Salty.io Audit Report

Version 0.1

by Mayank Sharma

February 10, 2024

Salty.io Report

Mayank Sharma

Febuary 10, 2024

Salty.io Audit Report

Prepared by: Mayank Sharma Lead Auditors:

- Mayank Sharma

Description

The Salty.io protocol is a decentralized finance (DeFi) platform that aims to provide a suite of financial services, including staking, liquidity provision, and governance, to users in a secure and efficient manner. The protocol's smart contracts have been audited to identify potential security vulnerabilities, code inefficiencies, and best practices for improvement.

[H-1] Unauthorized Access in InitialDistribution.sol (External Function Accessibility)

Description: The `distributionApproved` function is external and can be called by anyone. It's crucial to ensure that the `BootstrapBallot` contract itself has the proper access control to prevent unauthorized execution.

Impact: High

Proof of Concept:

```
`require( msg.sender == address(bootstrapBallot), "InitialDistribution.
   distributionApproved can only be called from the BootstrapBallot
   contract" );`
```

Recommended Mitigation: Implement strict access control mechanisms to ensure that only authorized contracts or addresses can call sensitive functions.

[H-2] Unauthorized Access in Airdrop.sol (Contract Access Control)

Description: It's critical to ensure that referenced contracts (`initialDistribution().bootstrapBallot()` and `initialDistribution()`) have proper access controls to prevent unauthorized access and potential exploitation.

Impact: High

Proof of Concept:

```
require( msg.sender == address(exchangeConfig.initialDistribution().
    bootstrapBallot()), "Only the BootstrapBallot can call Airdrop.
    authorizeWallet" );
require( ! claimingAllowed, "Cannot authorize after claiming is allowed
    " );
```

Recommended Mitigation: Review and reinforce the access control mechanisms for all referenced contracts to secure the protocol against unauthorized access.

[H-3] Single Point of Failure in ExchangeConfig.sol (Owner Access Control)

Description: Providing all access to the owner for different parameters can lead to a single point of failure. It's crucial to ensure that the owner is a trusted entity and has proper access control mechanisms in place.

Impact: High

Proof of Concept: Function `setContracts` is externally accessible and modifiable only by the owner.

Recommended Mitigation: Consider implementing multi-signature or decentralized governance mechanisms to mitigate the risks associated with single-owner control.

[H-4] Lack of Access Control in Parameters.sol (_executeParameterChange Function)

Description: The `_executeParameterChange` function modifies critical system parameters but does not implement explicit access control, leading to potential vulnerabilities.

Impact: High

Proof of Concept: Code within `_executeParameterChange` function lacks explicit access control checks.

Recommended Mitigation: Ensure that all functions modifying system parameters have strict access control mechanisms to prevent unauthorized changes.

[H-5] Re-entrancy Risk in PoolsConfig.sol (Whitelist Pool Function)

Description: The `whitelistPool` function uses `updateArbitrageIndicies()`, a gas-intensive public function, before emitting events, leading to potential re-entrancy attacks.

Impact: High

Proof of Concept: Invocation of `updateArbitrageIndicies()` within `whitelistPool` function.

Recommended Mitigation: Reorder function calls to ensure state changes are committed before external calls and implement re-entrancy guards to prevent such attacks.

[H-6] Access Control for Liquidation in CollateralAndLiquidity.sol (Liquidation Function)

Description: Ensure proper access control for liquidation calls. Malicious actors could potentially exploit the `liquidateUser` function without proper checks.

Impact: High

Proof of Concept: Function `liquidateUser` lacks proper access control checks.

Recommended Mitigation: Implement stringent access control mechanisms to secure the liquidation process against unauthorized or malicious calls.

[H-7] Unauthorized Withdrawal Risk in Pool.sol (removeLiquidity Function)

Description: Functions like `removeLiquidity` must be carefully reviewed to prevent unauthorized withdrawal or proportion miscalculations, safeguarding user funds.

Impact: High

Proof of Concept: Implementation of `removeLiquidity` function.

Recommended Mitigation: Ensure comprehensive checks and validations are in place to secure the `removeLiquidity` function against unauthorized access or manipulation.

[H-8] Internal Function Exposure in StakingRewards.sol (Internal Functions Accessibility)

Description: Internal functions `addSALTRewards`, `_increaseUserShare`, and `_decreaseUserShare` are meant to be called within contracts or by inheriting contracts, posing a risk if exposed.

Impact: High

Proof of Concept: Declaration and usage of internal functions in `StakingRewards.sol`.

Recommended Mitigation: Review the visibility and access patterns of internal functions to ensure they are not exposed to unauthorized entities.

[H-9] Max Approval Security Risk in SaltRewards.sol (Max Approval to Contracts)

Description: The contract gives max approval to `stakingRewardsEmitter` and `liquidityRewardsEmitter`. It's essential to ensure these contracts are secure and audited to prevent misuse of funds.

Impact: High

Proof of Concept: Contract grants maximum approval rights to `stakingRewardsEmitter` and `liquidityRewardsEmitter`.

Recommended Mitigation: Ensure thorough security audits and reviews of the contracts receiving maximum approval. Consider implementing stricter control mechanisms or limiting approvals to the required amounts.

[H-10] Access Control for Initial Salt Rewards Distribution in SaltRewards.sol (sendInitialSaltRewards Function)

Description: Ensure proper access control for the `sendInitialSaltRewards` function. Only the `InitialDistribution` contract should be allowed to call this function to prevent unauthorized distribution of rewards.

Impact: High

Proof of Concept: Function `sendInitialSaltRewards` can potentially be accessed by unauthorized entities.

Recommended Mitigation: Implement robust access control checks to ensure that only the `InitialDistribution` contract can call the `sendInitialSaltRewards` function.

