing `poolId`. However, due to its internal invocation of the `verifyNewPool` function, the `updatePoolAddress` function always reverts. This occurs because the `verifyNewPool` function itself reverts when the specified `poolId` already exists. Consequently, it is not possible to update the pool address for an existing `poolId`.

Proof of Concept

The issue occurs in the `updatePoolAddress` function below :

File: PoolUtils.sol Line 55-65

```
    function updatePoolAddress(
        uint8 _poolId,
        address _newPoolAddress
    ) external override onlyExistingPoolId(_poolId) onlyRole(DEFAULT
        UtilLib.checkNonZeroAddress(_newPoolAddress);
        // @audit always revert on exsiting poolId
        verifyNewPool(_poolId, _newPoolAddress);
        poolAddressById[_poolId] = _newPoolAddress;
        emit PoolAddressUpdated(_poolId, _newPoolAddress);
    }
```

As it can be seen from the code above, the `updatePoolAddress` function contains the `onlyExistingPoolId` modifier which means it can only be called for updating the pool address of an already exiting `poolId`.

Before updating the pool address, the `updatePoolAddress` function calls the `verifyNewPool` function below:

```
    function verifyNewPool(uint8 _poolId, address _poolAddress) inte
        if (
            INodeRegistry(IStaderPoolBase(_poolAddress).getNodeRegis
            isExistingPoolId(_poolId)
        ) {
            revert ExistingOrMismatchingPoolId();
        }
    }
```

It's clear that the function reverts when the `poolId` already exists meaning `isExistingPoolId(_poolId) == true`.

To summarize, the `updatePoolAddress` function reverts when the `poolId` does not exist and the `verifyNewPool` function reverts when the `poolId` exists. These two functions work on opposite conditions, which means when the `verifyNewPool` function is called inside the `updatePoolAddress` function it will automatically revert and the pool address of already existing `poolId` can never be updated.

Recommended Mitigation Steps

Remove the `verifyNewPool` call inside the `updatePoolAddress` function and replace it with the following:

```
    function updatePoolAddress(
        uint8 _poolId,
        address _newPoolAddress
    ) external override onlyExistingPoolId(_poolId) onlyRole(DEFAULT
        UtilLib.checkNonZeroAddress(_newPoolAddress);
        // @audit revert only when mismatch in poolId
        if (INodeRegistry(IStaderPoolBase(_poolAddress).getNodeRegis
            revert MismatchingPoolId();
        }
        poolAddressById[_poolId] = _newPoolAddress;
        emit PoolAddressUpdated(_poolId, _newPoolAddress);
    }
```

Assessed type

Error

[Picodes (judge) decreased severity to Medium]()

[manoj9april (Stader) confirmed and commented]():

> Thanks! We will fix this.

[sanjay-staderlabs (Stader) commented]():

> This is fixed.

[M-05] `StaderOracle` - Strict equal can cause no consensus if trusted nodes are removed before consensus

Submitted by dwward3n

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/StaderOracle.sol#L148
https://github.com/code-423n4/2023-06-stader/blob/main/contracts/StaderOracle.sol#L290

```
            if (
                submissionCount == trustedNodesCount / 2 + 1 &&
                _exchangeRate.reportingBlockNumber > exchangeRate.re
            ) {
                updateWithInLimitER(
                    _exchangeRate.totalETHBalance,
                    _exchangeRate.totalETHXSupply,
                    _exchangeRate.reportingBlockNumber
                );
            }
```

In `submitExchangeRateData`, consensus is reached if `submissionCount` is strictly equal to desired number. However, `trustNodesCount` can be decreased and this condition can be never met.

```
            if ((submissionCount == (2 * trustedNodesCount) / 3 + 1)
                lastReportedSDPriceData = _sdPriceData;
                lastReportedSDPriceData.sdPriceInETH = getMedianValu
                delete sdPrices;
```

In `submitSDPrice`, if this case happens, `sdPrices` doesn't get deleted and it will affect the next submission batch's price.

Proof of Concept

In the above snippet, let's assume `trustedNodesCount` = 10, `submissionCount` = 5.
The condition doesn't meet for now (5 != 10/2+1). Then `trustedNodesCount` decreases to 9.
Next time when a node submits, `trustedNodesCount` = 9, `submissionCount` = 6.
Then the condition cannot be met since (6 != 9/2+1).

Recommended Mitigation Steps

Replace strict equal with equal or greater than. Or replace it with greater than and decrease the right side.

Not sure about adding cooldown for add/remove trusted nodes.

manoj9april (Stader) confirmed and commented:

> Thanks, we will fix this.

Picodes (judge) commented:

> Keeping medium severity because of the sponsor's label, but the justification of why it qualifies for Med and what the impacts of the bug are is insufficient.

> For most functions, it seems the `submissionCount` depends on the `reportingBlockNumber` so the impact would only be one period where the oracle couldn't be updated, which doesn't qualify without additional justification. For example, for `submitSocializingRewardsMerkleRoot` and `submitMissedAttestationPenalties`, the impact may be more important.

sanjay-staderlabs (Stader) commented:

> This is fixed.

[M-06] Protocol will not benefit from slashing mechanism when remaining penalty bigger than `minThreshold`

Submitted by CoOnan, also found by CoOnan and 0xWaitress

During the withdraw process, the function `settleFunds()` get called. This function first calculates the `operatorShare` and the `penaltyAmount`. If the `operatorShare < penaltyAmount`, the function calls `slashValidatorSD` in order to slash the operator and start new auction to cover the loss.

The issue here, is `slashValidatorSD` determines the amount to be reduced based on the smallest value between operators current SD balance and the `poolThreshold.minThreshold`. In this case, where the `operatorShare` (1ETH) is too small than the `penaltyAmount` (10ETH) the system should reduce an equivalent amount to cover the remaining ETH (`9ETH_`). However, the function choose the smallest value which could end up being 4e17. In such cases, the protocol will not favor because starting a new auction with SD amount = 4e17 will not end up with a 9ETH in exchange.

Proof of Concept

1. Implementation of `settleFunds`:

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/ValidatorWithdrawalVault.sol#L54

```
    function settleFunds() external override {
            uint8 poolId = VaultProxy(payable(address(this))).poolIc
            uint256 validatorId = VaultProxy(payable(address(this)))
            IStaderConfig staderConfig = VaultProxy(payable(address(
            address nodeRegistry = IPoolUtils(staderConfig.getPoolUt
            if (msg.sender != nodeRegistry) {
                revert CallerNotNodeRegistryContract();
            }
            (uint256 userSharePrelim, uint256 operatorShare, uint256

            uint256 penaltyAmount = getUpdatedPenaltyAmount(poolId,

            if (operatorShare < penaltyAmount) {
                ISDCollateral(staderConfig.getSDCollateral()).slashV
                penaltyAmount = operatorShare;
            }

            uint256 userShare = userSharePrelim + penaltyAmount;
            operatorShare = operatorShare - penaltyAmount;

            // Final settlement
            vaultSettleStatus = true;
            IPenalty(staderConfig.getPenaltyContract()).markValidato
            IStaderStakePoolManager(staderConfig.getStakePoolManager
            UtilLib.sendValue(payable(staderConfig.getStaderTreasury
            IOperatorRewardsCollector(staderConfig.getOperatorRewarc
            getOperatorAddress(poolId, validatorId, staderConfig
        );
        emit SettledFunds(userShare, operatorShare, protocolShar
    }
```

2. Let's suppose `operatorShare` is 1 ETH, `penaltyAmount` is 5ETH. In this case, the function will enter the `If` condition on L67.

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/ValidatorWithdrawalVault.sol#L67

```
 ISDCollateral(staderConfig.getSDCollateral()).slashValidatorSD(
```

3. Implementation of `slashValidatorSD` and `slashSD`.

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/SDCollateral.sol#L78

```
function slashValidatorSD(uint256 _validatorId, uint8 _poolId) e
        address operator = UtilLib.getOperatorForValidSender(_po
        isPoolThresholdValid(_poolId);
        PoolThresholdInfo storage poolThreshold = poolThresholdb
        uint256 sdToSlash = convertETHToSD(poolThreshold.minThre
        slashSD(operator, sdToSlash);
    }

    /// @notice used to slash operator SD, incase of operator de
    /// @dev do provide SD approval to auction contract using `m
    /// @param _operator which operator SD collateral to slash
    /// @param _sdToSlash amount of SD to slash
    function slashSD(address _operator, uint256 _sdToSlash) inte
        uint256 sdBalance = operatorSDBalance[_operator];
        uint256 sdSlashed = Math.min(_sdToSlash, sdBalance);
        if (sdSlashed == 0) {
            return;
        }
        operatorSDBalance[_operator] -= sdSlashed;
        IAuction(staderConfig.getAuctionContract()).createLot(sd
        emit SDSlashed(_operator, staderConfig.getAuctionContrac
    }
```

4. As you can see, on line 82, the function gets the `minThreshold` and passes it to `slashSD`.
5. On line 92, it selects the smallest value between the current balance of the operator and the `minThreshold`:

```
uint256 sdSlashed = Math.min(_sdToSlash, sdBalance);
```

6. If the `minThreshold` < remaining penalty, which is 4 ETH in this case, the function simply ignores that and reduces the operator amount with "`minThreshold`" instead. In this case, it's < current SD balance.
7. The function then starts new auction with the smallest value.

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/SDCollateral.sol#L97

```
operatorSDBalance[_operator] -= sdSlashed;
        IAuction(staderConfig.getAuctionContract()).createLot(sc
```

Despite how much the user should pay, the auction will start with the min value and the `penaltyAmount` will not be paid in full.

Recommended Mitigation Steps

The function shouldn't use `minThreshold`. It should catch the remaining penalty (difference between `operatorShare` and `penaltyAmount`) and use it to calculate the required SD amount to be slashed.

Assessed type

Context

manoj9april (Stader) acknowledged and commented:

> This update is slated for a future release.

[M-07] MEV bots can win all the auctions when `Auction` is paused

Submitted by DadeKuma, also found by SovaSlava

MEV bots may bid on all the ongoing auctions before `Auction` is paused by frontrunning it with the `addBid` function.

This can lead to bots winning all the `auctions` for a fraction of their price. Especially `auctions` that are almost ended, as no one is able to bid until the contract is unpaused.

Proof of Concept

No one will be able to bid when the contract is paused, but bots can simply front-run the `pause` by looking at the mempool:

```
    function addBid(uint256 lotId) external payable override
        // reject payments of 0 ETH
        if (msg.value == 0) revert InSufficientETH();

        LotItem storage lotItem = lots[lotId];
        if (block.number > lotItem.endBlock) revert AuctionE

        uint256 totalUserBid = lotItem.bids[msg.sender] + ms

        if (totalUserBid < lotItem.highestBidAmount + bidInc

        lotItem.highestBidder = msg.sender;
        lotItem.highestBidAmount = totalUserBid;
        lotItem.bids[msg.sender] = totalUserBid;

        emit BidPlaced(lotId, msg.sender, totalUserBid);
    }
```

A word about severity; the contract extends `PausableUpgradeable`:

```
    contract Auction is IAuction, Initializable, AccessContr
```

Also, it has several functions that have the `whenNotPaused` modifier:

```
        function createLot(uint256 _sdAmount) external override

        function addBid(uint256 lotId) external payable override
```

However, it doesn't have any `pause` / `unpause` functions that are `external` / `public`, as they have `internal` visibility by default in `PausableUpgradeable`; as such, the contract can't be paused with the current implementation.

However, this may change in the future as the contract is upgradeable, so I still consider this issue valid.

Recommended Mitigation Steps

Consider removing the `whenNotPaused` modifier from `addBid`, so ongoing auctions may be ended even when the contract is paused.

