



pStake Finance

Smart Contract Security Audit

Prepared by ShellBoxes

Aug 16th, 2023 - Aug 29th, 2023

[Shellboxes.com](https://shellboxes.com)

contact@shellboxes.com

Document Properties

Client	Persistence
Version	1.0
Classification	Public

Scope

Repository	Commit Hash
https://github.com/persistenceOne/pstake-stkETH	c558158203a185f4b0bc62740c920acca6c3c580

Re-Audit

Repository	Commit Hash
https://github.com/persistenceOne/pstake-stkETH	5ea6c32c05386b200693bf37c507666cb5f57f15

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1	Introduction	5
1.1	About Persistence	5
1.2	Approach & Methodology	5
1.2.1	Risk Methodology	6
2	Findings Overview	7
2.1	Disclaimer	7
2.2	Summary	7
2.3	Key Findings	7
3	Finding Details	10
SHB.1	Multiple Candidate Votes Accepted for the Same Epoch	10
SHB.2	Replay Attack on Accepted ConsensusData	12
SHB.3	Exited Balance of Validators and Staker Rewards Permanently Locked in the WithdrawalCredential Contract	14
SHB.4	Permanent Locking of Validator Rewards Due to Lack of depositedValidators Update	15
SHB.5	L2 Funds Cannot Be Bridged to L1 Due to Flawed Slippage Calculation	18
SHB.6	Stuck MEV Rewards in the WithdrawalCredential	21
SHB.7	Desynchronization Risk Due to Epoch-Based Data Submission	22
SHB.8	Premature Reward Allocation Due to Ignoring Queue Wait Time	24
SHB.9	Loss of User-Supplied Fees when Interacting with Optimism Messenger	26
SHB.10	Improper Handling of Exiting Validators Allowing Last-Time Reward Claims	31
SHB.11	Desynchronization of pricePerShare Between L1 and L2	35
SHB.12	Inequitable Reward Distribution for New Validators	37
SHB.13	Incorrect Condition Prevents Governor from Updating Commission Fees	40
SHB.14	First Staker can Grief Others using an Inflation Attack	42
SHB.15	Innaccurate rewardDebt Calculation for nodeOperators Modifying Validator Count	46
SHB.16	Uninitialized socketRegistry Address Leading to Potential Loss of Funds	49
SHB.17	Lack of Blacklist Mechanism for Malicious Node Operators	51
SHB.18	Owner Can Set Critical Values to Zero	53
SHB.19	Oracle Members Can Vote on Multiple ConsensusData Inputs	54

SHB.20	Need for Whitelisting Trusted Relayers in MEV Boost	55
SHB.21	Requirement for Node Operators to Set Fee Recipient to Protocol-Managed Address	56
SHB.22	Missing Socket API Payload Check	57
SHB.23	WITHDRAWAL_CREDENTIAL_BYTES32 Setter Desynchronizes Old Validators	59
SHB.24	Governor Has Full Control Over Oracle Quorum	60
SHB.25	Minimum Stake Amount Bypass	61
SHB.26	Inability to Update stkETH Exchange Rate When All Rewards Are Slashed .	64
SHB.27	Uninitialized optimismReceiver and arbitrumReceiver Can Lead to DoS . .	66
SHB.28	Hard-coded Slippage Causes DoS	68
SHB.29	Block Number Difference Between Chains results in Desynchronized Events	69
4	Best Practices	71
BP.1	Remove Unused variables	71
BP.2	Remove Redundant Initializations with Default Type Values	72
BP.3	Remove Tautological Statements	72
BP.4	Unchanged Variables Should Be Declared as Constants	73
BP.5	Correct Misleading Comments	74
BP.6	Optimize For Loop Counter Increment	74
BP.7	Remove Unused Modifier	75
5	Tests	76
5.1	L1-contracts	76
5.2	L2-contracts	78
6	Conclusion	79
7	Scope Files	80
7.1	Audit	80
7.2	Re-Audit	81
8	Disclaimer	83

1 Introduction

Persistence engaged **ShellBoxes** to conduct a security assessment on the pStake Finance beginning on Aug 16th, 2023 and ending Aug 29th, 2023. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About Persistence

pSTAKE is a liquid staking protocol unlocking the liquidity of staked assets. Stakers of PoS tokens can now stake their assets while maintaining the liquidity of these assets. On staking with pSTAKE, users earn staking rewards and also receive staked representative tokens (stkASSETS) which can be used in DeFi to generate additional yield (yield on top of staking rewards).

Issuer	Persistence
Website	https://pstake.finance/
Type	Solidity Smart Contract
Documentation	pSTAKE for Ethereum (stkETH) on Layer 2s
Audit Method	Whitebox

1.2 Approach & Methodology

ShellBoxes used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and

implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by ShellBoxes are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

Impact	High	Likelihood		
		High	Medium	Low
	Medium	High	Medium	Low
	Low	Medium	Low	Low

2 Findings Overview

2.1 Disclaimer

Please note that our review and subsequent findings related to the smart contracts do not cover the Socket Bridge Aggregator. The functionality, security, and integrity of the Socket Bridge Aggregator are outside the scope of this audit. The implementation of the bridging solution can have a significant impact on the security of the protocol.

Furthermore, within the smart contract system, there exists a role designated as the **Governor**. This role possesses significant permissions, including the ability to influence the oracle that submits consensus data on-chain. For the purposes of this audit, we have treated the governor role as a trusted entity. However, users and stakeholders should be aware of the extensive capabilities and influence this role holds within the system.

2.2 Summary

The following is a synopsis of our conclusions from our analysis of the pStake Finance implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.3 Key Findings

While the smart contracts exhibit a structured approach, our review identified several areas of concern that need to be addressed to ensure the robustness and security of the system. The issues include **6** critical-severity, **7** high-severity, **10** medium-severity, **5** low-severity, **1** informational-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Multiple Candidate Votes Accepted for the Same Epoch	CRITICAL	Fixed

SHB.2. Replay Attack on Accepted ConsensusData	CRITICAL	Fixed
SHB.3. Exited Balance of Validators and Staker Rewards Permanently Locked in the WithdrawalCredential Contract	CRITICAL	Acknowledged
SHB.4. Permanent Locking of Validator Rewards Due to Lack of depositedValidators Update	CRITICAL	Fixed
SHB.5. L2 Funds Cannot Be Bridged to L1 Due to Flawed Slippage Calculation	CRITICAL	Fixed
SHB.6. Stuck MEV Rewards in the WithdrawalCredential	CRITICAL	Acknowledged
SHB.7. Desynchronization Risk Due to Epoch-Based Data Submission	HIGH	Acknowledged
SHB.8. Premature Reward Allocation Due to Ignoring Queue Wait Time	HIGH	Acknowledged
SHB.9. Loss of User-Supplied Fees when Interacting with Optimism Messenger	HIGH	Fixed
SHB.10. Improper Handling of Exiting Validators Allowing Last-Time Reward Claims	HIGH	Fixed
SHB.11. Desynchronization of pricePerShare Between L1 and L2	HIGH	Acknowledged
SHB.12. Inequitable Reward Distribution for New Validators	HIGH	Acknowledged
SHB.13. Incorrect Condition Prevents Governor from Updating Commission Fees	HIGH	Fixed
SHB.14. First Staker can Grief Others using an Inflation Attack	MEDIUM	Fixed
SHB.15. Innacurate rewardDebt Calculation for node-Operators Modifying Validator Count	MEDIUM	Fixed

SHB.16. Uninitialized <code>socketRegistry</code> Address Leading to Potential Loss of Funds	MEDIUM	Fixed
SHB.17. Lack of Blacklist Mechanism for Malicious Node Operators	MEDIUM	Acknowledged
SHB.18. Owner Can Set Critical Values to Zero	MEDIUM	Fixed
SHB.19. Oracle Members Can Vote on Multiple <code>ConsensusData</code> Inputs	MEDIUM	Acknowledged
SHB.20. Need for Whitelisting Trusted Relayers in MEV Boost	MEDIUM	Acknowledged
SHB.21. Requirement for Node Operators to Set Fee Recipient to Protocol-Managed Address	MEDIUM	Acknowledged
SHB.22. Missing Socket API Payload Check	MEDIUM	Acknowledged
SHB.23. <code>WITHDRAWAL_CREDENTIAL_BYTES32</code> Setter Desynchronizes Old Validators	MEDIUM	Acknowledged
SHB.24. Governor Has Full Control Over Oracle Quorum	LOW	Acknowledged
SHB.25. Minimum Stake Amount Bypass	LOW	Fixed
SHB.26. Inability to Update <code>stkETH</code> Exchange Rate When All Rewards Are Slashed	LOW	Fixed
SHB.27. Uninitialized <code>optimismReceiver</code> and <code>arbitrumReceiver</code> Can Lead to DoS	LOW	Fixed
SHB.28. Hard-coded Slippage Causes DoS	LOW	Acknowledged
SHB.29. Block Number Difference Between Chains results in Desynchronized Events	INFORMATIONAL	Acknowledged

3 Finding Details

SHB.1 Multiple Candidate Votes Accepted for the Same Epoch

- Severity: **CRITICAL**
- Likelihood: 3
- Status: Fixed
- Impact: 3

Description:

The `pushData` function in the contract allows the execution of different `ConsensusData` inputs for the same transaction epoch ($n \times 200$ epochs). The contract assumes the `ConsensusData` to be correct every time the number of votes is equal to or exceeds the `quorum`. This design flaw can lead to the acceptance of two or more different `ConsensusData` inputs for the same transaction epoch.

Exploit Scenario:

Consider a scenario where the quorum is initialized to 2 (as it is in the contract now) and there are 4 oracle members. If two members agree on a particular input and the other two members agree on a different input, both inputs can be executed (for 6 oracle members we will have 3 accepted inputs ...). This can lead to an incorrect state in the contract, as the contract would accept both inputs as valid even though they might be contradictory.

Files Affected:

SHB.1.1: Oracle.sol

```
251 function pushData(  
252     ConsensusData memory _consensusData  
253 ) external override whenNotPaused onlyOracle {  
254     if (beaconData.getNextTxEpoch(lastCompletedEpoch) != beaconData.  
        ↪ getCurrentEpoch()) {  
255         revert VotedEarly();
```

```

256     }
257     bytes32 candidateId = keccak256(abi.encode(_consensusData,
        ↪ beaconData.getCurrentEpoch()));
258     bytes32 voteId = keccak256(abi.encode(msg.sender, candidateId));
259     if (submittedVotes[voteId]) {
260         revert AlreadyVoted(msg.sender);
261     }
262     submittedVotes[voteId] = true;
263     uint256 candidateNewVotes = candidates[candidateId] + 1;
264     candidates[candidateId] = candidateNewVotes;
265     if (candidateNewVotes >= quorum) {

```

Recommendation:

- Implement a mechanism to ensure that only one candidate data is accepted for a given transaction epoch
- Use a percentage as a quorum instead of relying on static number of votes for accepting the input, the data input given by the oracle should only be accepted if it is voted on by the majority of the members (more than 50% as a minimum so we can only have one accepted data per epoch).

Updates

The team resolved the issue, by reverting with **VotedEarly** in the **pushData** whenever the current epoch was already voted on.

SHB.1.2: Oracle.sol

```

244 function pushData(
245     ConsensusData memory _consensusData
246 ) external override whenNotPaused onlyOracle {
247     if(!_executedConsensusData[keccak256(abi.encode(_consensusData))])
        ↪ revert DuplicateDataSubmitted();
248     // revert if voted for completed Epoch or if voted early
249     if (beaconData.getCurrentEpoch() == lastCompletedEpoch

```

```
250 beaconData.getNextTxEpoch(lastCompletedEpoch) != beaconData.  
    ↳ getCurrentEpoch()) revert VotedEarly();
```

SHB.2 Replay Attack on Accepted ConsensusData

- Severity: **CRITICAL**
- Likelihood: 3
- Status: Fixed
- Impact: 3

Description:

The `pushData` function in the contract accepts and processes `ConsensusData` if it receives votes greater than or equal to the "quorum". However, there is no mechanism in place to ensure that the same `ConsensusData` isn't processed multiple times. This oversight allows for a potential replay attack where the same `ConsensusData` can be submitted and accepted multiple times, leading to incorrect state updates.