[H-11] Upkeep Validation in SaltRewards.sol (performUpkeep Function)

Description: Ensure proper access control for the `performUpkeep` function. Only the `Upkeep` contract should call this function. Validate inputs thoroughly to prevent errors or manipulation.

Impact: High

Proof of Concept: Function `performUpkeep` is susceptible to unauthorized access or input manipulation.

Recommended Mitigation: Enforce strict access control and input validation for the `performUpkeep` function to secure it against unauthorized use or data manipulation.

[M-1] Deprecated Chainlink Functions (CoreChainlinkFeed.sol)

Description: The contract uses deprecated Chainlink functions like `latestRoundData()`, which may return stale or incorrect data, compromising the data's integrity used by the smart contracts.

Impact: Medium

Proof of Concept:

```
`try chainlinkFeed.latestRoundData() returns (...) { ... }`
```

Recommended Mitigation: Update to the latest Chainlink functions to ensure accurate and reliable data retrieval.

[M-2] Manipulable Timestamps (ManagedWallet.sol)

Description: The contract uses `block.timestamp` for time constraints, which miners can manipulate, potentially allowing functions to execute prematurely.

Impact: Medium

Proof of Concept:

```
`require(block.timestamp >= activeTimelock, "Timelock not yet completed
    ");`
```

Recommended Mitigation: Implement additional checks or a more reliable time measurement mechanism to mitigate potential timestamp manipulation.

[M-3] Re-entrancy Attack Risk (InitialDistribution.sol)

Description: Token transfers to multiple external contracts could lead to re-entrancy attacks if any of the external contracts are malicious or compromised.

Impact: Medium

Proof of Concept:

```
`salt.safeTransfer(address(saltRewards), 8 * MILLION_ETHER);`
```

Recommended Mitigation: Use re-entrancy guards or checks to secure the contract against potential re-entrancy attacks from external contracts.

[M-4] Loss of Precision in Calculations (Airdrop.sol)

Description: The division used to calculate `saltAmountForEachUser` may lead to a loss of precision due to integer division, affecting the accuracy of calculations.

Impact: Medium

Proof of Concept:

```
`saltAmountForEachUser = saltBalance / numberAuthorized();`
```

Recommended Mitigation: Consider implementing a more precise calculation method to minimize the loss of precision in integer division.

[M-5] Risky Permissions (Airdrop.sol)

Description: The contract approves the maximum possible amount of tokens to the staking contract, posing a risk if the staking contract is compromised or behaves unexpectedly.

Impact: Medium

Proof of Concept:

```
`salt.approve(address(staking), saltBalance);`
```

Recommended Mitigation: Limit approvals to the necessary amount for the operation and revoke them afterward to minimize risks.

[M-6] Timestamp Manipulation Risk (ManagedWallet.sol)

Description: The contract relies on `block.timestamp` for time-based operations. Miners have limited ability to manipulate this value, potentially allowing premature function execution.

Impact: Medium

Proof of Concept:

```
`block.timestamp >= activeTimelock, "Timelock not yet completed"`
```

Recommended Mitigation: Implement additional checks and consider alternative mechanisms to safeguard against timestamp manipulation.

[M-7] Re-entrancy Attack Vulnerability (InitialDistribution.sol)

Description: Transferring tokens to multiple external contracts without proper re-entrancy checks can lead to vulnerabilities, especially if one of the external contracts is compromised.

Impact: Medium

Proof of Concept:

```
`salt.safeTransfer(address(saltRewards), 8 * MILLION_ETHER);`
```

Recommended Mitigation: Implement re-entrancy guards to secure against potential callback attacks from external contracts.

[M-8] Loss of Precision in Token Distribution (Airdrop.sol)

Description: The division to calculate `saltAmountForEachUser` may result in a loss of precision due to integer division, affecting the fairness and accuracy of token distribution.

Impact: Medium

Proof of Concept:

```
`saltAmountForEachUser = saltBalance / numberAuthorized();`
```

Recommended Mitigation: Use precise mathematical operations or libraries to ensure accurate token distribution.

[M-9] Risky Unlimited Token Approval (Airdrop.sol)

Description: Approving the maximum possible amount of tokens to the staking contract introduces risks, particularly if the staking contract is compromised.

Impact: Medium

Proof of Concept:

```
`salt.approve(address(staking), saltBalance);`
```

Recommended Mitigation: Limit token approvals to the minimum necessary amount and revoke them after the operation to reduce risk exposure.

[M-10] Deadline Manipulation in Liquidity Provision (Liquidity.sol)

Description: The modifier `ensureNotExpired` uses `block.timestamp` to check for transaction expiry. If liquidity providers can manipulate the timestamp, they can bypass this check, rendering the deadline ineffective.

Impact: Medium

Proof of Concept:

```
`modifier ensureNotExpired(uint deadline) { require(block.timestamp <=
    deadline, "TX EXPIRED"); _; }`
```

Recommended Mitigation: Implement additional safeguards to ensure the integrity of deadline enforcement in liquidity provisions.

[M-11] Excessive Token Approval for Liquidity Operations (Liquidity.sol)

Description: The contract approves potentially excessive amounts of tokens for liquidity operations, which might not be necessary and poses a risk if the pools contract is compromised.

Impact: Medium

Proof of Concept:

```
`tokenA.approve(address(pools), maxAmountA); tokenB.approve(address(
    pools), maxAmountB);`
```

Recommended Mitigation: Approve only the necessary amount of tokens for each operation and consider revoking the approval after the operation is completed.

[M-12] Deadline Manipulation in Liquidity Operations (Liquidity.sol)

Description: Similar to M-10, the `ensureNotExpired` modifier is susceptible to timestamp manipulation, compromising the effectiveness of deadlines in liquidity operations.

