// HALBORN

Lodestar Finance – Staking

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **July 24th, 2023 - July 28th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	07/24/2023	Roberto Reigada
0.2	Document Updates	07/28/2023	Roberto Reigada
0.3	Draft Review	07/28/2023	Gokberk Gulgun
0.4	Draft Review	07/28/2023	Gabi Urrutia
0.5	Document Updates	08/07/2023	Roberto Reigada
0.6	Added fuzzing section	08/07/2023	Roberto Reigada
1.0	Remediation Plan	08/14/2023	Roberto Reigada
1.1	Remediation Plan Review	08/14/2023	Gokberk Gulgun
1.2	Remediation Plan Review	08/14/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Gokberk Gulgun	Halborn	Gokberk.Gulgun@halborn.com
Roberto Reigada	Halborn	Roberto.Reigada@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Lodestar Finance is an algorithmic borrowing and lending protocol built on the Arbitrum network. It aims to bring decentralized money markets to Arbitrum communities, enabling users to earn interest in lending assets and access liquidity through collateralized borrowing. The protocol's goals include expanding lending services to emerging cryptocurrency communities and collaborating with layer 2 native communities. The security assessment report focuses on the staking rewards, voting power, and governance contracts to ensure their security, functionality, and compliance with the industry best practices.

Lodestar engaged Halborn to conduct a security assessment on their smart contracts beginning on July 24th, 2023 and ending on July 28th, 2023. The security assessment was scoped to the smart contracts provided in the following GitHub repository:

- Lodestar-Finance/lodestar-staking.

1.2 ASSESSMENT SUMMARY

The team at Halborn was provided a week for the engagement and assigned a full-time security engineer to verify the security of the smart contracts. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some security risks that were mostly addressed by the Lodestar team.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions. (solgraph)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Testnet deployment. (Foundry)

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 SEVERITY COEFFICIENT

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2.4 SCOPE

1. IN-SCOPE TREE & COMMIT :

The security assessment was scoped to the following smart contracts:

GitHub repository: Lodestar-Finance/lodestar-protocol
Assessed Commit ID #1: a21ecb23a4308c2602ac63ee86d576f78d73c6e6
Assessed Commit ID #2: aba53cfd19189720cb8c32368176648d6aead960
Assessed Commit ID #3: b049afc3bf5635250033f03fc9e4684eb332d373
Assessed Commit ID #4: 97c84d98ef6011507183fcd37570084137b6626b
Assessed Commit ID #5: e8277ff24975c3d418dc8124909452f1d24e8251
Final Commit ID: 1c15f28d1a71b59262dbd3a5c542159c6091ad49
Smart contracts in scope:

- esLODE.sol
- LodestarHandler.sol
- RewardRouter.sol
- StakingRewards.sol
- TokenClaim.sol
- VotingPower.sol
- RouterConstants.sol
- StakingConstants.sol
- Swap.sol
- WETHUtils.sol
- Whitelist.sol

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
6	6	4	0	4

In the process of assessment of the provided smart contracts codebase, a number of issues were identified that have serious implications for the reliability, security and maintainability of the application. Over the course of the assessment, multiple critical issues were uncovered.

This in itself is a significant concern, and points to a broader issue regarding the state of the code. It's worth noting that the identification of a substantial number of issues in a relatively short time span may suggest that there is a greater potential for further, yet undiscovered, vulnerabilities within the code.

While the scope of the assessment is thorough and guided by best industry practices, it is inherently limited by time and resources. Hence, the discovery of any number of critical issues cannot conclusively imply that all possible vulnerabilities have been identified. An assessment, while valuable, should not be viewed as a blanket guarantee against future exploits.

It is strongly recommended that the team responsible for the smart contract take these findings as an opportunity to not only rectify the identified vulnerabilities, but also to conduct a comprehensive review and potential re-design of the whole code. This would entail enforcing stronger coding standards, improving testing and review processes, and also aiming to follow the best practices in Solidity smart contract development.

Additionally, consider engaging in regular and repeated assessments to ensure ongoing security and proper functioning of your smart contracts, especially after changes have been made. The use of automated testing

and static analysis tools can also assist in maintaining a high standard of code quality.

Please note that this assessment does not absolve the developers of their responsibility to maintain, update, and secure their code. Rather, it serves as a professional evaluation of its current state, and a guideline for mitigating known issues and improving overall security and design patterns.

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) ESLODE TOKENS CAN BE DRAINED BY CALLING WITHDRAWESLODE() FUNCTION INDEFINITELY	Critical (10)	SOLVED - 08/01/2023
(HAL-02) VOTING POWER CAN BE MANIPULATED BY STAKING, VOTING, UNSTAKING 10 SECONDS LATER AND TRANSFERRING LODE TO A NEW WALLET	Critical (10)	SOLVED - 08/01/2023
(HAL-03) VOTING POWER CAN BE STOLEN BY CALLING VOTINGPOWER.DELEGATE FUNCTION	Critical (10)	SOLVED - 08/01/2023
(HAL-04) THE ONE YEAR VESTING PERIOD FOR ESLODE TOKENS CAN BE BYPASSED	Critical (10)	SOLVED - 08/01/2023
(HAL-05) LODE TOKENS CAN BE PERMANENTLY STUCK IN THE STAKINGREWARDS CONTRACT DUE TO WRONG MINTING/BURNING LOGIC IN STAKE/UNSTAKE LODE FUNCTIONS	Critical (10)	SOLVED - 08/03/2023
(HAL-06) UNSTAKELODE FUNCTION MAY REVERT UNDER CERTAIN CONDITIONS AS VOTING POWER IS INCORRECTLY BURNT	Critical (10)	SOLVED - 08/07/2023
(HAL-07) ESLODE TOKENS COULD GET LOCKED PERMANENTLY IN THE STAKINGREWARDS CONTRACT	High (7.5)	SOLVED - 08/01/2023
(HAL-08) VOTING POWER CAN BE MANIPULATED WITH A LODE FLASHLOAN AS STARTTIME VARIABLE IS ONLY UPDATED DURING THE INITIAL LOCK	High (7.5)	SOLVED - 08/01/2023
(HAL-09) LODE LOCKING CAN BE BYPASSED ABUSING THE REWARDS SYSTEM	High (7.5)	SOLVED - 08/01/2023
(HAL-10) RELOCKING FULL BONUS CAN BE OBTAINED AFTER A 10 SECONDS LOCK	High (7.5)	SOLVED - 08/01/2023
(HAL-11) CONVERTESLODETOLODE FUNCTION EXCESSIVELY BURNS VOTING POWER, REVERTING	High (7.5)	SOLVED - 08/07/2023
(HAL-12) STAKERS MAY NOT BE ABLE TO UNSTAKE THEIR LODE TOKENS DUE TO ROUNDING ERROR	High (7.5)	SOLVED - 08/11/2023

(HAL-13) UPDATEWEEKLYREWARDS FUNCTION DOES NOT GUARANTEE THAT THE STAKINGREWARDS CONTRACT HAS ENOUGH REWARDS FOR ALL THE STAKERS	Medium (5.6)	SOLVED - 08/01/2023
(HAL-14) CALLING STAKINGREWARDS.UPDATEVOTINGCONTRACT FUNCTION CAN BREAK THE STAKINGREWARDS CONTRACT	Medium (5.0)	SOLVED - 08/14/2023
(HAL-15) POSSIBLE DENIAL OF SERVICE BY REACHING BLOCK GAS LIMIT WHEN CALLING CONVERTESLODETOLODE FUNCTION	Medium (5.0)	SOLVED - 08/01/2023
(HAL-16) UPDATESHARES IS NOT CALLED IN CERTAIN CASES IN THE RELOCK FUNCTION	Medium (5.0)	SOLVED - 08/01/2023
(HAL-17) STATE VARIABLES MISSING IMMUTABLE MODIFIER	Informational (0.0)	SOLVED - 08/01/2023
(HAL-18) STATE VARIABLES MISSING CONSTANT MODIFIER	Informational (0.0)	SOLVED - 08/01/2023
(HAL-19) STRUCTS DO NOT FOLLOW THE TIGHT VARIABLE PACKING PATTERN	Informational (0.0)	ACKNOWLEDGED
(HAL-20) FLOATING PRAGMA	Informational (0.0)	SOLVED - 08/01/2023

FINDINGS & TECH DETAILS

4.1 (HAL-01) ESLODE TOKENS CAN BE DRAINED BY CALLING WITHDRAWESLODE() FUNCTION INDEFINITELY - CRITICAL(10)

Commit IDs affected:
- a21ecb23a4308c2602ac63ee86d576f78d73c6e6

Description:

The StakingRewards contract implements the function withdrawEsLODE() that allows users to unstake their esLODE without converting them to LODE:

Listing 1: StakingRewards.sol (Line 347)

```
338 /**
339  * @notice Withdraw esLODE
340  * @dev can only be called by the end user when withdrawing of
 ↳ esLODE is allowed
341  */
342 function withdrawEsLODE() external nonReentrant {
343     require(withdrawEsLODEAllowed == true, "esLODE Withdrawals Not
 ↳  Permitted");
344     //harvest();
345     StakingInfo storage account = stakers[msg.sender];
346     uint256 totalEsLODE = account.totalEsLODEStakedByUser;
347     esLODE.safeTransfer(msg.sender, totalEsLODE);
348     emit UnstakedEsLODE(msg.sender, totalEsLODE);
349 }
```

Although, this function does not update the storage after withdrawing, allowing any user to repeatedly call this function, draining all the esLODE from the contract.

Proof of Concept:

The proof of concept shows how the user1 is able to repeatedly call the withdrawEsLODE() function, draining all the esLODE from the contract.

```
USER1 CALLS ---> < contract_esLODE.approve(contract_StakingRewards, type(uint256).max) >

USER1 CALLS ---> < contract_StakingRewards.stakeEsLODE(100e18) >

USER2 CALLS ---> < contract_esLODE.approve(contract_StakingRewards, type(uint256).max) >

USER2 CALLS ---> < contract_StakingRewards.stakeEsLODE(2000e18) >

OWNER CALLS ---> < contract_StakingRewards._allowEsLODEWithdraw(true) >

USER1 CALLS ---> < contract_StakingRewards.withdrawEsLODE() >
  contract_esLODE.balanceOf(user1) -> 10000000000000000000
      USER1 CALLS ---> < contract_StakingRewards.withdrawEsLODE() >
  contract_esLODE.balanceOf(user1) -> 20000000000000000000
      USER1 CALLS ---> < contract_StakingRewards.withdrawEsLODE() >
  contract_esLODE.balanceOf(user1) -> 30000000000000000000
      USER1 CALLS ---> < contract_StakingRewards.withdrawEsLODE() >
  contract_esLODE.balanceOf(user1) -> 40000000000000000000
      USER1 CALLS ---> < contract_StakingRewards.withdrawEsLODE() >
  contract_esLODE.balanceOf(user1) -> 50000000000000000000
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:C/Y:N/R:N/S:U (10)

Recommendation:

It is recommended to reset the `stakers[msg.sender].totalEsLODEStakedByUser` storage to 0 after the `withdrawEsLODE()` call.

Remediation Plan:

SOLVED: The Lodestar team solved the issue by implementing the recommended solution.

Commit ID : aba53cfd19189720cb8c32368176648d6aead960.

4.2 (HAL-02) VOTING POWER CAN BE MANIPULATED BY STAKING, VOTING, UNSTAKING 10 SECONDS LATER AND TRANSFERRING LODE TO A NEW WALLET - CRITICAL(10)

Commit IDs affected:
- a21ecb23a4308c2602ac63ee86d576f78d73c6e6

Description:

The VotingPower contract calculates the users' voting power through the StakingRewards.accountVoteShare() function:

Listing 2: StakingRewards.sol

```
694 /**
695  * @notice Function used to calculate a user's voting power for
 ↳ emissions voting
696  * @param account The staker's address
697  * @return Returns the user's voting power as a percentage of the
 ↳ total voting power
698  */
699 function accountVoteShare(address account) public view returns (
 ↳ uint256) {
700     uint256 stLODEStaked = stakers[account].stLODEAmount;
701     uint256 vstLODEStaked = stakers[account].relockStLODEAmount;
702     uint256 totalStLODEStaked = totalSupply() - totalRelockStLODE;
703     uint256 totalVstLODEStaked = totalRelockStLODE;
704
705     uint256 totalStakedAmount = stLODEStaked + vstLODEStaked;
706     uint256 totalStakedBalance = totalStLODEStaked +
 ↳ totalVstLODEStaked;
707
708     if (totalStakedBalance == 0) {
709         return 0;
710     }
711
```

```
712     return (totalStakedAmount * 1e18) / totalStakedBalance;
713 }
```

As soon as LODE is staked in the StakingRewards contract, the voting power is increased. Although, the minimum period of time that LODE can be staked is just 10 seconds. Hence, any user could:

1. Stake LODE for 10 seconds.
2. Vote with their increased voting power.
3. Wait 10 seconds, unstake, transfer the LODE tokens to another wallet.
4. Stake LODE using the new wallet.
5. Vote again.
6. Repeat.

```
USER5 CALLS ---> < contract_StakingRewards.stakeLODE(1000e18, 10) >

USER5 CALLS ---> < contract_VotingPower.vote(['ARB'], [BORROW], [5e17]) >

USER1 CALLS ---> < contract_LODE.approve(address(contract_StakingRewards), type(uint256).max) >
  contract_LODE.balanceOf(user1) -> 10000000000000000000
  contract_StakingRewards.accountVoteShare(user1) -> 0

USER1 CALLS ---> < contract_StakingRewards.stakeLODE(1000e18, 10) >
  contract_LODE.balanceOf(user1) -> 0
  contract_StakingRewards.accountVoteShare(user1) -> 500000000000000000

USER1 CALLS ---> < contract_VotingPower.vote(['ARB'], [SUPPLY], [5e17]) >
  totalVotes['ARB'][OperationType.SUPPLY] -> 500000000000000000
  totalVotes['ARB'][OperationType.BORROW] -> 500000000000000000
  Token -> ARB
  OperationType -> 0
  Votes -> 301369863000000000
  Token -> ARB
  OperationType -> 1
  Votes -> 301369863000000000

10 seconds later...


USER1 CALLS ---> < contract_StakingRewards.unstakeLODE(1000e18) >
  contract_LODE.balanceOf(user1) -> 10000000000000000000

USER1 CALLS ---> < contract_LODE.transfer(user2, 1000e18) >

USER2 CALLS ---> < contract_LODE.approve(address(contract_StakingRewards), type(uint256).max) >
  contract_LODE.balanceOf(user2) -> 10000000000000000000
  contract_StakingRewards.accountVoteShare(user2) -> 0

USER2 CALLS ---> < contract_StakingRewards.stakeLODE(1000e18, 10) >
  contract_LODE.balanceOf(user2) -> 0
  contract_StakingRewards.accountVoteShare(user2) -> 500000000000000000
```

```
USER2 CALLS ---> < contract_VotingPower.vote(['ARB'], [SUPPLY], [5e17]) >
  totalVotes['ARB'][OperationType.SUPPLY] -> 1000000000000000000
  totalVotes['ARB'][OperationType.BORROW] -> 500000000000000000
  Token -> ARB
  OperationType -> 0
  Votes -> 401826483999999999
  Token -> ARB
  OperationType -> 1
  Votes -> 200913241999999999
```

This way, the votes can be easily manipulated.

BVSS:

AO:A/AC:L/AX:L/C:N/I:C/A:N/D:N/Y:N/R:N/S:C (10)

Recommendation:

It is recommended to set the minimum stake period to one week or simply do not assign any voting power to users which lockTime is set only to 10 seconds.

Remediation Plan:

SOLVED: The Lodestar team solved the issue by implementing the recommended solution.

Commit ID : aba53cfd19189720cb8c32368176648d6aead960.

4.3 (HAL-03) VOTING POWER CAN BE STOLEN BY CALLING VOTINGPOWER.DELEGATE FUNCTION - CRITICAL(10)

Commit IDs affected:
- a21ecb23a4308c2602ac63ee86d576f78d73c6e6

Description:

The VotingPower contract contains the function delegate() which supposedly is used to delegate voting power to another address of the user's choice:

Listing 3: VotingPower.sol

```
95 /// @notice Delegates existing voting power to an address of the
   ↳ user's choice.
96 /// @param to The address in which the user wishes to delegate
   ↳ their voting power to.
97 function delegate(address to) external resetVotesIfNeeded
   ↳ whenNotPaused {
98     voteDelegates[msg.sender] = to;
99     emit DelegateChanged(msg.sender, to);
100 }
```

Moreover, this voteDelegates mapping is incorrectly used in the VotingPower.vote() function to "calculate" the voting power:

Listing 4: VotingPower.sol (Line 146)

```
128 function vote(
129     string[] calldata tokens,
130     OperationType[] calldata operations,
131     uint256[] calldata shares
132 ) external resetVotesIfNeeded whenNotPaused {
133     require(
134         tokens.length == operations.length && tokens.length ==
```

```
↳ shares.length,
135         "Arrays must have the same length"
136     );
137
138     uint256 currentWeek = getCurrentWeek();
139     require(
140         // Will pass if lastVotedWeek[msg.sender] is less than
 ↳ currentWeek (i.e., the user hasn't voted this week) or if
 ↳ currentWeek and lastVotedWeek[msg.sender] are both 0
141         // (i.e., it's the first week since the contract was
 ↳ deployed and the user hasn't voted before).
142         lastVotedWeek[msg.sender] < currentWeek || (currentWeek ==
 ↳  0 && lastVotedWeek[msg.sender] == 0 && !previouslyVoted[msg.
 ↳ sender]),
143         "You have already voted this week"
144     );
145
146     uint256 userVotingPower = stakingRewards.accountVoteShare(
 ↳ voteDelegates[msg.sender] == address(0) ? msg.sender :
 ↳ voteDelegates[msg.sender]);
147     uint256 totalShares = 0;
148
149     for (uint256 i = 0; i < tokens.length; i++) {
150         string memory token = tokens[i];
151         OperationType operation = operations[i];
152         uint256 share = shares[i];
153
154         require(share > 0, "Share must be greater than 0");
155         require(tokenEnabled[token], "Token is not enabled for
 ↳ voting");
156         if (!bothOperationsAllowed[token]) {
157             require(operation == OperationType.SUPPLY, "Only
 ↳ supply emissions are allowed for this token");
158         }
159
160         userVotes[msg.sender][i] = Vote(share, token, operation);
161         totalVotes[token][operation] = totalVotes[token][operation
 ↳ ].add(share);
162         totalShares = totalShares.add(share);
163
164         emit VoteCast(msg.sender, token, operation, share);
165     }
166
167     require(totalShares <= userVotingPower, "Voted shares exceed
```

```
 ↳ user's voting power");
168     lastVotedWeek[msg.sender] = currentWeek;
169     previouslyVoted[msg.sender] = true;
170 }
```

Basically, when stakingRewards.accountVoteShare() is called, if the voteDelegates mapping is not pointing to the address(0), the voting power of the voteDelegates mapping address will be used. This flawed implementation allows anyone to vote with the voting power of another user by simply calling delegate(<high voting power address>).

Proof of Concept:

```
USER5 CALLS ---> < contract_LODE.approve(address(contract_StakingRewards), type(uint256).max) >

USER5 CALLS ---> < contract_StakingRewards.stakeLODE(1000e18, 10) >

USER5 CALLS ---> < contract_VotingPower.vote(['ARB'], [BORROW], [5e17]) >

USER1 CALLS ---> < contract_VotingPower.delegate(user5) >
  contract_LODE.balanceOf(user1) -> 0
  contract_StakingRewards.accountVoteShare(user1) -> 0

USER1 CALLS ---> < contract_VotingPower.vote(['ARB'], [SUPPLY], [5e17]) >
  totalVotes['ARB'][OperationType.SUPPLY] -> 500000000000000000
  totalVotes['ARB'][OperationType.BORROW] -> 500000000000000000
  Token -> ARB
  OperationType -> 0
  Votes -> 301369863000000000
  Token -> ARB
  OperationType -> 1
  Votes -> 301369863000000000

USER2 CALLS ---> < contract_VotingPower.delegate(user5) >
  contract_LODE.balanceOf(user2) -> 0
  contract_StakingRewards.accountVoteShare(user2) -> 0

USER2 CALLS ---> < contract_VotingPower.vote(['ARB'], [SUPPLY], [5e17]) >
  totalVotes['ARB'][OperationType.SUPPLY] -> 1000000000000000000
  totalVotes['ARB'][OperationType.BORROW] -> 500000000000000000
  Token -> ARB
  OperationType -> 0
  Votes -> 401826483999999999
  Token -> ARB
  OperationType -> 1
  Votes -> 200913241999999999
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:C/A:N/D:N/Y:N/R:N/S:C (10)

Recommendation:

It is recommended to either rebuild from scratch the delegating logic or otherwise remove it. Some example implementation of voting power delegation can be found in OpenZeppelin's Votes contract.