Assessed type

MEV

manoj9april (Stader) disputed and commented:

> Bots can win the auction by frontrunning if that ensures a better price than the current bid (which it does) Also this is an duplicate issue of #70

Picodes (judge) commented:

> I don't think this is a duplicate of #70. This issue is specifically about the fact that pausing an auction decreases its time and that MEV bots/the admin could benefit from this. It is a valid issue, in the sense that nothing is done to prevent `auctions` from being closed by a call to `pause`, which would break the system.

sanjay-staderlabs (Stader) commented:

> Hi @Picodes - thanks for pointing it out. It turns out for the mitigation of issue [383](), we have removed `PausableUpgradeable` from the `Auction` contract; hence, removing the `whenNotPaused` modifier from `addBid` function. This issue is no longer relevant. Please see if we can close this or let me know what you think.

[M-08] Corruption of oracle data

Submitted by [etherhood]()

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/StaderOracle.sol#L270
https://github.com/code-423n4/2023-06-stader/blob/main/contracts/StaderOracle.sol#L285

Proof of Concept

Block for `lastReportedSDPriceData` = 7200

Let's make the current block = 21601
Now `StaderOracle` will have data for 14400 and 21600, both blocks are being pushed by nodes and in the prices array.
It will be all mixed up. Also, As soon as the 14400 block is finalised, the data for block 21600 is all lost as well.

Recommended Mitigation Steps

Add `if (_sdPriceData.reportingBlockNumber == getSDPriceReportableBlock())` to ensure it is always the latest reportable block data.
Add `mapping(uint256 => uint256[]) blockPrices` to store the prices array separately for each block being reported, to avoid mixing and corruption of data. Or have `uint256 currentEpochBlock`, so when a new block of data is pushed, previous data is deleted before pushing the new data.

```
    if(_sdPriceData.reportingBlockNumber!=currentEpochBlock){
        delete prices;
    }
```

Assessed type

Oracle

manoj9april (Stader) confirmed and commented:

> Sure we will fix this.

sanjay-staderlabs (Stader) commented:

> This is fixed.

[M-09] `depositETHOverTargetWeight()` malicious modifications `poolIdArrayIndexForExcessDeposit`

Submitted by bin2chen, *also found by* sces60107

Malicious modification in favor of your own funds' allocation rounds.

Proof of Concept

`poolIdArrayIndexForExcessDeposit` is used to save `depositETHOverTargetWeight()`, which `Pool` is given priority for allocation in the next round.

The current implementation rolls over to the next `pool`, regardless of whether the current balance is sufficient or not.

`poolAllocationForExcessETHDeposit`:

```
        function poolAllocationForExcessETHDeposit(uint256 _excessET
            external
            override
            returns (uint256[] memory selectedPoolCapacity, uint8[]
        {
    ..
            for (uint256 j; j < poolCount; ) {
                uint256 poolCapacity = poolUtils.getQueuedValidator(
            uint256 poolDepositSize = ETH_PER_NODE - poolUtils.g
            uint256 remainingValidatorsToDeposit = ethToDeposit
            selectedPoolCapacity[i] = Math.min(
                poolAllocationMaxSize - selectedValidatorCount,
                Math.min(poolCapacity, remainingValidatorsToDepo
            );
            selectedValidatorCount += selectedPoolCapacity[i];
            ethToDeposit -= selectedPoolCapacity[i] * poolDeposi
@>          i = (i + 1) % poolCount;
            //For ethToDeposit < ETH_PER_NODE, we will be able t
            //but that will introduce complex logic, hence we ar
@>          if (ethToDeposit < ETH_PER_NODE || selectedValidator
@>              poolIdArrayIndexForExcessDeposit = i;
                break;
            }
```

Suppose now, the balance of `StaderStakePoolsManager` is 0 and `poolIdArrayIndexForExcessDeposit` = 1

If I have a `Validator` with funds to be allocated at `pool` = 2, I can maliciously transfer 1 wei and let `poolIdArrayIndexForExcessDeposit` roll over to 2.

This way, the next round of funding will be allocated in favor of my `Validator`.

Normally, if the current funds are not enough to allocate one `Validator`, then `poolIdArrayIndexForExcessDeposit` should not be rolled over. This is fairer.

Suggested: If `poolAllocationForExcessETHDeposit()` returns all 0's, revert to avoid rolling `poolIdArrayIndexForExcessDeposit`.

Recommended Mitigation Steps

```
    function depositETHOverTargetWeight() external override nonR
..

+       bool findValidator;
        for (uint256 i = 0; i < poolCount; i++) {
            uint256 validatorToDeposit = selectedPoolCapacity[i]
            if (validatorToDeposit == 0) {
                continue;
            }
+           findValidator = true;
            address poolAddress = IPoolUtils(poolUtils).poolAddr
            uint256 poolDepositSize = staderConfig.getStakedEthP
                IPoolUtils(poolUtils).getCollateralETH(poolIdArr

            lastExcessETHDepositBlock = block.number;
            //slither-disable-next-line arbitrary-send-eth
            IStaderPoolBase(poolAddress).stakeUserETHToBeaconCha
            emit ETHTransferredToPool(i, poolAddress, validatorT
        }
+       require(findValidator,"not valid validator");
```

Assessed type

Context

manoj9april (Stader) confirmed and commented:

> Sure, we will fix this.

manoj9april (Stader) disagreed with severity and commented:

> We expect this issue to be in QA. As in, `Cooldown` eventually evens out every pool.

Picodes (judge) commented:

> Technically valid, although the impact should remain small. Especially as the `cooldown` prevents it from using this frequently and really imbalancing among validators. Keeping Medium severity as funds are at stake and it wasn't the intended design.

sanjay-staderlabs (Stader) commented:

> @Picodes - can you please elaborate how funds are at stake? I think funds will never be at stake, as funds might go to a different `pool`, and there is no risk of funds being at stake in this case.

Picodes (judge) commented:

> My reasoning was although the impact is very limited, as the system can be gamed to send funds to the incorrect validator, which then could behave incorrectly, this report qualifies for Med severity under "leak value with a hypothetical attack path with stated assumptions, but external requirements".

sanjay-staderlabs (Stader) commented:

> @Picodes - just to clarify, there is no incorrect validator. All validators work as expected. We have penalty mechanics in place to penalize and exit validators if they behave incorrectly, so the question of sending ETH to the incorrect validator or fund loss does not exist.

Picodes (judge) commented:

> Yes, it is not an incorrect validator in the sense of a malicious one as safeguards exist, just compared to what it would have been without this bug. Unless I am missing something, you still could slightly game the system; in favor of, for example, your own validators, less performing ones, or validators starting to behave incorrectly.

sanjay-staderlabs (Stader) commented:

> This is fixed.

[M-10] Owner in `VaultProxy.sol` is `address(0)`

Submitted by josephdara, also found by ksk2345, Aymen0909, bin2chen, ChrisTina, and NoamYakov

The owner variable in the `VaultProxy.sol` is going to be zero because of a bug in `staderConfig`. The `intialize` function in `staderConfig` grants the admin role to an admin passed in the native `_grantRole()`, but does not update the mapping that `getAdmin` reads from.

```
        function initialise(
            bool _isValidatorWithdrawalVault,
            uint8 _poolId,
            uint256 _id,
            address _staderConfig
        ) external {
            if (isInitialized) {
                revert AlreadyInitialized();
            }
            UtilLib.checkNonZeroAddress(_staderConfig);
            isValidatorWithdrawalVault = _isValidatorWithdrawalVault
            isInitialized = true;
            poolId = _poolId;
            id = _id;
            staderConfig = IStaderConfig(_staderConfig);

            //@audit-issue Admin is zero on initialize
            //address(0) is going to  be returned, wrt stader.initia
            owner = staderConfig.getAdmin();
        }
```

https://github.com/code-423n4/2023-06-stader/blob/7566b5a35f32ebd55d3578b8bd05c038feb7d9cc/contracts/StaderConfig.sol#L85-L103
https://github.com/code-423n4/2023-06-stader/blob/7566b5a35f32ebd55d3578b8bd05c038feb7d9cc/contracts/StaderConfig.sol#L361-L363

The intialization of the vaults would not revert and all functions restricted to the owner would be inaccessible, unless function `updateAdmin` is called.

https://github.com/code-423n4/2023-06-stader/blob/7566b5a35f32ebd55d3578b8bd05c038feb7d9cc/contracts/StaderConfig.sol#L176-L183

The update admin does not revoke functions for the current admin in the `AccessControl` contract either, because the address returned is address zero.

Proof of Concept

```
pragma solidity 0.8.16;

import '../../contracts/library/UtilLib.sol';
import '../../contracts/StaderConfig.sol';
import '../../contracts/VaultProxy.sol';
import '../../contracts/ValidatorWithdrawalVault.sol';
import '../../contracts/OperatorRewardsCollector.sol';

import './mocks/PoolUtilsMock.sol';
import './mocks/PenaltyMockForVault.sol';
import './mocks/SDCollateralMock.sol';
import './mocks/StakePoolManagerMock.sol';

import 'forge-std/Test.sol';
import '@openzeppelin/contracts/proxy/transparent/TransparentUpg
import '@openzeppelin/contracts/proxy/transparent/ProxyAdmin.sol

contract ValidatorWithdrawalVaultTest is Test {
    address staderAdmin;
    address staderManager;
    address staderTreasury;
    PoolUtilsMock poolUtils;

    uint8 poolId;
    uint256 validatorId;

    StaderConfig staderConfig;
    VaultProxy withdrawVault;
    OperatorRewardsCollector operatorRC;

    function setUp() public {
        poolId = 1;
        validatorId = 1;

        staderAdmin = vm.addr(100);
        staderManager = vm.addr(101);
        address ethDepositAddr = vm.addr(102);

        ProxyAdmin proxyAdmin = new ProxyAdmin();

        StaderConfig configImpl = new StaderConfig();
        TransparentUpgradeableProxy configProxy = new Transparen
            address(configImpl),
            address(proxyAdmin),
            ''
        );
        staderConfig = StaderConfig(address(configProxy));
        staderConfig.initialize(staderAdmin, ethDepositAddr);
        OperatorRewardsCollector operatorRCImpl = new OperatorRe
        TransparentUpgradeableProxy operatorRCProxy = new Transp
            address(operatorRCImpl),
            address(proxyAdmin),
            ''
        );
        operatorRC = OperatorRewardsCollector(address(operatorRC
        operatorRC.initialize(staderAdmin, address(staderConfig)

        poolUtils = new PoolUtilsMock(address(staderConfig));
        PenaltyMockForVault penaltyContract = new PenaltyMockFor
        SDCollateralMock sdCollateral = new SDCollateralMock();
        ValidatorWithdrawalVault withdrawVaultImpl = new Validat

        vm.startPrank(staderAdmin);
        staderConfig.updatePoolUtils(address(poolUtils));
        staderConfig.updatePenaltyContract(address(penaltyContra
        staderConfig.updateSDCollateral(address(sdCollateral));
        staderConfig.updateOperatorRewardsCollector(address(oper
        staderConfig.updateValidatorWithdrawalVaultImplementatic

        vm.stopPrank();

        withdrawVault = new VaultProxy();
        withdrawVault.initialise(true, poolId, validatorId, addr
    }

    function testAdminRights() public {
       vm.startPrank(staderAdmin);
      staderConfig.grantRole(staderConfig.MANAGER(), staderManag
      vm.stopPrank();
    }
    function testFetchAdminAndVaultOwner() public{
        address _admin = staderConfig.getAdmin();
        assertEq(_admin, address(0));
        address _owner = withdrawVault.owner();
        assertEq(_owner, address(0));
    }

}
```

Tools Used

Foundry

Recommended Mitigation Steps

`StaderConfig.initialize` should add `setAccount(ADMIN, _admin);` at the last line.