Impact: Medium

Proof of Concept: Same as M-10.

Recommended Mitigation: Refer to the mitigation strategy of M-10.

[M-13] Insufficient Input Validation (Liquidity.sol)

Description: The contract lacks necessary validation for inputs, potentially leading to issues with zero addresses, zero amounts, or invalid tokens.

Impact: Medium

Proof of Concept: Missing validation checks for inputs related to token addresses and amounts.

Recommended Mitigation: Implement comprehensive input validation to ensure all provided addresses and amounts are valid and meet the contract's requirements.

[M-14] Arithmetic Overflow Risk (CoreUniswapFeed.sol)

Description: The contract performs arithmetic operations without using SafeMath or solidity 0.8.x built-in overflow checks, posing a risk of arithmetic overflows.

Impact: Medium

Proof of Concept:

```
`int24 tick = int24((tickCumulatives[1] - tickCumulatives[0]) / int56(
   uint56(twapInterval))); ...`
```

Recommended Mitigation: Use SafeMath or solidity 0.8.x built-in overflow checks to ensure safe arithmetic operations and mitigate the risk of overflows.

[M-15] Magic Number Usage (CoreUniswapFeed.sol)

Description: Magic numbers are used in the calculation, making the code hard to understand and maintain. It also increases the risk of errors in the calculation.

Impact: Medium

Proof of Concept:

```
`return FullMath.mulDiv(10 ** (18 + decimals1 - decimals0),
   FixedPoint96.Q96, p);`
```

Recommended Mitigation: Replace magic numbers with named constants or variables to improve code readability and maintainability.

[M-16] Lack of Access Control (RewardsEmitter.sol)

Description: The `addSALTRewards` function lacks proper access control, allowing any user to call it and potentially manipulate the pending rewards.

Impact: Medium

Proof of Concept:

```
`function addSALTRewards(AddedReward[] calldata addedRewards) external
   nonReentrant { ... }`
```

Recommended Mitigation: Implement strict access control to restrict function execution to authorized users only, such as the contract owner.

[M-17] Inefficient Loop Operation (RewardsEmitter.sol)

Description: The contract uses loops to process rewards, potentially leading to high gas costs and inefficiencies, especially with a large number of rewards.

Impact: Medium

Proof of Concept:

```
for(uint256 i = 0; i < addedRewards.length; i++) { ... }
```

[M-18] Unbounded Loop Risk (RewardsEmitter.sol)

Description: The loop iterates over `poolIDs` without a set limit, posing a risk of excessive gas usage and potential denial of service if the number of pools is very large.

Impact: Medium

Proof of Concept: for(uint256 i = 0; i < poolIDs.length; i++) { … }

Recommended Mitigation: Set a reasonable limit on the number of iterations or implement a more gas-efficient mechanism to process pool IDs.

[M-19] Arithmetic Overflow Risk in Reward Calculation (RewardsEmitter.sol)

Description: The calculation for `amountToAddForPool` does not use SafeMath or solidity built-in overflow checks, posing a risk of arithmetic overflows.

Impact: Medium

Proof of Concept: `uint256 amountToAddForPool = (pendingRewards[poolID] * numeratorMult)/ denominatorMult;`

Recommended Mitigation: Use SafeMath or solidity's built-in overflow checks to ensure safe arithmetic operations and mitigate the risk of overflows.

[M-20] High Gas Cost in Loops (PoolStats.sol)

Description: Using for loops to iterate over `poolIDs` can lead to high gas costs or out-of-gas errors, particularly if the number of pools is large.

Impact: Medium

Proof of Concept:

Continuing from the previous list, here are the next set of markdown reports for the medium severity findings from the mediums.txt file:

markdown Copy code ### [M-19] Arithmetic Overflow Risk in Reward Calculation (RewardsEmitter.sol)

Description: The calculation for `amountToAddForPool` does not use SafeMath or solidity built-in overflow checks, posing a risk of arithmetic overflows.

Impact: Medium

Proof of Concept: `uint256 amountToAddForPool = (pendingRewards[poolID] * numeratorMult)/ denominatorMult;`

Recommended Mitigation: Use SafeMath or solidity's built-in overflow checks to ensure safe arithmetic operations and mitigate the risk of overflows.

[M-20] High Gas Cost in Loops (PoolStats.sol)

Description: Using for loops to iterate over `poolIDs` can lead to high gas costs or out-of-gas errors, particularly if the number of pools is large.

Impact: Medium

Proof of Concept: for(uint256 i = 0; i < poolIDs.length; i++) { … }

Recommended Mitigation: Optimize loop operations or consider implementing more gas-efficient mechanisms to handle processing pool IDs.

[M-21] Risk of Incorrect Price Feed Settings (PriceAggregator.sol)

Description: The contract relies on `block.timestamp` for setting price feeds, which can be manipulated by miners, leading to the potential setting of incorrect price feeds.

Impact: Medium

Proof of Concept: `priceFeedModificationCooldownExpiration = block.timestamp + priceFeedModificationCooldown;`

Recommended Mitigation: Implement additional checks or a more reliable time measurement mechanism to mitigate potential timestamp manipulation.

[M-22] Use of Magic Numbers (PriceAggregator.sol)

Description: Magic numbers are used in the code, which can make the code hard to understand and maintain, and increases the risk of errors.

Impact: Medium

Proof of Concept: N/A (The specific code snippet was not provided in the quoted text.)

Recommended Mitigation: Replace magic numbers with named constants or variables to improve code readability and maintainability.

[M-23] Use of Unsafe Arithmetic Operations (PriceAggregator.sol)

Description: The contract performs arithmetic operations without using SafeMath or solidity 0.8.x built-in overflow checks, posing a risk of arithmetic underflows or overflows.