Remediation Plan:

SOLVED: The Lodestar team solved the issue by implementing the recommended solution.

Commit ID : aba53cfd19189720cb8c32368176648d6aead960.

4.4 (HAL-04) THE ONE YEAR VESTING PERIOD FOR ESLODE TOKENS CAN BE BYPASSED - CRITICAL(10)

Commit IDs affected:
- a21ecb23a4308c2602ac63ee86d576f78d73c6e6

Description:

In the StakingRewards contract, the function convertEsLODEToLODE() is used to convert vested esLODE to LODE. The vesting period of the esLODE tokens is 365 days:

Listing 5: StakingRewards.sol (Line 252)

```
239 /**
240  * @notice Converts vested esLODE to LODE and updates user reward
 ↳ shares accordingly accounting for current lock time and relocks
241  * @param user The staker's address
242  */
243 function convertEsLODEToLODE(address user) public returns (uint256
 ↳ ) {
244     //since this is also called on unstake and harvesting, we exit
 ↳  out of this function if user has no esLODE staked.
245     if (stakers[msg.sender].totalEsLODEStakedByUser == 0) {
246         return 0;
247     }
248
249     uint256 lockTime = stakers[user].lockTime;
250     uint256 threeMonthCount = stakers[user].threeMonthRelockCount;
251     uint256 sixMonthCount = stakers[user].sixMonthRelockCount;
252     uint256 totalDays = 365 days;
253     uint256 amountToTransfer;
254     uint256 stLODEAdjustment;
255     uint256 conversionAmount;
256
257     Stake[] memory userStakes = esLODEStakes[msg.sender];
258
259     for (uint256 i = 0; i < userStakes.length; i++) {
```

```
260         uint256 timeDiff = (block.timestamp - userStakes[i].
 ↳ startTimestamp);
261         uint256 alreadyConverted = userStakes[i].alreadyConverted;
262 
263         if (timeDiff >= totalDays) {
264             conversionAmount = userStakes[i].amount;
265             amountToTransfer += conversionAmount;
266             userStakes[i].amount = 0;
267             if (lockTime == 90 days) {
268                 stLODEAdjustment +=
269                     (conversionAmount *
270                         ((stLODE3M - 1e18) +
271                             (threeMonthCount * relockStLODE3M) +
272                             (sixMonthCount * relockStLODE6M))) /
273                     BASE;
274             } else if (lockTime == 180 days) {
275                 stLODEAdjustment +=
276                     (conversionAmount *
277                         ((stLODE6M - 1e18) +
278                             (threeMonthCount * relockStLODE3M) +
279                             (sixMonthCount * relockStLODE6M))) /
280                     BASE;
281             }
282         } else if (timeDiff < totalDays) {
283             uint256 conversionRatioMantissa = (timeDiff * BASE) /
 ↳ totalDays;
284             conversionAmount = ((userStakes[i].amount *
 ↳ conversionRatioMantissa) / BASE) - alreadyConverted;
285             amountToTransfer += conversionAmount;
286             esLODEStakes[msg.sender][i].alreadyConverted +=
 ↳ conversionAmount;
287             esLODEStakes[msg.sender][i].amount -= conversionAmount
 ↳ ;
288             if (lockTime == 90 days) {
289                 stLODEAdjustment +=
290                     (conversionAmount *
291                         ((stLODE3M - 1e18) +
292                             (threeMonthCount * relockStLODE3M) +
293                             (sixMonthCount * relockStLODE6M))) /
294                     BASE;
295             } else if (lockTime == 180 days) {
296                 stLODEAdjustment +=
297                     (conversionAmount *
298                         ((stLODE6M - 1e18) +
```

```
299                             (threeMonthCount * relockStLODE3M) +
300                             (sixMonthCount * relockStLODE6M))) /
301                         BASE;
302                 }
303             }
304         }
305
306         stakers[user].lodeAmount += amountToTransfer;
307         stakers[user].totalEsLODEStakedByUser -= amountToTransfer;
308         totalEsLODEStaked -= amountToTransfer;
309
310         if (stLODEAdjustment != 0) {
311             stakers[user].stLODEAmount += stLODEAdjustment;
312             UserInfo storage userRewards = userInfo[user];
313
314             uint256 _prev = totalSupply();
315
316             updateShares();
317
318             unchecked {
319                 userRewards.amount += uint96(stLODEAdjustment);
320                 shares += uint96(stLODEAdjustment);
321             }
322
323             userRewards.wethRewardsDebt =
324                 userRewards.wethRewardsDebt +
325                 int128(uint128(_calculateRewardDebt(accWethPerShare,
 ↳ uint96(stLODEAdjustment))));
326
327             _mint(address(this), stLODEAdjustment);
328
329             unchecked {
330                 if (_prev + stLODEAdjustment != totalSupply()) revert
 ↳ DEPOSIT_ERROR();
331             }
332         }
333
334         esLODE.transfer(address(0), amountToTransfer);
335         return conversionAmount;
336 }
```

The `convertEsLODEToLODE(address user)` function has a `user` parameter that when used with an account different from `msg.sender` causes the following

exploit:

1. User1 stakes 1000e18 esLODE and waits 365 days. User1's esLODE tokens are vested at this point.
2. User1 creates another wallet, let's call this wallet User2.
3. User2 stakes 1000e18 esLODE tokens.
4. User1 calls convertEsLODEToLODE(<user2 address>).
5. User2 esLODE tokens are converted right away to LODE
6. Repeat this process with other wallets to totally bypass the vesting period.

```
USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS ---> < contract_StakingRewards.stakeEsLODE(1000e18) >

STAKING_INFO: User 0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90
  ------------------------------
  lodeAmount -> 0
  stLODEAmount -> 10000000000000000000
  startTime -> 0
  lockTime -> 0
  relockStLODEAmount -> 0
  nextStakeId -> 1
  totalEsLODEStakedByUser -> 10000000000000000000
  threeMonthRelockCount -> 0
  sixMonthRelockCount -> 0

365 DAYS LATER...


USER2(0x88C0e901bd1fd1a77BdA342f0d2210fDC71Cef6B) CALLS ---> < contract_StakingRewards.stakeEsLODE(1000e18) >

STAKING_INFO: User 0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90
  ------------------------------
  lodeAmount -> 0
  stLODEAmount -> 10000000000000000000
  startTime -> 0
  lockTime -> 0
  relockStLODEAmount -> 0
  nextStakeId -> 1
  totalEsLODEStakedByUser -> 10000000000000000000
  threeMonthRelockCount -> 0
  sixMonthRelockCount -> 0

STAKING_INFO: User 0x88C0e901bd1fd1a77BdA342f0d2210fDC71Cef6B
  ------------------------------
  lodeAmount -> 0
  stLODEAmount -> 10000000000000000000
  startTime -> 0
  lockTime -> 0
  relockStLODEAmount -> 0
  nextStakeId -> 1
  totalEsLODEStakedByUser -> 10000000000000000000
  threeMonthRelockCount -> 0
  sixMonthRelockCount -> 0

USER1(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90) CALLS ---> < contract_StakingRewards.convertEsLODEToLODE(user2) >

STAKING_INFO: User 0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90
  ------------------------------
  lodeAmount -> 0
  stLODEAmount -> 10000000000000000000
  startTime -> 0
  lockTime -> 0
  relockStLODEAmount -> 0
  nextStakeId -> 1
  totalEsLODEStakedByUser -> 10000000000000000000
  threeMonthRelockCount -> 0
  sixMonthRelockCount -> 0
```

```
STAKING_INFO: User 0x88C0e901bd1fd1a77BdA342f0d2210fDC71Cef6B

  ------------------------------
  lodeAmount -> 10000000000000000000000
  stLODEAmount -> 10000000000000000000000
  startTime -> 0
  lockTime -> 0
  relockStLODEAmount -> 0
  nextStakeId -> 1
  totalEsLODEStakedByUser -> 0
  threeMonthRelockCount -> 0
  sixMonthRelockCount -> 0
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:C/A:N/D:N/Y:N/R:N/S:C (10)

Recommendation:

It is recommended to remove the user parameter from the convertEsLODEToLODE() function and perform all the function logic using msg.sender.

Remediation Plan:

SOLVED: The Lodestar team solved the issue by implementing the recommended solution.

Commit ID : aba53cfd19189720cb8c32368176648d6aead960.

4.5 (HAL-05) LODE TOKENS CAN BE PERMANENTLY STUCK IN THE STAKINGREWARDS CONTRACT DUE TO WRONG MINTING/BURNING LOGIC IN STAKE/UNSTAKE LODE FUNCTIONS - CRITICAL(10)

Commit IDs affected:
- aba53cfd19189720cb8c32368176648d6aead960

Description:

In the StakingRewards contract, VotingPower is minted every time a user calls stakeLODE() and burnt every time a user calls the unstakeLODE() function.

Although, in the convertEsLODEToLODE() function, when lockTime is different from 10 seconds, VotingPower is also burnt:

Listing 6: StakingRewards.sol (Line 367)

```
364 //Adjust voting power if user is locking LODE
365 if(stakers[msg.sender].lockTime != 10 seconds) {
366     //if user is unlocked, we need to burn their converted amount
 ↳ of voting power
367     votingContract.burn(msg.sender, conversionAmount);
368 } else {
```

lockTime will be different from 10 seconds if:
1. User staked LODE with 90 or 180 days lockTime.
2. User has not staked LODE yet.

Based on this implementation, the following flow would cause an overflow, not allowing users to unstake their LODE tokens:

1. Alice stakes 100 esLODE tokens. She receives 100 VotingPower.
2. 1 year later, when the vesting is completed, Alice calls convertEsLODEToLODE() converting those 100 esLODE tokens into 100 LODE tokens. As she hadn't staked any LODE before, her lockTime is 0, entering the if logic mentioned above. This means that her 100 VotingPower is burnt.
3. Alice tries to unstake her 100 LODE token, but she can't as it reverts with [FAIL. Reason: Burn amount exceeds voting power]. Contract is incorrectly trying to burn VotingPower that she does not have anymore, as it was already burnt before during the convertEsLODEToLODE() call.

Proof of Concept:

```
USER1 CALLS ---> < contract_esLODE.approve(address(contract_StakingRewards), 100e18) >

USER1 CALLS ---> < contract_StakingRewards.stakeEsLODE(100e18) >
  contract_VotingPower.getVotes2(user1) -> 100000000000000000000

365 DAYS LATER...

  contract_VotingPower.getVotes2(user1) -> 100000000000000000000

USER1 CALLS ---> < contract_StakingRewards.convertEsLODEToLODE() >
  contract_VotingPower.getVotes2(user1) -> 0

1 DAY LATER...


STAKING_INFO: User 0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90

  ----------------------------
  lodeAmount -> 100000000000000000000
  stLODEAmount -> 100000000000000000000
  startTime -> 0
  lockTime -> 0
  relockStLODEAmount -> 0
  nextStakeId -> 1
  totalEsLODEStakedByUser -> 0
  threeMonthRelockCount -> 0
  sixMonthRelockCount -> 0
  contract_VotingPower.getVotes2(user1) -> 0

USER1 CALLS ---> < contract_StakingRewards.unstakeLODE(100000000000000000000) >

    ├─ [797] VotingPower::burn(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90, 100000000000000000000)
    │   └─ ← "Burn amount exceeds voting power"
    └─ ← "Burn amount exceeds voting power"

Test result: FAILED. 0 passed; 1 failed; finished in 340.37ms

Failing tests:
Encountered 1 failing test in test/foundry/StakingRewards.t.sol:StakingRewardsTests
[FAIL. Reason: Burn amount exceeds voting power] test_2_POC() (gas: 923146)
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:C/A:N/D:C/Y:N/R:N/S:U (10)

Recommendation:

It is recommended to not mint VotingPower when users stake for 10 seconds in the stakeLODE() function. Accordingly, the VotingPower should not be burnt when users unstake with a 0 or 10 lockTime in the unstakeLODE() function.

Remediation Plan:

SOLVED: The Lodestar team solved the issue by implementing the recommended solution.

Commit ID : b049afc3bf5635250033f03fc9e4684eb332d373.

4.6 (HAL-06) UNSTAKELODE FUNCTION MAY REVERT UNDER CERTAIN CONDITIONS AS VOTING POWER IS INCORRECTLY BURNT - CRITICAL(10)

Commit IDs affected:
- b049afc3bf5635250033f03fc9e4684eb332d373

Description:

During the re-assess of the new code introduced in the Commit ID b049afc3bf5635250033f03fc9e4684eb332d373 a new issue was found in the stakeLODE()-unstakeLODE() functions.

Basically, the following flow would cause a revert Burn amount exceeds voting power during an unstakeLODE() function call:
1. Alice stakes 700 esLODE. She receives 700 VotingPower.
2. 2 weeks later, Alice calls convertEsLODEToLODE(). She converts 26,8 esLODE into LODE. Her VotingPower is burnt accordingly. She now has 673,2 VotingPower.
3. A few weeks later, Alice calls stakeLODE(1500, 10). She receives no voting power as she staked with a lockTime of 10. Although, the lockTime was not updated by the contract properly in the stakeLODE() function:

Listing 7: StakingRewards.sol

```
308 if (stakers[msg.sender].lodeAmount == 0) {
309     stakers[msg.sender].startTime = block.timestamp;
310     stakers[msg.sender].lockTime = lockTime;
311 }
```

When Alice called the stakeLODE() function, as she had previously called convertEsLODEToLODE(), her lodeAmount was >0, hence this code block was not entered.
4. Later on, Alice calls unstakeLODE() but it reverts with a Burn

amount exceeds voting power error. Why does this occurs? Because this code block is entered incorrectly during the unstakeLODE() call (as the lockTime was not properly updated before when she staked):

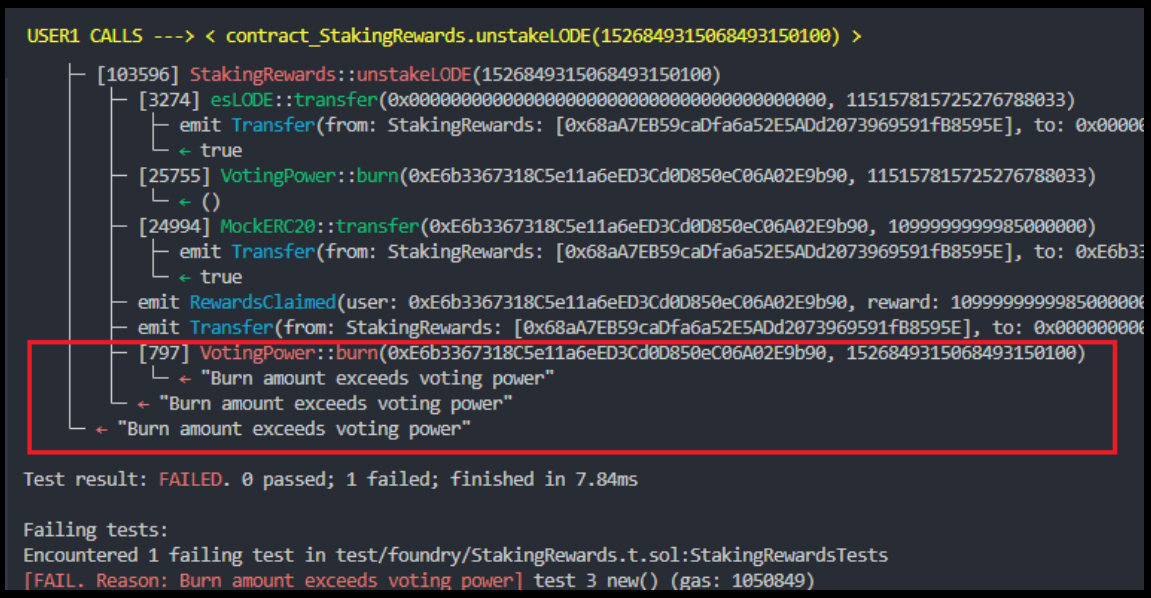
Listing 8: StakingRewards.sol (Line 265)

```
263 //Adjust voting power
264 if (lockTimePriorToUpdate != 10 seconds) {
265     votingContract.burn(msg.sender, stLODEReduction);
266 }
```

As the contract tries to burn VotingPower that Alice does not have, it reverts, not allowing her to ever unstake her LODE tokens.