Assessed type

Access Control

Picodes (judge) decreased severity to Medium

sanjay-staderlabs (Stader) disagreed with severity, acknowledged and commented:

> We are handling it as a part of the deployment process; `updateAdmin(address _admin)` is called during deployment process.

sanjay-staderlabs (Stader) commented:

> @Picodes - can you please check this? It think it should be part for QA. Thanks

Picodes (judge) commented:

> Considering the audited version of the code, especially in combination with issue 390, I still think this qualifies for Medium severity.

[M-11] `ValidatorWithdrawalVault.distributeRewards` can be called to make operator slashable

Submitted by rvierdiiev

An attacker can call `distributeRewards` right before `settleFunds` to make `operatorShare < penaltyAmount`. As a result, the validator will face loses.

Proof of Concept

`ValidatorWithdrawalVault.distributeRewards` can be called by anyone. It's purpose is to distribute validators rewards among the stakers protocol and the operator. After the call, the balance of `ValidatorWithdrawalVault` becomes 0.

`ValidatorWithdrawalVault.settle` is called when a validator is withdrawn from beacon chain. In this case, the balance of the contract is used to find operatorShare. If it's less than the accrued penalty by the validator, then operator is slashed.

Because `distributeRewards` is permissionless, then next situation is possible:

1. Operator decided to withdraw a validator. At the moment of that call, the balance of `ValidatorWithdrawalVault` is not 0 and `operatorShare` is 1 eth. Also the validator accrued 4.5 eth of penalty.
2. A malicious user sees when 32 eth of a validator's deposit is sent to the `ValidatorWithdrawalVault` and frontruns it with `distributeRewards` call. This makes balance to be 32 eth.
3. `operatorShare` will be 4 eth this time (permisssionless) and the penalty is 4.5, so the user is slashed.
4. In this case, if a malicious user didn't call `distributeRewards`, then the slash would not occur.

Also in same way, a permissioned operator can call `distributeRewards` to get their rewards when they are going to be slashed. As permissioned validators are not forced to have collateral to be slashed, they rescued their earnings; otherwise, they would have been sent to the `pool`.

Tools Used

VsCode

Recommended Mitigation Steps

Maybe think about restricting access to the `distributeRewards` function.

Assessed type

Access Control

manoj9april (Stader) disputed and commented:

> This is intended, as we want to `slash SD` for an operator if the penalty is greater than operator share.
> Rewards share of vaults is going to be distributed by Stader every so often to enforce this.

Picodes (judge) commented:

> This report shows how an operator can be slashed depending on the order of actions. It seems a valid Medium severity issue to me, as front-running/changing a sequence of action could lead to a loss of funds compared to what the `operator` and `users` thought they should get.

> Wouldn't it solve this to also fetch and solve `penaltyAmount` during `distributeRewards`?

CoOnan (warden) commented:

> Not sure if I'm following the impact here.

> At the moment of that call, balance of `ValidatorWithdrawalVault` is not 0 and `operatorShare` is 1 eth. Also validator accrued 4.5 eth of penalty.

> This case means the validator has to be slashed. In both cases, the user must be slashed according to the design. I can't see the exact root cause. Also, `distributeRewards` can be called at any time; no need to make a front-run, so the system behaves as intended.

rvierdiyev (warden) commented:

> @CoOnan - the problem here is that once `distributeRewards` is called, then all balances of the vault becomes 0. As I described in the example, the user had 1 eth inside the vault before the `distributeRewards` call and he has 4.5 eth of penalty.

> When withdrawals of 32 eth should come, the user's balance will be 1 + 4 eth, which is more than penalty. But because a malicious user calls `distributeRewards` before 32 eth will come, then the user in the end has only 4 eth, which is less than penalty.

[M-12] `ValidatorWithdrawalVault.settleFunds` doesn’t check amount that user has inside `NodeELRewardVault` to pay for penalty

Submitted by [rvierdiiev]

`ValidatorWithdrawalVault.settleFunds` doesn’t check amount that user has inside `NodeELRewardVault` to pay for penalty. That value can increase operator’s earned amount, which can avoid slashing.

Proof of Concept

When a validator withdraws from beacon chain the `ValidatorWithdrawalVault.settleFunds` function is called. This function calculates amount that a validator [has earned] for attestations as a validator. So only the balance of this contract [is considered].

The function [fetches penalty amount]. This penalty amount contains [of 3 points]: `_mevTheftPenalty`, `_missedAttestationPenalty` and `_missedAttestationPenalty`.

In this case, if the penalty amount is bigger than the validator’s earning on `ValidatorWithdrawalVault`, the [SD collateral is slashed].

Now, we need to understand how validator receives funds in this system. All attestation payments come to `ValidatorWithdrawalVault`, while `mev` / `block` proposal funds are coming to `SocializingPool` or `NodeELRewardVault` (depends on user’s choice). So actually, `_missedAttestationPenalty` is responding to `ValidatorWithdrawalVault` earning, while `_mevTheftPenalty` is responding to `NodeELRewardVault` earnings.

That means, `NodeELRewardVault` balance should also be checked in order to find out how many earnings a validator has and they should be also counted when applying the penalty.

Simple example:

1. A validator wants to exit.
2. An operator earning is 0.1 eth inside `ValidatorWithdrawalVault`.
3. The accrued penalty is 0.11, which means the user will be slashed.
4. The operator also has `NodeELRewardVault` where their operator's reward is 0.05 eth.
5. As result, the user has enough balance to cover penalty, but they were still penalized.

Tools Used

VsCode

Recommended Mitigation Steps

As you accrue `_mevTheftPenalty` inside `ValidatorWithdrawalVault`, you also should calculate the operator's rewards inside `NodeELRewardVault`.

Assessed type

Error

manoj9april (Stader) acknowledged and commented:

> Rewards are treated separately between CL and EL. We will take this suggestion into account for next upgrades.

Picodes (judge) commented:

> Keeping medium severity as this report shows how an operator could be slashed, despite having earned more than the penalty if we combine EL and CL rewards, which could lead to a loss of funds.

[M-13] No bidder has incentive to bid in the Auction except doing last-minute `MEV` due to fixed `endBlock`

Submitted by 0xWaitress, also found by LaScaloneta, Josiah, peanuts, RaymondFam, and T1MOH

Proof of Concept

The auction of `SD Token` has a fixed `endBlock`. Bidder(s) would like to get SD Token with the least amount of ETH and they are all incentivized to just bid at the last block, leading to loss of protocol principle during the auction.

```
    function createLot(uint256 _sdAmount) external override wher
        lots[nextLot].startBlock = block.number;
        lots[nextLot].endBlock = block.number + duration;
        lots[nextLot].sdAmount = _sdAmount;
```

Genearally, auctions with a fixed endtime has the known vulnerability of being bid on at the last block. Essentially, the validator/MEVer who has the ability to slip in transaction at the last block has the highest likelihood to get the bid. This basically gives them an advantage and would lead to the auction to end at lower price.

Recommended Mitigation Steps

Extend the final `endBlock` at each bid. This can be activated at the end of 1h, for example, to ensure the highest bidder can take the auction on in a fair manner.

Assessed type

MEV

manoj9april (Stader) disputed and commented:

> MEV bots are welcome to be bidders.

Picodes (judge) commented:

> This is a valid vulnerability in my opinion. MEV are, of course welcome to bid, but the problem is the current auction system doesn't allow for a proper price discovery. From a game theory standpoint, no one has any interest in revealing their bid before the end of the auction; and when bidding, you tend to not reveal your max bid but try to guess the "second-highest bid". Therefore, the current design can quickly lead to a suboptimal final price as someone may have increased their bid, given more time.

> Note as well, that it is currently relatively cheap to censor a transaction for a few blocks by bribing block builders. So with the current design, value may easily be lost trying to censor other bidders or bribing miners to be included.

manoj9april (Stader) acknowledged, disagreed with severity and commented:

> Yes, We agree with your point. But for now, we would prefer to go with a current simpler approach and improve later phases, based your recommendation.

> I think this should be considered as low severity.

[M-14] Chainlink's `latestRoundData` may return a stale or incorrect result

Submitted by Madalad, *also found by* MohammedRizwan, turvy_fuzz, Breeje, etherhood, dwward3n, Aymen0909, LaScaloneta, erictee, tallo, DadeKuma, bin2chen, peanuts, Bauchibred, Bauchibred, Bauchibred, Bauchibred, whimints, whimints, piyushshukla, rvierdiiev, saneryee, Hama, Madalad, *and* kutugu

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/StaderOracle.sol#L646
https://github.com/code-423n4/2023-06-stader/blob/main/contracts/StaderOracle.sol#L648

Chainlink's `latestRoundData` is used here to retrieve price feed data; however, there is insufficient protection against price staleness.

Return arguments other than `int256 answer` are necessary to determine the validity of the returned price, as it is possible for an outdated price to be received. See here for reasons why a price feed might stop updating.

The return value `updatedAt` contains the timestamp at which the received price was last updated, and can be used to ensure that the price is not outdated. See more information about `latestRoundID` in the Chainlink docs. Inaccurate price data can lead to functions not working as expected and/or loss of funds.

Proof of Concept

```
    function getPORFeedData()
        internal
        view
        returns (
            uint256,
            uint256,
            uint256
        )
    {
        (, int256 totalETHBalanceInInt, , , ) = AggregatorV3Inte
            .latestRoundData();
        (, int256 totalETHXSupplyInInt, , , ) = AggregatorV3Inte
            .latestRoundData();
        return (uint256(totalETHBalanceInInt), uint256(totalETHX
    }
```

Recommended Mitigation Steps

Add a check for the `updatedAt` returned value from `latestRoundData`.

```
    function getPORFeedData()
        internal
        view
        returns (
            uint256,
            uint256,
            uint256
        )
    {
-       (, int256 totalETHBalanceInInt, , , ) = AggregatorV3Inte
+       (, int256 totalETHBalanceInInt, , uint256 balanceUpdatec
            .latestRoundData();
+       require(block.timestamp - balanceUpdatedAt <= MAX_DELAY,

-       (, int256 totalETHXSupplyInInt, , , ) = AggregatorV3Inte
+       (, int256 totalETHXSupplyInInt, , uint256 supplyUpdatedA
            .latestRoundData();
+       require(block.timestamp - supplyUpdatedAt <= MAX_DELAY,

        return (uint256(totalETHBalanceInInt), uint256(totalETHX
    }
```

Assessed type

Oracle

manoj9april (Stader) acknowledged and commented:

> Solution with chainlink is not finalized.

Low Risk and Non-Critical Issues

For this audit, 24 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **RaymondFam** received the top score from the judge.

The following wardens also submitted reports: niser93, turvy_fuzz, 0xhacksmithh, ernestognw, naman1778, SAAJ, catellatech, btk, LaScaloneta, hals, hunter_w3b, mgf15, DadeKuma, ChrisTina, jaraxxus, Rolezn, matrix_0wl, bin2chen, solsaver, T1MOH, fatherOfBlocks, Sathish9098 *and* 0xWaitress.

Bid Withdrawal Mechanism in `Auction` Contract

The current design of the `Auction` contract includes the `withdrawUnselectedBid` function, which allows a withdrawal of funds only after the auction ends. While this safeguards the auction process from disruptions, it locks up participants' funds for a prolonged period.

To enhance the fluidity of the auction, it is proposed to refactor the `addBid` function, thereby eliminating the need for `withdrawUnselectedBid`. In the refactored `addBid`, the previous highest bidder's bid will be deleted from `lotItem.bids[lotItem.highestBidder]`, and the bidder's funds will be refunded before updating `lotItem.highestBidder`, `lotItem.highestBidAmount`, and `lotItem.bids[msg.sender]`.