Exploit Scenario:

An oracle submits a specific `ConsensusData` that garners more than the "quorum" votes, leading the contract to update its state based on this data. Another oracle, either maliciously or inadvertently, submits the same `ConsensusData` again. Since there's no check to prevent the same data from being processed multiple times, the contract will again update its state based on the same data, leading to incorrect or duplicated state changes in the same transaction epoch.

Files Affected:

SHB.2.1: Oracle.sol

```
251 function pushData(  
252     ConsensusData memory _consensusData  
253 ) external override whenNotPaused onlyOracle {
```

```

254     if (beaconData.getNextTxEpoch(lastCompletedEpoch) != beaconData.
        ↪ getEpoch()) {
255         revert VotedEarly();
256     }
257     bytes32 candidateId = keccak256(abi.encode(_consensusData,
        ↪ beaconData.getEpoch()));
258     bytes32 voteId = keccak256(abi.encode(msg.sender, candidateId));
259     if (submittedVotes[voteId]) {
260         revert AlreadyVoted(msg.sender);
261     }
262     submittedVotes[voteId] = true;
263     uint256 candidateNewVotes = candidates[candidateId] + 1;
264     candidates[candidateId] = candidateNewVotes;
265     if (candidateNewVotes >= quorum) {

```

Recommendation:

Before processing any **ConsensusData**, consider checking against the stored entries to ensure it has not been processed before on the same tx epoch.

Updates

The team resolved the issue by adding a mapping called **_executedConsensusData** to track the executed consensus data and prevent it from being replayed.

SHB.2.2: Oracle.sol

```

244 function pushData(
245     ConsensusData memory _consensusData
246 ) external override whenNotPaused onlyOracle {
247     if(_executedConsensusData[keccak256(abi.encode(_consensusData))])
        ↪ revert DuplicateDataSubmitted();

```

SHB.2.3: Oracle.sol

```

298 _executedConsensusData[keccak256(abi.encode(_consensusData))] = true;

```

SHB.3 Exited Balance of Validators and Staker Rewards Permanently Locked in the `WithdrawalCredential` Contract

- Severity: **CRITICAL**
- Likelihood: 3
- Status: Acknowledged
- Impact: 3

Description:

The `setRewardsSlashedAmount` function in the contract is designed to set rewards, slashed amounts, and exit balances. However, there's an oversight in the handling of the exited balance of validators. When a validator exits, their balance remains locked in the `WithdrawalCredential` contract and isn't transferred back to the `Issuer` contract. The same goes for the accumulated staker rewards, the contract only handles validators and treasury rewards.

Exploit Scenario:

Consider a scenario where multiple validators exit over time. Their combined exited balances accumulate in the `WithdrawalCredential` contract. This accumulated balance remains idle and isn't utilized to generate rewards or for any other productive purpose. Over time, this can lead to a significant amount of the users' funds being locked without any utility.

Files Affected:

SHB.3.1: `WithdrawalCredential.sol`

```
94 function setRewardsSlashedAmount(  
95     uint256 _rewards,  
96     uint256 _slashed_amount,  
97     uint256 exit_balance  
98 ) external override onlyOracle {
```

```

99     newRewards = _rewards;
100    totalRewards += _rewards;
101    totalSlashedAmount = _slashed_amount;
102    exitBalance += exit_balance;
103 }

```

Recommendation:

Modify the `setRewardsSlashedAmount` function to transfer the exited balance back to the `Issuer` contract upon a validator's exit.

Updates

The team acknowledged the issue, stating that proper fund movement will be implemented with withdrawal feature (unstaking) as the exit balance will majorly serve the purpose of filling the withdrawal requests or provide liquidity to different validator for continuous reward generation.

SHB.4 Permanent Locking of Validator Rewards Due to Lack of `depositedValidators` Update

- Severity: **CRITICAL**
- Likelihood : 3
- Status: Fixed
- Impact : 3

Description:

The `updateRewardPerValidator` function in the contract is designed to update the rewards for validators. However, there's a critical oversight related to the handling of exited validators. The contract fails to update the `depositedValidators` in the `Issuer` when a validator exits. As a result, the contract still considers exited validators when calculating rewards. Since exited validators cannot claim these rewards, an important portion of each accumulated reward becomes permanently locked in the contract.

Exploit Scenario:

Consider a situation where a significant number of validators exit over a period. Due to the lack of updates to `depositedValidators`, the contract continues to allocate rewards considering these exited validators. Over time, a substantial portion of the rewards becomes locked and unclaimable. As more validators exit, the percentage of lost rewards for each allocation increases, leading to a significant loss of funds over time.

Files Affected:

SHB.4.1: Oracle.sol

```
310 function validatorExited(ExitedValidator[] memory _validators) internal
    ↪ returns (uint256) {
311     bytes[] memory pub_key = new bytes[](_validators.length);
312     uint256 exitValidatorBalance = 0;
313     for (uint i; i < _validators.length; ) {
314         pub_key[i] = _validators[i].publicKey;
315         exitValidatorBalance += _validators[i].amount;
316         unchecked {
317             ++i;
318         }
319     }
320     IKeysManager(core().keysManager()).exitedValidator(pub_key);
321     return exitValidatorBalance;
322 }
```

SHB.4.2: KeysManager.sol

```
81 function exitedValidator(bytes[] memory publicKeys) external override {
82     require(msg.sender == core().oracle(), "KeysManager: Only oracle can
    ↪ activate");
83     for (uint256 i; i < publicKeys.length; ) {
84         Validator storage validator = _validators[publicKeys[i]];
85         require(
86             validator.state == State.DEPOSITED,
87             "KeysManager: Validator not in valid state"
```



```

88         );
89         // node operator active validator count decreases
90         nodeOperatorValidatorCount[validator.nodeOperator] -= 1;
91         validator.state = State.EXITED;
92         unchecked {
93             ++i;
94         }
95     }
96     emit ExitValidator(publicKeys);
97 }

```

SHB.4.3: StakingPool.sol

```

67 function updateRewardPerValidator(uint256 newReward) public override {
68     IERC20Upgradeable(address(stkEth)).safeTransferFrom(_msgSender(),
        ↪ address(this), newReward);
69     accRewardPerValidator += (newReward * 1e12) / IIssuer(core.issuer())
        ↪ .depositedValidators();
70 }

```

Recommendation:

Consider updating the `depositedValidators` count in the `Issuer` whenever a validator exits.

Updates

The team resolved the issue, by updating the `depositedValidators` count using the `validator-sExited` function from the `Issuer` contract.

SHB.4.4: Oracle.sol

```

305 function validatorExited(ExitedValidator[] memory _validators) internal
    ↪ returns (uint256) {
306     bytes[] memory pubKey = new bytes[](_validators.length);
307     uint256 exitValidatorBalance;
308     for (uint i; i < _validators.length; ) {
309         pubKey[i] = _validators[i].publicKey;

```

```

310         exitValidatorBalance += _validators[i].amount;
311         unchecked {
312             ++i;
313         }
314     }
315     IKeysManager(core().keysManager()).exitedValidator(pubKey);
316     IIssuer(core().issuer()).validatorsExited(_validators.length);
317     return exitValidatorBalance;
318 }

```

SHB.4.5: Oracle.sol

```

306 function validatorsExited(uint256 _numValidatorExited) external
    ↪ whenNotPaused {
307     if(msg.sender != core.oracle()) revert UnauthorizedCall(msg.sender);
308     depositedValidators -= _numValidatorExited;
309 }

```

SHB.5 L2 Funds Cannot Be Bridged to L1 Due to Flawed Slippage Calculation

- Severity: **CRITICAL**
- Likelihood: 3
- Status: Fixed
- Impact: 3

Description:

The smart contract `IssuerUpgradable` contains a function named `getDepositL2` that serves as the entry point for receiving ETH from L2 stakers. This function employs a slippage control mechanism designed to accommodate delays in the bridge process. However, there is a critical oversight in the calculation of the L2 exchange rate. The issue arises from the omission of a necessary adjustment for the multiplication by `1e18` in the `pricePerShare`, leading to incorrect slippage calculations resulting in reverted deposits, even for exchange rates

that are not stale. so basically meaning that the transaction to `getDepositL2` will revert and ETH will remain stuck in L2.

Exploit Scenario:

1. Initially, the exchange rate between stkETH and ETH is 1:1.

`pricePerShare = 1e18`

2. Given the 1:1 exchange rate, `msg.value / _stkEthMinted` will be approximately 1 due to the exchange rate.
3. The slippage check in `getDepositL2` involves:

SHB.5.1: IssuerUpgradable.sol

```
exchangeRate - exchangeRate / 100 > (msg.value / _stkEthMinted)
(msg.value / _stkEthMinted) > exchangeRate + exchangeRate / 100
```

4. The check does not account for the `pricePerShare` being inflated by a multiplication by `1e18`.
5. Consequently, the slippage check is erroneous and consistently reverts deposits, even when the exchange rate is not stale.

This results in an inability to bridge L2 ETH to L1, rendering the L2 ETH stuck.

Files Affected:

SHB.5.2: IssuerUpgradable.sol

```
261 function getDepositL2(
262     uint256 _stkEthMinted,
263     uint256 _sourceChainId
264 ) external payable onlySocketReceiver {
265     // accept 1% error in exchange rate due to delay in bridging
266     uint256 exchangeRate = core.stkEth().pricePerShare();
267     if (
268         exchangeRate - exchangeRate / 100 > (msg.value / _stkEthMinted)
269         (msg.value / _stkEthMinted) > exchangeRate + exchangeRate / 100
```

```
270     ) revert InvalidExchangeRateReceived();
```

SHB.5.3: StkEth.sol

```
102 function pricePerShare() public view override returns (uint256) {  
103     return IOracle(core().oracle()).pricePerShare();  
104 }
```

SHB.5.4: Oracle.sol

```
182 function changeCValue(int256 calculatedRewards) internal whenNotPaused {  
183     if (calculatedRewards > 0) {  
184         uint256 valEthShare = (valCommission * uint256(  
            ↪ calculatedRewards)) / BASIS_POINT;  
185         uint256 protocolEthShare = (pStakeCommission * uint256(  
            ↪ calculatedRewards)) /  
186             BASIS_POINT;  
187         IIssuer issuer = IIssuer(core().issuer());  
188         pricePerShare =  
189             ((withdrawals.getTotalRewards() +  
190                 issuer.ethStaked() -  
191                 withdrawals.getTotalSlashedAmount() -  
192                 valEthShare -  
193                 protocolEthShare) * 1e18) /  
194             issuer.stkEthMinted();  
195     }  
196 }
```

Recommendation:

Consider comparing the `exchangeRate` with `msg.value * 1e18 / _stkEthMinted`.

Updates

The team resolved the issue by adding a `1e18` multiplication to balance the ratio with the exchange rate.

SHB.5.5: IssuerUpgradable.sol

```
265 function getDepositL2(  
266     uint256 _stkEthMinted,  
267     uint256 _sourceChainId  
268 ) external payable onlySocketReceiver {  
269     // accept 1% error in exchange rate due to delay in bridging  
270     uint256 exchangeRate = core.stkEth().pricePerShare();  
271     if (  
272         exchangeRate - exchangeRate / 100 > (msg.value * 1e18 /  
            ↪ _stkEthMinted)  
273         (msg.value * 1e18 / _stkEthMinted) > exchangeRate + exchangeRate  
            ↪ / 100  
274     ) revert InvalidExchangeRateReceived();
```

SHB.6 Stuck MEV Rewards in the WithdrawalCredential

- Severity: **CRITICAL**
- Likelihood: 3
- Status: Acknowledged
- Impact: 3

Description:

The contract is designed to receive MEV rewards (when node operators run mev boost) in Ether. However, once the Ether is received and added to the [mevRewards](#) variable, there is no mechanism in place to withdraw or utilize these funds. This design flaw can result in a significant amount of Ether being permanently locked in the contract, rendering them inaccessible and unusable.