Proof of Concept:

```
USER1 CALLS ---> < contract_esLODE.approve(address(contract_StakingRewards), 700e18) >

USER1 CALLS ---> < contract_StakingRewards.stakeEsLODE(700e18) >
  contract_VotingPower.getVotes2(user1) -> 700000000000000000000

2 WEEKS LATER...

  contract_VotingPower.getVotes2(user1) -> 700000000000000000000

USER1 CALLS ---> < contract_StakingRewards.convertEsLODEToLODE() >
  contract_VotingPower.getVotes2(user1) -> 673150684931506849900

4 WEEKS LATER...


USER1 CALLS ---> < contract_LODE.approve(address(contract_StakingRewards), 1500e18) >
  contract_VotingPower.getVotes2(user1) -> 673150684931506849900

USER1 CALLS ---> < contract_StakingRewards.stakeLODE(1500e18, 10) >
  contract_VotingPower.getVotes2(user1) -> 673150684931506849900

5 WEEKS LATER...


STAKING_INFO: User 0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90

  lodeAmount -> 1526849315068493150100
  stLODEAmount -> 2200000000000000000000
  startTime -> 3628801
  lockTime -> 0
  relockStLODEAmount -> 0
  nextStakeId -> 1
  totalEsLODEStakedByUser -> 673150684931506849900
  threeMonthRelockCount -> 0
  sixMonthRelockCount -> 0
  contract_VotingPower.getVotes2(user1) -> 673150684931506849900
```



BVSS:

AO:A/AC:L/AX:L/C:N/I:C/A:N/D:C/Y:N/R:N/S:U (10)

Recommendation:

It is recommended to update the `lockTime` variable when `convertEsLODEToLODE()` is called and the current `lockTime` of the user is `0`.

Remediation Plan:

SOLVED: The `Lodestar team` solved the issue by implementing the recommended solution.

`Commit ID : 97c84d98ef6011507183fcd37570084137b6626b`.

4.7 (HAL-07) ESLODE TOKENS COULD GET LOCKED PERMANENTLY IN THE STAKINGREWARDS CONTRACT - HIGH (7.5)

Commit IDs affected:
- a21ecb23a4308c2602ac63ee86d576f78d73c6e6

Description:

In the StakingRewards contract, the function convertEsLODEToLODE() could totally lock the esLODE in the contract under certain circumstances.

It was found that the following steps would lock Alice's esLODE permanently in the contract:

1. Alice stakes 1000 esLODE.
2. On the day 250, Alice calls convertEsLODEToLODE(Alice) receiving 685 LODE tokens.
3. 115 days later (day 365), Alice stakes another 1000 esLODE, and then she calls convertEsLODEToLODE(Alice) receiving the remaining 315 LODE tokens. Let's remember, though, that now there is a new stake of another 1000 esLODE tokens.
4. 500 days later, Alice calls convertEsLODEToLODE(Alice) but it fails with ERROR: [FAIL. Reason: Arithmetic over/underflow]. Alice's esLODE tokens are now stuck in the StakingRewards contract.

```
    │   └─ ← "Arithmetic over/underflow"
    └─ ← "Arithmetic over/underflow"

Test result: FAILED. 0 passed; 1 failed; finished in 5.19s

Failing tests:
Encountered 1 failing test in test/foundry/StakingRewards.t.sol:StakingRewardsTests
[FAIL. Reason: Arithmetic over/underflow] test_9() (gas: 1752189)
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:H/Y:N/R:N/S:U (7.5)

Recommendation:

It is recommended to fix the convertEsLODEToLODE() function logic to prevent this Denial of Service situation.

Remediation Plan:

SOLVED: The Lodestar team solved the issue by implementing the recommended solution.

Commit ID : aba53cfd19189720cb8c32368176648d6aead960.

4.8 (HAL-08) VOTING POWER CAN BE MANIPULATED WITH A LODE FLASHLOAN AS STARTTIME VARIABLE IS ONLY UPDATED DURING THE INITIAL LOCK - HIGH (7.5)

Commit IDs affected:
- a21ecb23a4308c2602ac63ee86d576f78d73c6e6

Description:

In the StakingRewards contract, the function stakeLODE() is used to stake LODE with or without a lock time to earn rewards:

Listing 9: StakingRewards.sol (Line 77)

```
54 /**
55  * @notice Stake LODE with or without a lock time to earn rewards
56  * @param amount the amount the user wishes to stake (denom. in
↳ wei)
57  * @param lockTime the desired lock time. Must be 10 seconds, 90
↳ days (in seconds) or 180 days (in seconds)
58  */
59 function stakeLODE(uint256 amount, uint256 lockTime) external
↳ whenNotPaused nonReentrant {
60     require(amount != 0, "StakingRewards: Invalid stake amount");
61     require(
62         lockTime == 10 seconds || lockTime == 90 days || lockTime
↳ == 180 days,
63         "StakingRewards: Invalid lock time"
64     );
65     uint256 currentLockTime = stakers[msg.sender].lockTime;
66     uint256 startTime = stakers[msg.sender].startTime;
67     uint256 unlockTime = startTime + currentLockTime;
68
69     if (currentLockTime != 0) {
70         require(lockTime == currentLockTime, "StakingRewards:
↳ Cannot add stake with different lock time");
71     }
72
```

```
73      if (currentLockTime != 10 seconds && currentLockTime != 0) {
74          require(block.timestamp < unlockTime, "StakingRewards:
↳ Staking period expired");
75      }
76
77      stakeLODEInternal(msg.sender, amount, lockTime);
78  }
```

This stakeLODE() function internally calls the stakeLODEInternal() function:

Listing 10: StakingRewards.sol (Lines 91-94)

```
80  function stakeLODEInternal(address staker, uint256 amount, uint256
↳  lockTime) internal {
81      require(LODE.transferFrom(staker, address(this), amount), "
↳ StakingRewards: Transfer failed");
82
83      uint256 mintAmount = amount;
84
85      if (lockTime == 90 days) {
86          mintAmount = (amount * stLODE3M) / 1e18; // Scale the mint
↳  amount for 3 months lock time
87      } else if (lockTime == 180 days) {
88          mintAmount = (amount * stLODE6M) / 1e18; // Scale the mint
↳  amount for 6 months lock time
89      }
90
91      if (stakers[staker].lodeAmount == 0) {
92          stakers[staker].startTime = block.timestamp;
93          stakers[staker].lockTime = lockTime;
94      }
95
96      stakers[staker].lodeAmount += amount; // Update LODE staked
↳ amount
97      stakers[staker].stLODEAmount += mintAmount; // Update stLODE
↳ minted amount
98      totalStaked += amount;
99
100     UserInfo storage user = userInfo[staker];
101
102     uint256 _prev = totalSupply();
103
```

```
104     updateShares();
105
106     unchecked {
107         user.amount += uint96(mintAmount);
108         shares += uint96(mintAmount);
109     }
110
111     user.wethRewardsDebt =
112         user.wethRewardsDebt +
113         int128(uint128(_calculateRewardDebt(accWethPerShare,
 ↳ uint96(mintAmount))));
114
115     _mint(address(this), mintAmount);
116
117     unchecked {
118         if (_prev + mintAmount != totalSupply()) revert
 ↳ DEPOSIT_ERROR();
119     }
120
121     emit StakedLODE(staker, amount, lockTime);
122 }
```

Although, the stakeLODEInternal() only updates the stakers[staker].startTime when the stakers[staker].lodeAmount is equal to 0. Hence, this flaw in the stakeLODEInternal() function logic can be abused to take a flashloan (if it was possible) of LODE, stake it, increase voting power, vote, unstake and repay the flashloan.

```
USER1 CALLS ---> < contract_LODE.approve(address(contract_StakingRewards), type(uint256).max) >
  contract_LODE.balanceOf(user1) -> 10000000000000000000
  contract_StakingRewards.accountVoteShare(user1) -> 0

USER1 CALLS ---> < contract_StakingRewards.stakeLODE(1, 10) >
  contract_LODE.balanceOf(user1) -> 9999999999999999999
  contract_StakingRewards.accountVoteShare(user1) -> 100000000000000000

12 SECONDS LATER...

  contract_LODE.balanceOf(user1) -> 9999999999999999999
  contract_StakingRewards.accountVoteShare(user1) -> 100000000000000000

USER1 CALLS ---> < contract_StakingRewards.stakeLODE(1000e18 - 1, 10) >
  contract_LODE.balanceOf(user1) -> 0
  contract_StakingRewards.accountVoteShare(user1) -> 100000000000000000

USER1 CALLS ---> < contract_StakingRewards.unstakeLODE(1000e18) >
  contract_LODE.balanceOf(user1) -> 10000000000000000000
```

AT THIS POINT, STAKELODE AND UNSTAKE LODE CAN BE CALLED WITHIN THE SAME TX

BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:N/Y:N/R:N/S:U (7.5)

Recommendation:

It is recommended to always update the stakers[staker].startTime variable every time a user stakes.

Remediation Plan:

SOLVED: The Lodestar team solved the issue in the following Commit ID.

Commit ID : aba53cfd19189720cb8c32368176648d6aead960.

The VotingPower logic was rebuilt and is now based on the OpenZeppelin Votes contract.

4.9 (HAL-09) LODE LOCKING CAN BE BYPASSED ABUSING THE REWARDS SYSTEM - HIGH (7.5)

Commit IDs affected:
- a21ecb23a4308c2602ac63ee86d576f78d73c6e6

Description:

As mentioned in the previous issue, stakers[staker].startTime variable is only updated when the stakers[staker].lodeAmount is equal to 0. Based on this, the following exploit would be possible to get higher rewards without any risk:

```
USER1 CALLS ---> < contract_LODE.approve(address(contract_StakingRewards), type(uint256).max) >

USER1 CALLS ---> < contract_StakingRewards.stakeLODE(1, 180 days) >

180 DAYS LATER - 1...


USER1 CALLS ---> < contract_StakingRewards.claimRewards() >

USER1 CALLS ---> < contract_StakingRewards.stakeLODE(1000e18 - 1, 180 days) >

USER2 CALLS ---> < contract_LODE.approve(address(contract_StakingRewards), type(uint256).max) >

USER2 CALLS ---> < contract_StakingRewards.stakeLODE(1000e18, 180 days) >


 DAY: 0
  contract_StakingRewards.pendingRewards(user1) -> 0
  contract_StakingRewards.pendingRewards(user2) -> 0


 DAY: 1
  contract_StakingRewards.pendingRewards(user1) -> 71428571428560000000
  contract_StakingRewards.pendingRewards(user2) -> 71428571428560000000


 DAY: 2
  contract_StakingRewards.pendingRewards(user1) -> 142857142857140000000
  contract_StakingRewards.pendingRewards(user2) -> 142857142857140000000

USER1 CALLS ---> < contract_StakingRewards.unstakeLODE(1000e18) >
   contract_LODE.balanceOf(user1) -> 1000000000000000000000
   contract_WETH.balanceOf(user1) -> 25928569775132254016847
```

1. User1 stakes 0.000000001 LODE with a 180 days lock and waits 180 days - 1 second.
2. User1 stakes 10000 LODE.
3. User2 stakes 10000 LODE.
4. Both users are generating the same rewards. Although User1 now can unstake his LODE at any given time, User2 needs to wait another 180 days which is not fair for the User2.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:H/R:N/S:U (7.5)

Recommendation:

As suggested in the previous issue, always update the `stakers[staker].startTime` variable every time a user stakes.

Remediation Plan:

SOLVED: The Lodestar team solved the issue in the following Commit ID.

Commit ID : aba53cfd19189720cb8c32368176648d6aead960.

A different mitigation was chosen by Lodestar team. Users will not be able to re-stake if the `cutoffTime` was reached. This approach does not totally mitigate the issue, although the Lodestar team states that users will be informed of this logic, so it is a fair situation for all the stakers.

4.10 (HAL-10) RELOCKING FULL BONUS CAN BE OBTAINED AFTER A 10 SECONDS LOCK - HIGH (7.5)

Commit IDs affected:
- a21ecb23a4308c2602ac63ee86d576f78d73c6e6

Description:

In the StakingRewards contract, the function relock() allows re-staking for boosted rewards. Currently, there are 3 different lock times:
- 10 seconds.
- 90 days.
- 180 days.

Although, the function allows relocking stakes with 10 seconds as lockTime and still applies the bonus multiplier to them. Taking this into account, it would be possible for a user to get higher rewards by relocking 10 seconds lockTime stake than simply staking the LODE tokens for 180 days through the stakeLODE() function.

Proof of Concept:

```
USER1 CALLS ---> < contract_StakingRewards.stakeLODE(100e18, 180 days) >

USER4 CALLS ---> < contract_StakingRewards.stakeLODE(100e18, 10) >

10 SECONDS LATER...


USER4 CALLS ---> < contract_StakingRewards.relock(180 days) >

STAKING_INFO: User 0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90

  ____________________________
  lodeAmount -> 100000000000000000000
  stLODEAmount -> 200000000000000000000
  startTime -> 1690560494
  lockTime -> 15552000
  relockStLODEAmount -> 0
  nextStakeId -> 0
  totalEsLODEStakedByUser -> 0
  threeMonthRelockCount -> 0
  sixMonthRelockCount -> 0
```

```
STAKING_INFO: User 0x043aEd06383F290Ee28FA02794Ec7215CA099683
  ------------------------------
  lodeAmount -> 10000000000000000000
  stLODEAmount -> 21000000000000000000
  startTime -> 1690560504
  lockTime -> 15552000
  relockStLODEAmount -> 10000000000000000000
  nextStakeId -> 0
  totalEsLODEStakedByUser -> 0
  threeMonthRelockCount -> 0
  sixMonthRelockCount -> 1


 DAY: 0
  contract_StakingRewards.pendingRewards(user1) -> 11022927688000000
  contract_StakingRewards.pendingRewards(user4) -> 5511463844000000


 DAY: 30
  contract_StakingRewards.pendingRewards(user1) -> 2040827349458298000000
  contract_StakingRewards.pendingRewards(user4) -> 2244903470647515000000


 DAY: 60
  contract_StakingRewards.pendingRewards(user1) -> 4081643675988908000000
  contract_StakingRewards.pendingRewards(user4) -> 4489801429831186000000


 DAY: 90
  contract_StakingRewards.pendingRewards(user1) -> 6122460002519518000000
  contract_StakingRewards.pendingRewards(user4) -> 6734699389014857000000


 DAY: 120
  contract_StakingRewards.pendingRewards(user1) -> 8163276329050128000000
  contract_StakingRewards.pendingRewards(user4) -> 8979597348198528000000


 DAY: 150
  contract_StakingRewards.pendingRewards(user1) -> 10204092655580738000000
  contract_StakingRewards.pendingRewards(user4) -> 11224495307382199000000


 DAY: 180
  contract_StakingRewards.pendingRewards(user1) -> 12244908982111348000000
  contract_StakingRewards.pendingRewards(user4) -> 13469393266565870000000
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:H/R:N/S:U (7.5)

Recommendation:

It is recommended to restrict users from relocking when their stakes' lockTime is equal to 10.

Remediation Plan:

SOLVED: The Lodestar team solved the issue by implementing the recommended solution.

Commit ID : aba53cfd19189720cb8c32368176648d6aead960.

4.11 (HAL-11) CONVERTESLODETOLODE FUNCTION EXCESSIVELY BURNS VOTING POWER, REVERTING - HIGH (7.5)

Commit IDs affected:
- b049afc3bf5635250033f03fc9e4684eb332d373

Description:

In the StakingRewards contract, the function convertEsLODEToLODE() is used to convert the esLODE tokens to LODE with a vesting period of one year. This function performs some calculations and burns VotingPower accordingly after converting the esLODE tokens to LODE. Although, these calculations are done incorrectly and under some scenarios more VotingPower than what the stakes actually has is burnt, reverting. This locks permanently the LODE tokens in the contract, as any unstake/withdraw function calls the convertEsLODEToLODE() function.