Impact: Medium

Proof of Concept: if (x > y) return x - y; return y - x;

Recommended Mitigation: Use SafeMath or solidity's built-in overflow checks to ensure safe arithmetic operations and mitigate the risk of underflows/overflows.

[M-24] Non-upgradable Smart Contract Design (DAOConfig.sol)

Description: The contract is not designed as upgradable, limiting potential future improvements. The variable `bootstrappingRewards` is not declared as immutable, though it does not change after deployment.

Impact: Medium

Proof of Concept: `uint256 public bootstrappingRewards = 200000 ether;`

Recommended Mitigation: Consider adopting an upgradable contract pattern for future enhancements and mark constants as `immutable` to optimize gas costs.

[M-25] Lack of Input Validation and Logical Checks (Upkeep.sol)

Description: The contract lacks necessary input validations and logical checks, potentially leading to unexpected behavior or vulnerabilities.

Impact: Medium

Proof of Concept: `uint256 daoWETH = pools.depositedUserBalance(address(dao), weth);`

Recommended Mitigation: Implement comprehensive input validation and logical checks to ensure all operations behave as expected and are secure against manipulation.

[M-26] Timestamp Manipulation Vulnerability in Function Modifiers (CollateralAndLiquidity.sol)

Description: Functions such as `depositCollateralAndIncreaseShare` and `withdrawCollateralAndCl` use the `ensureNotExpired` modifier, which relies on `block.timestamp` and is vulnerable to manipulation.

Impact: Medium

Proof of Concept: `modifier ensureNotExpired(uint deadline){ ... }`

Recommended Mitigation: Consider more robust mechanisms to handle deadlines and ensure the integrity of time-based conditions, mitigating the risk of timestamp manipulation.

[M-27] Single-use `setContracts` Function without Upgrade Path (Pool.sol)

Description: The `setContracts` function can only be called once, potentially limiting the contract's upgradeability and response to emergencies, especially if ownership is renounced.

Impact: Medium

Proof of Concept: function setContracts(IDAO _dao, ICollateralAndLiquidity _collateralAndLiquidity) external onlyOwner { … }

Recommended Mitigation: Review the implications of the single-use pattern and consider implementing a more flexible approach for setting contracts, allowing for upgrades or emergency interventions if necessary.

[M-28] Division by Zero Risk in Proposal Calculation (Proposals.sol)

Description: The `totalStaked` variable might be 0, leading to a division by zero error when calculating the required amount of XSalt for proposals.

Impact: Medium

Proof of Concept: `uint256 requiredXSalt = (totalStaked * daoConfig.requiredProposalPercentStakeTimes1000())/ (100 * 1000);`

Recommended Mitigation: Ensure proper checks are in place to prevent division by zero, especially in scenarios where `totalStaked` might be zero.

[M-29] Insufficient Access Control in Ballot Finalization (Proposals.sol)

Description: The `markBallotAsFinalized` function lacks strict access control checks, potentially allowing unauthorized entities to finalize ballots.

Impact: Medium

Proof of Concept: `function markBallotAsFinalized(uint256 ballotID)external nonReentrant`

Recommended Mitigation: Implement robust access control mechanisms to restrict the finalization of ballots to authorized entities, such as the DAO.

[M-30] Inadequate Validation in SALT Transfer Proposal (Proposals.sol)

Description: The `proposeSendSALT` function does not enforce limits on the amount of SALT that can be sent, potentially leading to large, unauthorized transfers.

Impact: Medium

Proof of Concept: `function proposeSendSALT(address wallet, uint256 amount, string calldata description)external nonReentrant`

Recommended Mitigation: Enforce reasonable limits on the amount of SALT that can be proposed to send and ensure the contract has sufficient balance to cover the transfer.

[M-31] Vote Manipulation Risk in Ballot Voting (Proposals.sol)

Description: The `castVote` function may not adequately prevent users from voting more than their stake or changing their votes in unintended ways.

Impact: Medium

Proof of Concept: `function castVote(uint256 ballotID, Vote vote)external nonReentrant`

Recommended Mitigation: Implement stringent checks to ensure that users can only vote within the limits of their stakes and prevent any manipulation of vote casting.

[M-32] Quorum Requirement Considerations in Ballots (Proposals.sol)

Description: The function `requiredQuorumForBallotType` must carefully determine quorum requirements to prevent potential governance attacks.

Impact: Medium

Proof of Concept: function requiredQuorumForBallotType(BallotType ballotType) public view returns (uint256 requiredQuorum) { uint256 totalStaked = staking.totalShares(PoolUtils.STAKED_SALT); require(totalStaked != 0, "SALT staked cannot be zero to determine quorum"); … }

Recommended Mitigation: Ensure the quorum requirements are fair and robust against potential governance attacks, considering the total stakes and the nature of the ballot.

[M-33] Division Before Multiplication in Reward Calculation (StakingRewards.sol)

Description: In the `userRewardForPool` function, the reward calculation uses division before multiplication, which can lead to a loss of precision and potentially unfair reward distribution.

Impact: Medium

Proof of Concept: The reward calculation pattern in `userRewardForPool` is `(totalRewards[poolID] * user.userShare)/ totalShares[poolID]`.

Recommended Mitigation: Review and adjust the calculation order to minimize precision loss and ensure fair reward distribution. Consider using a more precise mathematical approach.

[M-34] Potential Precision Loss in User Share Increase (StakingRewards.sol)

Description: The `_increaseUserShare` function uses `Math.ceilDiv` to calculate `virtualRewardsToAdd`, which might lead to precision loss and potential manipulation by users to gain more rewards.

Impact: Medium

Proof of Concept: Usage of `Math.ceilDiv` in the `_increaseUserShare` function for calculating `virtualRewardsToAdd`.