Files Affected:

SHB.6.1: WithdrawalCredential.sol

```
72 /// @dev This function is responsible for receiving eth MEV rewards
```

```
73 receive() external payable {  
74     emit MEVReceived(msg.value);  
75     mevRewards += msg.value;
```

Recommendation:

Implement a function that allows the withdrawal or reallocation of the MEV rewards. This function should have appropriate access controls to ensure only authorized entities can execute it.

Updates

The team acknowledged the issue, stating that the feature to withdraw MEV rewards will be implemented with `stkETH withdrawal(unstaking)`.

SHB.7 Desynchronization Risk Due to Epoch-Based Data Submission

- Severity: **HIGH**
- Likelihood : 2
- Status : Acknowledged
- Impact : 3

Description:

The `pushData` function in the contract is designed to accept `ConsensusData` from off-chain oracles based on a voting system. The data submission is restricted to every **200th** epoch. However, there's a potential desynchronization issue if oracle members do not consistently submit data every 200 epochs. This can lead to scenarios where different oracles submit data covering different epoch ranges, resulting in a lack of consensus even if the data from each oracle is correct.

Exploit Scenario:

Consider a scenario with four oracle members over a span of **400** epochs (quorum = 2):

- Oracle A and B submit ConsensusData for the first 200 epochs then another one for the other 200 epochs.
- Oracle C and D submit ConsensusData covering the entire 400 epochs.

In this situation, even though both oracles' groups might be providing accurate data, they won't reach a consensus due to the overlapping epoch ranges. If there are more oracle members, this desynchronization can lead to various issues, such as:

- Failure to reach consensus on correct values.
- Potential state corruption if a ConsensusData for a 200 epoch range is accepted, followed by another ConsensusData from a delayed oracle covering a larger epoch range (e.g., $n \times 200$ epochs), effectively replaying data from previous epochs.

Files Affected:

SHB.7.1: Oracle.sol

```

251 function pushData(
252     ConsensusData memory _consensusData
253 ) external override whenNotPaused onlyOracle {
254     if (beaconData.getNextTxEpoch(lastCompletedEpoch) != beaconData.
        ↪ getCurrentEpoch()) {
255         revert VotedEarly();
256     }

```

SHB.7.2: BeaconData.sol

```

20 function getNextTxEpoch(
21     Values storage beaconValues,
22     uint64 lastEpoch
23 ) internal view returns (uint64) {
24     if ((beaconValues.getCurrentEpoch() - lastEpoch) % beaconValues.
        ↪ epochsPerTimePeriod == 0) {
25         return beaconValues.getCurrentEpoch();
26     } else {
27         uint64 n = (beaconValues.getCurrentEpoch() - lastEpoch) /

```

```

28         beaconValues.epochsPerTimePeriod;
29         return lastEpoch + ((n + 1) * beaconValues.epochsPerTimePeriod);
30     }
31 }
32
33 function getCurrentEpoch(Values storage beaconValues) internal view
    ↪ returns (uint64) {
34     return
35         uint64(
36             (uint64(block.timestamp) - beaconValues.genesisTime) /
37             (beaconValues.slotsPerEpoch * beaconValues.secondsPerSlot)
38         );
39 }

```

Recommendation:

Consider adding more information in the [ConsensusData](#) about epoch range associated to it. This information should be taken into consideration to assure reaching consensus ,and to avoid replaying previously accounted data.

Updates

The team acknowledged the issue, stating that the team will be running the off chain oracle initially and making sure to send correct data. Also, they are planning to implement epoch range in consensus data to prevent desynchronisation risk in the next update with Withdrawal feature.

SHB.8 Premature Reward Allocation Due to Ignoring Queue Wait Time

- Severity: **HIGH**
- Likelihood: 3
- Status: Acknowledged
- Impact: 2

Description:

Ethereum's proof of stake (PoS) consensus mechanism uses enter and exit queues to manage validators waiting to begin staking or to unstake, ensuring the stability of the network. The network has a rate limit, known as churn, on how many validators can be processed per epoch. If the number of validators trying to enter or exit exceeds this limit, they are placed in the respective queue. However, the contract's `depositToEth2` function in the `Issuer` contract immediately accounts a validator as deposited after staking in the beacon chain, without considering the queue wait time. Note that the queue wait time changes over time (currently at 23.01 days), the queue times can be checked here: [Validator Queue](#).

Exploit Scenario:

A validator stakes and is instantly recognized as deposited by the `Issuer` contract (Eligible for protocol rewards). This premature recognition allows the validator to start earning rewards even before they begin attesting to and proposing blocks in the consensus layer (generating rewards for the protocol). As a result, validators can earn rewards without actively participating in the consensus process, undermining the incentive structure of the PoS mechanism.

Files Affected:

SHB.8.1: IssuerUpgradable.sol

```
280 function depositToEth2(bytes calldata publicKey) external whenNotPaused
    ↪ {
281     require(
282         address(this).balance >= VALIDATOR_DEPOSIT + VERIFICATION_DEPOSIT
    ↪ ,
283     "Issuer: Not enough ether deposited"
284 );
285 IKeysManager.Validator memory validator = IKeysManager(core.
    ↪ keysManager()).validators(
286     publicKey
287 );
288
```

```

289     withdrawalVerificationDeposit(validator.nodeOperator);
290
291     IKeysManager(core.keysManager()).depositValidator(publicKey);
292
293     depositedValidators = depositedValidators + 1;
294     DEPOSIT_CONTRACT.deposit{ value: VALIDATOR_DEPOSIT }(
295         publicKey,
296         abi.encodePacked(core.withdrawalCredential()),
297         validator.signature,
298         validator.deposit_root
299     );
300 }

```

Recommendation:

Implement checks to ensure that validators only start earning rewards after they begin attesting to and proposing blocks. This can be achieved by relying on the oracle to provide data that allows the contract to switch a validator from deposited to eligible to rewards after they start proposing and attesting to blocks.

Updates

The team acknowledged the issue, stating that they will be implementing the fix with the withdrawal feature by introducing additional info to [pushData](#) function through an offchain oracle, mark [Deposited](#) and update the required state changes with proper checks.

SHB.9 Loss of User-Supplied Fees when Interacting with Optimism Messenger

- Severity: **HIGH**
- Likelihood: 2
- Status: Fixed
- Impact: 3

Description:

The `Issuer` contract on Layer 1 contains functions such as `mintL2`, `transferToL2`, and `mintWethL2`, which are responsible for minting or transferring `stkETH` to Layer 2 (e.g., `Arbitrum` or `Optimism`). In the case of `Arbitrum`, the `_callValue` is used for retry-able L2 message, but a critical issue arises when interacting with `Optimism`. Specifically, when calling `mintstkETHL2` within `OptimismMessenger` or `changeCValueL2` within `Oracle`, the user-supplied value goes unused as the first 1.92 million gas on L2 OP is free. Therefore, the sent value becomes trapped in the contract without a method to retrieve it. This results in a loss of user-supplied fees when interacting with `OptimismMessenger`.

Files Affected:

SHB.9.1: IssuerUpgradable.sol

```
164 function mintL2(  
165     uint256 _messengerId,  
166     uint256 _callValue,  
167     address _receiverAddress,  
168     bytes memory _payload  
169 )  
170     external  
171     payable  
172     whenNotPaused  
173     minimumStakeAmount(msg.value)  
174     onlyExistingMessenger(_messengerId)  
175 {  
176     uint256 ethToStake = msg.value - _callValue;  
177     emit Stake(msg.sender, ethToStake, block.timestamp);  
178     uint256 stkEthToMint = (ethToStake * 1e18) / core.stkEth().  
        ↪ pricePerShare();  
179     stkEthMinted = stkEthMinted + stkEthToMint;  
180     ethStaked = ethStaked + ethToStake;  
181     IL1Messenger(messengers[_messengerId].messenger).mintstkETHL2(  
        ↪ value: _callValue )(  
182         _receiverAddress,
```

```

183         stkEthToMint,
184         _payload
185     );

```

SHB.9.2: IssuerUpgradable.sol

```

240 function transferToL2(
241     uint256 _messengerId,
242     uint256 _amount,
243     address _receiverAddress,
244     bytes memory _payload
245 ) external payable whenNotPaused onlyExistingMessenger(_messengerId)
    ⇨ {
246     uint256 amountTotal = core.stkEth().balanceOf(msg.sender);
247     if (amountTotal >= _amount) {
248         core.stkEth().burn(msg.sender, _amount);
249         IL1Messenger(messengers[_messengerId].messenger).mintstkETHL2
            ⇨ { value: msg.value }(
250             _receiverAddress,
251             _amount,
252             _payload
253         );

```

SHB.9.3: IssuerUpgradable.sol

```

207 function mintWethL2(
208     uint256 _messengerId,
209     uint256 _amount,
210     address _receiverAddress,
211     bytes memory _payload
212 )
213     external
214     payable
215     whenNotPaused
216     minimumStakeAmount(_amount)
217     onlyExistingMessenger(_messengerId)

```

```

218     {
219         // Transfer WETH from user to issuer
220         IERC20Upgradeable(WETH).safeTransferFrom(msg.sender, address(this)
            ↪ ), _amount);
221         // withdraw ETH by burying WETH token
222         IWETH(WETH).withdraw(_amount);
223         emit Stake(msg.sender, _amount, block.timestamp);
224         ethStaked += _amount;
225         uint256 stkEthToMint = (_amount * 1e18) / core.stkEth().
            ↪ pricePerShare();
226         stkEthMinted += stkEthToMint;
227         IL1Messenger(messengers[_messengerId].messenger).mintstkETHL2{
            ↪ value: msg.value }(
228             _receiverAddress,
229             stkEthToMint,
230             _payload
231         );

```

SHB.9.4: Oracle.sol

```

211 function changeCValueL2(
212     uint256 _messengerId,
213     bytes memory _payload
214 ) external payable whenNotPaused {
215     IIssuer issuer = IIssuer(core().issuer());
216     (bool messengerStatus, address messenger) = issuer.getMessenger(
        ↪ _messengerId);
217     if (!messengerStatus (messenger == address(0))) revert
        ↪ InvalidMessenger();
218     IL1Messenger(messenger).changeCValueL2{ value: msg.value }(
219         msg.sender,
220         pricePerShare,
221         _payload
222     );

```

SHB.9.5: OptimismMessenger.sol

```

38 function _sendMessage(bytes memory _message) internal {
39     optimismMessenger.sendMessage(optimismReceiver, _message, 12gas);
40 }

```

SHB.9.6: OptimismMessenger.sol

```

42 function changeCValueL2(
43     address,
44     uint256 cValue,
45     bytes memory
46 ) external payable override onlyOracle whenNotPaused {
47     bytes memory message = abi.encodeWithSelector(
48         IL2MessageContract.changeCValue.selector,
49         cValue
50     );
51     _sendMessage(message);
52     emit CValueChangedL2(block.number, cValue, destinationChainID);
53 }

```

SHB.9.7: OptimismMessenger.sol

```

55 function mintstkETHL2(
56     address user,
57     uint256 amount,
58     bytes memory
59 ) external payable override onlyIssuer whenNotPaused {
60     bytes memory message = abi.encodeWithSelector(
61         IL2MessageContract.mintstkETH.selector,
62         user,
63         amount
64     );
65     _sendMessage(message);
66     emit MintStkETHL2(msg.sender, user, amount, destinationChainID);
67 }

```

Recommendation:

Consider refunding the call value to the user when he chooses the [OptimismMessenger](#).

Updates

The team resolved the issue by adding a [call](#) to refund the user when interacting with the [OptimismMessenger](#).