Proof of Concept:

```
USER1 CALLS ---> < contract_esLODE.approve(address(contract_StakingRewards), 6000e18) >

USER1 CALLS ---> < contract_StakingRewards.stakeEsLODE(6000e18) >
  contract_VotingPower.getVotes2(user1) -> 6000000000000000000000

22 WEEKS LATER...


USER1 CALLS ---> < contract_LODE.approve(address(contract_StakingRewards), 2300e18) >
  contract_VotingPower.getVotes2(user1) -> 6000000000000000000000

USER1 CALLS ---> < contract_StakingRewards.stakeLODE(2300e18, 90 days) >
  contract_VotingPower.getVotes2(user1) -> 9220000000000000000000

1 WEEKS LATER...


USER1 CALLS ---> < contract_LODE.approve(address(contract_StakingRewards), 2700e18) >
  contract_VotingPower.getVotes2(user1) -> 9220000000000000000000

USER1 CALLS ---> < contract_StakingRewards.stakeLODE(2700e18, 90 days) >
  contract_VotingPower.getVotes2(user1) -> 13000000000000000000000

9 WEEKS LATER...

  contract_VotingPower.getVotes2(user1) -> 13000000000000000000000

USER1 CALLS ---> < contract_StakingRewards.convertEsLODEToLODE() >
  contract_VotingPower.getVotes2(user1) -> 1079068493150684931640O

4 WEEKS LATER...


STAKING_INFO: User 0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90
  ------------------------------
  lodeAmount -> 868219178082191780600O
  stLODEAmount -> 1447287671232876712240O
  startTime -> 13305601
  lockTime -> 7776000
  relockStLODEAmount -> 0
  nextStakeId -> 1
  totalEsLODEStakedByUser -> 2317808219178082194000
  threeMonthRelockCount -> 0
  sixMonthRelockCount -> 0
  contract_VotingPower.getVotes2(user1) -> 1079068493150684931640O

USER1 CALLS ---> < contract_StakingRewards.unstakeLODE(868219178082191780600O) >
    ├─ [70359] StakingRewards::unstakeLODE(868219178082191780600O)
    │   ├─ emit Transfer(from: 0x0000000000000000000000000000000000000000, to: StakingRewards: [0x68aA7EB59caDfa6a52E5ADd2073969591fB8595E], value: 184109589041095891200)
    │   ├─ [3274] esLODE::transfer(0x0000000000000000000000000000000000000000, 460273972602739728000)
    │   │   ├─ emit Transfer(from: StakingRewards: [0x68aA7EB59caDfa6a52E5ADd2073969591fB8595E], to: 0x0000000000000000000000000000000000000000, value: 460273972602739728000)
    │   │   └─ ← true
    │   ├─ [25755] VotingPower::burn(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90, 2761643835616438368O0)
    │   │   └─ ← ()
    │   ├─ [24994] MockERC20::transfer(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90, 3599640000001752437235T)
    │   │   ├─ emit Transfer(from: StakingRewards: [0x68aA7EB59caDfa6a52E5ADd2073969591fB8595E], to: 0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90, value: 3599640000001752437235T)
    │   │   └─ ← true
    │   ├─ emit RewardsClaimed(user: 0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90, reward: 3599640000001752437235T)
    │   ├─ emit Transfer(from: StakingRewards: [0x68aA7EB59caDfa6a52E5ADd2073969591fB8595E], to: 0x0000000000000000000000000000000000000000, value: 1279945205479452054760O)
    │   ├─ [797] VotingPower::burn(0xE6b3367318C5e11a6eED3Cd0D850eC06A02E9b90, 1279945205479452054760O)
    │   │   └─ ← "Burn amount exceeds voting power"
    │   └─ ← "Burn amount exceeds voting power"
    └─ ← "Burn amount exceeds voting power"

Test result: FAILED. 0 passed; 1 failed; finished in 466.37ms

Failing tests:
Encountered 1 failing test in test/foundry/StakingRewards.t.sol:StakingRewardsTests
[FAIL. Reason: Burn amount exceeds voting power] test latestrem() (gas: 1205256)
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:H/Y:N/R:N/S:U (7.5)

Recommendation:

It is recommended to correct the convertEsLODEToLODE() function logic, so it does not excessively burn VotingPower, reverting.

Remediation Plan:

SOLVED: The Lodestar team solved the issue by implementing the recommended solution.

Commit ID : 97c84d98ef6011507183fcd37570084137b6626b.

4.12 (HAL-12) STAKERS MAY NOT BE ABLE TO UNSTAKE THEIR LODE TOKENS DUE TO ROUNDING ERROR - HIGH (7.5)

Commit IDs affected:
- 97c84d98ef6011507183fcd37570084137b6626b

Description:

The StakingRewards contract performs multiple calculations without using any kind of specific Math library. In Solidity, which is a statically typed language, each variable type has a fixed range of values that it can represent. For instance, an uint8 can represent values between 0 and 255, while an uint256 can represent values between 0 and approximately 10^77.

When it comes to mathematical operations involving fractional numbers, Solidity poses a challenge because it does not natively support floating-point numbers. All numbers in Solidity are considered to be integers. So, when you perform an operation that would ordinarily result in a fractional number, Solidity simply discards the fractional part, leading to precision loss.

For example, if you attempt to calculate 3 / 2 in Solidity, instead of getting 1.5 (the correct answer), you would receive 1 because the fractional part (.5) is discarded. This is an example of precision loss.

To avoid precision loss, we have to be aware of this behavior and implement strategies that take it into account:

1. Order of Operations: Always perform multiplication before division because this will help to preserve precision. For example, if you wanted to calculate (3 * 2) / 2, Solidity would give you the correct answer (3) because the multiplication is performed first.
2. Fixed-Point Libraries: Consider using libraries that provide

fixed-point arithmetic, such as the ABDK Math 64.64 library, or the FixedPoint library from OpenZeppelin. These libraries provide functions to work with fractional numbers with a high degree of precision.

3. Rounding: If you know you're going to be losing precision and you can't avoid it, consider whether you should be rounding up, rounding down, or rounding to the nearest number, and implement this in your contract.
4. Scaling: In many cases, it may be helpful to use a scaling factor. For instance, if you're working with Ether amounts, instead of storing them as Ether, you could store them as Wei (1 Ether = 10^18 Wei). This allows you to work with whole numbers while still keeping track of fractional amounts of Ether.
5. SafeMath Library: Use SafeMath library for basic arithmetic operations. This library includes safety checks that prevent overflow and underflow errors.

The StakingRewards contract does not really have any mitigation in place to avoid precision loss. Because of this, it was found through fuzzing that it is possible that users can not unstake their LODE tokens after performing multiple stakeLODE() calls. See Proof of Concept below.

Proof of Concept:

Steps:

1. LODEStaked;User(0xce. . .);Amount(1067762960486231292124);LockTime(7776000).
2. LODEStaked;User(0xce. . .);Amount(276120821005269572232);LockTime(7776000).
3. LODEStaked;User(0xce. . .);Amount(378398211923089032081);LockTime(7776000).
4. Relock;User(0xce. . .);RelockTime(7776000).

```
    ├─ [30394] StakingRewards::unstakeLODE(17222819934145890896437 [1.722e21])
    │   ├─ [24994] MockERC20::transfer(0xCE7Af7fa498727F32B767429333dDea67200f0B0, 650621464074716660342 [6.506e20])
    │   │   ├─ emit Transfer(from: StakingRewards: [0x68aA7EB59caDfa6a52E5ADd2073969591fB8595E], to: 0xCE7Af7fa498727F32B767429333dDea67200f0B0, value: 650621464074716660342 [6.506e20])
    │   │   └─ ← true
    │   ├─ emit RewardsClaimed(user: 0xCE7Af7fa498727F32B767429333dDea67200f0B0, reward: 650621464074716660342 [6.506e20])
    │   ├─ emit Debug(a: user.amount, b: 2497308890451155349831 [2.497e21])
    │   ├─ emit Debug(a: stLODEReduction, b: 2497308890451155349832 [2.497e21])
    │   └─ ← "WITHDRAW_ERROR()"
    └─ ← "WITHDRAW_ERROR()"

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 278.64ms
Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:H/Y:N/R:N/S:U (7.5)

Recommendation:

It is recommended to make use of the FixedPointMathLib to perform all the Mathematical operations, avoiding precision loss. Adjust the rounding direction accordingly to avoid, for example, in this case, that user.amount can ever be lower than the stLODEReduction calculation.

Remediation Plan:

SOLVED: The Lodestar team solved the issue by implementing the recommended solution.

Commit ID : 1c15f28d1a71b59262dbd3a5c542159c6091ad49.

4.13 (HAL-13) UPDATEWEEKLYREWARDS FUNCTION DOES NOT GUARANTEE THAT THE STAKINGREWARDS CONTRACT HAS ENOUGH REWARDS FOR ALL THE STAKERS - MEDIUM (5.6)

Commit IDs affected:
- a21ecb23a4308c2602ac63ee86d576f78d73c6e6

Description:

In the StakingRewards contract, the function updateWeeklyRewards() is a permissioned function, called by the RewardsRouter to update the weekly rewards:

Listing 11: StakingRewards.sol (Line 689)

```
679 /**
680  * @notice Permissioned function to update weekly rewards
681  * @param _weeklyRewards The amount of incoming weekly rewards
682  * @dev Can only be called by the router contract
683  */
684 function updateWeeklyRewards(uint256 _weeklyRewards) external {
685     require(msg.sender == routerContract, "StakingRewards:
 ↳ Unauthorized");
686     weeklyRewards = _weeklyRewards;
687     lastUpdateTimestamp = block.timestamp;
688     setStartTime(uint32(block.timestamp));
689     uint256 _wethPerSecond = calculateWethPerSecond(_weeklyRewards
 ↳ );
690     setEmission(_wethPerSecond);
691     emit WeeklyRewardsUpdated(_weeklyRewards);
692 }
```

This updateWeeklyRewards() function is called through the withdrawRewards() function, by a privileged account:

Listing 12: RewardRouter.sol (Lines 130-132)

```
68 /**
69  * @notice Permissioned function to reduce reserves on all markets
↳  and send to distributor
70  * @param lTokens[] The list of lTokens to draw reserves from
71  * @dev Verifies authorization via external Whitelist contract,
↳ loops over markets array and
72  * @dev reduces reserves according to the amount accrued since the
↳  previous withdrawal multiplied
73  * @dev by the withdrawMantissa. Swaps using different DEX's/
↳ contracts based on the underlying token
74  * @dev down to WETH, which is unwrapped into native ETH. The ETH
↳ is transferred using sendValue to
75  * @dev the distributor contract. After the transfer the hook to
↳ initiate rewards distribution is called
76  * @dev and the RewardsDistributed event is emitted.
77  */
78 function withdrawRewards(address[] memory lTokens) external
↳ nonReentrant {
79     require(WHITELIST.isWhitelisted(msg.sender), "RewardRouter:
↳ UNAUTHORIZED");
80     for (uint256 i = 0; i < lTokens.length; i++) {
81         uint256 currentReserves = ICERC20(lTokens[i]).
↳ totalReserves();
82         uint256 previousReserves = PreviousReserves[lTokens[i]];
83         uint256 delta = currentReserves - previousReserves;
84         uint256 withdrawAmount = (delta * withdrawMantissa) / BASE
↳ ;
85         if (delta == 0 || withdrawAmount == 0) {
86             continue;
87         }
88         require(ICERC20(lTokens[i])._reduceReserves(withdrawAmount
↳ ) == 0, "RewardRouter: Withdrawal Failed");
89         string memory name = ICERC20(lTokens[i]).symbol();
90         if (compareStrings(name, "lplvGLP") && withdrawAmount < 1
↳ ether) {
91             continue;
92         }
93         string memory underlyingSymbol;
94         IERC20Extended underlying;
95         uint256 minAmountOut;
96         if (!compareStrings(name, "lETH")) {
97             underlying = IERC20Extended(ICERC20(lTokens[i]).
↳ underlying());
```

```
98              underlyingSymbol = underlying.symbol();
99              if (!compareStrings(underlyingSymbol, "plvGLP")) {
100                 minAmountOut = Swap.getMinimumSwapAmountOut(
101                     underlying,
102                     IERC20Extended(address(WETH)),
103                     withdrawAmount,
104                     MAX_SLIPPAGE
105                 );
106             }
107         }
108         if (
109             compareStrings(underlyingSymbol, "USDC") ||
110             compareStrings(underlyingSymbol, "USDT") ||
111             compareStrings(underlyingSymbol, "DAI") ||
112             compareStrings(underlyingSymbol, "WBTC") ||
113             compareStrings(underlyingSymbol, "ARB") ||
114             compareStrings(underlyingSymbol, "GMX")
115         ) {
116             Swap.swapThroughUniswap(address(underlying), address(
 ↳ WETH), withdrawAmount, minAmountOut);
117         } else if (compareStrings(underlyingSymbol, "MAGIC") ||
 ↳ compareStrings(underlyingSymbol, "DPX")) {
118             Swap.swapThroughSushiswap(address(underlying), address
 ↳ (WETH), withdrawAmount, minAmountOut);
119         } else if (compareStrings(underlyingSymbol, "FRAX")) {
120             Swap.swapThroughFraxswap(address(underlying), address(
 ↳ WETH), withdrawAmount, minAmountOut);
121         } else if (compareStrings(underlyingSymbol, "plvGLP")) {
122             Swap.unwindPlutusPosition();
123         } else {
124             uint256 ethBalance = address(this).balance;
125             Swap.wrapEther(ethBalance);
126         }
127         uint256 newReserves = ICERC20(lTokens[i]).totalReserves();
128         PreviousReserves[lTokens[i]] = newReserves;
129     }
130     uint256 wethBalance = WETH.balanceOf(address(this));
131     require(WETH.transferFrom(address(this), DISTRIBUTOR,
 ↳ wethBalance), "RewardRouter: WETH Transfer Failed.");
132     StakingRewardsInterface(DISTRIBUTOR).updateWeeklyRewards(
 ↳ wethBalance);
133     emit RewardsDistributed(wethBalance, block.timestamp);
134 }
```

The RewardsRouter sends the WETH to the StakingRewards contract, and the StakingRewards contract adjusts the wethPerSecond state variable accordingly. At this point, the contract "assumes" that every week that amount of WETH will be distributed. Although nothing guarantees that this function will be called with a frequency lower than 7 days.

If the privileged user takes longer than 1 week to call the RewardsRouter.withdrawRewards() function, it is entirely possible that the StakingRewards contract becomes insolvent and users cannot claim their WETH rewards. This situation would always occur if all the stakers claimed their rewards during that week period.

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:L/R:N/S:U (5.6)

Recommendation:

It is recommended to guarantee that the RewardsRouter.withdrawRewards() function is called often enough. Consider also sending some extra WETH to the RewardsRouter contract.

Remediation Plan:

SOLVED: The Lodestar team states that this is known and that they will call this function periodically and will never take longer than 1 week to call it. Moreover, they mention that the contract will also have always more WETH rewards than the one sent through the withdrawRewards() function to account for possible precision loss.

4.14 (HAL-14) CALLING STAKINGREWARDS.UPDATEVOTINGCONTRACT FUNCTION CAN BREAK THE STAKINGREWARDS CONTRACT - MEDIUM (5.0)

Commit IDs affected:
- a21ecb23a4308c2602ac63ee86d576f78d73c6e6

Description:

The StakingRewards contract implements the function _updateVotingContract() that can be called by the contract owner to update the VotingPower contract address:

Listing 13: StakingRewards.sol (Line 992)

```
986 /**
987  * @notice Admin function to update the voting power contract
988  * @dev Can only be called by contract owner
989  */
990 function _updateVotingContract(address _votingContract) external
 ↳ onlyOwner {
991     require(_votingContract != address(0), "StakingRewards:
 ↳ Invalid Voting Contract");
992     votingContract = IVotingPower(_votingContract);
993     emit VotingContractUpdated(address(votingContract));
994 }
```

Although, if this function was called and in the new VotingPower contract the users did not have the same amount of _votingPower a Denial of Service would be caused in the StakingRewards contract blocking all the stakes/unstake/emergency withdraw operations and hence, blocking all the funds in the contract.

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:U (5.0)

Recommendation:

It is recommended to remove the _updateVotingContract() function. Consider also making the VotingPower contract upgradeable in case that new functionality has to be added.

Remediation Plan:

SOLVED: The Lodestar team partially solved the issue by making the VotingPower contract upgradeable.
Commit ID : 1c15f28d1a71b59262dbd3a5c542159c6091ad49.

4.15 (HAL-15) POSSIBLE DENIAL OF SERVICE BY REACHING BLOCK GAS LIMIT WHEN CALLING CONVERTESLODETOLODE FUNCTION - MEDIUM (5.0)

Commit IDs affected:
- a21ecb23a4308c2602ac63ee86d576f78d73c6e6

Description:

In the StakingRewards contract, every time esLODE is staked, a Stake struct is pushed into the esLODEStakes array:

Listing 14: StakingRewards.sol (Line 138)

```
124 /**
125  * @notice Stake esLODE tokens to earn rewards
126  * @param amount the amount the user wishes to stake (denom. in
 ↳ wei)
127  */
128 function stakeEsLODE(uint256 amount) external whenNotPaused
 ↳ nonReentrant {
129     require(esLODE.balanceOf(msg.sender) >= amount, "
 ↳ StakingRewards: Insufficient balance");
130     require(amount > 0, "StakingRewards: Invalid amount");
131     stakeEsLODEInternal(amount);
132 }
133
134 function stakeEsLODEInternal(uint256 amount) internal {
135     require(esLODE.transferFrom(msg.sender, address(this), amount)
 ↳ , "StakingRewards: Transfer failed");
136     stakers[msg.sender].nextStakeId += 1;
137
138     esLODEStakes[msg.sender].push(Stake({amount: amount,
 ↳ startTimestamp: block.timestamp, alreadyConverted: 0}));
139
140     stakers[msg.sender].totalEsLODEStakedByUser += amount; //
 ↳ Update total EsLODE staked by user
141     stakers[msg.sender].stLODEAmount += amount;
```

```
142
143     totalEsLODEStaked += amount;
144     totalStaked += amount;
145
146     UserInfo storage user = userInfo[msg.sender];
147
148     uint256 _prev = totalSupply();
149
150     updateShares();
151
152     unchecked {
153         user.amount += uint96(amount);
154         shares += uint96(amount);
155     }
156
157     user.wethRewardsDebt =
158         user.wethRewardsDebt +
159         int128(uint128(_calculateRewardDebt(accWethPerShare,
 ↳ uint96(amount))));
160
161     _mint(address(this), amount);
162
163     unchecked {
164         if (_prev + amount != totalSupply()) revert DEPOSIT_ERROR
 ↳ ();
165     }
166     emit StakedEsLODE(msg.sender, amount);
167 }
```

This esLODEStakes array is never decremented and is iterated over by the convertEsLODEToLODE() function. The size of esLODEStakes array is never limited, and any user can perform as many esLODE stakes as they wish.

This means that eventually this array could be large enough to cause reach the block gas limit when calling the convertEsLODEToLODE() function, locking permanently the esLODE tokens in the contract.

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:U (5.0)

Recommendation:

It is recommended to limit the size of the esLODEStakes array to a max. value. On the other hand, it is recommended to pop from the esLODEStakes array every time a Stake has been full converted to LODE tokens.

Remediation Plan:

SOLVED: The Lodestar team solved the issue by implementing the recommended solution.

Commit ID : 1c15f28d1a71b59262dbd3a5c542159c6091ad49.