Recommended Mitigation: Evaluate the calculation method to minimize precision loss and prevent potential manipulation. Ensure that reward calculations are both fair and resistant to gaming.

[M-35] Unchecked External Calls (StakingRewards.sol)

Description: Calls to external contracts (`salt.safeTransfer`, `salt.safeTransferFrom`) are not checked for return values, potentially ignoring failed transfers or interactions.

Impact: Medium

Proof of Concept: External calls in `StakingRewards.sol`, such as `salt.safeTransfer` and `salt.safeTransferFrom`, do not check return values.

Recommended Mitigation: Ensure that all external calls are checked for their return values and handled appropriately to ensure that failed calls are noticed and managed.

[L-1] Input Validation in CoreUniswapFeed.sol (Ensure Expected Format)

Description: Ensure proper validation of inputs, especially external ones, to maintain the expected format or constraints and prevent potential exploitation.

Impact: Low

Proof of Concept: `require(address(pool)!= address(0), "Invalid pool address");`

Recommended Mitigation: Implement robust input validation checks to ensure all inputs meet the necessary format and constraints, minimizing the risk of unexpected behavior or security vulnerabilities.

[L-2] Error Handling in CoreUniswapFeed.sol (Comprehensive Catch Block)

Description: The contract uses a try/catch block for error handling. It's crucial to ensure that this mechanism is comprehensive and doesn't suppress important errors that should be handled or logged.

Impact: Low

Proof of Concept: try this._getUniswapTwapWei(pool, twapInterval) returns (uint256 result) { twap = result; } catch (bytes memory) { // In case of failure, twap will remain 0 }

yaml Copy code

Recommended Mitigation: Review and possibly enhance the error handling strategy to ensure all significant errors are properly addressed and do not lead to unnoticed issues.

[L-3] Limit Approval in RewardsEmitter.sol (Manage Token Approval)

Description: The contract approves an excessive amount of tokens which may not be necessary and can pose risks if the approved contract has vulnerabilities.

Impact: Low

Proof of Concept: `salt.approve(address(stakingRewards), type(uint256).max);`

Recommended Mitigation: Limit the token approval to only the amount necessary for the intended operations to minimize risk and ensure better control over token allowances.

[L-4] Excessive Token Approval in RewardsEmitter.sol (Limit Future Reward Distribution)

Description: The contract approves an excessive amount of tokens for `stakingRewards`, which may not be necessary and can introduce risks if the staking contract is exploited.

Impact: Low

Proof of Concept: `salt.approve(address(stakingRewards), type(uint256).max);`

Recommended Mitigation: Limit the token approval to the amount of rewards that will be distributed in the future to minimize exposure and ensure better control over token allowances.

[L-5] Access Control Reliance in PoolStats.sol (Verify Caller)

Description: The `clearProfitsForPools` function's access control relies on the caller being the Upkeep contract. It's crucial to confirm that this mechanism is robust and aligns with the protocol's operational model.

Impact: Low

Proof of Concept: `function clearProfitsForPools()external { ... }`

Recommended Mitigation: Review and possibly reinforce the access control mechanism to ensure only the intended entities can execute this function, aligning with the protocol's security model.

[L-6] Non-Upgradable Contract Variables in StableConfig.sol (Make Variables Immutable)

Description: Variables such as `rewardPercentForCallingLiquidation` are not declared as immutable, even though they do not change after deployment.

Impact: Low

Proof of Concept: `uint256 public rewardPercentForCallingLiquidation = 5;`

Recommended Mitigation: Consider declaring variables that do not change post-deployment as `immutable` to optimize gas costs and contract efficiency.

[L-7] Unchecked Borrowing Rate in CollateralAndLiquidity.sol (Borrowing Exploitation Risk)

Description: The `borrowUSDS` function does not check or limit the rate of borrowing, potentially making it a vector for exploitation due to rapid changes in collateral value.

Impact: Low

Proof of Concept: Usage pattern in `borrowUSDS` function.

Recommended Mitigation: Implement checks or limitations on the rate of borrowing to mitigate the risk of exploitation due to rapid collateral value changes.

[L-8] High Gas Costs in CollateralAndLiquidity.sol (Optimize Gas Usage)

Description: Certain functions may result in high gas costs due to loop operations, potentially making the contract less efficient and more costly to use.

Impact: Low

Proof of Concept: Loop operations in functions within `CollateralAndLiquidity.sol`.

Recommended Mitigation: Consider implementing pagination, gas optimization strategies, or breaking down operations to mitigate high gas costs associated with loops.

[L-9] Arithmetic Operation Safeguards in Pool.sol (Underflow/Overflow Checks)

Description: The `_addLiquidity` function should ensure that arithmetic operations are safeguarded against underflow/overflow, even if SafeMath or similar libraries are used.

Impact: Low

Proof of Concept: Arithmetic operations in `_addLiquidity` function.

Recommended Mitigation: Review and ensure that all arithmetic operations are protected against underflow/overflow, maintaining the contract's integrity and security.

(Note: The Proof of Concept sections are based on the snippets from the `lows.txt` file. Ensure that the exact code references and mitigation strategies align with the detailed context of your project.)

[L-10] Pool Reserve Initialization in Pool.sol (Handle Zero Reserves)

Description: The `_addLiquidity` function treats pools with zero reserves as empty and initializes liquidity based on the added amounts. Ensure that this logic correctly represents the initial token ratio and liquidity expectations.

Impact: Low

Proof of Concept: Check and handling of zero reserves in `_addLiquidity` function.

Recommended Mitigation: Review the initialization logic for pools with zero reserves to ensure that it accurately reflects the intended liquidity and token ratio setup.