SHB.9.8: OptimismMessenger.sol

```
38  if(msg.value > 0){
39      (bool success, ) = user.call{value: msg.value}("");
40      if(!success) revert RefundFailed();
41  }
```

SHB.9.9: OptimismMessenger.sol

```
72  if(msg.value > 0){
73      (bool success, ) = user.call{value: msg.value}("");
74      if(!success) revert RefundFailed();
75  }
```

SHB.10 Improper Handling of Exiting Validators Allowing Last-Time Reward Claims

- Severity: **HIGH**
- Likelihood: 3
- Status: Fixed
- Impact: 2

Description:

The function [exitedValidator](#) within the [KeysManager](#) contract successfully marks validators as exited when they cease staking, either voluntarily or due to slashing. This process includes a necessary decrement of the number of validators for a given [nodeOperator](#). However, the flaw here is that the function does not trigger the

`claimAndUpdateRewardDebt` function in the `StakingPool` contract. Consequently, a `nodeOperator` can call `claimAndUpdateRewardDebt` and still get rewards for the reported exited validator, enabling them to collect rewards meant for that validator. Essentially, this allows a last-minute reward claim by a `nodeOperator` after their validator has been marked as exited.

Exploit Scenario:

1. A validator under the control of a `nodeOperator` is reported as exited through the `exitValidator` function.
2. The function in `KeysManager` decrements the `nodeOperatorValidatorCount`, reflecting the exited validator.
3. Despite the validator being reported as exited, the `nodeOperator` identifies the absence of a call to `claimAndUpdateRewardDebt`.
4. The `nodeOperator` exploits this gap by calling `claimAndUpdateRewardDebt` for the exited validator, subsequently accumulating rewards originally designated for active validators.
5. This unauthorized accumulation of rewards results in an unfair distribution of rewards and undermines the integrity of the reward system.
6. In a scenario with 20 validators and 10000 wei in fees, each validator should receive 500 wei as their share of the fees.
7. Although `nodeOperator` A has no active validators, they can still claim 500 wei in rewards, essentially gaining rewards one last time for their exited validator.

Files Affected:

SHB.10.1: StakingPool.sol

```
74 function claimAndUpdateRewardDebt(address usr) external override {  
75     UserInfo storage user = userInfos[usr];  
76  
77     uint256 userValidators = IKeysManager(core.keysManager()).  
        ↪ nodeOperatorValidatorCount(usr);
```



```

78
79     uint256 pending = ((accRewardPerValidator * user.amount) / 1e12)
        ↪ - user.rewardDebt;
80
81     if (pending > 0) {
82         IERC20Upgradeable(address(stkEth)).safeTransfer(usr, pending)
            ↪ ;
83         emit RewardRedeemed(pending, usr);
84     }
85
86     user.rewardDebt = (accRewardPerValidator * userValidators) / 1e12
        ↪ ;
87     user.amount = userValidators;
88 }

```

SHB.10.2: KeysManager.sol

```

81 function exitedValidator(bytes[] memory publicKeys) external override {
82     require(msg.sender == core().oracle(), "KeysManager: Only oracle
        ↪ can activate");
83     for (uint256 i; i < publicKeys.length; ) {
84         Validator storage validator = _validators[publicKeys[i]];
85         require(
86             validator.state == State.DEPOSITED,
87             "KeysManager: Validator not in valid state"
88         );
89         // node operator active validator count decreases
90         nodeOperatorValidatorCount[validator.nodeOperator] -= 1;

```

Recommendation:

Consider calling `claimAndUpdateRewardDebt` after decreasing the validator count for the `nodeOperator`.

Updates

The team resolved the issue by adding a call to the `claimAndUpdateRewardDebt` function after updating the validator count.

SHB.10.3: KeysManager.sol

```
94 function exitedValidator(bytes[] memory publicKeys) external override {
95     require(msg.sender == core().oracle(), "KeysManager: Only oracle can
        ↳ activate");
96     for (uint256 i; i < publicKeys.length; ) {
97         Validator storage validator = _validators[publicKeys[i]];
98         require(
99             validator.state == State.DEPOSITED,
100             "KeysManager: Validator not in valid state"
101         );
102         // node operator active validator count decreases
103         nodeOperatorValidatorCount[validator.nodeOperator] -= 1;
104         IStakingPool(core().validatorPool()).claimAndUpdateRewardDebt(
            ↳ validator.nodeOperator);
105         validator.state = State.EXITED;
106         unchecked {
107             ++i;
108         }
109     }
110     emit ExitValidator(publicKeys);
111 }
```

SHB.11 Desynchronization of `pricePerShare` Between L1 and L2

- Severity: **HIGH**
- Likelihood: 3
- Status: Acknowledged
- Impact: 2

Description:

After reaching consensus on an oracle report, the contract updates the `pricePerShare` of `stkETH` on L1 based on accumulated rewards and slashing penalties using the `changeCValue` function. However, this updated price is not automatically reflected on Layer 2 (L2). The synchronization only occurs when someone explicitly calls the `changeCValueL2` function. This leads to a significant desynchronization in the `pricePerShare` between L1 and L2, allowing `stkETH` to be minted at inconsistent prices across layers.

Exploit Scenario:

An actor observes the desynchronization between L1 and L2 `pricePerShare`. They exploit this discrepancy by minting `stkETH` on the layer where the price is more favorable, potentially leading to arbitrage opportunities or undue advantage.

Files Affected:

SHB.11.1: Oracle.sol

```
182 function changeCValue(int256 calculatedRewards) internal whenNotPaused {
183     if (calculatedRewards > 0) {
184         uint256 valEthShare = (valCommission * uint256(calculatedRewards)
            ↪ ) / BASIS_POINT;
185         uint256 protocolEthShare = (pStakeCommission * uint256(
            ↪ calculatedRewards)) /
186             BASIS_POINT;
187         IIssuer issuer = IIssuer(core().issuer());
```

```

188     pricePerShare =
189         ((withdrawals.getTotalRewards() +
190             issuer.ethStaked() -
191             withdrawals.getTotalSlashedAmount() -
192             valEthShare -
193             protocolEthShare) * 1e18) /
194         issuer.stkEthMinted();
195     withdrawals.distributeRewards(protocolEthShare, valEthShare,
196         ↪ pricePerShare);
197     emit RewardRateChanged(pricePerShare);
198 } else if (calculatedRewards < 0) {
199     IIssuer issuer = IIssuer(core().issuer());
200     pricePerShare =
201         ((withdrawals.getTotalRewards() +
202             issuer.ethStaked() -
203             withdrawals.getTotalSlashedAmount()) * 1e18) /
204         issuer.stkEthMinted();
205     emit RewardRateChanged(pricePerShare);
206 }

```

SHB.11.2: Oracle.sol

```

211 function changeCValueL2(
212     uint256 _messengerId,
213     bytes memory _payload
214 ) external payable whenNotPaused {
215     IIssuer issuer = IIssuer(core().issuer());
216     (bool messengerStatus, address messenger) = issuer.getMessenger(
217         ↪ _messengerId);
218     if (!messengerStatus (messenger == address(0))) revert
219         ↪ InvalidMessenger();
220     IL1Messenger(messenger).changeCValueL2{ value: msg.value }(
221         msg.sender,
222         pricePerShare,

```

```
221         _payload
222     );
223 }
```

Recommendation:

Consider calling the `changeCValueL2` for each chain whenever the `pricePerShare` changes to keep it synchronized between the chains.

Updates

The team acknowledged the risk, stating that they will call the `changeCValueL2` from an off chain oracle after `pushData` gets executed as different payloads are required for different messengers. Also, for long term they will be using an aggregator service for cross chain communication and update the `cValueL2` inside `pushData` itself. .

SHB.12 Inequitable Reward Distribution for New Validators

- Severity: **HIGH**
- Likelihood : 3
- Status : Acknowledged
- Impact : 2

Description:

Within the `KeysManager` contract, the function `depositValidator` is designed to facilitate the addition of validators from the `nodeOperators` count, which subsequently influences the rewards allocated to `nodeOperators`. However, the issue stems from the fact that these rewards are not updated before the addition of a new validator. As a consequence, new validators end up sharing rewards allocated to old validators.

Exploit Scenario:

- NodeOperator Bob currently has 0 validators under his control.

- Bob decides to add 1 new validator to his list Prior to the rewards being distributed among validators.
- When the rewards are subsequently given out, Bob receives a share of the rewards that were initially accumulated by other validators who were active before he added his new validator.
- Bob benefits from rewards he did not contribute to, thereby gaining an unfair advantage over other validators who earned their rewards through actual participation and contribution.

Files Affected:

SHB.12.1: KeysManager.sol

```

101 function depositValidator(bytes memory publicKey) external override {
102     require(msg.sender == core().issuer(), "KeysManager: Only issuer can
        ↳ activate");
103
104     Validator storage validator = _validators[publicKey];
105
106     // num of Valudators allowed is specified as type(uint256).max .
        ↳ Hence, using the same here
107     require(
108         type(uint256).max > nodeOperatorValidatorCount[validator.
            ↳ nodeOperator],
109         "KeysManager: validator deposit not added by node operator"
110     );
111
112     require(validator.state == State.ACTIVATED, "KeysManager: Key not
        ↳ activated");
113     validator.state = State.DEPOSITED;
114     // node operator active validator count increases
115     nodeOperatorValidatorCount[validator.nodeOperator] += 1;
116

```

```

117     IStakingPool(core().validatorPool()).claimAndUpdateRewardDebt(
        ↪ validator.nodeOperator);
118
119     emit DepositValidator(publicKey);
120 }

```

SHB.12.2: StakingPool.sol

```

74 function claimAndUpdateRewardDebt(address usr) external override {
75     UserInfo storage user = userInfos[usr];
76
77     uint256 userValidators = IKeysManager(core.keysManager()).
        ↪ nodeOperatorValidatorCount(usr);
78
79     uint256 pending = ((accRewardPerValidator * user.amount) / 1e12) -
        ↪ user.rewardDebt;
80
81     if (pending > 0) {
82         IERC20Upgradeable(address(stkEth)).safeTransfer(usr, pending);
83         emit RewardRedeemed(pending, usr);
84     }
85
86     user.rewardDebt = (accRewardPerValidator * userValidators) / 1e12;
87     user.amount = userValidators;
88 }

```

Recommendation:

Consider implementing a mechanism to update the rewards before new deposits. This can be achieved by gathering an array of pending depositors (who called the [depositToEth2](#)) and stating them as [DEPOSITED](#) after updating the rewards in the [Oracle](#) contract.

Updates

The team acknowledged the issue, stating that they will be implementing the fix by using the oracle to make the validator eligible for rewards.

SHB.13 Incorrect Condition Prevents Governor from Updating Commission Fees

- Severity: **HIGH**
- Likelihood: 2
- Status: Fixed
- Impact: 3

Description:

The `updateCommissions` function is designed to allow the Governor to update commission fees. However, there's an oversight in the condition that checks the validity of the provided commission values. The condition reverts if both `_pStakeCommission` and `_valCommission` are less than `BASIS_POINT`, and their sum is also less than `BASIS_POINT`. This incorrect condition can prevent the Governor from updating the commission fees to valid values.

Exploit Scenario:

The Governor attempts to update the commission fees using the `updateCommissions` function. Due to the incorrect condition, even if the provided values are valid and within the acceptable range, the function might revert with the `InvalidValues` error, preventing the Governor from setting the desired commission rates.

Files Affected:

SHB.13.1: Oracle.sol

```
118 function updateCommissions(  
119     uint32 _pStakeCommission,  
120     uint32 _valCommission  
121 ) external onlyGovernor {  
122     if (  
123         _pStakeCommission < BASIS_POINT &&  
124         _valCommission < BASIS_POINT &&  
125         (_pStakeCommission + _valCommission) < BASIS_POINT
```



```

126     ) {
127         revert InvalidValues();
128     }
129     pStakeCommission = _pStakeCommission;
130     valCommission = _valCommission;
131     emit CommissionsUpdated(_pStakeCommission, _valCommission);
132 }

```

Recommendation:

Review and correct the condition in the `updateCommissions` function to ensure that it accurately checks the validity of the provided commission values.