4.16 (HAL-16) UPDATESHARES IS NOT CALLED IN CERTAIN CASES IN THE RELOCK FUNCTION - MEDIUM (5.0)

Commit IDs affected:
- a21ecb23a4308c2602ac63ee86d576f78d73c6e6

Description:

In the StakingRewards contract, every time shares are updated the updateShares() function should be called:

Listing 15: StakingRewards.sol (Lines 585,588)

```
568 /**
569  * @notice Update the staking rewards information to be current
570  * @dev Called before all reward state changing functions
571  */
572 function updateShares() public {
573     // if block.timestamp <= lastRewardSecond, already updated.
574     if (block.timestamp <= lastRewardSecond) {
575         return;
576     }
577
578     // if pool has no supply
579     if (shares == 0) {
580         lastRewardSecond = uint32(block.timestamp);
581         return;
582     }
583
584     unchecked {
585         accWethPerShare += rewardPerShare(wethPerSecond);
586     }
587
588     lastRewardSecond = uint32(block.timestamp);
589 }
```

This call is also necessary to update the accWethPerShare and lastRewardSecond state variables. Although, the updateShares()

function is not called properly in the relock() function, in the if (stLODEAdjustment >= relockStLODEAmount) code block.

This could lead to wrong WETH rewards calculations.

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:U (5.0)

Recommendation:

It is recommended to call the updateShares() function in the if (stLODEAdjustment >= relockStLODEAmount) code block.

Remediation Plan:

SOLVED: The Lodestar team solved the issue by implementing the recommended solution.

Commit ID : aba53cfd19189720cb8c32368176648d6aead960.

4.17 (HAL-17) STATE VARIABLES MISSING IMMUTABLE MODIFIER - INFORMATIONAL (0.0)

Commit IDs affected:
- a21ecb23a4308c2602ac63ee86d576f78d73c6e6

Description:

The `immutable` keyword was added to Solidity in 0.6.5. State variables can be marked `immutable` which causes them to be read-only, but only assignable in the constructor. The following state variables are missing the `immutable` modifier:

LodestarHandler.sol
- Line 11: `StakingRewardsInterface public STAKING;`

TokenClaim.sol
- Line 11: `address public admin;`

VotingPower.sol
- Line 16: `IStakingRewards public stakingRewards;`
- Line 17: `uint256 public votingStartTimestamp;`

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation:

It is recommended to add the `immutable` modifier to the state variables mentioned.

Remediation Plan:

SOLVED: The Lodestar team solved the issue by implementing the recommended solution.

Commit ID : aba53cfd19189720cb8c32368176648d6aead960.

4.18 (HAL-18) STATE VARIABLES MISSING CONSTANT MODIFIER - INFORMATIONAL (0.0)

Commit IDs affected:
- a21ecb23a4308c2602ac63ee86d576f78d73c6e6

Description:

State variables can be declared as constant or immutable. In both cases, the variables cannot be modified after the contract has been constructed. For constant variables, the value has to be fixed at compile-time, while for immutable, it can still be assigned at construction time. The following state variable is missing the constant modifier:

StakingConstants.sol
- Line 51: `bool public lockCanceled;`
(Consider removing this state variable as it is not used anywhere in the code)

VotingPower.sol
- Line 18: `uint256 public votingPeriod = 1 weeks;`

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation:

It is recommended to add the constant modifier to the state variables mentioned.

Remediation Plan:

SOLVED: The Lodestar team solved the issue by implementing the recommended solution.

Commit ID : aba53cfd19189720cb8c32368176648d6aead960.

4.19 (HAL-19) STRUCTS DO NOT FOLLOW THE TIGHT VARIABLE PACKING PATTERN - INFORMATIONAL (0.0)

Commit IDs affected:
- a21ecb23a4308c2602ac63ee86d576f78d73c6e6

Description:

In Solidity, data type packing within struct variables is a recommended practice to optimize gas usage and efficiency in smart contracts. This concept, called tight packing, can effectively save storage costs when implemented correctly.

This technique leverages the fact that Ethereum's storage model stores variables in slots, with each slot offering a capacity of 32 bytes. When data types that consume less than 32 bytes, such as uint8, bool, or address, are declared individually, each occupies a whole storage slot. However, when these smaller variables are grouped into a struct, they can share a storage slot, resulting in a significant reduction in storage requirements and, by extension, gas costs.

Despite these benefits, packing variables in structs requires caution and should be performed understanding its implications:
- Ordering Matters: The variables should be ordered from largest to smallest to utilize the storage space optimally. Incorrect ordering could lead to unused space within the storage slots.
- Explicit Type Sizes: It's recommended to use explicit type sizes (like uint8, uint16, uint32) over int256 and uint256.
- Potential Overflow: Be aware of potential overflows when using smaller integer types.

Code Location:

StakingContract.sol

Listing 16: StakingContract.sol

```
 9 struct Stake {
10     uint256 amount;
11     uint256 startTimestamp;
12     uint256 alreadyConverted;
13 }
14
15 struct StakingInfo {
16     uint256 lodeAmount;
17     uint256 stLODEAmount;
18     uint256 startTime;
19     uint256 lockTime;
20     uint256 relockStLODEAmount;
21     uint256 nextStakeId;
22     uint256 totalEsLODEStakedByUser;
23     uint256 threeMonthRelockCount;
24     uint256 sixMonthRelockCount;
25 }
```

References:

Tight Variable Packing

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

To optimize gas consumption and smart contract efficiency, it is recommended to pack smaller data types into structs, keeping in mind to order variables from largest to smallest, use explicit type sizes, and be wary of potential overflows when using smaller integer types.

Remediation Plan:

ACKNOWLEDGED: The Lodestar team acknowledged this.

4.20 (HAL-20) FLOATING PRAGMA - INFORMATIONAL (0.0)

Commit IDs affected:
- a21ecb23a4308c2602ac63ee86d576f78d73c6e6

Description:

Contracts should be deployed with the same compiler version and flags used during development and testing. Locking the pragma helps to ensure that contracts do not accidentally get deployed using another pragma. For example, an outdated pragma version might introduce bugs that affect the contract system negatively.

Code Location:

Most of the contracts in the repository are using the `pragma solidity ^0.8.0;` floating pragma.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:C (0.0)

Recommendation:

Consider locking the pragma version in the smart contracts. It is not recommended to use a floating pragma in production.

For example: `pragma solidity 0.8.19;`

Remediation Plan:

SOLVED: The Lodestar team solved the issue by implementing the recommended solution.

Commit ID : aba53cfd19189720cb8c32368176648d6aead960.

RECOMMENDATIONS OVERVIEW

1. Reset the stakers[msg.sender].totalEsLODEStakedByUser storage to 0 after the withdrawEsLODE() call in the StakingRewards contract.
2. Set the minimum stake period to one week or simply do not assign any voting power to users, which lockTime is set only to 10 seconds in the StakingRewards contract.
3. Either rebuild from scratch the delegating logic or otherwise remove it in the VotingPower contract.
4. Remove the user parameter from the convertEsLODEToLODE() function, and perform all the function logic using msg.sender in the StakingRewards contract.
5. Do not mint VotingPower when users stake for 10 seconds in the stakeLODE() function. Accordingly, the VotingPower should not be burnt when users unstake with a 0 or 10 lockTime in the unstakeLODE() function.
6. Update the lockTime variable when convertEsLODEToLODE() is called and the current lockTime of the user is 0.
7. Fix the convertEsLODEToLODE() function logic to prevent that esLODE tokens from getting stuck in the StakingRewards contract.
8. Always update the stakers[staker].startTime variable every time a user stakes in the StakingRewards contract.
9. Restrict users from relocking when their stakes' lockTime is equal to 10 in the StakingRewards contract.
10. Correct the convertEsLODEToLODE() function logic, so it does not excessively burn VotingPower, reverting.
11. Make use of the FixedPointMathLib to perform all the Mathematical operations in the StakingRewards contract to avoid precision loss. Adjust the rounding direction accordingly.
12. Make sure that the RewardsRouter.withdrawRewards() function is called often enough. Consider also sending some extra WETH to the RewardsRouter contract.
13. Remove the _updateVotingContract() function from the StakingRewards contract. Consider also making the VotingPower contract upgradeable in case that new functionality has to be added.
14. Limit the size of the esLODEStakes array to a max. value. On the other hand, it is recommended to pop from the esLODEStakes array every time a Stake has been full converted to LODE tokens in the StakingRewards contract.

15. Call the `updateShares()` function in the `if (stLODEAdjustment >= relockStLODEAmount) code block`
16. Add the `immutable` modifier to the state variables mentioned.
17. Add the `constant` modifier to the state variables mentioned.
18. Consider packing smaller data types into structs, keeping in mind to order variables from largest to smallest, use explicit type sizes, and be wary of potential overflows when using smaller integer types.
19. Lock the pragma version in all the smart contracts.

FUZZ TESTING

FUZZ TESTING

Fuzz testing is a testing technique that involves sending randomly generated or mutated inputs to a target system to identify unexpected behavior or vulnerabilities. In the context of smart contract assessment, fuzz testing can help identify potential security issues by exposing the smart contracts to a wide range of inputs that they may not have been designed to handle.

In this assessment, we conducted comprehensive fuzzing tests on the StakingRewards contract to assess its resilience to unexpected inputs. Our goal was to identify any potential vulnerabilities or flaws that could be exploited by an attacker or any wrong or unintended logic.

The following section provides a detailed description of the fuzzing methodology we used and the tools we employed. We believe that this information will be useful in helping the development team to understand and address the identified vulnerabilities, thereby improving the overall security posture of the smart contract.

Foundry is a smart contract development toolchain, and it was used to perform all the fuzz testing.

6.1 FUZZ TESTING SCRIPTS

In order to perform the fuzz testing, 5 different files were created:

- Fuzzer.t.sol: Implements the core logic of the fuzzer.
- FuzzHelper.t.sol: Implements all the wrappers used to call the different functions in the protocol. The whole project deployment is also defined in this file.
- FuzzProperties.t.sol: Implements all the functions used to test different properties/invariants.
- FuzzRandomizer.t.sol: Contract used to generate random numbers.
- FuzzStorage.t.sol: Contract used to hold the storage of the fuzzer.

These files were pushed to the following repository:
Halborn_LodestarStaking_Fuzzer

6.2 SETUP INSTRUCTIONS

To run the fuzzer a single run:

Listing 17

```
export FUZZ_ENTROPY=$(echo -n $RANDOM);forge test -vvvv --match-
↳ contract Fuzzer --match-test test_all_properties
```

To run the fuzzer with 10 runs:

Listing 18

```
for i in `seq 1 10`; do export FUZZ_ENTROPY=$(echo -n $RANDOM);
↳ forge test -vvvv --match-contract Fuzzer --match-test
↳ test_all_properties; done
```

AUTOMATED TESTING

7.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their ABIS and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

Slither results:

esLODE.sol

```
INFO:Detectors:
Pragma version^0.8.0 (contracts/esLODE.sol#2) allows old versions
solc-0.8.17 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Contract esLODE (contracts/esLODE.sol#6-54) is not in CapWords
Constant esLODE.initialSupply (contracts/esLODE.sol#10) is not in UPPER_CASE_WITH_UNDERSCORES
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
In a function esLODE.transfer(address,uint256) (contracts/esLODE.sol#27-33) variable esLODE._balances (contracts/esLODE.sol#11) is read multiple times
In a function esLODE.transferFrom(address,address,uint256) (contracts/esLODE.sol#45-53) variable esLODE._balances (contracts/esLODE.sol#11) is read multiple times
Reference: https://github.com/pessimistic-io/slitherin/blob/master/docs/multiple_storage_read.md
```

LodestarHandler.sol

```
INFO:Detectors:
LodestarHandler.addMarket(address,string,uint256,uint256,bool).market (contracts/LodestarHandler.sol#185) is a local variable never initialized
LodestarHandler.constructor(RouterInterface,VotingInterface,StakingRewardsInterface,ComptrollerInterface,Whitelist,address[],string[],uint256[],uint256[]).market (contracts/LodestarHandler.sol#47) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
Reentrancy in LodestarHandler.update() (contracts/LodestarHandler.sol#96-101):
        External calls:
        - updateStakingRewards() (contracts/LodestarHandler.sol#98)
                - ROUTER.withdrawRewards(markets) (contracts/LodestarHandler.sol#59)
        - updateLODEStreams() (contracts/LodestarHandler.sol#99)
                - UNITROLLER._setCompSpeeds(marketAddresses,supplySpeeds,borrowSpeeds) (contracts/LodestarHandler.sol#92)
        Event emitted after the call(s):
        - Updated(block.timestamp) (contracts/LodestarHandler.sol#100)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
Function LodestarHandler.updateRouter(RouterInterface) (contracts/LodestarHandler.sol#141-146) has a dubious typecast: address<=address
Function LodestarHandler.updateVoting(VotingInterface) (contracts/LodestarHandler.sol#148-153) has a dubious typecast: address<=address
Function LodestarHandler.updateUnitroller(ComptrollerInterface) (contracts/LodestarHandler.sol#155-160) has a dubious typecast: address<=address
Function LodestarHandler.updateWhitelist(Whitelist) (contracts/LodestarHandler.sol#162-167) has a dubious typecast: address<=address
Function LodestarHandler.addMarket(address,string,uint256,uint256,bool) (contracts/LodestarHandler.sol#169-194) has a dubious typecast: address<=address
Reference: https://github.com/pessimistic-io/slitherin/blob/master/docs/dubious_typecast.md
INFO:Detectors:
Function ComptrollerInterface._setCompSpeeds(address[],uint256[],uint256[]) (contracts/Interfaces/HandlerInterfaces.sol#20) is not in mixedCase
Variable LodestarHandler.ROUTER (contracts/LodestarHandler.sol#9) is not in mixedCase
Variable LodestarHandler.VOTING (contracts/LodestarHandler.sol#10) is not in mixedCase
Variable LodestarHandler.STAKING (contracts/LodestarHandler.sol#11) is not in mixedCase
Variable LodestarHandler.UNITROLLER (contracts/LodestarHandler.sol#12) is not in mixedCase
Variable LodestarHandler.WHITELIST (contracts/LodestarHandler.sol#13) is not in mixedCase
Parameter Whitelist.updateWhitelist(address,bool)._address (contracts/Utils/Whitelist.sol#11) is not in mixedCase
Parameter Whitelist.updateWhitelist(address,bool)._isActive (contracts/Utils/Whitelist.sol#12) is not in mixedCase
Parameter Whitelist.getWhitelisted(address)._address (contracts/Utils/Whitelist.sol#17) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
LodestarHandler.STAKING (contracts/LodestarHandler.sol#11) should be immutable
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-immutable
INFO:Detectors:
In a function LodestarHandler.addMarket(address,string,uint256,uint256,bool) (contracts/LodestarHandler.sol#169-194) variable LodestarHandler.tokenMapping (contracts/LodestarHandler.sol#24) is read multiple times
Reference: https://github.com/pessimistic-io/slitherin/blob/master/docs/multiple_storage_read.md
```