The refactored `addBid` function could look like this:

```
    function addBid(uint256 lotId) external payable override whenNot
        // reject payments of 0 ETH
        if (msg.value == 0) revert InSufficientETH();

        LotItem storage lotItem = lots[lotId];
        if (block.number > lotItem.endBlock) revert AuctionEnded();

        uint256 totalUserBid = lotItem.bids[msg.sender] + msg.value;

        if (totalUserBid < lotItem.highestBidAmount + bidIncrement)

        // Refund previous highest bidder and delete their bid befor
        if(lotItem.highestBidder != address(0)) {
            payable(lotItem.highestBidder).transfer(lotItem.highestE
            delete lotItem.bids[lotItem.highestBidder];
        }

        lotItem.highestBidder = msg.sender;
        lotItem.highestBidAmount = totalUserBid;
        lotItem.bids[msg.sender] = totalUserBid;

        emit BidPlaced(lotId, msg.sender, totalUserBid);
    }
```

This refactor makes sure that before a new highest bid is accepted, the previous highest bid is refunded and removed from `lotItem.bids` . It's important to note this design would need a proper audit and testing to make sure it behaves as expected in all scenarios. Additionally, take into account that the refund operation could fail due to gas constraints, so a more robust mechanism to handle such scenarios might be needed.

Streamlining the `addBid` function in `Auction` Contract: Removal of Redundant Code

The current implementation of the `addBid` function in the `Auction` contract contains a line of code that checks whether the sent ETH amount (`msg.value`) is zero. Specifically, the line `if (msg.value == 0) revert InSufficientETH();` appears to be redundant, due to the subsequent condition that checks if the total bid is less than the highest current bid, plus the bid increment.

In the event `msg.value` is zero, the latter condition `if (totalUserBid < lotItem.highestBidAmount + bidIncrement) revert InSufficientBid();`

would effectively handle the scenario, as it encompasses the case where no new ETH is added to the bid.

While separate error messages could aid in user experience by precisely pointing out the mistake, in this context, it seems superfluous; a user not sending any ETH with the bid and a user not meeting the minimum bid increment inherently suggest the same issue - the bid is insufficient.

Therefore, it is recommended to remove the line `if (msg.value == 0) revert InSufficientETH();` from the `addBid` function to simplify the code and remove redundancy, without compromising the function's integrity or user experience.

Initiating Role Management in `VaultFactory` Contract Initialization and Subsequent Function Call Permissions

The `VaultFactory` contract could greatly benefit from incorporating a role management function for the `NODE_REGISTRY_CONTRACT` role within the `initialize` function. This initial role assignment is key to ensuring secure and authorized contract functionality from the onset.

Upon contract deployment, it's crucial that functions `deployWithdrawVault` and `deployNodeELRewardVault` are callable, particularly considering their `onlyRole(NODE_REGISTRY_CONTRACT) visibility`. Setting the `NODE_REGISTRY_CONTRACT` role within the `initialize` function can facilitate this.

Here's a sample implementation within the initialize function:

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/factory/VaultFactory.sol#L23-L32

```
    function initialize(address _admin, address _staderConfig) exter
        ...
        _grantRole(DEFAULT_ADMIN_ROLE, _msgSender());
+       _grantRole(NODE_REGISTRY_CONTRACT, _msgSender());
    }
```

The same shall apply to the *‘MINTERROLE’ and ‘BURNERROLE’* in the ETHx contract too.

Comments and Code Mismatch

The withdraw function in the `SDCollateral` contract lacks the implementation of a withdrawal delay, as indicated in the function’s NatSpec documentation. This omission contradicts the intended behavior specified in the documentation, potentially leading to inconsistencies and unintended consequences in the withdrawal process.

https://github.com/code-423n4/2023-06-stader/blob/7566b5a35f32ebd55d3578b8bd05c038feb7d9cc/contracts/SDCollateral.sol#L54-L73

```
    /// @notice for operator to request withdraw of sd
    /// @dev it does not transfer sd tokens immediately
    /// operator should come back after withdrawal-delay time tc
    /// this requested sd is subject to slashes
    function withdraw(uint256 _requestedSD) external override {
        address operator = msg.sender;
        uint256 opSDBalance = operatorSDBalance[operator];

        if (opSDBalance < getOperatorWithdrawThreshold(operator)
            revert InsufficientSDToWithdraw(opSDBalance);
        }
        operatorSDBalance[operator] -= _requestedSD;

        // cannot use safeERC20 as this contract is an upgradeak
        if (!IERC20(staderConfig.getStaderToken()).transfer(paya
            revert SDTransferFailed();
        }

        emit SDWithdrawn(operator, _requestedSD);
    }
```

It is recommended to address this discrepancy by implementing the appropriate withdrawal delay mechanism in the withdraw function. By adhering to the intended withdrawal delay, operators can claim their requested SD tokens only after the specified time period, ensuring the proper functioning of the collateral slashing mechanism and maintaining the expected behavior of the contract.

Identical Code Logic

In `ValidatorWithdrawalVault.sol`, functions `distributeRewards` and `settleFunds` have the following identical code snippet:

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/ValidatorWithdrawalVault.sol#L31-L33
https://github.com/code-423n4/2023-06-stader/blob/main/contracts/ValidatorWithdrawalVault.sol#L55-L57

```
        uint8 poolId = VaultProxy(payable(address(this))).poolIc
        uint256 validatorId = VaultProxy(payable(address(this)))
        IStaderConfig staderConfig = VaultProxy(payable(address(
```

It is recommended to refactor the duplicate code snippet by grouping it into a private function. This will improve code maintainability and reduce redundancy. By extracting the common code snippet into a separate function, you can avoid duplicating the same code and make future updates or modifications easier.

Missing Length Check in `claim` Function

The `claim` function in the `SocializingPool` contract should include a length check for the input parameters `_index`, `_amountSD`, `_amountETH`, and `_merkleProof` before invoking the `_claim` internal function. This check ensures that all arrays have the same length, preventing potential errors or inconsistencies during the reward claiming process.

By verifying the lengths of these arrays, you can ensure that the corresponding values align properly and avoid any unexpected behavior or unintended consequences. Adding this length check will help maintain the integrity of the claim operation and enhance the overall robustness of the contract.

Consider adding the following length check at the beginning of the claim function:

```
    require(
        _index.length == _amountSD.length &&
        _index.length == _amountETH.length &&
        _index.length == _merkleProof.length,
        "Input lengths mismatch"
    );
```

Unnecessary payable Keyword in `withdraw` Function

The `payable` keyword in the line `if (!IERC20(staderConfig.getStaderToken()).transfer(payable(operator), _requestedSD))` of the `withdraw` function in the `SDCollateral` contract is unnecessary and can be safely removed.

Since you're not dealing with the transfer of native tokens (ETH) in this specific line, using the `payable` keyword is not required. The `payable` keyword is typically used when transferring ETH or performing operations related to ETH transfers.

You can simplify the line to:

https://github.com/code-423n4/2023-06-stader/blob/7566b5a35f32ebd55d3578b8bd05c038feb7d9cc/contracts/SDCollateral.sol#LL68C1-L68C96

```
    -        if (!IERC20(staderConfig.getStaderToken()).transfer(pay
    +        if (!IERC20(staderConfig.getStaderToken()).transfer(ope
```

By removing the `payable` keyword, you ensure clarity and eliminate any confusion regarding the transfer of native tokens, making the code more concise and accurate.

Removal of Unused `receive` and `fallback` Functions

The `receive` and `fallback` functions in the contract `PermissionedPool` and `StaderStakePoolsManager.sol` can be safely removed, as they serve as protection against accidental submissions by calling non-existent functions. By removing these functions, direct transfers of Ether to the contract will be disallowed, achieving the same outcome as having the functions present with the reverting logic.

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/PermissionedPool.sol#L51-L59
https://github.com/code-423n4/2023-06-stader/blob/main/contracts/StaderStakePoolsManager.sol#L62-L70

```
    // protection against accidental submissions by calling non-
    receive() external payable {
        revert UnsupportedOperation();
    }

    // protection against accidental submissions by calling non-
    fallback() external payable {
        revert UnsupportedOperation();
    }
```

Stale Values Attributable to Unpredictably Delayed Updates

The following concensus in `StaderOracle.sol` will have to be exactly met in order to have the update logics executed, highly leading to stale answers:

1. Function `submitExchangeRateData`

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/StaderOracle.sol#L147-L156

```
        if (
            submissionCount == trustedNodesCount / 2 + 1 &&
            _exchangeRate.reportingBlockNumber > exchangeRate.re
        ) {
            updateWithInLimitER(
                _exchangeRate.totalETHBalance,
                _exchangeRate.totalETHXSupply,
                _exchangeRate.reportingBlockNumber
            );
        }
```

2. Function `submitSocializingRewardsMerkleRoot`

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/StaderOracle.sol#L255-L267

```
        if ((submissionCount == trustedNodesCount / 2 + 1)) {
            address socializingPool = IPoolUtils(staderConfig.ge
                _rewardsData.poolId
            );
            ISocializingPool(socializingPool).handleRewards(_rew

            emit SocializingRewardsMerkleRootUpdated(
                _rewardsData.index,
                _rewardsData.merkleRoot,
                _rewardsData.poolId,
                block.number
            );
        }
```

3. Function `submitSDPrice`

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/StaderOracle.sol#L290-L297

```
        if ((submissionCount == (2 * trustedNodesCount) / 3 + 1)
            lastReportedSDPriceData = _sdPriceData;
            lastReportedSDPriceData.sdPriceInETH = getMedianValu
            delete sdPrices;

            // Emit SD Price updated event
            emit SDPriceUpdated(_sdPriceData.sdPriceInETH, _sdPr
        }
```

4. Function `submitValidatorStats`

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/StaderOracle.sol#L371-L388

```
        if (
            submissionCount == trustedNodesCount / 2 + 1 &&
            _validatorStats.reportingBlockNumber > validatorStat
        ) {
            validatorStats = _validatorStats;

            // Emit stats updated event
            emit ValidatorStatsUpdated(
                _validatorStats.reportingBlockNumber,
                _validatorStats.exitingValidatorsBalance,
                _validatorStats.exitedValidatorsBalance,
                _validatorStats.slashedValidatorsBalance,
                _validatorStats.exitingValidatorsCount,
                _validatorStats.exitedValidatorsCount,
                _validatorStats.slashedValidatorsCount,
                block.timestamp
            );
        }
```

5. Function `submitWithdrawnValidators`

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/StaderOracle.sol#L431-L445

```
        if (
            submissionCount == trustedNodesCount / 2 + 1 &&
            _withdrawnValidators.reportingBlockNumber > reportir
        ) {
            reportingBlockNumberForWithdrawnValidators = _withdr
            INodeRegistry(_withdrawnValidators.nodeRegistry).wit

            // Emit withdrawn validators updated event
            emit WithdrawnValidatorsUpdated(
                _withdrawnValidators.reportingBlockNumber,
                _withdrawnValidators.nodeRegistry,
                _withdrawnValidators.sortedPubkeys,
                block.timestamp
            );
        }
```

6. Function `submitMissedAttestationPenalties`

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/StaderOracle.sol#L482-L493

```
        if ((submissionCount == trustedNodesCount / 2 + 1)) {
            lastReportedMAPDIndex = _mapd.index;
            uint256 keyCount = _mapd.sortedPubkeys.length;
            for (uint256 i; i < keyCount; ) {
                bytes32 pubkeyRoot = UtilLib.getPubkeyRoot(_mapc
                missedAttestationPenalty[pubkeyRoot]++;
                unchecked {
                    ++i;
                }
            }
            emit MissedAttestationPenaltyUpdated(_mapd.index, bl
        }
```

Too Harsh Deactivation Rule on Operators

It only requires one front run validator as a culprit to deactivate an operator, as is evidenced in the following code logics of:

1. `PermissionedNodeRegistry.sol` with functions `allocateValidatorsAndUpdateOperatorId` and `getTotalQueuedValidatorCount` getting denied:

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/PermissionedNodeRegistry.sol#L670-L677

```
    // handle front run validator by changing their status and c
    function handleFrontRun(uint256 _validatorId) internal {
        validatorRegistry[_validatorId].status = ValidatorStatus
        uint256 operatorId = validatorRegistry[_validatorId].ope
        if (operatorStructById[operatorId].active) {
            _deactivateNodeOperator(operatorId);
        }
    }
```

2. `PermissionlessNodeRegistry.sol` with functions `addValidatorKeys`, `changeSocializingPoolState` and `updateOperatorDetails` getting denied:

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/PermissionlessNodeRegistry.sol#L624-L629

```
    // handle front run validator by changing their status, deac
    function handleFrontRun(uint256 _validatorId) internal {
        validatorRegistry[_validatorId].status = ValidatorStatus
        uint256 operatorId = validatorRegistry[_validatorId].ope
        operatorStructById[operatorId].active = false;
    }
```

Consider enforcing this rule only when exceeding a threshold percentage of validators front running.