SHB.13.2: Oracle.sol

```

118 function updateCommissions(
119     uint32 _pStakeCommission,
120     uint32 _valCommission
121 ) external onlyGovernor {
122     if (
123         _pStakeCommission >= BASIS_POINT
124         _valCommission >= BASIS_POINT
125         (_pStakeCommission + _valCommission) >= BASIS_POINT
126     ) {
127         revert InvalidValues();
128     }
129     pStakeCommission = _pStakeCommission;
130     valCommission = _valCommission;
131     emit CommissionsUpdated(_pStakeCommission, _valCommission);
132 }

```

Updates

The team resolved the issue by correcting the commission checks.

SHB.13.3: Oracle.sol

```

113 function updateCommissions(
114     uint32 _pStakeCommission,
115     uint32 _valCommission
116 ) external onlyGovernor {
117     if (
118         _pStakeCommission > BASIS_POINT
119         _valCommission > BASIS_POINT
120         (_pStakeCommission + _valCommission) > BASIS_POINT
121     ) revert InvalidValues();
122     pStakeCommission = _pStakeCommission;
123     valCommission = _valCommission;
124     emit CommissionsUpdated(_pStakeCommission, _valCommission);
125 }

```

SHB.14 First Staker can Grief Others using an Inflation Attack

- Severity: **MEDIUM**
- Likelihood: 2
- Status: Fixed
- Impact: 2

Description:

A malicious actor can front-run a call to the `pushData` function within `Oracle`. This call updates the exchange rate and subsequently influences the `stkETH` price calculation. By exploiting the issue described in (minimum stake amount bypass), the attacker can stake 1 wei, and mint 1 wei of `stkETH`, and then proceed to deposit a substantial ETH value in `withdrawalCredential`. Which leads to an inflation of the price of `stkETH`. This inflation impacts subsequent users' ability to stake, leading to a cascading effect of artificially inflated token prices. Users attempting to stake face rounding down issues, intensifying the inflation attack's consequences. it can be even more impacting if the protocol implements an unstaking mechanism.

Exploit Scenario:

1. Consider a scenario where the protocol has just been deployed and has no stakers and that oracle members are trying to activate some validators.
2. Malicious user Bob observes the mempool for `pushData` transactions and anticipates that a particular vote will satisfy the quorum check.
3. Bob performs a front-running attack by submitting two transactions:
 - (a) He exploits the issue: SHB.25. Minimum Stake Amount Bypass, he will be able to stake 1 wei of Ether and receives 1 wei of stkETH due to the 1:1 exchange rate.
 - (b) Bob then deposits a large amount of ETH into `withdrawalCredential`.
4. The `pushData` transaction, which was expected to pass the quorum check, succeeds and updates the exchange rate, leading to a higher stkETH price calculation. The price calculation incorporates `pricePerShare`:

SHB.14.1: IssuerUpgradable.sol

```
pricePerShare =  
    ((withdrawals.getTotalRewards() +  
     issuer.ethStaked() -  
     withdrawals.getTotalSlashedAmount()) * 1e18) /  
    issuer.stkEthMinted();
```

5. As a result of the inflated `pricePerShare`, the price of 1 wei of stkETH becomes very high due to the large ETH deposit in the `WithdrawalCredential` contract.
6. Subsequent users attempting to stake will have to do so at the inflated price until the `pushData` function is called again to update the exchange rate.
7. Users trying to stake less than the balance of `WithdrawalCredential` contract will receive no shares due to `stkEthToMint` rounding down to zero, exacerbating the impact of the inflation attack.

Files Affected:

SHB.14.2: IssuerUpgradable.sol

```
164 function mintL2(  
165     uint256 _messengerId,  
166     uint256 _callValue,  
167     address _receiverAddress,  
168     bytes memory _payload  
169 )  
170     external  
171     payable  
172     whenNotPaused  
173     minimumStakeAmount(msg.value)  
174     onlyExistingMessenger(_messengerId)  
175 {  
176     uint256 ethToStake = msg.value - _callValue;  
177     emit Stake(msg.sender, ethToStake, block.timestamp);  
178     uint256 stkEthToMint = (ethToStake * 1e18) / core.stkEth().  
        ↪ pricePerShare();
```

SHB.14.3: Oracle.sol

```
182 function changeCValue(int256 calculatedRewards) internal whenNotPaused {  
183     if (calculatedRewards > 0) {  
184         uint256 valEthShare = (valCommission * uint256(calculatedRewards)  
            ↪ ) / BASIS_POINT;  
185         uint256 protocolEthShare = (pStakeCommission * uint256(  
            ↪ calculatedRewards)) /  
186         BASIS_POINT;  
187         IIssuer issuer = IIssuer(core().issuer());  
188         pricePerShare =  
189             ((withdrawals.getTotalRewards() +  
190                 issuer.ethStaked() -  
191                 withdrawals.getTotalSlashedAmount() -  
192                 valEthShare -  
193                 protocolEthShare) * 1e18) /  
194             issuer.stkEthMinted();
```

```

195         withdrawals.distributeRewards(protocolEthShare, valEthShare,
            ↪ pricePerShare);
196         emit RewardRateChanged(pricePerShare);
197     } else if (calculatedRewards < 0) {
198         IIssuer issuer = IIssuer(core().issuer());
199         pricePerShare =
200             ((withdrawals.getTotalRewards() +
201              issuer.ethStaked() -
202              withdrawals.getTotalSlashedAmount()) * 1e18) /
203             issuer.stkEthMinted();
204         emit RewardRateChanged(pricePerShare);
205     }
206 }

```

Recommendation:

It is recommended to correct the minimum stake check to remediate the risk of the inflation attack.

Updates

The team resolved the issue by applying the `minimumStakeAmount` check on the `msg.value - _callValue` instead of `msg.value`.

SHB.14.4: IssuerUpgradable.sol

```

168 function mintL2(
169     uint256 _messengerId,
170     uint256 _callValue,
171     address _receiverAddress,
172     bytes memory _payload
173 )
174     external
175     payable
176     whenNotPaused
177     minimumStakeAmount(msg.value - _callValue)

```

```
178     onlyExistingMessenger(_messengerId)
179     {
```

SHB.15 Inaccurate `rewardDebt` Calculation for `nodeOperators` Modifying Validator Count

- Severity: **MEDIUM**
- Likelihood: 2
- Status: Fixed
- Impact: 2

Description:

Within the `StakingPool` contract, the function `claimAndUpdateRewardDebt` allows `nodeOperators` to claim their rewards based on the number of validators under their management. However, a discrepancy in the calculation of `user.rewardDebt` leads to inconsistent outcomes for `nodeOperators` when they add or remove validators from their control.

Exploit Scenario:

Scenario A: Inaccurate Reward Debt Upon Adding Validators

1. In a scenario involving 20 validators and a total of 10000 wei in validator rewards, the ideal distribution dictates that each validator should receive 500 wei as their proportionate share of the rewards.
2. Node Operator Bob, responsible for managing 2 validators, makes the decision to introduce a 3rd validator under his supervision.
3. Upon Bob's invocation of the `claimAndUpdateRewardDebt` function, he receives rewards meant for the total number of `user.amount` validators, which amounts to 2. Consequently, he gains rewards equivalent to 1000 wei (500 wei per validator * 2 validators).

4. However, the variable `user.rewardDebt`, intended to represent the amount Bob has received, is inaccurately calculated using the formula $(\text{accRewardPerValidator} * \text{userValidators}) / 1e12$. In this case, it is set to 1500 wei (500 wei * 3 validators).
5. Consequently, the protocol erroneously assumes that Bob has obtained 1500 wei, when in reality, he has only received 1000 wei. This miscalculation leads to Bob receiving fewer rewards than he should during the subsequent invocation of `claimAndUpdateRewardDebt`.

Scenario B: Incorrect Reward Debt After Exiting a Validator

1. In a scenario with 20 validators and 10000 wei in validator rewards, each validator should receive 500 wei as their share of the rewards.
2. Node Operator Bob, managing 2 validators, Bob decides to exit 1 validator, reducing his validator count to 1.
3. Upon Bob's invocation of the `claimAndUpdateRewardDebt` function, he receives rewards meant for the total number of `user.amount` validators, which amounts to 2. Consequently, he gains rewards equivalent to 1000 wei (500 wei per validator * 2 validators).
4. However, the variable `user.rewardDebt`, intended to represent the amount Bob has received, is inaccurately calculated using the formula $(\text{accRewardPerValidator} * \text{userValidators}) / 1e12$. In this case, it is set to 500 wei (500 wei * 1 validators).
5. Consequently, the protocol erroneously assumes that Bob has obtained 500 wei, when in reality, he has received 1000 wei. This miscalculation leads to Bob receiving more rewards than he should during the subsequent invocation of `claimAndUpdateRewardDebt`.

Files Affected:

SHB.15.1: StakingPool.sol

```
74 function claimAndUpdateRewardDebt(address usr) external override {  
75     UserInfo storage user = userInfos[usr];  
76
```

```

77     uint256 userValidators = IKeysManager(core.keysManager()).
        ↳ nodeOperatorValidatorCount(usr);
78
79     uint256 pending = ((accRewardPerValidator * user.amount) / 1e12) -
        ↳ user.rewardDebt;
80
81     if (pending > 0) {
82         IERC20Upgradeable(address(stkEth)).safeTransfer(usr, pending);
83         emit RewardRedeemed(pending, usr);
84     }
85
86     user.rewardDebt = (accRewardPerValidator * userValidators) / 1e12;
87     user.amount = userValidators;

```

Recommendation:

Consider correcting `user.rewardDebt` calculation to be : `user.rewardDebt = (accRewardPerValidator * user.amount) / 1e12` when `user.amount` is not zero , and using the current formula of `user.rewardDebt = (accRewardPerValidator * userValidators) / 1e12` otherwise.

Updates

The team resolved the issue by using `user.amount` when it's different from zero.

SHB.15.2: StakingPool.sol

```

74     function claimAndUpdateRewardDebt(address usr) external override {
75         UserInfo storage user = userInfos[usr];
76
77         uint256 userValidators = IKeysManager(core.keysManager()).
            ↳ nodeOperatorValidatorCount(usr);
78
79         uint256 pending = ((accRewardPerValidator * user.amount) / 1e12) -
            ↳ user.rewardDebt;
80
81         if (pending > 0) {

```



```

82         IERC20Upgradeable(address(stkEth)).safeTransfer(usr, pending);
83         emit RewardRedeemed(pending, usr);
84     }
85
86     if(user.amount != 0) {
87         user.rewardDebt = (accRewardPerValidator * user.amount) / 1e12;
88     } else {
89         user.rewardDebt = (accRewardPerValidator * userValidators) / 1e12
90         ↪ ;
91     }
92     user.amount = userValidators;
93 }

```

SHB.16 Uninitialized `socketRegistry` Address Leading to Potential Loss of Funds

- Severity: **MEDIUM**
- Likelihood: 1
- Status: Fixed
- Impact: 3

Description:

The Issuer contract in L2 contains a function `transferEthMainnet` designed to transfer ETH from Layer 2 (L2) to Layer 1 (L1). However, there's a critical oversight related to the `socketRegistry` address. This address is not initialized in the contract's `constructor`. If the owner does not set this address post-deployment, any attempt to transfer ETH using the `transferEthMainnet` function can result in a loss of funds, as the funds would be sent to an uninitialized address.

Files Affected:

SHB.16.1: Issuer.sol

```

118 function transferEthMainnet(
119     uint256 _stkEthMinted,
120     uint256 _amount,
121     uint256 _slippageFee,
122     bytes calldata _payload
123 ) external override onlyOracle returns (bool) {
124     if (address(this).balance < _amount + _slippageFee) revert
        ↳ InsufficientBalance();
125     // update correct amount
126     newEthStaked = newEthStaked - _amount;
127     newStkEthMinted = newStkEthMinted - _stkEthMinted;
128     slippageColleted -= _slippageFee;
129     (bool success, ) = socketRegistry.call{ value: _amount +
        ↳ _slippageFee }(_payload);
130     if (!success) revert BridgeCallFailed();
131     emit EthBridgedToL1(address(this).balance);
132     return success;
133 }

```

Recommendation:

Ensure that the `socketRegistry` address is initialized during the contract deployment, preferably in the `constructor`.