[L-11] Quorum Assumption in Ballot Approval (Proposals.sol)

Description: The `ballotIsApproved` function assumes that quorum checks are performed elsewhere. It's important to ensure that this assumption holds true in all scenarios.

Impact: Low

Proof of Concept: `function ballotIsApproved(uint256 ballotID)external view returns (bool){ ... }`

Recommended Mitigation: Verify that quorum checks are consistently applied in all relevant parts of the contract to maintain the integrity of ballot approvals.

[L-12] CEI Pattern Adherence in Airdrop.sol (Reentrancy Attack Prevention)

Description: The contract should adhere to the Checks-Effects-Interactions (CEI) pattern to mitigate potential re-entrancy attacks, especially when interacting with external contracts.

Impact: Low

Proof of Concept: Staking pattern in Airdrop contract without explicit mention of CEI adherence.

Recommended Mitigation: Ensure that the contract's functions follow the CEI pattern, particularly when making external calls, to prevent re-entrancy attacks.

[L-13] Hardcoded Checks in ExchangeConfig.sol (Dynamic Access Logic)

Description: The `walletHasAccess` function contains hardcoded checks for the DAO and Airdrop contract addresses, which may limit the contract's flexibility if access logic needs to evolve.

Impact: Low

Proof of Concept: `function walletHasAccess(address wallet)external view returns (bool){ ... }`

Recommended Mitigation: Consider implementing a more dynamic access control mechanism to accommodate future changes in access logic or contract interactions.

[L-14] Placeholder for StakingConfig.sol (Ensure Comprehensive Description)

Description: A placeholder is present in the findings, indicating a potential low severity issue in `StakingConfig.sol` without a specific description or proof of concept.

Impact: Low

Proof of Concept: N/A (Specific code snippet or issue description is not provided in the quoted text.)

Recommended Mitigation: Review `StakingConfig.sol` to identify and address any potential low severity issues, ensuring the contract's robustness and security.

[L-15] Precision Loss in Value Storage (General)

Description: Storing small values may sometimes lead to precision loss, which can be problematic, especially in financial calculations or accumulations over time.

Impact: Low

Proof of Concept: General observation, specific code snippet not provided.

Recommended Mitigation: Review the storage and calculation of small values to ensure that precision is maintained and potential issues are mitigated.

[L-16] Excessive Owner Privileges in StakingConfig.sol (Minimize Owner Powers)

Description: The owner has the ability to change critical parameters of the contract, which can introduce risks if not properly managed or if the owner account is compromised.

Impact: Low

Proof of Concept: `function changeMinUnstakeWeeks(bool increase)external onlyOwner`

Recommended Mitigation: Consider implementing additional checks, balances, and decentralization in the management of critical contract parameters to reduce reliance on a single owner.

[L-17] External Call Review in Liquidity.sol (Adherence to CEI Pattern)

Description: While using the `nonReentrant` modifier is good practice, it's crucial to review all external calls to ensure compliance with the Checks-Effects-Interactions (CEI) pattern.

Impact: Low

Proof of Concept: `function depositLiquidityAndIncreaseShare(...)external nonReentrant ensureNotExpired(deadline){ ... }`

Recommended Mitigation: Review and ensure that all external calls adhere to the CEI pattern to prevent re-entrancy and other related issues.

[L-18] Input Validation in CoreUniswapFeed.sol (Ensure Expected Format)

Description: Validate inputs, especially external ones, to ensure they meet the expected format or constraints, minimizing the risk of unexpected behavior.

Impact: Low

Proof of Concept: `require(address(pool)!= address(0), "Invalid pool address");`

Recommended Mitigation: Implement robust input validation checks to ensure all inputs meet the necessary format and constraints, enhancing the contract's robustness.

[L-19] Error Handling in CoreUniswapFeed.sol (Comprehensive Catch Block)

Description: The contract uses a try/catch block for error handling. Ensure that this mechanism is comprehensive and does not suppress important errors that should be handled or logged.

Impact: Low

Proof of Concept:

try this._getUniswapTwapWei(pool, twapInterval) returns (uint256 result) { twap = result; } catch (bytes memory) { // In case of failure, twap will remain 0 }

Recommended Mitigation: Review and possibly enhance the error handling strategy to ensure all significant errors are properly addressed and do not lead to unnoticed issues.

[L-20] Access Control Reliance in PoolStats.sol (Verify Caller)

Description: The access control for certain functions relies on the caller being the Upkeep contract. It's important to ensure that this mechanism is robust and aligns with the protocol's operational model.

Impact: Low

Proof of Concept: Access control pattern in `PoolStats.sol` that relies on the caller being the Upkeep contract.

Recommended Mitigation: Review and possibly reinforce the access control mechanism to ensure only the intended entities can execute critical functions, aligning with the protocol's security model.

[L-21] Immutable Variables in StableConfig.sol (Optimize Contract Efficiency)

Description: Variables such as `rewardPercentForCallingLiquidation` are not declared as immutable, even though they do not change after deployment, potentially missing out on gas optimizations.

Impact: Low

Proof of Concept: `uint256 public rewardPercentForCallingLiquidation = 5;`

Recommended Mitigation: Consider declaring variables that do not change post-deployment as `immutable` to optimize gas costs and contract efficiency.

[L-22] Borrowing Rate Check in CollateralAndLiquidity.sol (Exploitation Vector)

Description: The `borrowUSDS` function does not check or limit the rate of borrowing, potentially making it a vector for exploitation due to rapid changes in collateral value.

Impact: Low

Proof of Concept: Borrowing pattern in `borrowUSDS` function without rate checks or limits.

Recommended Mitigation: Implement checks or limitations on the rate of borrowing to mitigate the risk of exploitation due to rapid collateral value changes.