Typo mistakes

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/PoolSelector.sol#L46

```
-     * @notice calculates the count of validator to deposit on
+     * @notice calculates the count of validators to deposit or
```

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/ValidatorWithdrawalVault.sol#L97

```
-         uint256 collateralETH = getCollateralETH(poolId, stader
+         uint256 collateralETH = getCollateralETH(poolId, stader
```

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/PermissionedPool.sol#L170

```
-     * @notice transfer the excess ETH sent by some EAO or non
+     * @notice transfer the excess ETH sent by some EOA or non
```

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/PermissionedNodeRegistry.sol#L520

```
-     * @param _endIndex  up to end index of validator queue to
```

```
+      * @param _endIndex  up to end index of validator queue to
```

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/PermissionedNodeRegistry.sol#L104

```
-      * @return feeRecipientAddress fee recipient address for al
+      * @return feeRecipientAddress fee recipient address for al
```

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/VaultProxy.sol#L38

```
-     /**route all call to this proxy contract to the respective
+     /**route all calls to this proxy contract to the respective
```

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/VaultProxy.sol#L64

```
-      * @notice @update the owner of vault proxy contrat
+      * @notice @update the owner of vault proxy contract
```

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/PoolSelector.sol#L32

```
-      * @dev pool index start from 1 with permission less pool
+      * @dev pool index starts from 1 with permissionless pool
```

Repeatedly Used Instances May be Cached

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/PermissionedNodeRegistry.sol#L106-L131

```
        function onboardNodeOperator(string calldata _operatorName,
            external
            override
            whenNotPaused
            returns (address feeRecipientAddress)
        {
            address poolUtils = staderConfig.getPoolUtils();
    +        IPoolUtils _poolUtils = IPoolUtils(poolUtils);
    -        if (IPoolUtils(poolUtils).poolAddressById(POOL_ID) != s
    +        if (_poolUtils.poolAddressById(POOL_ID) != staderConfig
                revert DuplicatePoolIDOrPoolNotAdded();
            }
    -        IPoolUtils(poolUtils).onlyValidName(_operatorName);
    +        _poolUtils.onlyValidName(_operatorName);
            UtilLib.checkNonZeroAddress(_operatorRewardAddress);
            if (nextOperatorId > maxOperatorId) {
                revert MaxOperatorLimitReached();
            }
            if (!permissionList[msg.sender]) {
                revert NotAPermissionedNodeOperator();
            }
            //checks if operator already onboarded in any pool of pr
    -        if (IPoolUtils(poolUtils).isExistingOperator(msg.sender
    +        if (_poolUtils.isExistingOperator(msg.sender)) {
                revert OperatorAlreadyOnBoardedInProtocol();
            }
            feeRecipientAddress = staderConfig.getPermissionedSocial
            onboardOperator(_operatorName, _operatorRewardAddress);
            return feeRecipientAddress;
        }
```

Essential Early Checks

Key checks should be placed at the beginning part of a function logic where possible:

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/PermissionedNodeRegistry.sol#L106-L131

```
    function onboardNodeOperator(string calldata _operatorName,
        external
        override
        whenNotPaused
        returns (address feeRecipientAddress)
    {
+        if (!permissionList[msg.sender]) {
```

```
+             revert NotAPermissionedNodeOperator();
+         }
        address poolUtils = staderConfig.getPoolUtils();
        if (IPoolUtils(poolUtils).poolAddressById(POOL_ID) != st
            revert DuplicatePoolIDOrPoolNotAdded();
        }
        IPoolUtils(poolUtils).onlyValidName(_operatorName);
        UtilLib.checkNonZeroAddress(_operatorRewardAddress);
        if (nextOperatorId > maxOperatorId) {
            revert MaxOperatorLimitReached();
        }
-         if (!permissionList[msg.sender]) {
-             revert NotAPermissionedNodeOperator();
-         }
        //checks if operator already onboarded in any pool of pr
        if (IPoolUtils(poolUtils).isExistingOperator(msg.sender)
            revert OperatorAlreadyOnBoardedInProtocol();
        }
        feeRecipientAddress = staderConfig.getPermissionedSocial
        onboardOperator(_operatorName, _operatorRewardAddress);
        return feeRecipientAddress;
    }
```

Activate the optimizer

Before deploying your contract, activate the optimizer when compiling using “ `solc --optimize --bin sourceFile.sol` ”. By default, the optimizer will optimize the contract assuming it is called 200 times across its lifetime. If you want the initial contract deployment to be cheaper and the later function executions to be more expensive, set it to “ `--optimize-runs=1` ”. Conversely, if you expect many transactions and do not care for higher deployment cost and output size, set “ `--optimize-runs` ” to a high number.

```
module.exports = {
solidity: {
version: "0.8.16",
settings: {
  optimizer: {
    enabled: true,
    runs: 1000,
  },
},
    },
    };
```

Please visit the following site for further information:

https://docs.soliditylang.org/en/v0.5.4/using-the-compiler.html#using-the-commandline-compiler

Here's one example of instance on opcode comparison that delineates the gas saving mechanism:

```
    for !=0 before optimization
    PUSH1 0x00
    DUP2
    EQ
    ISZERO
    PUSH1 [cont offset]
    JUMPI

    after optimization
    DUP1
    PUSH1 [revert offset]
    JUMPI
```

Disclaimer: There have been several bugs with security implications related to optimizations. For this reason, Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised. High-severity security issues due to optimization bugs have occurred in the past . A high-severity bug in the `emscripten -generated solc-js` *compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity* `CHANGELOG` *. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. Please measure the gas savings from optimizations, and carefully weigh them against the possibility of an optimization-related bug. Also, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.*

manoj9april (Stader) acknowledged

Picodes (judge) commented:

> Note as well #236 and #106 by the warden.

Note: Issues 236 and 106 have been included below, at the judge's request.

`SD Collateral` Contract's Deposit Mechanism

https://github.com/code-423n4/2023-06-stader/blob/7566b5a35f32ebd55d3578b8bd05c038feb7d9cc/contracts/SDCollateral.sol#L151-L191
https://github.com/code-423n4/2023-06-stader/blob/7566b5a35f32ebd55d3578b8bd05c038feb7d9cc/contracts/SDCollateral.sol#L210-L213

The `SD Collateral` contract contains a vulnerability related to the deposit mechanism `depositSDAsCollateral function` due to price fluctuations in the `SD token`. This arises from the calculation of the minimum SD token amount required for onboarding validators and the subsequent check for sufficient collateral in the `hasEnoughSDCollateral` function.

Specifically, `PermissionedNodeRegistry.checkInputKeysCountAndCollateral()` and `PermissionlessNodeRegistry.checkInputKeysCountAndCollateral()` externally calling the `hasEnoughSDCollateral` function are possibly going to run into DoS:

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/PermissionedNodeRegistry.sol#L706-L716
https://github.com/code-423n4/2023-06-stader/blob/main/contracts/PermissionlessNodeRegistry.sol#L667-L676

```
        //checks if operator has enough SD collateral for adding
        //SD threshold for permissioned NOs is 0 for phase1
        if (
            !ISDCollateral(staderConfig.getSDCollateral()).hasEn
                msg.sender,
                POOL_ID,
                totalNonTerminalKeys + keyCount
            )
        ) {
            revert NotEnoughSDCollateral();
        }
```

Proof of Concept

The vulnerability can be demonstrated through the following steps:

1. Price Fluctuation: Assume that the price of the `SD token` experiences a significant drop after an operator calculates and [deposits](https://github.com/code-423n4/2023-06-stader/blob/7566b5a35f32ebd55d3578b8bd05c038feb7d9cc/contracts/SDCollateral.sol#L183-L191) the minimum `SD token` amount required for onboarding validators.
2. Calculation Discrepancy: The `getMinimumSDToBond` function, which determines the required `SD token` amount based on the current price is eventually invoked. However, due to the price drop, the calculated minimum `SD token` amount becomes larger than initially expected.
3. Inadequate Collateral: Because the calculated minimum amount has increased, the operator's balance is now considered insufficient, resulting in a false negative indication stemming from the returned values of functions `getRemainingSDToBond`,

https://github.com/code-423n4/2023-06-stader/blob/7566b5a35f32ebd55d3578b8bd05c038feb7d9cc/contracts/SDCollateral.sol#L183-L191

```
    function getRemainingSDToBond(
        address _operator,
        uint8 _poolId,
        uint256 _numValidator
    ) public view override returns (uint256) {
        uint256 sdBalance = operatorSDBalance[_operator];
        uint256 minSDToBond = getMinimumSDToBond(_poolId, _numVa
        return (sdBalance >= minSDToBond ? 0 : minSDToBond - sdB
    }
```

and `hasEnoughSDCollateral`:

https://github.com/code-423n4/2023-06-stader/blob/7566b5a35f32ebd55d3578b8bd05c038feb7d9cc/contracts/SDCollateral.sol#L155-L161

```
    function hasEnoughSDCollateral(
        address _operator,
        uint8 _poolId,
        uint256 _numValidator
    ) external view override returns (bool) {
        return (getRemainingSDToBond(_operator, _poolId, _numVal
    }
```

4. Rejection: As a consequence, `PermissionedNodeRegistry.checkInputKeysCountAndCollateral()` or `PermissionlessNodeRegistry.checkInputKeysCountAndCollateral()` reverts, disrupting the onboarding process for validators and creating inconsistency in collateral management.

Recommended Mitigation Steps

To address this vulnerability and ensure a more reliable deposit mechanism, the following mitigation steps are recommended:

1. Atomic Deposit: Modify the deposit process to ensure atomicity. Instead of depositing `SD tokens` upfront, operators should perform the necessary calculations and checks first before depsoiting the exact required amount of `SD tokens` and executing the onboarding operation. This approach minimizes exposure to price fluctuations.
2. Regular Monitoring: Implement a system to monitor and track the price of the `SD token` regularly. By staying informed about price fluctuations, operators can make informed decisions and adjust their actions accordingly.
3. Dynamic Threshold Adjustment: Consider implementing dynamic threshold mechanisms that account for price fluctuations. By adjusting the thresholds based on the current `SD token` price, operators can maintain a more stable and consistent collateral management process.

Assessed type

Oracle

manoj9april (Stader) disputed and commented:

> TWAP of 24 hours solves this issue and is implemented in Oracles.

Picodes (judge) decreased severity to QA and commented:

> Indeed there is already a TWAP to avoid sudden price drops which partially mitigates this. Downgrading to Low/QA.