Updates

The team resolved the issue by initializing the `socketRegistry` address in the `initialize` function.

SHB.16.2: Issuer.sol

```

69 function initialize(IStkEth _stketh, address _socketRegistry) public
    ↳ initializer {
70     __Ownable_init();
71     stketh = _stketh;
72     socketRegistry = _socketRegistry;

```

SHB.17 Lack of Blacklist Mechanism for Malicious Node Operators

- Severity: **MEDIUM**
- Likelihood: 1
- Status: Acknowledged
- Impact: 3

Description:

The current contract implementation addresses the scenario where a validator acts maliciously and subsequently gets slashed. However, post-slashing, there's nothing in place to prevent the same node operator from creating a new validator, calling the [depositToEth2](#) function, and potentially repeating the malicious actions. This oversight can allow malicious actors to continually exploit and grieve the protocol.

Exploit Scenario:

A validator acts maliciously, leading to them being slashed. Post-slashing, the validator, leveraging the lack of preventive measures in the contract, calls the [depositToEth2](#) function to create a new validator. They can then repeat their malicious actions, causing repeated harm to the protocol and its participants.

Files Affected:

SHB.17.1: IssuerUpgradable.sol

```
280 function depositToEth2(bytes calldata publicKey) external whenNotPaused
    ↪ {
281     require(
282         address(this).balance >= VALIDATOR_DEPOSIT + VERIFICATION_DEPOSIT
    ↪ ,
```

```

283         "Issuer: Not enough ether deposited"
284     );
285     IKeysManager.Validator memory validator = IKeysManager(core.
        ↪ keysManager()).validators(
286         publicKey
287     );
288
289     withdrawalVerificationDeposit(validator.nodeOperator);
290
291     IKeysManager(core.keysManager()).depositValidator(publicKey);
292
293     depositedValidators = depositedValidators + 1;
294     DEPOSIT_CONTRACT.deposit{ value: VALIDATOR_DEPOSIT }(
295         publicKey,
296         abi.encodePacked(core.withdrawalCredential()),
297         validator.signature,
298         validator.deposit_root
299     );
300 }

```

Recommendation:

Track and monitor validators that get slashed due to malicious actions, and implement a blacklist mechanism within the contract.

Updates

The team acknowledged the issue, stating that they will be implementing a blacklist mechanism with the withdrawal feature.

SHB.18 Owner Can Set Critical Values to Zero

- Severity: **MEDIUM**
- Likelihood: 1
- Status: Fixed
- Impact: 3

Description:

The `setValues` function allows the owner to set the values of `mevRewards` and `exitBalance` to zero. While the comment suggests that this function is meant for initialization, it's redundant since uint variables in Solidity are initialized to zero by default. Moreover, allowing the owner to reset these values post-initialization can lead to unintended consequences.

Files Affected:

SHB.18.1: WithdrawalCredential.sol

```
84 /// @notice this function will be used to initialize mev rewards and  
    ↪ exit balance  
85 function setValues() external onlyOwner {  
86     mevRewards = 0;  
87     exitBalance = 0;  
88 }
```

Recommendation:

Consider removing the `setValues` function as it does not add the intended functionality.

Updates

The team resolved the issue by removing the `setValues` function.

SHB.19 Oracle Members Can Vote on Multiple ConsensusData Inputs

- Severity: MEDIUM
- Status: Acknowledged
- Likelihood: 1
- Impact: 3

Description:

The `pushData` function is designed to allow oracle members to vote on a specific `ConsensusData`. While the function restricts an oracle member from voting more than once on a specific `ConsensusData`, it doesn't prevent them from voting on multiple and different `ConsensusData` inputs within the same tx epoch. This oversight can allow a malicious oracle to produce multiple attestations in the same epoch, undermining the consensus logic.

Exploit Scenario:

A malicious oracle member, aiming to disrupt the consensus mechanism, submits votes on multiple different `ConsensusData` inputs within the same epoch. This behavior can lead to confusion, potential desynchronization, and could compromise the integrity of the consensus mechanism.

Files Affected:

SHB.19.1: Oracle.sol

```
251 function pushData(  
252     ConsensusData memory _consensusData  
253 ) external override whenNotPaused onlyOracle {  
254     if (beaconData.getNextTxEpoch(lastCompletedEpoch) != beaconData.  
        ↪ getCurrentEpoch()) {  
255         revert VotedEarly();  
256     }
```

```

257     bytes32 candidateId = keccak256(abi.encode(_consensusData,
        ↪ beaconData.getCurrentEpoch()));
258     bytes32 voteId = keccak256(abi.encode(msg.sender, candidateId));
259     if (submittedVotes[voteId]) {
260         revert AlreadyVoted(msg.sender);
261     }
262     submittedVotes[voteId] = true;
263     uint256 candidateNewVotes = candidates[candidateId] + 1;
264     candidates[candidateId] = candidateNewVotes;
265     if (candidateNewVotes >= quorum) {

```

Recommendation:

Adapt the `pushData` function to ensure that an oracle member can only vote once per tx epoch, regardless of the `ConsensusData` input.

Updates

The team acknowledged the issue, stating that they are planning to implement the recommendation with the withdrawal feature.

SHB.20 Need for Whitelisting Trusted Relayers in MEV Boost

- | | |
|---------------------------|-----------------|
| • Severity: MEDIUM | • Likelihood: 1 |
| • Status: Acknowledged | • Impact: 3 |

Description:

In the context of MEV Boost, Relays play a crucial role as a data-availability layer and communication bridge between builders and validators. They are doubly-trusted: builders trust them for unbiased payload routing, while proposers trust them for block validity, accuracy, and data availability. Given their specialization in Denial of Service (DoS) protection and

networking, it's essential to ensure that only trustworthy relayers are allowed to participate. Without a mechanism to whitelist trusted relayers, the system is exposed to potential risks, especially since there's an inherent trust assumption on relayers in PBS before the integration of in-protocol PBS in Ethereum.

Recommendation:

Implement a mechanism to whitelist a set of trusted relayers within the MEV Boost system.

Updates

The team acknowledged the issue, stating that they are planning to implement the recommendation with the withdrawal feature.

SHB.21 Requirement for Node Operators to Set Fee Recipient to Protocol-Managed Address

- | | |
|---------------------------|-----------------|
| • Severity: MEDIUM | • Likelihood: 1 |
| • Status: Acknowledged | • Impact: 3 |

Description:

For the pStake system, Node Operators who run validators should be mandated to set the fee recipient for their respective validators to an address that is managed by the protocol. This address is specifically for managing Execution Layer Rewards. It's important to note that this address is distinct from the Withdrawal Credentials in the consensus layer.

Exploit Scenario:

If Node Operators set the fee recipient to an address other than the protocol-managed one, the Execution Layer Rewards will not be fairly distributed between the stakers, the protocol and the validators. This leads to a loss of rewards.

Recommendation:

Implement a mechanism within the pStake system to enforce Node Operators to set the fee recipient to a protocol-managed address. This can be achieved by monitoring the node operators to make sure the fee recipient address is set to the correct address.

Updates

The team acknowledged the issue, stating that they are planning to implement the recommendation with the withdrawal feature.

SHB.22 Missing Socket API Payload Check

- Severity: **MEDIUM**
- Likelihood : 1
- Status : Acknowledged
- Impact : 3

Description:

The protocol currently employs the socket bridge to facilitate the transfer of ether from L2 to L1. Given the potential risks associated with a compromise in Socket API servers, it's crucial to have an additional layer of validation for the payload data. Implementing the [Socket V2 Verifier](#) can serve as this additional validation layer, ensuring the integrity and authenticity of the data being transferred.

Exploit Scenario:

If the Socket API servers are compromised, malicious actors could manipulate or inject malicious payload data during the transfer from L2 to L1. This could lead to incorrect or fraudulent transfers, potentially causing financial losses or undermining the trust in the protocol.

Files Affected:

SHB.22.1: Issuer.sol

```
118 function transferEthMainnet(  
119     uint256 _stkEthMinted,  
120     uint256 _amount,  
121     uint256 _slippageFee,  
122     bytes calldata _payload  
123 ) external override onlyOracle returns (bool) {  
124     if (address(this).balance < _amount + _slippageFee) revert  
125         ↳ InSufficientBalance();  
126     // update correct amount  
127     newEthStaked = newEthStaked - _amount;  
128     newStkEthMinted = newStkEthMinted - _stkEthMinted;  
129     slippageColleted -= _slippageFee;  
130     (bool success, ) = socketRegistry.call{ value: _amount +  
131         ↳ _slippageFee }(_payload);  
132     if (!success) revert BridgeCallFailed();  
133     emit EthBridgedToL1(address(this).balance);  
134     return success;  
135 }
```

Recommendation:

Integrate the [Socket V2 Verifier](#) into the protocol's transfer mechanism.

Updates

The team acknowledged the issue, stating that the [Socket V2 Verifier](#) is not yet in production mode.

SHB.23 WITHDRAWAL_CREDENTIAL_BYTES32 Setter Desynchronizes Old Validators

- Severity: MEDIUM
- Likelihood: 1
- Status: Acknowledged
- Impact: 3

Description:

The smart contract `Core` features a variable named `WITHDRAWAL_CREDENTIAL_BYTES32`, which stores the withdrawal address for rewards and full withdrawals. This withdrawal address, once set in the protocol, remains immutable and cannot be changed. As a result, validators who have registered with a specific withdrawal address are unable to modify it after registration. The only way to alter this address is through the intervention of the governor, who can call the `setWithdrawalCredential` function. However, even if the withdrawal address is changed by the governor, previously registered validators will continue to retain the initially assigned withdrawal address, which will cause a desynchronization between validators.

Files Affected:

SHB.23.1: Core.sol

```
58 function setWithdrawalCredential(bytes32 withdrawcreds) external
    ↪ onlyGovernor {
59 //0x01000000000000000000000003d80b31a78c30fc628f20b2c89d7ddbf6e53cedc
60 WITHDRAWAL_CREDENTIAL_BYTES32 = withdrawcreds;
```

Recommendation:

Consider setting the `WITHDRAWAL_CREDENTIAL_BYTES32` only once, as the protocol should rely on upgradeability to modify the `WithdrawalCredential`'s code.

Updates

The team acknowledged the issue, stating that the **Core** contract is already deployed and it is not upgradeable.

SHB.24 Governor Has Full Control Over Oracle Quorum

- Severity: **LOW**
- Likelihood: 1
- Status: Acknowledged
- Impact: 2

Description:

There exists a function within the contract that permits the governor to modify the **quorum**, which is the number of required votes needed by oracle members to validate the data. This capability grants the governor undue influence and control over the oracle, potentially compromising its decentralized nature and integrity.

Files Affected:

SHB.24.1: Oracle.sol

```
110 function updateQuorum(uint32 latestQuorum) external onlyGovernor
    ↪ NonZeroQuorum(latestQuorum) {
111     emit QuorumUpdated(latestQuorum, quorum);
112     quorum = latestQuorum;
```

Recommendation:

Consider implementing a decentralized governance that should be responsible for critical changes like adjusting the quorum. Also, it is recommended to limit the governor's ability to change the quorum or introduce a range within which the quorum can be adjusted to prevent extreme values.

Updates

The team acknowledged the issue, stating that they are planning to implement proper governance with DAO but will start by a multisig with time-lock to update any admin, governor functionalities.

SHB.25 Minimum Stake Amount Bypass

- Severity: **LOW**
- Likelihood: 2
- Status: Fixed
- Impact: 1

Description:

The function `mintL2` within the `Issuer` L1 contract is designed to facilitate the minting of stkETH tokens in Layer 2 (e.g., Arbitrum or Optimism). This function includes a check to ensure that the amount of ETH supplied in `msg.value` is greater than or equal to the minimum stake amount. However, the actual amount of stkETH minted to the user is determined by the variable `ethToStake`, which is derived from `msg.value - _callValue`. This discrepancy enables users to exploit the protocol by minting arbitrarily low amounts of stkETH through manipulation of the `_callValue`, bypassing the intended minimum stake requirement. This exploitation contradicts the security assumption mentioned in the `minimumStakeAmount` modifier comment. It's important to note that this issue is particularly applicable in Arbitrum, where the remainder of `_callValue` is reimbursed to the user. This reimbursement mechanism effectively allows users to mint stkETH with minimal monetary commitment.