RewardRouter.sol

```
INFO:Detectors:
Reentrancy in RewardsRouter.withdrawRewards(address[]) (contracts/RewardRouter.sol#78-134):
        External calls:
        - currentReserves = ICERC20(lTokens[1]).totalReserves() (contracts/RewardRouter.sol#81)
        - require(bool,string)(ICERC20(lTokens[1])._reduceReserves(withdrawAmount) == 0,RewardRouter: Withdrawal Failed) (contracts/RewardRouter.sol#88)
        - Swap.swapThroughUniswap(address(underlying),address(WETH),withdrawAmount,minAmountOut) (contracts/RewardRouter.sol#116)
                - amountOut = UNI_ROUTER.exactInput(params) (contracts/Utils/Swap.sol#25)
        - Swap.swapThroughSushiswap(address(underlying),address(WETH),withdrawAmount,minAmountOut) (contracts/RewardRouter.sol#118)
                - SUSHI_ROUTER.swapExactTokensForTokensSupportingFeeOnTransferTokens(amountIn,minAmountOut,path,to,deadline) (contracts/Utils/Swap.sol#41)
        - Swap.swapThroughFraxswap(address(underlying),address(WETH),withdrawAmount,minAmountOut) (contracts/RewardRouter.sol#120)
                - FRAX_ROUTER.swapExactTokensForTokensSupportingFeeOnTransferTokens(amountIn,minAmountOut,path,to,deadline) (contracts/Utils/Swap.sol#55)
        - Swap.unwindPlutusPosition() (contracts/RewardRouter.sol#122)
                - PLUTUS_DEPOSITOR.redeemAll() (contracts/Utils/Swap.sol#60)
                - GLP_ROUTER.unstakeAndRedeemGlp(address(WETH),glpAmount,minOut,address(this)) (contracts/Utils/Swap.sol#64)
        - Swap.wrapEther(ethBalance) (contracts/RewardRouter.sol#125)
                - (sent) = address(WETH).call{value: amount}() (contracts/Utils/Swap.sol#77)
        - newReserves = ICERC20(lTokens[1]).totalReserves() (contracts/RewardRouter.sol#127)
        External calls sending eth:
        - Swap.wrapEther(ethBalance) (contracts/RewardRouter.sol#125)
                - (sent) = address(WETH).call{value: amount}() (contracts/Utils/Swap.sol#77)
```

```
        State variables written after the call(s):
        - PreviousReserves[ITokens[i]] = newReserves (contracts/RewardRouter.sol#128)
        RouterConstants.PreviousReserves (contracts/Utils/RouterConstants.sol#11) can be used in cross function reentrancies:
        - RouterConstants.PreviousReserves (contracts/Utils/RouterConstants.sol#11)
        - RewardsRouter._initializeMarkets(address[]) (contracts/RewardRouter.sol#156-167)
        - RewardsRouter.initialize(address[],address[],Whitelist,address) (contracts/RewardRouter.sol#27-55)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities
INFO:Detectors:
RewardsRouter RewardsRouter._withdraw(address[]) (contracts/RewardRouter.sol#173-189) might be vulnerable to double-entry token exploit
Reference: https://github.com/pessimistic-io/slitherin/blob/master/docs/double_entry_token_possibility.md
INFO:Detectors:
Function IUniswapV2Pair.initialize(address,address) (contracts/Interfaces/UniswapV2Interface.sol#81) is an unprotected initializer.
Reference: https://github.com/pessimistic-io/slitherin/blob/master/docs/unprotected_initialize.md
INFO:Detectors:
Swap.getMinimumSwapAmountOut(IERC20Extended,IERC20Extended,uint256,uint256) (contracts/Utils/Swap.sol#100-116) performs a multiplication on the result of a division:
        - conversionRatio = (tokenAPrice * BASE) / tokenBPrice (contracts/Utils/Swap.sol#111)
        - amountOutRaw = (swapAmount * conversionRatio) / BASE (contracts/Utils/Swap.sol#112)
Swap.getMinimumSwapAmountOut(IERC20Extended,IERC20Extended,uint256,uint256) (contracts/Utils/Swap.sol#100-116) performs a multiplication on the result of a division:
        - amountOutRaw = (swapAmount * conversionRatio) / BASE (contracts/Utils/Swap.sol#112)
        - (amountOutRaw * slippageFactor) / (decimalTruncation * BASE) (contracts/Utils/Swap.sol#115)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
Swap.getSequencerStatus(address).status (contracts/Utils/Swap.sol#139) is a local variable never initialized
RewardsRouter.withdrawRewards(address[]).underlyingSymbol (contracts/RewardRouter.sol#93) is a local variable never initialized
RewardsRouter.withdrawRewards(address[]).underlying (contracts/RewardRouter.sol#94) is a local variable never initialized
RewardsRouter.withdrawRewards(address[]).minAmountOut (contracts/RewardRouter.sol#95) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
RewardsRouter.initialize(address[],address[],Whitelist,address) (contracts/RewardRouter.sol#27-55) ignores return value by IERC20(underlyingTokens[i]).approve(address(SUSHI_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#39)
RewardsRouter.initialize(address[],address[],Whitelist,address) (contracts/RewardRouter.sol#27-55) ignores return value by IERC20(underlyingTokens[i]).approve(address(UNI_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#40)
RewardsRouter.initialize(address[],address[],Whitelist,address) (contracts/RewardRouter.sol#27-55) ignores return value by IERC20(underlyingTokens[i]).approve(address(FRAX_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#41)
RewardsRouter.initialize(address[],address[],Whitelist,address) (contracts/RewardRouter.sol#27-55) ignores return value by WETH.approve(address(GLP),type()(uint256).max) (contracts/RewardRouter.sol#47)
RewardsRouter.initialize(address[],address[],Whitelist,address) (contracts/RewardRouter.sol#27-55) ignores return value by WETH.approve(address(this),type()(uint256).max) (contracts/RewardRouter.sol#48)
RewardsRouter.initialize(address[],address[],Whitelist,address) (contracts/RewardRouter.sol#27-55) ignores return value by GLP.approve(address(GLP_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#49)
RewardsRouter.initialize(address[],address[],Whitelist,address) (contracts/RewardRouter.sol#27-55) ignores return value by SGLP.approve(address(GLP_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#50)
RewardsRouter.initialize(address[],address[],Whitelist,address) (contracts/RewardRouter.sol#27-55) ignores return value by PLVGLP.approve(address(PLUTUS_DEPOSITOR),type()(uint256).max) (contracts/RewardRouter.sol#51)
RewardsRouter._initializeMarkets(address[]) (contracts/RewardRouter.sol#156-167) ignores return value by underlyingToken.approve(address(SUSHI_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#163)
RewardsRouter._initializeMarkets(address[]) (contracts/RewardRouter.sol#156-167) ignores return value by underlyingToken.approve(address(UNI_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#164)
RewardsRouter._initializeMarkets(address[]) (contracts/RewardRouter.sol#156-167) ignores return value by underlyingToken.approve(address(FRAX_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#165)
Swap.unwindPlutusPosition() (contracts/Utils/Swap.sol#59-65) ignores return value by GLP_ROUTER.unstakeAndRedeemGlp(address(WETH),glpAmount,minOut,address(this)) (contracts/Utils/Swap.sol#64)
Swap.glpRedeem() (contracts/Utils/Swap.sol#71-74) ignores return value by GLP_ROUTER.unstakeAndRedeemGlp(address(WETH),balance,0,address(this)) (contracts/Utils/Swap.sol#73)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
INFO:Detectors:
Swap withdrawWETH parameter from is not related to msg.sender require(bool,string)(WETH.transferFrom(address(this),msg.sender,amount),Transfer must succeed) (contracts/Utils/Swap.sol#91)
RewardsRouter withdrawRewards parameter from is not related to msg.sender require(bool,string)(WETH.transferFrom(address(this),DISTRIBUTOR,wethBalance),RewardRouter: WETH Transfer Failed.) (contracts/RewardRouter.sol#131)
RewardsRouter._withdraw parameter from is not related to msg.sender require(bool,string)(IERC20(markets[i]).transferFrom(address(this),msg.sender,tokenBalance),Token Transfer Failed) (contracts/RewardRouter.sol#183-186)
Reference: https://ventral.digital/posts/2022/8/18/sznsdaos-bountyboard-unauthorized-transferfrom-vulnerability
INFO:Detectors:
RewardsRouter.initialize(address[],address[],Whitelist,address)._distributor (contracts/RewardRouter.sol#31) lacks a zero-check on :
        - DISTRIBUTOR = _distributor (contracts/RewardRouter.sol#35)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
INFO:Detectors:
RewardsRouter.initialize(address[],address[],Whitelist,address) (contracts/RewardRouter.sol#27-55) has external calls inside a loop: IERC20(underlyingTokens[i]).approve(address(SUSHI_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#39)
RewardsRouter.initialize(address[],address[],Whitelist,address) (contracts/RewardRouter.sol#27-55) has external calls inside a loop: IERC20(underlyingTokens[i]).approve(address(UNI_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#40)
RewardsRouter.initialize(address[],address[],Whitelist,address) (contracts/RewardRouter.sol#27-55) has external calls inside a loop: IERC20(underlyingTokens[i]).approve(address(FRAX_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#41)
RewardsRouter.initialize(address[],address[],Whitelist,address) (contracts/RewardRouter.sol#27-55) has external calls inside a loop: PreviousReserves[ITokens[i_scope_0]] = ICERC20(ITokens[i_scope_0]).totalReserves() (contracts/RewardRouter.sol#45)
RewardsRouter.withdrawRewards(address[]) (contracts/RewardRouter.sol#78-134) has external calls inside a loop: currentReserves = ICERC20(ITokens[i]).totalReserves() (contracts/RewardRouter.sol#81)
RewardsRouter.withdrawRewards(address[]) (contracts/RewardRouter.sol#78-134) has external calls inside a loop: require(bool,string)(ICERC20(ITokens[i])._reduceReserves(withdrawAmount) == 0,RewardRouter: Withdrawal Failed) (contracts/RewardRouter.sol#88)
RewardsRouter.withdrawRewards(address[]) (contracts/RewardRouter.sol#78-134) has external calls inside a loop: name = ICERC20(ITokens[i]).symbol() (contracts/RewardRouter.sol#89)
RewardsRouter.withdrawRewards(address[]) (contracts/RewardRouter.sol#78-134) has external calls inside a loop: underlying = ICERC20Extended(ICERC20(ITokens[i]).underlying()) (contracts/RewardRouter.sol#97)
RewardsRouter.withdrawRewards(address[]) (contracts/RewardRouter.sol#78-134) has external calls inside a loop: underlyingSymbol = underlying.symbol() (contracts/RewardRouter.sol#98)
Swap.getMinimumSwapAmountOut(IERC20Extended,IERC20Extended,uint256,uint256) (contracts/Utils/Swap.sol#100-116) has external calls inside a loop: tokenBDecimals = tokenB.decimals() (contracts/Utils/Swap.sol#108)
Swap.getPriceFromChainlink(AggregatorV3Interface) (contracts/Utils/Swap.sol#123-131) has external calls inside a loop: (roundId,price,startedAt,updatedAt,answeredInRound) = aggregator.latestRoundData() (contracts/Utils/Swap.sol#124-125)
Swap.getPriceFromChainlink(AggregatorV3Interface) (contracts/Utils/Swap.sol#123-131) has external calls inside a loop: uint256(price) * 10 ** (18 - uint256(aggregator.decimals())) (contracts/Utils/Swap.sol#130)
Swap.swapThroughUniswap(address,address,uint256,uint256) (contracts/Utils/Swap.sol#9-27) has external calls inside a loop: amountOut = UNI_ROUTER.exactInput(params) (contracts/Utils/Swap.sol#25)
RewardsRouter.withdrawRewards(address[]) (contracts/RewardRouter.sol#78-134) has external calls inside a loop: newReserves = ICERC20(ITokens[i]).totalReserves() (contracts/RewardRouter.sol#127)
Swap.swapThroughSushiswap(address,address,uint256,uint256) (contracts/Utils/Swap.sol#30-42) has external calls inside a loop: SUSHI_ROUTER.swapExactTokensForTokensSupportingFeeOnTransferTokens(amountIn,minAmountOut,path,to,deadline) (contracts/Utils/Swap.sol#41)
Swap.swapThroughFraxswap(address,address,uint256,uint256) (contracts/Utils/Swap.sol#44-56) has external calls inside a loop: FRAX_ROUTER.swapExactTokensForTokensSupportingFeeOnTransferTokens(amountIn,minAmountOut,path,to,deadline) (contracts/Utils/Swap.sol#55)
Swap.unwindPlutusPosition() (contracts/Utils/Swap.sol#59-65) has external calls inside a loop: PLUTUS_DEPOSITOR.redeemAll() (contracts/Utils/Swap.sol#60)
Swap.unwindPlutusPosition() (contracts/Utils/Swap.sol#59-65) has external calls inside a loop: glpAmount = GLP.balanceOf(address(this)) (contracts/Utils/Swap.sol#61)
Swap.unwindPlutusPosition() (contracts/Utils/Swap.sol#59-65) has external calls inside a loop: GLP_ROUTER.unstakeAndRedeemGlp(address(WETH),glpAmount,minOut,address(this)) (contracts/Utils/Swap.sol#64)
Swap.wrapEther(uint256) (contracts/Utils/Swap.sol#76-81) has external calls inside a loop: (sent) = address(WETH).call{value: amount}() (contracts/Utils/Swap.sol#77)
Swap.wrapEther(uint256) (contracts/Utils/Swap.sol#76-81) has external calls inside a loop: wethAmount = WETH.balanceOf(address(this)) (contracts/Utils/Swap.sol#79)
RewardsRouter._initializeMarkets(address[]) (contracts/RewardRouter.sol#156-167) has external calls inside a loop: reserves = ICERC20(newMarkets[i]).totalReserves() (contracts/RewardRouter.sol#160)
RewardsRouter._initializeMarkets(address[]) (contracts/RewardRouter.sol#156-167) has external calls inside a loop: underlyingToken = IERC20(ICERC20(newMarkets[i]).underlying()) (contracts/RewardRouter.sol#162)
RewardsRouter._initializeMarkets(address[]) (contracts/RewardRouter.sol#156-167) has external calls inside a loop: underlyingToken.approve(address(SUSHI_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#163)
RewardsRouter._initializeMarkets(address[]) (contracts/RewardRouter.sol#156-167) has external calls inside a loop: underlyingToken.approve(address(UNI_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#164)
RewardsRouter._initializeMarkets(address[]) (contracts/RewardRouter.sol#156-167) has external calls inside a loop: underlyingToken.approve(address(FRAX_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#165)
RewardsRouter._withdraw(address[]) (contracts/RewardRouter.sol#173-189) has external calls inside a loop: (sent) = msg.sender.call{value: ethBalance}() (contracts/RewardRouter.sol#179)
RewardsRouter._withdraw(address[]) (contracts/RewardRouter.sol#173-189) has external calls inside a loop: tokenBalance = IERC20(markets[i]).balanceOf(address(this)) (contracts/RewardRouter.sol#182)
RewardsRouter._withdraw(address[]) (contracts/RewardRouter.sol#173-189) has external calls inside a loop: require(bool,string)(IERC20(markets[i]).transferFrom(address(this),msg.sender,tokenBalance),Token Transfer Failed) (contracts/RewardRouter.sol#183-186)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#calls-inside-a-loop
INFO:Detectors:
Reentrancy in RewardsRouter._initializeMarkets(address[]) (contracts/RewardRouter.sol#156-167):
        External calls:
        - reserves = ICERC20(newMarkets[i]).totalReserves() (contracts/RewardRouter.sol#160)
        - underlyingToken.approve(address(SUSHI_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#163)
        - underlyingToken.approve(address(UNI_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#164)
        - underlyingToken.approve(address(FRAX_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#165)
        State variables written after the call(s):
        - PreviousReserves[newMarkets[i]] = reserves (contracts/RewardRouter.sol#161)
Reentrancy in RewardsRouter.initialize(address[],address[],Whitelist,address) (contracts/RewardRouter.sol#27-55):
        External calls:
        - IERC20(underlyingTokens[i]).approve(address(SUSHI_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#39)
        - IERC20(underlyingTokens[i]).approve(address(UNI_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#40)
        - IERC20(underlyingTokens[i]).approve(address(FRAX_ROUTER),type()(uint256).max) (contracts/RewardRouter.sol#41)
        - PreviousReserves[ITokens[i_scope_0]] = ICERC20(ITokens[i_scope_0]).totalReserves() (contracts/RewardRouter.sol#45)
        State variables written after the call(s):
        - PreviousReserves[ITokens[i_scope_0]] = ICERC20(ITokens[i_scope_0]).totalReserves() (contracts/RewardRouter.sol#45)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2
INFO:Detectors:
Reentrancy in RewardsRouter.withdrawRewards(address[]) (contracts/RewardRouter.sol#78-134):
        External calls:
        - currentReserves = ICERC20(ITokens[i]).totalReserves() (contracts/RewardRouter.sol#81)
        - require(bool,string)(ICERC20(ITokens[i])._reduceReserves(withdrawAmount) == 0,RewardRouter: Withdrawal Failed) (contracts/RewardRouter.sol#88)
        - Swap.swapThroughUniswap(address(underlying),address(WETH),withdrawAmount,minAmountOut) (contracts/RewardRouter.sol#116)
            - amountOut = UNI_ROUTER.exactInput(params) (contracts/Utils/Swap.sol#25)
        - Swap.swapThroughSushiswap(address(underlying),address(WETH),withdrawAmount,minAmountOut) (contracts/RewardRouter.sol#118)
            - SUSHI_ROUTER.swapExactTokensForTokensSupportingFeeOnTransferTokens(amountIn,minAmountOut,path,to,deadline) (contracts/Utils/Swap.sol#41)
        - Swap.swapThroughFraxswap(address(underlying),address(WETH),withdrawAmount,minAmountOut) (contracts/RewardRouter.sol#120)
            - FRAX_ROUTER.swapExactTokensForTokensSupportingFeeOnTransferTokens(amountIn,minAmountOut,path,to,deadline) (contracts/Utils/Swap.sol#55)
        - Swap.unwindPlutusPosition() (contracts/RewardRouter.sol#122)
            - PLUTUS_DEPOSITOR.redeemAll() (contracts/Utils/Swap.sol#60)
            - GLP_ROUTER.unstakeAndRedeemGlp(address(WETH),glpAmount,minOut,address(this)) (contracts/Utils/Swap.sol#64)
        - Swap.wrapEther(ethBalance) (contracts/RewardRouter.sol#125)
            - (sent) = address(WETH).call{value: amount}() (contracts/Utils/Swap.sol#77)
        - newReserves = ICERC20(ITokens[i]).totalReserves() (contracts/RewardRouter.sol#127)
        - require(bool,string)(WETH.transferFrom(address(this),DISTRIBUTOR,wethBalance),RewardRouter: WETH Transfer Failed.) (contracts/RewardRouter.sol#131)
        - StakingRewardsInterface(DISTRIBUTOR).updateWeeklyRewards(wethBalance) (contracts/RewardRouter.sol#132)
        External calls sending eth:
        - Swap.wrapEther(ethBalance) (contracts/RewardRouter.sol#125)
            - (sent) = address(WETH).call{value: amount}() (contracts/Utils/Swap.sol#77)
        Event emitted after the call(s):
        - RewardsDistributed(wethBalance,block.timestamp) (contracts/RewardRouter.sol#133)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
Swap.getSequencerStatus(address) (contracts/Utils/Swap.sol#138-147) uses timestamp for comparisons
        Dangerous comparisons:
        - answer == 0 && block.timestamp - startedAt > GRACE_PERIOD_TIME (contracts/Utils/Swap.sol#141)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
Function Swap.unwindPlutusPosition() (contracts/Utils/Swap.sol#59-65) has a dubious typecast: address<=address
Function Swap.glpRedeem() (contracts/Utils/Swap.sol#71-74) has a dubious typecast: address<=address
Function RewardsRouter.initialize(address[],address[],Whitelist,address) (contracts/RewardRouter.sol#27-55) has a dubious typecast: address<=IERC20
Function RewardsRouter.withdrawRewards(address[]) (contracts/RewardRouter.sol#78-134) has a dubious typecast: IERC20Extended<=address
Function RewardsRouter.withdrawRewards(address[]) (contracts/RewardRouter.sol#78-134) has a dubious typecast: address<=address
Function RewardsRouter._withdraw(address[]) (contracts/RewardRouter.sol#173-189) has a dubious typecast: address<=IERC20
Reference: https://github.com/pessimistic-io/slitherin/blob/master/docs/dubious_typecast.md
INFO:Detectors:
Swap.getSequencerStatus(address) (contracts/Utils/Swap.sol#138-147) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code
INFO:Detectors:
Pragma version^0.8.10 (contracts/Interfaces/AggregatorV3Interface.sol#2) allows old versions
Pragma version>=0.7.5 (contracts/Interfaces/ISwapRouter.sol#2) allows old versions
Pragma version>=0.5.0 (contracts/Interfaces/IUniswapV3SwapCallback.sol#2) allows old versions
```

```
Pragma version^0.8.10 (contracts/Interfaces/UniswapV2Interface.sol#2) allows old versions
Pragma version^0.8.10 (contracts/RewardRouter.sol#2) allows old versions
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Low level call in RewardsRouter._withdraw(address[]) (contracts/RewardRouter.sol#173-189):
        - (sent) = msg.sender.call{value: ethBalance}() (contracts/RewardRouter.sol#179)
Low level call in Swap.wrapEther(uint256) (contracts/Utils/Swap.sol#76-81):
        - (sent) = address(WETH).call{value: amount}() (contracts/Utils/Swap.sol#77)
Low level call in Swap.withdrawETH() (contracts/Utils/Swap.sol#94-98):
        - (sent) = msg.sender.call{value: balance}() (contracts/Utils/Swap.sol#96)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
INFO:Detectors:
Function ICERC20._reduceReserves(uint256) (contracts/Interfaces/Interfaces.sol#57) is not in mixedCase
Function SushiRouterInterface.WETH() (contracts/Interfaces/Interfaces.sol#61) is not in mixedCase
Function IUniswapV2Pair.DOMAIN_SEPARATOR() (contracts/Interfaces/UniswapV2Interface.sol#27) is not in mixedCase
Function IUniswapV2Pair.PERMIT_TYPEHASH() (contracts/Interfaces/UniswapV2Interface.sol#29) is not in mixedCase
Function IUniswapV2Pair.MINIMUM_LIQUIDITY() (contracts/Interfaces/UniswapV2Interface.sol#55) is not in mixedCase
Parameter RewardsRouter.initialize(address[],address[],Whitelist,address)._whitelist (contracts/RewardRouter.sol#30) is not in mixedCase
Parameter RewardsRouter.initialize(address[],address[],Whitelist,address)._distributor (contracts/RewardRouter.sol#31) is not in mixedCase
Function RewardsRouter._updateDistributor(address) (contracts/RewardRouter.sol#144-149) is not in mixedCase
Function RewardsRouter._initializeMarkets(address[]) (contracts/RewardRouter.sol#156-167) is not in mixedCase
Function RewardsRouter._withdraw(address[]) (contracts/RewardRouter.sol#173-189) is not in mixedCase
Variable RewardsRouter.WHITELIST (contracts/RewardRouter.sol#14) is not in mixedCase
Variable RewardsRouter.DISTRIBUTOR (contracts/RewardRouter.sol#15) is not in mixedCase
Variable RewardsRouter.MAX_SLIPPAGE (contracts/RewardRouter.sol#16) is not in mixedCase
Variable RouterConstants.PreviousReserves (contracts/Utils/RouterConstants.sol#11) is not in mixedCase
Constant RouterConstants.1ETH (contracts/Utils/RouterConstants.sol#13) is not in UPPER_CASE_WITH_UNDERSCORES
Function Swap._setAggregators(address[],address[]) (contracts/Utils/Swap.sol#150-162) is not in mixedCase
Parameter Whitelist.updateWhitelist(address,bool)._address (contracts/Utils/Whitelist.sol#11) is not in mixedCase
Parameter Whitelist.updateWhitelist(address,bool)._isActive (contracts/Utils/Whitelist.sol#12) is not in mixedCase
Parameter Whitelist.getWhitelisted(address)._address (contracts/Utils/Whitelist.sol#17) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
Variable IUniswapV3SwapCallback.uniswapV3SwapCallback(int256,int256,bytes).amount0Delta (contracts/Interfaces/IUniswapV3SwapCallback.sol#16) is too similar to IUniswapV3SwapCallback.uniswapV3SwapCallback(int256,int256,bytes).amount1Delta
 (contracts/Interfaces/IUniswapV3SwapCallback.sol#16)
Variable Swap.swapThroughFraxswap(address,address,uint256,uint256).token0Address (contracts/Utils/Swap.sol#45) is too similar to Swap.swapThroughUniswap(address,address,uint256,uint256).token1Address (contracts/Utils/Swap.sol#11)
Variable Swap.swapThroughFraxswap(address,address,uint256,uint256).token0Address (contracts/Utils/Swap.sol#45) is too similar to Swap.swapThroughFraxswap(address,address,uint256,uint256).token1Address (contracts/Utils/Swap.sol#46)
Variable Swap.swapThroughSushiswap(address,address,uint256,uint256).token0Address (contracts/Utils/Swap.sol#31) is too similar to Swap.swapThroughUniswap(address,address,uint256,uint256).token1Address (contracts/Utils/Swap.sol#11)
Variable Swap.swapThroughSushiswap(address,address,uint256,uint256).token0Address (contracts/Utils/Swap.sol#31) is too similar to Swap.swapThroughFraxswap(address,address,uint256,uint256).token1Address (contracts/Utils/Swap.sol#46)
Variable Swap.swapThroughUniswap(address,address,uint256,uint256).token0Address (contracts/Utils/Swap.sol#10) is too similar to Swap.swapThroughSushiswap(address,address,uint256,uint256).token1Address (contracts/Utils/Swap.sol#32)
Variable Swap.swapThroughSushiswap(address,address,uint256,uint256).token0Address (contracts/Utils/Swap.sol#31) is too similar to Swap.swapThroughSushiswap(address,address,uint256,uint256).token1Address (contracts/Utils/Swap.sol#32)
Variable Swap.swapThroughUniswap(address,address,uint256,uint256).token0Address (contracts/Utils/Swap.sol#10) is too similar to Swap.swapThroughUniswap(address,address,uint256,uint256).token1Address (contracts/Utils/Swap.sol#11)
Variable Swap.swapThroughFraxswap(address,address,uint256,uint256).token0Address (contracts/Utils/Swap.sol#45) is too similar to Swap.swapThroughSushiswap(address,address,uint256,uint256).token1Address (contracts/Utils/Swap.sol#32)
Variable Swap.swapThroughUniswap(address,address,uint256,uint256).token0Address (contracts/Utils/Swap.sol#10) is too similar to Swap.swapThroughFraxswap(address,address,uint256,uint256).token1Address (contracts/Utils/Swap.sol#46)
Variable Swap.getMinimumSwapAmountOut(IERC20Extended,IERC20Extended,uint256,uint256).tokenAAggregator (contracts/Utils/Swap.sol#106) is too similar to Swap.getMinimumSwapAmountOut(IERC20Extended,IERC20Extended,uint256,uint256).tokenBAggr
egator (contracts/Utils/Swap.sol#107)
Variable Swap.getMinimumSwapAmountOut(IERC20Extended,IERC20Extended,uint256,uint256).tokenAPrice (contracts/Utils/Swap.sol#109) is too similar to Swap.getMinimumSwapAmountOut(IERC20Extended,IERC20Extended,uint256,uint256).tokenBPrice (co
ntracts/Utils/Swap.sol#110)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#variable-names-too-similar
INFO:Detectors:
RouterConstants.USDC (contracts/Utils/RouterConstants.sol#14) is never used in RewardsRouter (contracts/RewardRouter.sol#12-192)
RouterConstants.PRICE_ORACLE (contracts/Utils/RouterConstants.sol#27-28) is never used in RewardsRouter (contracts/RewardRouter.sol#12-192)
RouterConstants.ETHUSD_AGGREGATOR (contracts/Utils/RouterConstants.sol#30-31) is never used in RewardsRouter (contracts/RewardRouter.sol#12-192)
RouterConstants.SEQUENCER (contracts/Utils/RouterConstants.sol#32-33) is never used in RewardsRouter (contracts/RewardRouter.sol#12-192)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-state-variable
INFO:Detectors:
Function Swap.swapThroughUniswap(address,address,uint256,uint256) (contracts/Utils/Swap.sol#9-27) contains magic number: 3000
Function Swap.getPriceFromChainlink(AggregatorV3Interface) (contracts/Utils/Swap.sol#123-131) contains magic number: 10
Function RewardsRouter.initialize(address[],address[],Whitelist,address) (contracts/RewardRouter.sol#27-55) contains magic numbers: 5e17, 1e16
Function RewardsRouter.withdrawRewards(address[]) (contracts/RewardRouter.sol#78-134) contains magic number: 10000000000000000000
Reference: https://github.com/pessimistic-io/slitherin/blob/master/docs/magic_number.md
INFO:Detectors:
In a function RewardsRouter.withdrawRewards(address[]) (contracts/RewardRouter.sol#78-134) variable RouterConstants.WETH (contracts/Utils/RouterConstants.sol#15) is read multiple times
Reference: https://github.com/pessimistic-io/slitherin/blob/master/docs/multiple_storage_read.md
```