Unrestricted Access to the `createLot` Function in the `Auction` Contract

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/Auction.sol#L48-L60

In the current design of the Stader ecosystem's smart contracts, the `createLot` function of the `Auction` contract can be called by anyone. This allows arbitrary users to create an auction lot, leading to unintentional donations of their Stader tokens. As these tokens are upon completion of the auction, they are then transferred to the `treasury` if there is no bidder:

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/Auction.sol#L114

```
            if (!IERC20(staderConfig.getStaderToken()).transfer(stac
```

or the ETH proceeds to the `stake pool manager`:

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/Auction.sol#L102

```
            IStaderStakePoolManager(staderConfig.getStakePoolManager
```

This can lead to a loss of Stader tokens for users without any benefit in return.

Proof of Concept

In the `Auction` contract, the `createLot` function is open to all users:

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/Auction.sol#L48

```
    function createLot(uint256 _sdAmount) external override wher
        ...
    }
```

In the `SDCollateral` contract, `createLot` is called in the context of slashing:

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/SDCollateral.sol#L97

```
    function slashSD(address _operator, uint256 _sdToSlash) inte
        ...
        IAuction(staderConfig.getAuctionContract()).createLot(sc
        ...
    }
```

Unfortunately, there is no restriction in place in the `Auction` contract that would prevent arbitrary users from calling `createLot` outside of a slashing context.

Recommended Mitigation Steps

To address this vulnerability, consider applying one or both of the following changes:

1. Modify the `Auction` contract to restrict the `createLot` function only to the `SDCollateral` contract or a specific privileged role. This can be achieved through the OpenZeppelin's `AccessControl` contract or similar mechanism and it would ensure that the function can't be trapping arbitrary users.
2. Implement checks within the `createLot` function to validate the context of the call. For instance, the function could verify that the transaction is part of a

slashing event before proceeding.

In addition, make sure to properly communicate to your users the consequences and conditions of interacting with the `createLot` function to avoid unintentional token losses. Thoroughly test all changes to the contract to ensure they behave as expected without introducing new vulnerabilities.

Assessed type

Access Control

manoj9april (Stader) disputed and commented:

> No person should be able to interact with this contract without understanding what it does.

Picodes (judge) decreased severity to QA and commented:

> This would be a safety check to prevent users from doing mistakes, so is of Non-Critical severity.

Gas Optimizations

For this audit, 28 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **JCN** received the top score from the judge.

The following wardens also submitted reports: bigtone, niser93, 0xSmartContract, Tomio, c3phas, ernestognw, SM3_SS, 0xhacksmithh, naman1778, petrichor, SAQ, etherhood, Raihan, Rageur, LaScaloneta, koxuan, DavidGiladi, K42, mgf15, SAAJ, sebghatullah, matrix_0wl, shamsulhaq123, 0x70C9, Sathish9098, 0xWaitress *and* piyushshukla.

Summary

Some optimizations were benchmarked via the protocol's tests, i.e. using the following config: `solc version 0.8.16`, `optimizer on`, and `200 runs`. Optimizations that were not benchmarked are explained via EVM gas costs and opcodes.

Please note that a fair amount of functions that undergo optimizations are not benchmarked in the tests. Therefore, the total gas saved in the functions below may actually be greater when considering the gas saved from instances that don't include benchmarks.

Below are the overall average gas savings for the following tested functions (with all the optimizations applied):

Function	Before	After	Avg Gas Savings
Auction.claimSD	8693	8641	52
Auction.createLot	105780	103958	1822
NodeELRewardVault.withdraw	122581	120166	2415
OperatorRewardsCollector.claim	35757	35690	67
Penalty.updateTotalPenaltyAmount	53839	51089	2750
PermissionedNodeRegistry.addValidatorKeys	973836	962403	11433
PermissionedNodeRegistry.withdrawnValidators	30196	14323	15873
PermissionlessNodeRegistry.addValidatorKeys	839030	829233	9797
PermissionlessNodeRegistry.markValidatorReadyToDeposit	81811	81623	188
PermissionlessNodeRegistry.withdrawnValidators	35136	18608	16528
PermissionlessNodeRegistry.updateDepositStatusAndBlock	34034	14278	19756
SDCollateral.withdraw	21499	21169	330
SDCollateral.slashValidatorSD	98107	93733	4374
ValidatorWithdrawalVault.distributeRewards	85468	81847	3621
ValidatorWithdrawalVault.settleFunds	130849	117750	13099

Total gas saved across all listed functions: 102105

Notes:

- The Gas report output, after all optimizations have been applied, can be found at the end of the report.
- The final diff for the contract, with all the optimizations applied, can be found here.
- The `Avg` gas for `NodeELRewardVault.withdraw` changes between tests and so the `Max` column (which remains the same) is used when benchmarking this function.

Gas Optimizations Summary

Number	Issue	Instances
[G-01]	State variables can be cached instead of re-reading them from storage	11
[G-02]	State variables can be packed into fewer storage slots	6
[G-03]	Structs can be packed into fewer storage slots	3
[G-04]	Refactor `external` / `internal` function to avoid unnecessary External Calls	4
[G-05]	Refactor `external` / `internal` function to avoid unnecessary SLOAD	8
[G-06]	Cache state variables outside of loop to avoid reading/writing storage on every iteration	4
[G-07]	Cache external calls outside of loop to avoid re-calling function on each iteration	2
[G-08]	Refactor code to avoid unnecessary SLOAD	1
[G-09]	Utilize return value from internal function to avoid unnecessary SLOAD	2
[G-10]	Multiple address mappings can be combined into a single mapping of an address to a struct, where appropriate	1
[G-11]	Using storage instead of memory for structs/arrays saves gas	1
[G-12]	Using bools for storage incurs overhead	-
[G-13]	Multiple accesses of a mapping/array should use a local variable cache	4
[G-14]	Combine events to save two `Glogtopic (375 gas)`	-

Number	Issue	Instances
[G-15]	Call environment variables directly instead of caching them	-
[G-16]	Use assembly to perform efficient back-to-back calls	3

[G-01] State variables can be cached instead of re-reading them from storage

Caching of a state variable replaces each `Gwarmaccess (100 gas)` with a much cheaper stack read.

There are 11 instances of this issue. (For in-depth details on this and all further gas optimizations with multiple instances, please see the warden's full report.)

[G-02] State variables can be packed into fewer storage slots

The EVM works with 32 byte words. Variables less than 32 bytes can be declared next to each other in storage and this will pack the values together into a single 32 byte storage slot (if the values combined are <= 32 bytes). If the variables packed together are retrieved together in functions, we will effectively save ~2000 gas with every subsequent SLOAD for that storage slot. This is due to us incurring a `Gwarmaccess (100 gas)` versus a `Gcoldsload (2100 gas)`.

There are 6 instances of this issue.

[G-03] Structs can be packed into fewer storage slots

The EVM works with 32 byte words. Variables less than 32 bytes can be declared next to each other in storage and this will pack the values together into a single 32 byte storage slot (if values combined are <= 32 bytes). If the variables packed together are retrieved together in functions (more likely with structs), we will effectively save ~2000 gas with every subsequent SLOAD for that storage slot. This is due to us incurring a `Gwarmaccess (100 gas)` versus a `Gcoldsload (2100 gas)`.

There are 3 instances of this issue.

[G-04] Refactor `external` / `internal` function to avoid unnecessary External Calls

The functions below perform external calls that are previously performed in the functions that invoke them. We can refactor the `external` / `internal` functions to pass the cached external calls into them and avoid the extra external calls that would otherwise take place in the `internal` functions.

There are 4 instances of this issue.

[G-05] Refactor `external` / `internal` function to avoid unnecessary SLOAD

The functions below read storage slots that are previously read in the functions that invoke them. We can refactor the `external` / `internal` functions to pass cached storage variables as stack variables and avoid the extra storage reads that would otherwise take place in the `internal` functions.

There are 8 instances of this issue.

[G-06] Cache state variables outside of loop to avoid reading/writing storage on every iteration

Reading from storage should always try to be avoided within loops. In the following instances, we are able to cache state variables outside of the loop to save a `Gwarmaccess (100 gas)` per loop iteration.

There are 4 instances of this issue.

[G-07] Cache external calls outside of loop to avoid re-calling function on each iteration

Performing `STATICCALL` s that do not depend on variables incremented in loops should always try to be avoided within the loop. In the following instances, we are able to cache the external calls outside of the loop to save a `STATICCALL (100 gas)` per loop iteration.

Total Instances: 2

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/UserWithdrawalManager.sol#L132-L139

Cache `staderConfig.getMinBlockDelayToFinalizeWithdrawRequest()` outside of loop to save 1 `STATICCALL` per loop iteration

Note: This will also save 1 SLOAD (`staderConfig`) per loop iteration. In addition, this function is not benchmarked in the protocol's tests and therefore the exact gas savings are not included here.

```
File: contracts/UserWithdrawalManager.sol
132:         for (requestId = nextRequestIdToFinalize; requestId
133:             UserWithdrawInfo memory userWithdrawInfo = userW
134:             uint256 requiredEth = userWithdrawInfo.ethExpect
135:             uint256 lockedEthX = userWithdrawInfo.ethXAmount
136:             uint256 minEThRequiredToFinalizeRequest = Math.m
137:             if (
138:                 (ethToSendToFinalizeRequest + minEThRequirec
139:                 (userWithdrawInfo.requestBlock + staderConfi
```

```
diff --git a/contracts/UserWithdrawalManager.sol b/contracts/Use
index f563434..204239f 100644
--- a/contracts/UserWithdrawalManager.sol
+++ b/contracts/UserWithdrawalManager.sol
@@ -128,6 +128,7 @@ contract UserWithdrawalManager is
         uint256 lockedEthXToBurn;
         uint256 ethToSendToFinalizeRequest;
         uint256 requestId;
+        uint256 minBlockDelay = staderConfig.getMinBlockDelayTo
         uint256 pooledETH = poolManager.balance;
         for (requestId = nextRequestIdToFinalize; requestId <=
             UserWithdrawInfo memory userWithdrawInfo = userWith
@@ -136,7 +137,7 @@ contract UserWithdrawalManager is
             uint256 minEThRequiredToFinalizeRequest = Math.min(
             if (
                 (ethToSendToFinalizeRequest + minEThRequiredToF
-                (userWithdrawInfo.requestBlock + staderConfig.g
+                (userWithdrawInfo.requestBlock + minBlockDelay
                     block.number)
```

```
                ) {
                    break;
```

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/PermissionlessPool.sol#L94-L109

Cache return values from `INodeRegistry((staderConfig).getPermissionlessNodeRegistry()).POOL_ID()` and `staderConfig.getETHDepositContract()` outisde of loop to save 3 `STATICCALL (100 GAS)` per iteration