Exploit Scenario:

1. Assuming an initial exchange rate of 1:1 between stkETH and ETH.
2. The `mintL2` function includes a check to ensure that `msg.value` (the amount of ETH supplied) is greater than or equal to the minimum stake amount.

3. However, the actual amount of `stkETH` minted is determined by `ethToStake`, calculated as `msg.value - _callValue`.
4. Exploiting this discrepancy, a user (e.g., Bob) can manipulate `_callValue` to make `ethToStake` an arbitrarily low value.
5. Bob calls `mintL2` with the following parameters:
`msg.value = 10000 gwei`
`_callValue = 10000 gwei - 1 wei`
`messengerId` of the `ArbitrumMessenger` contract
Therefore :
`ethToStake = msg.value - _callValue = 1 wei`
6. Bob successfully mints an amount of `stkETH` equivalent to 1 wei, which significantly deviates from the intended minimum stake requirement, and gets refunded back `_callValue - arbitrum Fees`.
7. This enables Bob to bypass the minimum stake constraint, violating the security assumption.

Files Affected:

SHB.25.1: IssuerUpgradable.sol

```
164 function mintL2(  
165     uint256 _messengerId,  
166     uint256 _callValue,  
167     address _receiverAddress,  
168     bytes memory _payload  
169 )  
170     external  
171     payable  
172     whenNotPaused  
173     minimumStakeAmount(msg.value)  
174     onlyExistingMessenger(_messengerId)  
175 {
```

```

176     uint256 ethToStake = msg.value - _callValue;
177     emit Stake(msg.sender, ethToStake, block.timestamp);
178     uint256 stkEthToMint = (ethToStake * 1e18) / core.stkEth().
        ↪ pricePerShare();

```

Recommendation:

Consider performing the `minimumStakeAmount` check on `ethToStake` instead of `msg.value`.

Updates

The team resolved the issue by applying the `minimumStakeAmount` check on the `msg.value - _callValue` instead of `msg.value`.

SHB.25.2: IssuerUpgradable.sol

```

168 function mintL2(
169     uint256 _messengerId,
170     uint256 _callValue,
171     address _receiverAddress,
172     bytes memory _payload
173 )
174     external
175     payable
176     whenNotPaused
177     minimumStakeAmount(msg.value - _callValue)
178     onlyExistingMessenger(_messengerId)
179 {

```

SHB.26 Inability to Update `stkETH` Exchange Rate When All Rewards Are Slashed

- Severity: **LOW**
- Likelihood: 1
- Status: Fixed
- Impact: 2

Description:

Within the `Oracle` contract, the function `pushData` is essential for communicating key information from the consensus layer to the protocol. This information includes details about exited validators and the amounts that have been slashed. The variable `deltaBalanceChange` is responsible for representing rewards earned by validators, and the subsequent call to the `changeCValue` function facilitates the modification of the exchange rate for `stkETH`. However, an issue arises when the `deltaBalanceChange` is equal to the `slashed_amount`. In this situation, the exchange rate remains unchanged even if stakers have staked ETH in the tx epoch, which contradicts the intended behavior of the protocol.

Files Affected:

SHB.26.1: Oracle.sol

```
297 withdrawals.setRewardsSlashedAmount(  
298     deltaBalanceChange,  
299     _consensusData.slashedAmount,  
300     exitBalance  
301 );  
302 changeCValue(int256(deltaBalanceChange) - int256(slashed_amount));
```

SHB.26.2: Oracle.sol

```
182 function changeCValue(int256 calculatedRewards) internal whenNotPaused {  
183     if (calculatedRewards > 0) {
```



```

184     uint256 valEthShare = (valCommission * uint256(calculatedRewards)
        ↪ ) / BASIS_POINT;
185     uint256 protocolEthShare = (pStakeCommission * uint256(
        ↪ calculatedRewards)) /
186         BASIS_POINT;
187     IIssuer issuer = IIssuer(core().issuer());
188     pricePerShare =
189         ((withdrawals.getTotalRewards() +
190             issuer.ethStaked() -
191             withdrawals.getTotalSlashedAmount() -
192             valEthShare -
193             protocolEthShare) * 1e18) /
194             issuer.stkEthMinted();
195     withdrawals.distributeRewards(protocolEthShare, valEthShare,
        ↪ pricePerShare);
196     emit RewardRateChanged(pricePerShare);
197 } else if (calculatedRewards < 0) {
198     IIssuer issuer = IIssuer(core().issuer());
199     pricePerShare =
200         ((withdrawals.getTotalRewards() +
201             issuer.ethStaked() -
202             withdrawals.getTotalSlashedAmount()) * 1e18) /
203             issuer.stkEthMinted();
204     emit RewardRateChanged(pricePerShare);
205 }

```

Recommendation:

Consider including the case where `calculatedRewards` is equal to zero in the `else if` block:

SHB.26.3: Oracle.sol

```

} else if (calculatedRewards <= 0) {
    IIssuer issuer = IIssuer(core().issuer());
    pricePerShare =
        ((withdrawals.getTotalRewards() +

```

```

        issuer.ethStaked() -
        withdrawals.getTotalSlashedAmount()) * 1e18) /
        issuer.stkEthMinted();
    emit RewardRateChanged(pricePerShare);
}

```

Updates

The team resolved the issue by including the case where `calculatedRewards` is equal to zero in the `else if` block:

SHB.26.4: Oracle.sol

```

190 } else if (calculatedRewards <= 0) {
191     IIssuer issuer = IIssuer(core().issuer());
192     pricePerShare =
193         ((withdrawals.getTotalRewards() +
194             issuer.ethStaked() -
195             withdrawals.getTotalSlashedAmount()) * 1e18) /
196             issuer.stkEthMinted();
197     emit RewardRateChanged(pricePerShare);
198 }

```

SHB.27 Uninitialized `optimismReceiver` and `arbitrumReceiver` Can Lead to DoS

- Severity: **LOW**
- Likelihood: 1
- Status: Fixed
- Impact: 2

Description:

The `optimismReceiver` and `arbitrumReceiver` variables, crucial for cross-chain functionality, are not initialized in the contract's `constructor`. This oversight can lead to a Denial of

Service (DoS) attack on the cross-chain functionality until these variables are properly initialized at a later stage.

Files Affected:

SHB.27.1: OptimismMessenger.sol

```
15 address private optimismReceiver;
```

SHB.27.2: ArbitrumMessenger.sol

```
13 address private arbitrumReceiver;
```

Recommendation:

Ensure that all critical variables are properly initialized in the contract's constructor.

Updates

The team resolved the issue by initializing `optimismReceiver` and `arbitrumReceiver` in the contract's constructor.

SHB.27.3: OptimismMessenger.sol

```
26 constructor(address _messenger, address _core, address _optimismReceiver  
    ↪ ) L1MessengerBase(_core) {  
27     optimismMessenger = ICrossDomainMessenger(_messenger);  
28     optimismReceiver = _optimismReceiver;  
29 }
```

SHB.27.4: ArbitrumMessenger.sol

```
25 constructor(address _inbox, address _core, address _arbitrumReceiver)  
    ↪ L1MessengerBase(_core) {  
26     inbox = IInbox(_inbox);  
27     arbitrumReceiver = _arbitrumReceiver;  
28 }
```

SHB.28 Hard-coded Slippage Causes DoS

- Severity: **LOW**
- Likelihood: 1
- Status: Acknowledged
- Impact: 1

Description:

The **Issuer** L1 contract contains the function **getDepositL2**, which serves as a mechanism to receive ETH from Layer 2 stakers. This function is designed to implement a slippage control, intended to account for potential delays in the bridge process. However, a significant issue arises from the fact that the slippage control is hard-coded to a fixed value of 1%. This inflexible slippage setting can lead to complications, especially during periods of significant delay in the bridge process. In such cases, the contract may become incapable of receiving ETH from Layer 2 stakers, hindering its intended functionality.

Files Affected:

SHB.28.1: IssuerUpgradable.sol

```
261 function getDepositL2(  
262     uint256 _stkEthMinted,  
263     uint256 _sourceChainId  
264 ) external payable onlySocketReceiver {  
265     // accept 1% error in exchange rate due to delay in bridging  
266     uint256 exchangeRate = core.stkEth().pricePerShare();  
267     if (  
268         exchangeRate - exchangeRate / 100 > (msg.value /  
            ↪ _stkEthMinted)  
269         (msg.value / _stkEthMinted) > exchangeRate + exchangeRate /  
            ↪ 100  
270     ) revert InvalidExchangeRateReceived();
```

Recommendation:

Consider implementing a flexible slippage control to allow the contract to adapt to various bridging delays.

Updates

The team acknowledged the issue, stating that the accepted error rate is kept at max 1% so that the exchange rate does not get changed by a lot and the funds will be transferred once a day by the oracle (protocol) itself which will make sure to provide enough slippage by `addSlippage` functionality if required.

SHB.29 Block Number Difference Between Chains results in Desynchronized Events

- Severity: **INFORMATIONAL**
- Likelihood : 1
- Status : Acknowledged
- Impact : 0

Description:

The contracts `L2MessageContract.sol`, `L2MessageContractOptimism.sol`, and `L2MessageContractArbitrum.sol` contain the function `changeCValue`, which is responsible for minting `sktETH` for users on Layer 2 after receiving a message from the `crossDomainAccount`. This function emits an event `cValueChanged(block.number, _cValue)` to indicate the block number at which the `cValue` changed. However, a crucial issue arises due to the potential disparity between `block.number` on Layer 2 (Arbitrum or Optimism) and `block.number` on Layer 1. This mismatch can lead to the emission of an inaccurate block number in the event, causing confusion and potentially impacting front-end applications relying on accurate event information.

Files Affected:

SHB.29.1: L2MessageContract.sol

```
40 function changeCValue(uint256 _cValue) external
    ↪ onlyFromCrossDomainAccount(msg.sender) {
41     CValue = _cValue;
42     emit cValueChanged(block.number, _cValue);
43 }
```

SHB.29.2: L2MessageContractOptimism.sol

```
53 function changeCValue(
54     uint256 _cValue
55 ) external override onlyFromCrossDomainAccount whenNotPaused {
56     stkETH.changePricePerShare(_cValue);
57     emit cValueChanged(block.number, _cValue);
58 }
```

SHB.29.3: L2MessageContractArbitrum.sol

```
43 function changeCValue(
44     uint256 _cValue
45 ) external override onlyFromCrossDomainAccount whenNotPaused {
46     stkETH.changePricePerShare(_cValue);
47     emit cValueChanged(block.number, _cValue);
48 }
```

Recommendation:

Consider relying on `block.timestamp` instead to have a more accurate way to track event timing.

Updates

The team acknowledged the issue, stating that their UI is fetching data from subgraphs and not directly from the contracts. Also they have integrated different subgraphs for L1 and L2s.

4 Best Practices

BP.1 Remove Unused variables

Description:

The contracts contain multiple variables that are not utilized in their operations. These unused variables can introduce unnecessary complexity, increase gas costs, and potentially lead to confusion or misinterpretations when reviewing or interacting with the contracts. It is recommended to remove those variables.