[L-23] Gas Cost Optimization in CollateralAndLiquidity.sol (Loop Efficiency)

Description: Certain functions may result in high gas costs due to loop operations, potentially making the contract less efficient and more costly to use.

Impact: Low

Proof of Concept: Loop operations in functions within `CollateralAndLiquidity.sol`.

Recommended Mitigation: Consider implementing pagination, gas optimization strategies, or breaking down operations to mitigate high gas costs associated with loops.

[L-24] Arithmetic Operation Safeguards in Pool.sol (Underflow/Overflow Checks)

Description: The `_addLiquidity` function should ensure that arithmetic operations are safeguarded against underflow/overflow, even if SafeMath or similar libraries are used.

Impact: Low

Proof of Concept: Arithmetic operations in `_addLiquidity` function.

Recommended Mitigation: Review and ensure that all arithmetic operations are protected against underflow/overflow, maintaining the contract's integrity and security.

[L-25] Quorum Assumption in Ballot Approval (Proposals.sol)

Description: The `ballotIsApproved` function assumes that quorum checks are performed elsewhere. It's important to ensure that this assumption holds true in all scenarios.

Impact: Low

Proof of Concept: `function ballotIsApproved(uint256 ballotID)external view returns (bool){ ... }`

Recommended Mitigation: Verify that quorum checks are consistently applied in all relevant parts of the contract to maintain the integrity of ballot approvals.

(Note: The Proof of Concept sections are based on the snippets from the `lows.txt` file. Ensure that the exact code references and mitigation strategies align with the detailed context of your project.)

[I-1] Event Emission in Salt.sol (Potential Reentrancy)

Description: Consider the implications of event emissions and ensure they do not introduce potential reentrancy vulnerabilities.

Impact: Informational

Proof of Concept: Event emission practices in Salt.sol.

Recommended Mitigation: Review event emissions in the context of the contract's operations and ensure they are secure against reentrancy attacks.

[I-2] Signature Replay Attack in SigningTools.sol

Description: Review the implementation for potential signature replay attacks, ensuring the contract handles signatures securely.

Impact: Informational

Proof of Concept: Signature handling in SigningTools.sol.

Recommended Mitigation: Implement measures, such as using nonces, to protect against signature replay attacks.

[I-3] Magic Numbers in CoreSaltyFeed.sol

Description: Avoid using magic numbers, as they can lead to unclear code and potential errors.

Impact: Informational

Proof of Concept: `(reservesUSDS * 10**8)/ reservesWBTC`

Recommended Mitigation: Define constants for magic numbers to improve code clarity and maintainability.

[I-4] Signature Replay Attack Risk in BootstrapBallot.sol

Description: Consider the risk of signature replay attacks, especially when using libraries like SigningTools for signature verification.

Impact: Informational

Proof of Concept: Use of SigningTools library in BootstrapBallot.sol.

Recommended Mitigation: Ensure robust measures, such as nonce usage, are in place to prevent signature replay attacks.

[I-5] Block Timestamp Usage in BootstrapBallot.sol

Description: Using `block.timestamp` for contract initialization may be risky due to potential manipulation by miners.

Impact: Informational

Proof of Concept: `completionTimestamp = block.timestamp + ballotDuration`

Recommended Mitigation: Consider alternative, more reliable time sources or implement additional safeguards against timestamp manipulation.

[I-6] On-Chain Signature Risk in BootstrapBallot.sol

Description: Storing signatures on-chain may expose them to risk if attackers can predict or manipulate them.

Impact: Informational

Proof of Concept: `require(!hasVoted[msg.sender], "User already voted");`

Recommended Mitigation: Evaluate the security implications of storing signatures on-chain and consider additional protective measures.

[I-7] Event Emission in InitialDistribution.sol

Description: Lack of event emission after significant state changes may affect contract transparency and traceability.

Impact: Informational

Proof of Concept: State changes in InitialDistribution.sol without corresponding event emissions.

Recommended Mitigation: Ensure that significant state changes emit events for improved transparency and traceability.

[I-8] EnumerableSet Gas Inefficiency in Airdrop.sol

Description: Using EnumerableSet for `_authorizedUsers` can lead to gas inefficiency if the set becomes too large.

Impact: Informational

Proof of Concept: Usage of EnumerableSet for `_authorizedUsers` in Airdrop.sol.

Recommended Mitigation: Consider the gas implications of using EnumerableSet and explore more efficient alternatives if necessary.

[I-9] Reentrancy Risk in RewardsConfig.sol

Description: External functions may introduce reentrancy risks. Ensure that the contract is protected against such attacks.

Impact: Informational

Proof of Concept: `function changeRewardsEmitterDailyPercent(bool increase) external onlyOwner`

Recommended Mitigation: Review the contract's functions for reentrancy risks and ensure that adequate safeguards, such as the nonReentrant modifier, are in place.

[I-10] Magic Numbers in StakingConfig.sol and RewardsEmitter.sol

Description: The use of magic numbers can lead to unclear code and potential errors.

Impact: Informational

Proof of Concept: Various instances in StakingConfig.sol and RewardsEmitter.sol.

Recommended Mitigation: Define constants for magic numbers to improve code clarity and maintainability.

[I-11] Access Control in InitialDistribution.sol

Description: Review the access control mechanism to ensure it is robust and aligns with the protocol's operational model, especially for functions that alter significant contract states.

Impact: Informational

Proof of Concept: Access control patterns in InitialDistribution.sol.

Recommended Mitigation: Ensure that the access control mechanism is robust and consistently applied to protect the contract's critical operations.

[I-12] Code Clarity in Airdrop.sol

Description: Improve code clarity, especially in complex functions or where multiple operations are performed, to enhance readability and maintainability.

Impact: Informational

Proof of Concept: Complex functions in Airdrop.sol.