Note: Caching the `staderConfig.getPreDepositSize()` calls are not included since it was included in the automated report. In addition, this function is not benchmarked in the protocol's tests and therefore the exact gas savings are not included here.

```
File: contracts/PermissionlessPool.sol
94:         for (uint256 i; i < pubkeyCount; ) {
95:             address withdrawVault = IVaultFactory(vaultFactor
96:                 INodeRegistry((staderConfig).getPermissionles
97:                 _operatorId,
98:                 _operatorTotalKeys + i
99:             );
100:             bytes memory withdrawCredential = IVaultFactory(
101:
102:             bytes32 depositDataRoot = this.computeDepositDat
103:                 _pubkey[i],
104:                 _preDepositSignature[i],
105:                 withdrawCredential,
106:                 staderConfig.getPreDepositSize()
107:             );
108:             //slither-disable-next-line arbitrary-send-eth
109:             IDepositContract(staderConfig.getETHDepositContr
```

```
diff --git a/contracts/PermissionlessPool.sol b/contracts/Permis
index 046761d..ef0e3bf 100644
--- a/contracts/PermissionlessPool.sol
+++ b/contracts/PermissionlessPool.sol
@@ -83,17 +83,19 @@ contract PermissionlessPool is IStaderPoolBa
      * @param _operatorTotalKeys total keys of operator at the
```

```diff
         */
        function preDepositOnBeaconChain(
-           bytes[] calldata _pubkey,
-           bytes[] calldata _preDepositSignature,
+           bytes[] memory _pubkey,
+           bytes[] memory _preDepositSignature,
            uint256 _operatorId,
            uint256 _operatorTotalKeys
        ) external payable nonReentrant {
            UtilLib.onlyStaderContract(msg.sender, staderConfig, st
            address vaultFactory = staderConfig.getVaultFactory();
            uint256 pubkeyCount = _pubkey.length;
+           uint8 poolId = INodeRegistry((staderConfig).getPermissi
+           address ethDepositContract = staderConfig.getETHDeposit
            for (uint256 i; i < pubkeyCount; ) {
                address withdrawVault = IVaultFactory(vaultFactory)
-                   INodeRegistry((staderConfig).getPermissionlessN
+                   poolId,
                    _operatorId,
                    _operatorTotalKeys + i
                );
@@ -106,7 +108,7 @@ contract PermissionlessPool is IStaderPoolBa
                    staderConfig.getPreDepositSize()
                );
                //slither-disable-next-line arbitrary-send-eth
-               IDepositContract(staderConfig.getETHDepositContract
+               IDepositContract(ethDepositContract).deposit{value:
                    _pubkey[i],
                    withdrawCredential,
                    _preDepositSignature[i],
```

[G-08] Refactor code to avoid unnecessary SLOAD

Line 49 is essentially setting `balances[operator]` to 0. Instead of performing an extra SLOAD in the expression, we can simply set the mapping to 0.

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/OperatorRewardsCollector.sol#L46-L49

Gas Savings for `OperatorRewardsCollector.claim` *, obtained via protocol's tests: Avg 67 gas*

	Med	Max	Avg	# calls
Before	35757	35757	35757	3
After	35690	35690	35690	3

```
File: contracts/OperatorRewardsCollector.sol
46:     function claim() external whenNotPaused {
47:         address operator = msg.sender;
48:         uint256 amount = balances[operator]; // @audit: 1st s
49:         balances[operator] -= amount; // @audit: 2nd sload
```

```diff
diff --git a/contracts/OperatorRewardsCollector.sol b/contracts/
index e80db8a..d284367 100644
--- a/contracts/OperatorRewardsCollector.sol
+++ b/contracts/OperatorRewardsCollector.sol
@@ -46,7 +46,7 @@ contract OperatorRewardsCollector is
     function claim() external whenNotPaused {
         address operator = msg.sender;
         uint256 amount = balances[operator];
-        balances[operator] -= amount;
+        balances[operator] = 0;

         address operatorRewardsAddr = UtilLib.getOperatorRewarc
         UtilLib.sendValue(operatorRewardsAddr, amount);
```

[G-09] Utilize return value from internal function to avoid unnecessary SLOAD

The return value from the `onlyActiveOperator` internal function is `operatorIDByAddress[_operAddr]`. In the instances below, the return value is ignored and `operatorIDByAddress[msg.sender]` is being redundantly read from storage. Utilize the return value to save 1 SLOAD.

Total Instances: `2`

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/PermissionedNodeRegistry.sol#L401-L405 https://github.com/code-423n4/2023-06-stader/blob/main/contracts/PermissionedNodeRegistry.sol#L720-L721

Cache return value from `onlyActiveOperator` inside of `operatorId` to save 1 SLOAD

```
File: contracts/PermissionedNodeRegistry.sol
401:    function updateOperatorDetails(string calldata _operator
402:        IPoolUtils(staderConfig.getPoolUtils()).onlyValidNam
403:        UtilLib.checkNonZeroAddress(_rewardAddress);
404:        onlyActiveOperator(msg.sender); // @audit: 1st sloac
405:        uint256 operatorId = operatorIDByAddress[msg.sender]

720:    function onlyActiveOperator(address _operAddr) internal
721:        _operatorId = operatorIDByAddress[_operAddr];
```

```
diff --git a/contracts/PermissionedNodeRegistry.sol b/contracts/
index ca8aa81..80cfa7c 100644
--- a/contracts/PermissionedNodeRegistry.sol
+++ b/contracts/PermissionedNodeRegistry.sol
@@ -401,8 +401,7 @@ contract PermissionedNodeRegistry is
     function updateOperatorDetails(string calldata _operatorNam
         IPoolUtils(staderConfig.getPoolUtils()).onlyValidName(_
         UtilLib.checkNonZeroAddress(_rewardAddress);
-        onlyActiveOperator(msg.sender);
-        uint256 operatorId = operatorIDByAddress[msg.sender];
+        uint256 operatorId = onlyActiveOperator(msg.sender);
         operatorStructById[operatorId].operatorName = _operator
         operatorStructById[operatorId].operatorRewardAddress =
         emit UpdatedOperatorDetails(msg.sender, _operatorName,
```

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/PermissionlessNodeRegistry.sol#L366-L370

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/PermissionlessNodeRegistry.sol#L680-L681

Cache return value from `onlyActiveOperator` inside of `operatorId` to save 1 SLOAD

```
File: contracts/PermissionlessNodeRegistry.sol
366:    function updateOperatorDetails(string calldata _operato
367:        IPoolUtils(staderConfig.getPoolUtils()).onlyValidNa
368:        UtilLib.checkNonZeroAddress(_rewardAddress);
369:        onlyActiveOperator(msg.sender); // @audit: 1st sloa
370:        uint256 operatorId = operatorIDByAddress[msg.sender]

680:    function onlyActiveOperator(address _operAddr) internal
681:        _operatorId = operatorIDByAddress[_operAddr];
```

```diff
diff --git a/contracts/PermissionlessNodeRegistry.sol b/contrac
index 9640556..5c06f22 100644
--- a/contracts/PermissionlessNodeRegistry.sol
+++ b/contracts/PermissionlessNodeRegistry.sol
@@ -366,8 +366,7 @@ contract PermissionlessNodeRegistry is
     function updateOperatorDetails(string calldata _operatorNa
         IPoolUtils(staderConfig.getPoolUtils()).onlyValidName(_
         UtilLib.checkNonZeroAddress(_rewardAddress);
-        onlyActiveOperator(msg.sender);
-        uint256 operatorId = operatorIDByAddress[msg.sender];
+        uint256 operatorId = onlyActiveOperator(msg.sender);
         operatorStructById[operatorId].operatorName = _operato
         operatorStructById[operatorId].operatorRewardAddress =
         emit UpdatedOperatorDetails(msg.sender, _operatorName,
```

[G-10] Multiple address mappings can be combined into a single mapping of an address to a struct, where appropriate

We can combine multiple mappings below into structs. We can then pack the structs by modifying the `uint type` for the values. This will result in cheaper storage reads since multiple mappings are accessed in functions and those values are now occupying the same storage slot, meaning the slot will become warm after the first SLOAD. In addition, when writing to and reading from the struct values we will avoid a `Gsset (20000 gas)` and `Gcoldsload (2100 gas)`, since multiple struct values are now occupying the same slot.

Note: This instance was missed by the automated report.

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/SocializingPool.sol#L30-L32

Save at least two `Gsset (20000 gas)` when the `handleRewards` function is called (`handledRewards` , `poolId` , and `reportingBlockNumber` are all set in storage, but since they are contained in a single slot only one `Gsset (20000)` is paid).

In order for this optimization to result in impactful gas savings, we will need to pack the `poolId` *and* `reportingBlockNumber` *in the* `RewardsData` *struct (we can safely reduce the* `uint` *type for* `reportingBlockNumber` *to* `uint48` *). We can then also pack the* `handledRewards` *variable with* `poolId` *and* `reportingBlockNumber` *by declaring it after the* `RewardsData` *struct in our newly created* `RewardsInfo` *struct.*

```
File: contracts/SocializingPool.sol
30:     mapping(uint256 => bool) public handledRewards;
31:     RewardsData public lastReportedRewardsData;
32:     mapping(uint256 => RewardsData) public rewardsDataMap;
```

```
diff --git a/contracts/SocializingPool.sol b/contracts/Socializi
index 19d8cb2..90b68de 100644
--- a/contracts/SocializingPool.sol
+++ b/contracts/SocializingPool.sol
@@ -26,10 +26,15 @@ contract SocializingPool is
     uint256 public override totalOperatorSDRewardsRemaining;
     uint256 public override initialBlock;

+    struct RewardsInfo {
+        RewardsData rewardsDataMap;
+        bool handledRewards;
+    }
+
+    mapping(uint256 => RewardsInfo) rewardsInfo;
+
     mapping(address => mapping(uint256 => bool)) public overric
-    mapping(uint256 => bool) public handledRewards;
     RewardsData public lastReportedRewardsData;
-    mapping(uint256 => RewardsData) public rewardsDataMap;
```

```diff
     /// @custom:oz-upgrades-unsafe-allow constructor
     constructor() {
@@ -60,8 +65,9 @@ contract SocializingPool is

     function handleRewards(RewardsData calldata _rewardsData) e
         UtilLib.onlyStaderContract(msg.sender, staderConfig, st
-
-        if (handledRewards[_rewardsData.index]) {
+
+        RewardsInfo storage _rewardsInfo = rewardsInfo[_rewards
+        if (_rewardsInfo.handledRewards) {
             revert RewardAlreadyHandled();
         }
         if (
@@ -77,12 +83,12 @@ contract SocializingPool is
             revert InsufficientSDRewards();
         }

-        handledRewards[_rewardsData.index] = true;
+        _rewardsInfo.handledRewards = true;
         totalOperatorETHRewardsRemaining += _rewardsData.operat
         totalOperatorSDRewardsRemaining += _rewardsData.operatc

         lastReportedRewardsData = _rewardsData;
-        rewardsDataMap[_rewardsData.index] = _rewardsData;
+        _rewardsInfo.rewardsDataMap = _rewardsData;

         IStaderStakePoolManager(staderConfig.getStakePoolManage
             value: _rewardsData.userETHRewards
@@ -170,7 +176,7 @@ contract SocializingPool is
         if (_index == 0 || _index > lastReportedRewardsData.inc
             revert InvalidCycleIndex();
         }
-        bytes32 merkleRoot = rewardsDataMap[_index].merkleRoot;
+        bytes32 merkleRoot = rewardsInfo[_index].rewardsDataMap
         bytes32 node = keccak256(abi.encodePacked(_operator, _a
         return MerkleProofUpgradeable.verify(_merkleProof, merk
     }
@@ -214,15 +220,16 @@ contract SocializingPool is
         }

         // for past cycles
-        _startBlock = rewardsDataMap[_index - 1].reportingBlock
-        _endBlock = rewardsDataMap[_index].reportingBlockNumber
+        _startBlock = rewardsInfo[_index - 1].rewardsDataMap.re
+        _endBlock = rewardsInfo[_index].rewardsDataMap.reportir
```

```
         // for current cycle
-        if (rewardsDataMap[_index].reportingBlockNumber == 0) {
-            if (rewardsDataMap[_index - 1].reportingBlockNumber
+        if (rewardsInfo[_index].rewardsDataMap.reportingBlockNu
+            if (rewardsInfo[_index - 1].rewardsDataMap.reportin
                 revert FutureCycleIndex();
             }
-            _endBlock = rewardsDataMap[_index - 1].reportingBlo
+            _endBlock = rewardsInfo[_index - 1].rewardsDataMap.
         }
     }
 }



diff --git a/contracts/interfaces/ISocializingPool.sol b/contrac
index aceee8b..ab49093 100644
--- a/contracts/interfaces/ISocializingPool.sol
+++ b/contracts/interfaces/ISocializingPool.sol
@@ -7,14 +7,10 @@ import './IStaderConfig.sol';
 /// @title RewardsData
 /// @notice This struct holds rewards merkleRoot and rewards sp
 struct RewardsData {
-    /// @notice The block number when the rewards data was last
-    uint256 reportingBlockNumber;
     /// @notice The index of merkle tree or rewards cycle
     uint256 index;
     /// @notice The merkle root hash
     bytes32 merkleRoot;
-    /// @notice pool id of operators
-    uint8 poolId;
     /// @notice operator ETH rewards for index cycle
     uint256 operatorETHRewards;
     /// @notice user ETH rewards for index cycle
@@ -23,6 +19,10 @@ struct RewardsData {
     uint256 protocolETHRewards;
     /// @notice operator SD rewards for index cycle
     uint256 operatorSDRewards;
+    /// @notice pool id of operators
+    uint8 poolId;
+    /// @notice The block number when the rewards data was last
+    uint48 reportingBlockNumber;
 }

 interface ISocializingPool {
```

[G-11] Using storage instead of memory for structs/arrays saves gas

The function in the instance below only reads 3 out of 5 struct fields. It is performing two unnecessary `Gcoldsload (2100 gas)` . We can change `memory` to `storage` in order to save those two SLOADs. *Note: we will also need to cache the* `owner` *struct field since it is accessed multiple times.*

Note: This is an instance missed by the automated report.