StakingRewards.sol

```
INFO:Detectors:
StakingRewards.unstake(LODEInternal(address,uint256) (contracts/StakingRewards.sol#182-237) ignores return value by LODE.transfer(staker,amount) (contracts/StakingRewards.sol#234)
StakingRewards.convertEsLODEToLODE(address) (contracts/StakingRewards.sol#243-336) ignores return value by esLODE.transfer(address(0),amountToTransfer) (contracts/StakingRewards.sol#334)
StakingRewards._emergencyWithdraw() (contracts/StakingRewards.sol#758-763) ignores return value by LODE.transfer(owner(),LODEDelta) (contracts/StakingRewards.sol#761)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
INFO:Detectors:
StakingRewards.convertEsLODEToLODE(address) (contracts/StakingRewards.sol#243-336) performs a multiplication on the result of a division:
        - conversionRatioMantissa = (timeDiff * BASE) / totalDays (contracts/StakingRewards.sol#283)
        - conversionAmount = ((userStakes[i].amount * conversionRatioMantissa) / BASE) - alreadyConverted (contracts/StakingRewards.sol#284)
StakingRewards.relock(uint256) (contracts/StakingRewards.sol#355-566) performs a multiplication on the result of a division:
        - relockMultiplier = (relockMultiplier * (1e18 + relockStLODE6M)) / 1e18 (contracts/StakingRewards.sol#387)
        - relockMultiplier = (relockMultiplier * (1e18 + relockStLODE3M)) / 1e18 (contracts/StakingRewards.sol#529)
StakingRewards.relock(uint256) (contracts/StakingRewards.sol#355-566) performs a multiplication on the result of a division:
        - relockMultiplier = (relockMultiplier * (1e18 + relockStLODE6M)) / 1e18 (contracts/StakingRewards.sol#387)
        - relockMultiplier = (relockMultiplier * (1e18 + relockStLODE3M)) / 1e18 (contracts/StakingRewards.sol#428)
StakingRewards.relock(uint256) (contracts/StakingRewards.sol#355-566) performs a multiplication on the result of a division:
        - relockStLODEAmount = ((info.lodeAmount * relockMultiplier) / 1e18) - info.lodeAmount (contracts/StakingRewards.sol#388)
        - relockMultiplier = (relockMultiplier * (1e18 + relockStLODE3M)) / 1e18 (contracts/StakingRewards.sol#428)
StakingRewards.relock(uint256) (contracts/StakingRewards.sol#355-566) performs a multiplication on the result of a division:
        - relockMultiplier = (relockMultiplier * (1e18 + relockStLODE3M)) / 1e18 (contracts/StakingRewards.sol#428)
        - relockStLODEAmount = ((info.lodeAmount * relockMultiplier) / 1e18) - info.lodeAmount (contracts/StakingRewards.sol#429)
StakingRewards.relock(uint256) (contracts/StakingRewards.sol#355-566) performs a multiplication on the result of a division:
        - relockMultiplier = (relockMultiplier * (1e18 + relockStLODE3M)) / 1e18 (contracts/StakingRewards.sol#428)
        - relockStLODEAmount = ((info.lodeAmount * relockMultiplier) / 1e18) - info.lodeAmount (contracts/StakingRewards.sol#530)
StakingRewards.relock(uint256) (contracts/StakingRewards.sol#355-566) performs a multiplication on the result of a division:
        - relockMultiplier = (relockMultiplier * (1e18 + relockStLODE3M)) / 1e18 (contracts/StakingRewards.sol#428)
        - relockMultiplier = (relockMultiplier * (1e18 + relockStLODE6M)) / 1e18 (contracts/StakingRewards.sol#492)
StakingRewards.relock(uint256) (contracts/StakingRewards.sol#355-566) performs a multiplication on the result of a division:
        - relockMultiplier = (relockMultiplier * (1e18 + relockStLODE6M)) / 1e18 (contracts/StakingRewards.sol#492)
        - relockStLODEAmount = ((info.lodeAmount * relockMultiplier) / 1e18) - info.lodeAmount (contracts/StakingRewards.sol#493)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
Reentrancy in StakingRewards._harvest(address) (contracts/StakingRewards.sol#604-616):
        External calls:
        - convertEsLODEToLODE(msg.sender) (contracts/StakingRewards.sol#606)
                - esLODE.transfer(address(0),amountToTransfer) (contracts/StakingRewards.sol#334)
        State variables written after the call(s):
        - user.wethRewardsDebt = int128(uint128(_calculateRewardDebt(accWethPerShare,user.amount))) (contracts/StakingRewards.sol#611)
        StakingConstants.userInfo (contracts/Utils/StakingConstants.sol#65) can be used in cross function reentrancies:
        - StakingRewards.convertEsLODEToLODE(address) (contracts/StakingRewards.sol#243-336)
        - StakingRewards.pendingRewards(address) (contracts/StakingRewards.sol#632-642)
        - StakingConstants.userInfo (contracts/Utils/StakingConstants.sol#65)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
StakingRewards.convertEsLODEToLODE(address).amountToTransfer (contracts/StakingRewards.sol#253) is a local variable never initialized
StakingRewards.convertEsLODEToLODE(address).stLODEAdjustment (contracts/StakingRewards.sol#254) is a local variable never initialized
StakingRewards.convertEsLODEToLODE(address).conversionAmount (contracts/StakingRewards.sol#255) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
StakingRewards.initialize(address,address,address,address)._routerContract (contracts/StakingRewards.sol#34) lacks a zero-check on :
                - routerContract = _routerContract (contracts/StakingRewards.sol#44)
StakingRewards._updateRouterContract(address)._routerContract (contracts/StakingRewards.sol#739) lacks a zero-check on :
                - routerContract = _routerContract (contracts/StakingRewards.sol#740)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
INFO:Detectors:
Reentrancy in StakingRewards.relock(uint256) (contracts/StakingRewards.sol#355-566):
        External calls:
        - convertEsLODEToLODE(msg.sender) (contracts/StakingRewards.sol#359)
                - esLODE.transfer(address(0),amountToTransfer) (contracts/StakingRewards.sol#334)
        State variables written after the call(s):
        - totalRelockStLODE -= currentRelockStLODE (contracts/StakingRewards.sol#365)
        - totalRelockStLODE += relockStLODEAmount (contracts/StakingRewards.sol#400)
        - totalRelockStLODE += relockStLODEAmount (contracts/StakingRewards.sol#441)
        - totalRelockStLODE += relockStLODEAmount (contracts/StakingRewards.sol#504)
        - totalRelockStLODE += relockStLODEAmount (contracts/StakingRewards.sol#541)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2
INFO:Detectors:
Reentrancy in StakingRewards._emergencyWithdraw() (contracts/StakingRewards.sol#758-763):
        External calls:
        - LODE.transfer(owner(),LODEDelta) (contracts/StakingRewards.sol#761)
        Event emitted after the call(s):
        - EmergencyWithdrawal(LODEDelta) (contracts/StakingRewards.sol#762)
Reentrancy in StakingRewards._harvest(address) (contracts/StakingRewards.sol#604-616):
        External calls:
        - convertEsLODEToLODE(msg.sender) (contracts/StakingRewards.sol#606)
                - esLODE.transfer(address(0),amountToTransfer) (contracts/StakingRewards.sol#334)
        - WETH.safeTransfer(_user,wethPending) (contracts/StakingRewards.sol#613)
        Event emitted after the call(s):
        - RewardsClaimed(_user,wethPending) (contracts/StakingRewards.sol#615)
Reentrancy in StakingRewards.withdrawEsLODE() (contracts/StakingRewards.sol#342-349):
        External calls:
        - esLODE.safeTransfer(msg.sender,totalEsLODE) (contracts/StakingRewards.sol#347)
        Event emitted after the call(s):
        - UnstakedEsLODE(msg.sender,totalEsLODE) (contracts/StakingRewards.sol#348)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
```

```
INFO:Detectors:
StakingRewards.stakeLODE(uint256,uint256) (contracts/StakingRewards.sol#59-78) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(block.timestamp < unlockTime,StakingRewards: Staking period expired) (contracts/StakingRewards.sol#74)
StakingRewards.unstakeLODE(uint256) (contracts/StakingRewards.sol#173-180) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(stakers[msg.sender].startTime + stakers[msg.sender].lockTime <= block.timestamp,StakingRewards: Tokens are still locked) (contracts/StakingRewards.sol#175-178)
StakingRewards.convertEsLODEToLODE(address) (contracts/StakingRewards.sol#243-336) uses timestamp for comparisons
        Dangerous comparisons:
        - timeDiff >= totalDays (contracts/StakingRewards.sol#263)
        - timeDiff < totalDays (contracts/StakingRewards.sol#282)
        - stLODEAdjustment != 0 (contracts/StakingRewards.sol#310)
StakingRewards.relock(uint256) (contracts/StakingRewards.sol#355-566) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(info.startTime + info.lockTime <= block.timestamp,StakingRewards: Lock time not expired) (contracts/StakingRewards.sol#369)
StakingRewards.updateShares() (contracts/StakingRewards.sol#572-589) uses timestamp for comparisons
        Dangerous comparisons:
        - block.timestamp <= lastRewardSecond (contracts/StakingRewards.sol#574)
StakingRewards.pendingRewards(address) (contracts/StakingRewards.sol#632-642) uses timestamp for comparisons
        Dangerous comparisons:
        - block.timestamp > lastRewardSecond && shares != 0 (contracts/StakingRewards.sol#635)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
Function StakingRewards.stakeLODEInternal(address,uint256,uint256) (contracts/StakingRewards.sol#80-122) has a dubious typecast: int128<=uint128
Function StakingRewards.stakeEsLODEInternal(uint256) (contracts/StakingRewards.sol#134-167) has a dubious typecast: int128<=uint128
Function StakingRewards.unstakeLODEInternal(address,uint256) (contracts/StakingRewards.sol#182-237) has a dubious typecast: int128<=uint128
Function StakingRewards.convertEsLODEToLODE(address) (contracts/StakingRewards.sol#243-336) has a dubious typecast: int128<=uint128
Function StakingRewards.relock(uint256) (contracts/StakingRewards.sol#355-566) has a dubious typecast: int128<=uint128
Function StakingRewards._harvest(address) (contracts/StakingRewards.sol#604-616) has a dubious typecast: int128<=uint128
Function StakingRewards._calculatePending(int128,uint256,uint96) (contracts/StakingRewards.sol#644-654) has a dubious typecast: uint128<=uint128
Reference: https://github.com/pessimistic-io/slitherin/blob/master/docs/dubious_typecast.md
INFO:Detectors:
Setter function StakingRewards.setStartTime(uint32) (contracts/StakingRewards.sol#662-664) does not emit an event
Setter function StakingRewards.setEmission(uint256) (contracts/StakingRewards.sol#666-668) does not emit an event
Reference: https://github.com/pessimistic-io/slitherin/blob/master/docs/event_setter.md
INFO:Detectors:
StakingRewards.withdrawEsLODE() (contracts/StakingRewards.sol#342-349) compares to a boolean constant:
        -require(bool,string)(withdrawEsLODEAllowed == true,esLODE Withdrawals Not Permitted) (contracts/StakingRewards.sol#343)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#boolean-equality
INFO:Detectors:
Pragma version^0.8.0 (contracts/StakingRewards.sol#3) allows old versions
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
StakingRewards (contracts/StakingRewards.sol#16-764) should inherit from StakingRewardsInterface (contracts/Interfaces/Interfaces.sol#130-132)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-inheritance
INFO:Detectors:
Function ICERC20._reduceReserves(uint256) (contracts/Interfaces/Interfaces.sol#57) is not in mixedCase
Function SushiRouterInterface.WETH() (contracts/Interfaces/Interfaces.sol#61) is not in mixedCase
Parameter StakingRewards.initialize(address,address,address,address)._LODE (contracts/StakingRewards.sol#34) is not in mixedCase
Parameter StakingRewards.initialize(address,address,address,address)._WETH (contracts/StakingRewards.sol#34) is not in mixedCase
Parameter StakingRewards.initialize(address,address,address,address)._esLODE (contracts/StakingRewards.sol#34) is not in mixedCase
Parameter StakingRewards.initialize(address,address,address,address)._routerContract (contracts/StakingRewards.sol#34) is not in mixedCase
Parameter StakingRewards.rewardPerShare(uint256)._rewardRatePerSecond (contracts/StakingRewards.sol#622) is not in mixedCase
Parameter StakingRewards.pendingRewards(address)._user (contracts/StakingRewards.sol#632) is not in mixedCase
Parameter StakingRewards.setStartTime(uint32)._startTime (contracts/StakingRewards.sol#662) is not in mixedCase
Parameter StakingRewards.setEmission(uint256)._wethPerSecond (contracts/StakingRewards.sol#666) is not in mixedCase
Parameter StakingRewards.updateWeeklyRewards(uint256)._weeklyRewards (contracts/StakingRewards.sol#684) is not in mixedCase
Function StakingRewards._pauseStaking() (contracts/StakingRewards.sol#721-724) is not in mixedCase
Function StakingRewards._unpauseStaking() (contracts/StakingRewards.sol#730-733) is not in mixedCase
Function StakingRewards._updateRouterContract(address) (contracts/StakingRewards.sol#739-742) is not in mixedCase
Parameter StakingRewards._updateRouterContract(address)._routerContract (contracts/StakingRewards.sol#739) is not in mixedCase
Function StakingRewards._allowEsLODEWithdraw(bool) (contracts/StakingRewards.sol#749-752) is not in mixedCase
Function StakingRewards._emergencyWithdraw() (contracts/StakingRewards.sol#758-763) is not in mixedCase
Event StakingConstants.esLODEUnlocked(bool,uint256) (contracts/Utils/StakingConstants.sol#82) is not in CapWords
Variable StakingConstants.LODE (contracts/Utils/StakingConstants.sol#32) is not in mixedCase
Variable StakingConstants.WETH (contracts/Utils/StakingConstants.sol#33) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
Variable StakingRewards.relock(uint256)._prev_scope_3 (contracts/StakingRewards.sol#467) is too similar to StakingRewards.relock(uint256)._prev_scope_9 (contracts/StakingRewards.sol#545)
Variable StakingRewards.relock(uint256)._prev_scope_3 (contracts/StakingRewards.sol#467) is too similar to StakingRewards.relock(uint256)._prev_scope_6 (contracts/StakingRewards.sol#508)
Variable StakingRewards.relock(uint256)._prev_scope_6 (contracts/StakingRewards.sol#508) is too similar to StakingRewards.relock(uint256)._prev_scope_9 (contracts/StakingRewards.sol#545)
Variable StakingRewards.unstakeLODEInternal(address,uint256).esLODEBalance (contracts/StakingRewards.sol#189) is too similar to StakingRewards.unstakeLODEInternal(address,uint256).stLODEBalance (contracts/StakingRewards.sol#187)
Variable StakingRewards.relock(uint256).newStLODEBalance_scope_0 (contracts/StakingRewards.sol#430) is too similar to StakingRewards.relock(uint256).newStLODEBalance_scope_4 (contracts/StakingRewards.sol#494)
Variable StakingRewards.relock(uint256).newStLODEBalance_scope_4 (contracts/StakingRewards.sol#494) is too similar to StakingRewards.relock(uint256).newStLODEBalance_scope_7 (contracts/StakingRewards.sol#531)
Variable StakingRewards.relock(uint256).newStLODEBalance_scope_0 (contracts/StakingRewards.sol#430) is too similar to StakingRewards.relock(uint256).newStLODEBalance_scope_7 (contracts/StakingRewards.sol#531)
Variable StakingConstants.relockStLODE3M (contracts/Utils/StakingConstants.sol#42) is too similar to StakingConstants.relockStLODE6M (contracts/Utils/StakingConstants.sol#43)
Variable StakingConstants.totalEsLODEStaked (contracts/Utils/StakingConstants.sol#40) is too similar to StakingRewards.accountVoteShare(address).totalStLODEStaked (contracts/StakingRewards.sol#702)
Variable StakingRewards.relock(uint256).user_scope_1 (contracts/StakingRewards.sol#447) is too similar to StakingRewards.relock(uint256).user_scope_2 (contracts/StakingRewards.sol#465)
Variable StakingRewards.relock(uint256).user_scope_1 (contracts/StakingRewards.sol#447) is too similar to StakingRewards.relock(uint256).user_scope_5 (contracts/StakingRewards.sol#506)
Variable StakingRewards.relock(uint256).user_scope_1 (contracts/StakingRewards.sol#447) is too similar to StakingRewards.relock(uint256).user_scope_8 (contracts/StakingRewards.sol#543)
Variable StakingRewards.relock(uint256).user_scope_2 (contracts/StakingRewards.sol#465) is too similar to StakingRewards.relock(uint256).user_scope_8 (contracts/StakingRewards.sol#543)
Variable StakingRewards.relock(uint256).user_scope_2 (contracts/StakingRewards.sol#465) is too similar to StakingRewards.relock(uint256).user_scope_5 (contracts/StakingRewards.sol#506)
Variable StakingRewards.relock(uint256).user_scope_5 (contracts/StakingRewards.sol#506) is too similar to StakingRewards.relock(uint256).user_scope_8 (contracts/StakingRewards.sol#543)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#variable-names-too-similar
INFO:Detectors:
StakingRewards.initialize(address,address,address,address) (contracts/StakingRewards.sol#34-52) uses literals with too many digits:
        - stLODE3M = 14000000000000000000 (contracts/StakingRewards.sol#46)
StakingRewards.initialize(address,address,address,address) (contracts/StakingRewards.sol#34-52) uses literals with too many digits:
        - stLODE6M = 20000000000000000000 (contracts/StakingRewards.sol#47)
StakingRewards.initialize(address,address,address,address) (contracts/StakingRewards.sol#34-52) uses literals with too many digits:
        - relockStLODE3M = 5000000000000000000 (contracts/StakingRewards.sol#48)
StakingRewards.initialize(address,address,address,address) (contracts/StakingRewards.sol#34-52) uses literals with too many digits:
        - relockStLODE6M = 10000000000000000000 (contracts/StakingRewards.sol#49)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits
INFO:Detectors:
Function StakingRewards.initialize(address,address,address,address) (contracts/StakingRewards.sol#34-52) contains magic numbers: 14000000000000000000, 20000000000000000000, 5000000000000000000, 10000000000000000000
Function StakingRewards.stakeLODE(uint256,uint256) (contracts/StakingRewards.sol#59-78) contains magic numbers: 10, 10
Function StakingRewards.stakeLODEInternal(address,uint256,uint256) (contracts/StakingRewards.sol#80-122) contains magic numbers: 7776000, 15552000
Function StakingRewards.unstakeLODEInternal(address,uint256) (contracts/StakingRewards.sol#182-237) contains magic number: 10
Function StakingRewards.convertEsLODEToLODE(address) (contracts/StakingRewards.sol#243-336) contains magic numbers: 31536000, 7776000, 15552000, 7776000, 15552000
Function StakingRewards.relock(uint256) (contracts/StakingRewards.sol#355-566) contains magic numbers: 7776000, 15552000, 15552000
Function StakingRewards.calculateWethPerSecond(uint256) (contracts/StakingRewards.sol#674-677) contains magic number: 604800
Reference: https://github.com/pessimistic-io/slitherin/blob/master/docs/magic_number.md
INFO:Detectors:
StakingConstants.lockCanceled (contracts/Utils/StakingConstants.sol#52) should be constant
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-constant
INFO:Detectors:
In a function StakingRewards.stakeLODEInternal(address,uint256,uint256) (contracts/StakingRewards.sol#80-122) variable StakingConstants.stakers (contracts/Utils/StakingConstants.sol#30) is read multiple times
In a function StakingRewards.stakeEsLODEInternal(uint256) (contracts/StakingRewards.sol#134-167) variable StakingConstants.stakers (contracts/Utils/StakingConstants.sol#30) is read multiple times
In a function StakingRewards.unstakeLODEInternal(address,uint256) (contracts/StakingRewards.sol#182-237) variable StakingConstants.stakers (contracts/Utils/StakingConstants.sol#30) is read multiple times
In a function StakingRewards.convertEsLODEToLODE(address) (contracts/StakingRewards.sol#243-336) variable StakingConstants.BASE (contracts/Utils/StakingConstants.sol#47) is read multiple times
In a function StakingRewards.convertEsLODEToLODE(address) (contracts/StakingRewards.sol#243-336) variable StakingConstants.esLODEStakes (contracts/Utils/StakingConstants.sol#28) is read multiple times
In a function StakingRewards.convertEsLODEToLODE(address) (contracts/StakingRewards.sol#243-336) variable StakingConstants.relockStLODE3M (contracts/Utils/StakingConstants.sol#42) is read multiple times
In a function StakingRewards.convertEsLODEToLODE(address) (contracts/StakingRewards.sol#243-336) variable StakingConstants.relockStLODE6M (contracts/Utils/StakingConstants.sol#43) is read multiple times
In a function StakingRewards.convertEsLODEToLODE(address) (contracts/StakingRewards.sol#243-336) variable StakingConstants.stakers (contracts/Utils/StakingConstants.sol#30) is read multiple times
In a function StakingRewards.relock(uint256) (contracts/StakingRewards.sol#355-566) variable StakingConstants.accWethPerShare (contracts/Utils/StakingConstants.sol#61) is read multiple times
In a function StakingRewards.relock(uint256) (contracts/StakingRewards.sol#355-566) variable StakingConstants.shares (contracts/Utils/StakingConstants.sol#62) is read multiple times
In a function StakingRewards.relock(uint256) (contracts/StakingRewards.sol#355-566) variable StakingConstants.stakers (contracts/Utils/StakingConstants.sol#30) is read multiple times
In a function StakingRewards.relock(uint256) (contracts/StakingRewards.sol#355-566) variable StakingConstants.totalRelockStLODE (contracts/Utils/StakingConstants.sol#39) is read multiple times
In a function StakingRewards.relock(uint256) (contracts/StakingRewards.sol#355-566) variable StakingConstants.userInfo (contracts/Utils/StakingConstants.sol#65) is read multiple times
Reference: https://github.com/pessimistic-io/slitherin/blob/master/docs/multiple_storage_read.md
```