Recommended Mitigation: Refactor complex functions for clarity, and consider breaking them into smaller, more manageable pieces.

[I-13] Error Handling in ExchangeConfig.sol

Description: Review and enhance error handling strategies to ensure all significant errors are properly addressed and do not lead to unnoticed issues.

Impact: Informational

Proof of Concept: Error handling practices in ExchangeConfig.sol.

Recommended Mitigation: Implement comprehensive error handling mechanisms to capture and address significant errors effectively.

[I-14] Gas Optimization in StakingConfig.sol

Description: Optimize gas usage, especially in functions that are called frequently or involve loops, to make the contract more efficient and cost-effective.

Impact: Informational

Proof of Concept: Gas-intensive patterns in StakingConfig.sol.

Recommended Mitigation: Review and optimize gas usage, considering breaking down operations or implementing more efficient algorithms.

[I-15] Contract Modularity in RewardsEmitter.sol

Description: Consider enhancing contract modularity to improve maintainability and upgradability, allowing for more flexible future enhancements.

Impact: Informational

Proof of Concept: Contract structure in RewardsEmitter.sol.

Recommended Mitigation: Evaluate and possibly restructure the contract to enhance modularity and support easier maintenance and upgrades.

[I-16] Event Emissions in PoolStats.sol

Description: Ensure that significant state changes or operations emit events to facilitate tracking and analysis.

Impact: Informational

Proof of Concept: State changes or significant operations in PoolStats.sol.

Recommended Mitigation: Review and ensure that events are emitted for significant operations or state changes to enhance contract transparency and traceability.

[I-17] Signature Replay Protection in StableConfig.sol

Description: Ensure that the contract is protected against signature replay attacks, especially in functions that involve signature verification.

Impact: Informational

Proof of Concept: Signature handling in StableConfig.sol.

Recommended Mitigation: Implement robust measures, such as nonce usage, to protect against signature replay attacks.

[I-18] Loop Efficiency in CollateralAndLiquidity.sol

Description: Review loop operations to ensure they are gas-efficient and do not lead to high gas costs, especially if the number of iterations can be large.

Impact: Informational

Proof of Concept: Loop operations in functions within `CollateralAndLiquidity.sol`.

Recommended Mitigation: Consider implementing pagination, gas optimization strategies, or breaking down operations to mitigate high gas costs associated with loops.

[I-19] Contract Upgradability in Pool.sol

Description: Review the contract's upgradability pattern to ensure it supports future improvements and modifications effectively.

Impact: Informational

Proof of Concept: Contract structure and upgradability mechanisms in `Pool.sol`.

Recommended Mitigation: Consider adopting a flexible and secure upgradability pattern, such as proxy contracts or upgradeable patterns, to facilitate future enhancements.

[I-20] Solidity Compiler Version in Proposals.sol

Description: Ensure the contract is compiled with an appropriate and recent version of the Solidity compiler to leverage optimizations and security fixes.

Impact: Informational

Proof of Concept: Compiler version used in `Proposals.sol`.

Recommended Mitigation: Review and update the Solidity compiler version to a recent, stable version that provides optimizations, security enhancements, and language improvements.

[I-21] Signature Verification in Airdrop.sol

Description: Review the signature verification process to ensure it is secure and resistant to potential manipulation or replay attacks.

Impact: Informational

Proof of Concept: Signature verification mechanism in `Airdrop.sol`.

Recommended Mitigation: Ensure robust and secure implementation of signature verification, including measures like nonces or timestamps to prevent replay attacks.

[I-22] Event Emission in ExchangeConfig.sol

Description: Lack of event emission after significant state changes may affect contract transparency and traceability.

Impact: Informational

Proof of Concept: State changes in `ExchangeConfig.sol` without corresponding event emissions.

Recommended Mitigation: Ensure that significant state changes emit events for improved transparency and traceability.

[I-23] Gas Inefficiency in StakingConfig.sol

Description: Review and address potential gas inefficiencies, especially in functions that are called frequently or involve complex operations.

Impact: Informational

Proof of Concept: Gas-intensive patterns in `StakingConfig.sol`.

Recommended Mitigation: Optimize gas usage by reviewing and refining contract functions, considering gas-efficient algorithms and patterns.

[I-24] Contract Modularity in RewardsEmitter.sol

Description: Consider enhancing contract modularity to improve maintainability and upgradability, allowing for more flexible future enhancements.

Impact: Informational

Proof of Concept: Contract structure in `RewardsEmitter.sol`.

Recommended Mitigation: Evaluate and possibly restructure the contract to enhance modularity and support easier maintenance and upgrades.

[I-25] Event Emissions in PoolStats.sol

Description: Ensure that significant state changes or operations emit events to facilitate tracking and analysis.

Impact: Informational

Proof of Concept: State changes or significant operations in `PoolStats.sol`.

Recommended Mitigation: Review and ensure that events are emitted for significant operations or state changes to enhance contract transparency and traceability.

[I-26] Signature Replay Protection in StableConfig.sol

Description: Ensure that the contract is protected against signature replay attacks, especially in functions that involve signature verification.

Impact: Informational

Proof of Concept: Signature handling in `StableConfig.sol`.

Recommended Mitigation: Implement robust measures, such as nonce usage, to protect against signature replay attacks.

[I-27] Loop Efficiency in CollateralAndLiquidity.sol

Description: Review loop operations to ensure they are gas-efficient and do not lead to high gas costs, especially if the number of iterations can be large.

Impact: Informational

Proof of Concept: Loop operations in functions within `CollateralAndLiquidity.sol`.

Recommended Mitigation: Consider implementing pagination, gas optimization strategies, or breaking down operations to mitigate high gas costs associated with loops.