Files Affected:

BP.1.1: StakingPool.sol

```
32 IERC20Upgradeable public pstake;  
33 IUniswapRouter public router;  
34 address public WETH;
```

BP.1.2: StakingPool.sol

```
42 IPriceOracle public oracle;
```

BP.1.3: StakingPool.sol

```
46 uint256 public DEVIATION; // 5% deviation is acceptable  
47 uint256 public constant BASIS_POINT = 10000;
```

BP.1.4: WithdrawalCredential.sol

```
28 uint256 private newSlashedAmount;
```

BP.1.5: KeysManager.sol

```
17 uint256 public constant PUBKEY_LENGTH = 48;  
18 uint256 public constant SIGNATURE_LENGTH = 96;  
19 uint256 public constant VALIDATOR_DEPOSIT = 31e18;
```

Status - Acknowledged

BP.2 Remove Redundant Initializations with Default Type Values

Description:

The contract contains variables that are explicitly initialized with their default type values. In Solidity, variables are automatically initialized with their default values (e.g., 0 for integers, false for booleans). Remove these redundant initializations to simplify the contract and reduce deployment costs.

Files Affected:

BP.2.1: Oracle.sol

```
273 uint256 slashed_amount = 0;
```

BP.2.2: Oracle.sol

```
284 uint256 exitBalance = 0;
```

BP.2.3: Oracle.sol

```
312 uint256 exitValidatorBalance = 0;
```

Status - Fixed

BP.3 Remove Tautological Statements

Description:

The contract contains tautological statements, which are always true by their nature. Specifically, the require statement checks if `type(uint256).max` is greater than a value from `nodeOperatorValidatorCount`, which will always be true since `type(uint256).max` represents the maximum possible value for an uint256 and `nodeOperatorValidatorCount` will not reach it since it only grows increments.

Files Affected:

BP.3.1: KeysManager.sol

```
107 require(  
108     type(uint256).max > nodeOperatorValidatorCount[validator.  
        ↪ nodeOperator],  
109     "KeysManager: validator deposit not added by node operator"  
110 );
```

Status - Acknowledged

BP.4 Unchanged Variables Should Be Declared as Constants

Description:

The contract contains variables that remain unchanged throughout its lifecycle. These variables, which do not undergo any modifications post-deployment, should ideally be declared as constants. Using constants instead of regular state variables can lead to gas savings.

Files Affected:

BP.4.1: Oracle.sol

```
32 uint256 public minExitBal = 16 ether;
```

BP.4.2: Oracle.sol

```
34 uint256 public maxSlashing = 1 ether;
```

Status - Acknowledged

BP.5 Correct Misleading Comments

Description:

In the `Core` contract, the comments above `setWithdrawalCredential` state that the withdrawal address is in BLS form when it is not, it's an execution key (0x01).

Files Affected:

BP.5.1: Core.sol

[illegible]

Status - Acknowledged

BP.6 Optimize For Loop Counter Increment

Description:

In multiple contracts, the logic necessitates looping over a number of elements. A way to optimize incrementing the counter is using the unchecked keyword and to use post-increment. Here is an example:

Files Affected:

BP.6.1: Example

```
for (uint256 i; i < len;) {
    unchecked{
        ++i;
    }
}
```

Status - Acknowledged

BP.7 Remove Unused Modifier

Description:

The **CoreRef** contract defines a modifier named **ifMinterSelf**. However, throughout the contract's implementation, this modifier is not utilized in any of the functions or methods.

Files Affected:

BP.7.1: CoreRef.sol

```
19 modifier ifMinterSelf() {  
20     if (_core.isMinter(address(this))) {  
21         _;  
22     }  
23 }
```

Status - Acknowledged

5 Tests

Results:

5.1 L1-contracts

→ **admin actions**

- ✓ all contracts deploys successfully (146ms)
- ✓ upgradable contracts get upgraded by admin (280ms)
- ✓ only admin able to set values in core contract (593ms)
- ✓ only admin able to add values to oracle (178ms)
- ✓ only admin able to set l2 messaging address (63ms)

→ **keysmanager testing**

- ✓ only node operator can add validator (337ms)
- ✓ cannot add same validator again (84ms)

→ **Issuer testing**

- ✓ user should not stake less than 0 (161ms)
- ✓ user should be able to stake and get stkETH (101ms)
- ✓ user should be able to stake WETH and get stkETH (1973ms)
- ✓ user should be able to get stkETH on optimism (5601ms)
- ✓ user should be able to get stkETH on optimism by staking WETH (1999ms)
- ✓ user should be able to transfer stkETH on Optimism (2082ms)

- ✓ user should be able to get stkETH on Arbitrum (5240ms)
- ✓ user should be able to get stkETH on Arbitrum by staking WETH (868ms)
- ✓ user should be able to transfer stkETH on Arbitrum (2599ms)
- ✓ should not make deposit for validator when less than 32 eth in pool (72ms)
- ✓ should only make deposit for validator when key is activated (390ms)

→ Oracle Testing

- ✓ fetch beacon data
- ✓ push data for validator activation (277ms)
- ✓ No exit validators and no slashing (405ms)
- ✓ should update c value on Optimism and Arbitrum (3230ms)
- ✓ no slashing and wrong validator exiting (112ms)
- ✓ only slashing less than 1 eth accepted (92ms)
- ✓ delta balance more than minimum exit balance (159ms)
- ✓ exit validator with no slashing (338ms)
- ✓ slashing less than rewards (393ms)
- ✓ slashing more than rewards (212ms)

28 passing

→ Fuzz Tests

- ✓ testFuzz_stake(uint96) (runs: 256, μ : 153567, : 153567)

- ✓ testFuzz_stakeOnArbitrum(uint96) (runs: 256, μ : 219633, : 219633)
 - ✓ testFuzz_stakeOnOptimism(uint96) (runs: 256, μ : 654328, : 654328)
 - ✓ testFuzz_transferToArbitrum(uint96) (runs: 256, μ : 696284, : 696284)
 - ✓ testFuzz_transferToOptimism(uint96) (runs: 256, μ : 677649, : 677649)
- 5 passing

5.2 L2-contracts

→ Receive L1 Transaction

- ✓ contracts deploy successfully (143ms)
- ✓ only admin able to add minter, burner and l1Messgae addresses (544ms)
- ✓ only minter able to min (141ms)
- ✓ only l2 message contract can change price per share (107ms)
- ✓ user stake to get stkEth (606ms)

→ Receive L1 Transaction

- ✓ transfer Eth to mainnet successfully using socket (1885ms)

Conclusion:

The project offers a testing mechanism to improve the correctness of smart contracts; nonetheless, the number of tested scenarios are low; therefore, we advise on resolving this issue by covering more scenarios to handle most of the edge cases, in order to guarantee the integrity of the code and the functionality of the protocol.

6 Conclusion

In this audit, we examined the design and implementation of pStake Finance contract and discovered several issues of varying severity. Persistence team addressed 14 issues raised in the initial report and implemented the necessary fixes, while classifying the rest as a risk with low-probability of occurrence. Shellboxes' auditors advised Persistence Team to maintain a high level of vigilance and to keep those findings in mind in order to avoid any future complications.

7 Scope Files

7.1 Audit

Files	MD5 Hash
L1-contracts/contracts/Core.sol	480942fe2f929c558ef4f42e6687f89b
L1-contracts/contracts/CoreRef.sol	ca5d70f244774cd8173431a7398ecf7a
L1-contracts/contracts/IssuerUpgradable.sol	58f048e2bec0a1778887efa672253375
L1-contracts/contracts/KeysManager.sol	6f89a5be319402db1f50c9c1e90d8ec0
L1-contracts/contracts/Oracle.sol	b0707b809d0790f1c331f3f41c532341
L1-contracts/contracts/Permissions.sol	b625e559ec2e81856577dd5f18069ab5
L1-contracts/contracts/PriceOracle.sol	a20cd44a8b1287fc3a1b82be6f67e285
L1-contracts/contracts/StakingPool.sol	2e6eb81c4814cf53df2b5a71fe3eb4ee
L1-contracts/contracts/TimeLockController.sol	035e8800904d1f7554276ef4ffddda39
L1-contracts/contracts/WithdrawalCredential.sol	8cd0002e96af2b70b828841ab27ff14f
L1-contracts/contracts/token/StkEth.sol	a04a19e80f887f4cae0cc05b0e313d60
L1-contracts/contracts/messenger/ArbitrumMessenger.sol	324b65b7ac4846bb65ea073a1315ac44
L1-contracts/contracts/messenger/L1MessengerBase.sol	2c14cb6ec58b2f05d5d35b5ee716906e
L1-contracts/contracts/messenger/OptimismMessenger.sol	356c30f6bbb996412ac7873483079b9b
L1-contracts/contracts/library/BeaconData.sol	66539115a3844afc29324b8c9acf1ede

L2-contracts/contracts/Issuer.sol	7e50ff6318f1035d13820b3ed2a90736
L2-contracts/StkEth.sol	361ae6a1d670f5332f8d32842bb3fedd
L2-contracts/TimeLockController.sol	035e8800904d1f7554276ef4ffddda39
L2-contracts/optimism/L2MessageContractOptimism.sol	bf2211deb2cfee133854e90e9a4a7fc2
L2-contracts/arbitrum/L2MessageContractArbitrum.sol	e51de3b85e54e15a20a4b72ae1d84dd3

7.2 Re-Audit

Files	MD5 Hash
L1-contracts/contracts/Core.sol	480942fe2f929c558ef4f42e6687f89b
L1-contracts/contracts/CoreRef.sol	ca5d70f244774cd8173431a7398ecf7a
L1-contracts/contracts/IssuerUpgradable.sol	7094b28e372e01e4f4be515db61c1f0d
L1-contracts/contracts/KeysManager.sol	66d277a64dd13a52e4c5a5daba289b20
L1-contracts/contracts/Oracle.sol	9f442cb55de8c93d34acd48ce2787599
L1-contracts/contracts/Permissions.sol	b625e559ec2e81856577dd5f18069ab5
L1-contracts/contracts/PriceOracle.sol	a20cd44a8b1287fc3a1b82be6f67e285
L1-contracts/contracts/StakingPool.sol	d6fe6b1dfcb673f5e06447fb257e7f85
L1-contracts/contracts/TimeLockController.sol	035e8800904d1f7554276ef4ffddda39
L1-contracts/contracts/WithdrawalCredential.sol	cf5e699b004979f53f45d1411eabf19d
L1-contracts/contracts/token/StkEth.sol	a04a19e80f887f4cae0cc05b0e313d60

L1-contracts/contracts/messenger/ArbitrumMessenger.sol	bb76b8aa4b87beadf2dd4f7cca29dd67
L1-contracts/contracts/messenger/L1MessengerBase.sol	2c14cb6ec58b2f05d5d35b5ee716906e
L1-contracts/contracts/messenger/OptimismMessenger.sol	d4ea22d7c988335442abfe89f9a71e66
L1-contracts/contracts/library/BeaconData.sol	66539115a3844afc29324b8c9acf1ede
L2-contracts/contracts/Issuer.sol	ad509754109dd238adab9c9ec89dc44d
L2-contracts/contracts/StkEth.sol	361ae6a1d670f5332f8d32842bb3fedd
L2-contracts/contracts/TimeLockController.sol	035e8800904d1f7554276ef4ffddda39
L2-contracts/contracts/optimism/L2MessageContractOptimism.sol	bf2211deb2cfee133854e90e9a4a7fc2
L2-contracts/contracts/arbitrum/L2MessageContractArbitrum.sol	a805d73c20c711e18372f1b08c6d6406

8 Disclaimer

Shellboxes reports should not be construed as “endorsements” or “disapprovals” of particular teams or projects. These reports do not reflect the economics or value of any “product” or “asset” produced by any team or project that engages Shellboxes to do a security evaluation, nor should they be regarded as such. Shellboxes Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the examined technology, nor do they provide any indication of the technology’s proprietors, business model, business or legal compliance. Shellboxes Reports should not be used in any way to decide whether to invest in or take part in a certain project. These reports don’t offer any kind of investing advice and shouldn’t be used that way. Shellboxes Reports are the result of a thorough auditing process designed to assist our clients in improving the quality of their code while lowering the significant risk posed by blockchain technology. According to Shellboxes, each business and person is in charge of their own due diligence and ongoing security. Shellboxes does not guarantee the security or functionality of the technology we agree to research; instead, our purpose is to assist in limiting the attack vectors and the high degree of variation associated with using new and evolving technologies.



For a Contract Audit, contact us at contact@shellboxes.com