TokenClaim.sol

```
INFO:Detectors:
Reentrancy in TokenClaim.claimTokens(uint256,bytes32[]) (contracts/TokenClaim.sol#25-44):
        External calls:
        - require(bool,string)(token.transfer(msg.sender,scaledQuantity),Token transfer failed) (contracts/TokenClaim.sol#39)
        State variables written after the call(s):
        - isClaimed[msg.sender] = true (contracts/TokenClaim.sol#43)
        TokenClaim.isClaimed (contracts/TokenClaim.sol#15) can be used in cross function reentrancies:
        - TokenClaim.isClaimed (contracts/TokenClaim.sol#15)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
Reentrancy in TokenClaim.claimTokens(uint256,bytes32[]) (contracts/TokenClaim.sol#25-44):
        External calls:
        - require(bool,string)(token.transfer(msg.sender,scaledQuantity),Token transfer failed) (contracts/TokenClaim.sol#39)
        Event emitted after the call(s):
        - TokensClaimed(msg.sender,scaledQuantity) (contracts/TokenClaim.sol#40)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
Pragma version^0.8.0 (contracts/TokenClaim.sol#2) allows old versions
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Parameter TokenClaim.removeTokens(uint256)._amount (contracts/TokenClaim.sol#46) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
TokenClaim.admin (contracts/TokenClaim.sol#11) should be immutable
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-immutable
```

VotingPower.sol

```
INFO:Detectors:
Function VotingPower.resetVotesIfNeeded() (contracts/VotingPower.sol#112-122) is an unprotected initializer.
Reference: https://github.com/pessimistic-io/slitherin/blob/master/docs/unprotected_initialize.md
INFO:Detectors:
VotingPower.getResults() (contracts/VotingPower.sol#174-202) performs a multiplication on the result of a division:
        - votePercentage = voteCount * 1e18 / totalVoteCount (contracts/VotingPower.sol#194)
        - results[i_scope_0 * 2 + i_scope_1] = votePercentage * lodeSpeed / 1e18 (contracts/VotingPower.sol#195)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
VotingPower.vote(string[],VotingPower.OperationType[],uint256[]) (contracts/VotingPower.sol#128-170) uses a dangerous strict equality:
        - require(bool,string)(lastVotedWeek[msg.sender] < currentWeek || (currentWeek == 0 && lastVotedWeek[msg.sender] == 0 && ! previouslyVoted[msg.sender]),You have already voted this week) (contracts/VotingPower.sol#139-144)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
VotingPower.vote(string[],VotingPower.OperationType[],uint256[]) (contracts/VotingPower.sol#128-170) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(lastVotedWeek[msg.sender] < currentWeek || (currentWeek == 0 && lastVotedWeek[msg.sender] == 0 && ! previouslyVoted[msg.sender]),You have already voted this week) (contracts/VotingPower.sol#139-144)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
VotingPower.removeToken(string) (contracts/VotingPower.sol#79-93) has costly operations inside a loop:
        - enabledTokensList.pop() (contracts/VotingPower.sol#87)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#costly-operations-inside-a-loop
INFO:Detectors:
Pragma version^0.8.0 (contracts/Interfaces/IStakingRewards.sol#2) allows old versions
Pragma version^0.8.0 (contracts/VotingPower.sol#2) allows old versions
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Parameter VotingPower.removeToken(string)._token (contracts/VotingPower.sol#79) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
VotingPower.slitherConstructorVariables() (contracts/VotingPower.sol#13-203) uses literals with too many digits:
        - lodeSpeed = 602739726000000000 (contracts/VotingPower.sol#19)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits
INFO:Detectors:
Function VotingPower.slitherConstructorVariables() (contracts/VotingPower.sol#13-203) contains magic numbers: 604800, 602739726000000000
Reference: https://github.com/pessimistic-io/slitherin/blob/master/docs/magic_number.md
INFO:Detectors:
VotingPower.votingPeriod (contracts/VotingPower.sol#18) should be constant
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-constant
INFO:Detectors:
VotingPower.stakingRewards (contracts/VotingPower.sol#16) should be immutable
VotingPower.votingStartTimestamp (contracts/VotingPower.sol#17) should be immutable
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-immutable
INFO:Detectors:
In a function VotingPower.removeToken(string) (contracts/VotingPower.sol#79-93) variable VotingPower.enabledTokensList (contracts/VotingPower.sol#20) is read multiple times
In a function VotingPower.getResults() (contracts/VotingPower.sol#174-202) variable VotingPower.enabledTokensList (contracts/VotingPower.sol#20) is read multiple times
In a function VotingPower.resetVotesIfNeeded() (contracts/VotingPower.sol#112-122) variable VotingPower.enabledTokensList (contracts/VotingPower.sol#20) is read multiple times
Reference: https://github.com/pessimistic-io/slitherin/blob/master/docs/multiple_storage_read.md
```

- All the reentrancies flagged by Slither were checked individually and are false positives.
- All the unprotected initialize issues flagged by Slither were checked individually and are false positives.
- Double entry token issue flagged by Slither in the `RewardsRouter.withdraw()` function can also be considered a false positive.
- The unchecked transfers flagged by Slither can also be ignored, as the tokens used here are known: LODE & esLODE.
- No major issues found by Slither.

7.2 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and to identify low-hanging fruits on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the smart contracts and sent the compiled results to the analyzers to locate any vulnerabilities.

MythX results:

esLODE.sol

Line	SWC Title	Severity	Short Description
2	(SWC-103) Floating Pragma	Low	A floating pragma is set.

LodestarHandler.sol

Line	SWC Title	Severity	Short Description
13	(SWC-108) State Variable Default Visibility	Low	State variable visibility is not set.

RewardRouter.sol

Line	SWC Title	Severity	Short Description
2	(SWC-103) Floating Pragma	Low	A floating pragma is set.

StakingRewards.sol

Line	SWC Title	Severity	Short Description
3	(SWC-103) Floating Pragma	Low	A floating pragma is set.
81	(SWC-123) Requirement Violation	Low	Requirement violation.
129	(SWC-123) Requirement Violation	Low	Requirement violation.
574	(SWC-116) Timestamp Dependence	Low	A control flow decision is made based on The block.timestamp environment variable.
635	(SWC-116) Timestamp Dependence	Low	A control flow decision is made based on The block.timestamp environment variable.

TokenClaim.sol

Line	SWC Title	Severity	Short Description
2	(SWC-103) Floating Pragma	Low	A floating pragma is set.

VotingPower.sol

Line	SWC Title	Severity	Short Description
2	(SWC-103) Floating Pragma	Low	A floating pragma is set.

- Floating pragma was correctly flagged by MythX.
- MythX flagged some requirement violations, which were all considered to be false positives.
- No major issues were found by MythX.

THANK YOU FOR CHOOSING

// HALBORN