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/UserWithdrawalManager.sol#L169

```
File: contracts/UserWithdrawalManager.sol
169:         UserWithdrawInfo memory userRequest = userWithdrawRe
```

```
diff --git a/contracts/UserWithdrawalManager.sol b/contracts/Use
index f563434..a12919f 100644
--- a/contracts/UserWithdrawalManager.sol
+++ b/contracts/UserWithdrawalManager.sol
@@ -166,8 +166,9 @@ contract UserWithdrawalManager is
         if (_requestId >= nextRequestIdToFinalize) {
             revert requestIdNotFinalized(_requestId);
         }
-        UserWithdrawInfo memory userRequest = userWithdrawReque
-        if (msg.sender != userRequest.owner) {
+        UserWithdrawInfo storage userRequest = userWithdrawRequ
+        address payable _owner = userRequest.owner;
+        if (msg.sender != _owner) {
             revert CallerNotAuthorizedToRedeem();
         }
         // below is a default entry as no userRequest will be f
@@ -175,9 +176,9 @@ contract UserWithdrawalManager is
             revert RequestAlreadyRedeemed(_requestId);
         }
         uint256 etherToTransfer = userRequest.ethFinalized;
-        deleteRequestId(_requestId, userRequest.owner);
-        sendValue(userRequest.owner, etherToTransfer);
-        emit RequestRedeemed(msg.sender, userRequest.owner, eth
+        deleteRequestId(_requestId, _owner);
+        sendValue(_owner, etherToTransfer);
+        emit RequestRedeemed(msg.sender, _owner, etherToTransfe
     }
```

[G-12] Using bools for storage incurs overhead

Both state variables can potentially be set back and forth from `true` and `false`. This would result in a `Gsset (20000 gas)` everytime the values are set to `true` from `false`. We can instead use `uint(1)` in place of `true` and `uint(2)` in place of `false` and pay the `Gsset (20000 gas)` once during deployment (to set the slot to `uint(1)`. This would save two `Gsset (20000 gas)`. However, a more efficient mitigation would be to pack the variables into a slot with other variables that would inevitably be written to. Please see this finding for a more efficient solution.

Note: This is an instance missed by the automated report.

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/StaderOracle.sol#L18-L19

```
File: contracts/StaderOracle.sol
18:     bool public override erInspectionMode;
19:     bool public override isPORFeedBasedERData;
```

[G-13] Multiple accesses of a mapping/array should use a local variable cache

Caching a mapping's value in a storage pointer when the value is accessed multiple times saves ~40 gas per access due to not having to perform the same offset calculation every time. Help the Optimizer by saving a storage variable's reference instead of repeatedly fetching it.

To achieve this, declare a storage pointer for the variable and use it instead of repeatedly fetching the reference in a map or an array. As an example, instead of repeatedly calling `stakes[tokenId_]`, save its reference via a storage pointer: `StakeInfo storage stakeInfo = stakes[tokenId_]` and use the pointer instead.

There are 4 instances of this issue.

[G-14] Combine events to save 2 `Glogtopic (375 gas)`

The events below are only emitted once in the `handleRewards` function. We can combine the events into one singular event to save two `Glogtopic (375 gas)` that would otherwise be paid for the additional two events.

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/SocializingPool.sol#L96-L104

```
File: contracts/SocializingPool.sol
96:         emit OperatorRewardsUpdated(
97:             _rewardsData.operatorETHRewards,
98:             totalOperatorETHRewardsRemaining,
99:             _rewardsData.operatorSDRewards,
100:             totalOperatorSDRewardsRemaining
101:         );
102:
103:         emit UserETHRewardsTransferred(_rewardsData.userETHR
104:         emit ProtocolETHRewardsTransferred(_rewardsData.prot
```

[G-15] Call environment variables directly instead of caching them

In the instance below, instead of caching `msg.sender` and incurring unnecessary stack manipulation, we can call `msg.sender` directly. `msg.sender` costs 2 Gas while the extra stack manipulation will cost 3 Gas per `DUP`. *Note: that the cached variable is used multiple times within a loop (see `_claim`).*

Note: This function is not benchmarked in the protocol's tests and therefore exact gas savings is not included for this instance.

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/SocializingPool.sol#L107-L116

https://github.com/code-423n4/2023-06-stader/blob/main/contracts/SocializingPool.sol#L137-L161 https://github.com/code-423n4/2023-06-stader/blob/main/contracts/SocializingPool.sol#L163-L174

```
File: contracts/SocializingPool.sol
106:    function claim(
107:        uint256[] calldata _index,
108:        uint256[] calldata _amountSD,
109:        uint256[] calldata _amountETH,
110:        bytes32[][] calldata _merkleProof
111:    ) external override nonReentrant whenNotPaused {
112:        address operator = msg.sender;
113:        (uint256 totalAmountSD, uint256 totalAmountETH) = _c
114:
115:        address operatorRewardsAddr = UtilLib.getOperatorRew

137:    function _claim(
138:        uint256[] calldata _index,
139:        address _operator, // @audit: cached msg.sender
140:        uint256[] calldata _amountSD,
141:        uint256[] calldata _amountETH,
142:        bytes32[][] calldata _merkleProof
143:    ) internal returns (uint256 _totalAmountSD, uint256 _tot
144:        uint256 indexLength = _index.length;
145:        for (uint256 i = 0; i < indexLength; i++) {
146:            if (_amountSD[i] == 0 && _amountETH[i] == 0) {
147:                revert InvalidAmount();
148:            }
149:            if (claimedRewards[_operator][_index[i]]) {
150:                revert RewardAlreadyClaimed(_operator, _inde
151:            }
152:
153:            _totalAmountSD += _amountSD[i];
154:            _totalAmountETH += _amountETH[i];
155:            claimedRewards[_operator][_index[i]] = true;
156:
157:            if (!verifyProof(_index[i], _operator, _amountSI
158:                revert InvalidProof(_index[i], _operator);
159:            }
160:        }
161:    }

163:    function verifyProof(
164:        uint256 _index,
165:        address _operator, // @audit: cached msg.sender
166:        uint256 _amountSD,
        uint256 _amountETH,
        bytes32[] calldata _merkleProof
    ) public view returns (bool) {
        if (_index == 0 || _index > lastReportedRewardsData.
            revert InvalidCycleIndex();
        }
        bytes32 merkleRoot = rewardsDataMap[_index].merkleRo
        bytes32 node = keccak256(abi.encodePacked(_operator,
```

```
diff --git a/contracts/SocializingPool.sol b/contracts/Socializi
index 19d8cb2..c504d2c 100644
--- a/contracts/SocializingPool.sol
+++ b/contracts/SocializingPool.sol
@@ -110,10 +110,9 @@ contract SocializingPool is
         uint256[] calldata _amountETH,
         bytes32[][] calldata _merkleProof
     ) external override nonReentrant whenNotPaused {
-        address operator = msg.sender;
-        (uint256 totalAmountSD, uint256 totalAmountETH) = _clai
+        (uint256 totalAmountSD, uint256 totalAmountETH) = _clai

-        address operatorRewardsAddr = UtilLib.getOperatorRewarc
+        address operatorRewardsAddr = UtilLib.getOperatorRewarc

         bool success;
         if (totalAmountETH > 0) {
@@ -136,7 +135,6 @@ contract SocializingPool is

     function _claim(
         uint256[] calldata _index,
-        address _operator,
         uint256[] calldata _amountSD,
         uint256[] calldata _amountETH,
         bytes32[][] calldata _merkleProof
@@ -146,35 +144,44 @@ contract SocializingPool is
             if (_amountSD[i] == 0 && _amountETH[i] == 0) {
                 revert InvalidAmount();
             }
-            if (claimedRewards[_operator][_index[i]]) {
-                revert RewardAlreadyClaimed(_operator, _index[i
+            if (claimedRewards[msg.sender][_index[i]]) {
+                revert RewardAlreadyClaimed(msg.sender, _index[
             }
```

```diff
              _totalAmountSD += _amountSD[i];
              _totalAmountETH += _amountETH[i];
-             claimedRewards[_operator][_index[i]] = true;
+             claimedRewards[msg.sender][_index[i]] = true;

-             if (!verifyProof(_index[i], _operator, _amountSD[i]
-                 revert InvalidProof(_index[i], _operator);
+             if (!_verifyProof(_index[i], _amountSD[i], _amountE
+                 revert InvalidProof(_index[i], msg.sender);
              }
          }
      }

-     function verifyProof(
+     function _verifyProof(
          uint256 _index,
-         address _operator,
          uint256 _amountSD,
          uint256 _amountETH,
          bytes32[] calldata _merkleProof
-     ) public view returns (bool) {
+     ) internal view returns (bool) {
          if (_index == 0 || _index > lastReportedRewardsData.inc
              revert InvalidCycleIndex();
          }
          bytes32 merkleRoot = rewardsDataMap[_index].merkleRoot;
-         bytes32 node = keccak256(abi.encodePacked(_operator, _a
+         bytes32 node = keccak256(abi.encodePacked(msg.sender, _
          return MerkleProofUpgradeable.verify(_merkleProof, merk
      }

+     function verifyProof(
+         uint256 _index,
+         address _operator,
+         uint256 _amountSD,
+         uint256 _amountETH,
+         bytes32[] calldata _merkleProof
+     ) public view returns (bool) {
+         return _verifyProof(_index, _amountSD, _amountETH, _mer
+     }
+
      // SETTERS
      function updateStaderConfig(address _staderConfig) external
          UtilLib.checkNonZeroAddress(_staderConfig);
```

[G-16] Use assembly to perform efficient back-to-back calls

If similar external calls are performed back-to-back, we can use assembly to reuse any function signatures and function parameters that stay the same. In addition, we can also reuse the same memory space for each function call (`scratch space` + `free memory pointer`), which can potentially allow us to avoid memory expansion costs. In this case, we are also able to efficiently store the function signatures together in memory as one word, saving multiple `MLOAD` s in the process.

There are 3 instances of this issue.

`GasReport` output with all optimizations applied

Note: see GasReport *for more details.*

manoj9april (Stader) acknowledged

Picodes (judge) commented:

> Giving Best label over issue 388 for the readability and the gas diff.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top