Competitive Security Assessment

Lagrange_State_Committee

Mar 7th, 2024

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

• Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.

• Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.

• Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.

• Verify the code base is compliant with the most up-to-date industry standards and security best practices.

• Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

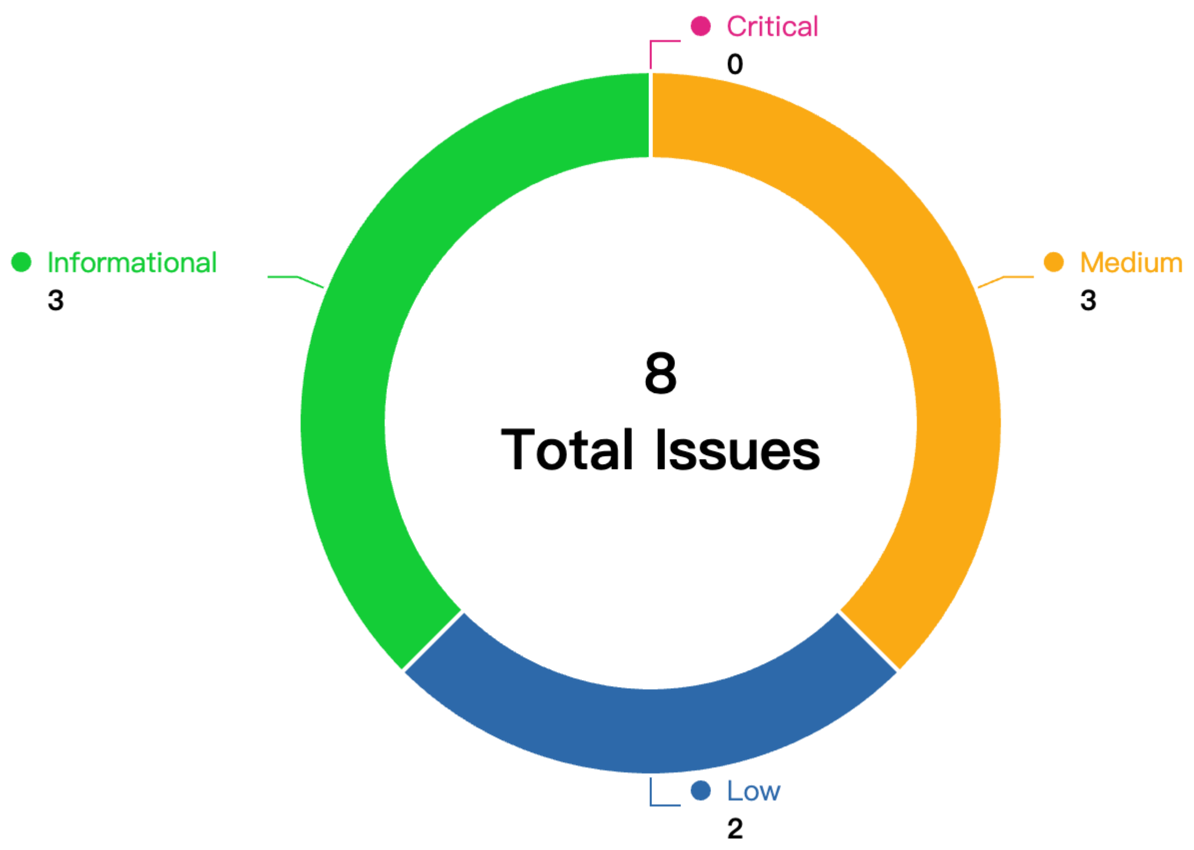
Overview

Project Name	Lagrange_State_Committee
Language	Solidity
Codebase	• https://github.com/Lagrange-Labs/lagrange-contracts-audit • audit version - c0165f561fea39f22104828355f3210c3435e095 • final version - 890fd4617f2915df597af5d33e252abd23302d6b
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

Audit Scope

File	SHA256 Hash
./src/protocol/LagrangeCommittee.sol	8ae3f7715b34644344f194c5b2df84b0558f5b96e64 66a06c1bae8a52c3e4ed9
./src/protocol/LagrangeService.sol	ebed6c51d333af6a1273e25036985d2ce613a838ac0 3a4f935b2eeafb17ee43f
./src/protocol/VoteWeigher.sol	3b6d921f535ede65ee4918f1a5e5e2613b0dac2a2e06 144a8c43acfd45d4c801
./src/library/StakeManager.sol	c9338080780935be2186ba652ed63f7fc32335cb1dd cf4ce5fe6c190c7d038a0
./src/interfaces/ILagrangeCommittee.sol	2efa6ddf37e64701e72df79f22b1736abc5d3b2f983a8 1a39a993a286ee5d4ea
./src/library/EigenAdapter.sol	f51f11ed92272e38dcce0767c81c88085e575d06e6d4 91619c0f7f179f55c68a
./src/interfaces/IVoteWeigher.sol	1ab0e139d36aaa08692a253ba4c4f610be7e9e39dcb 4f7cefdf0ef1092721ca8
./src/interfaces/ILagrangeService.sol	d67d17b195521550e36f289f408dc6af52366e41a351 65fb6e60e5b8e3228a92
./src/interfaces/IStakeManager.sol	da089cbb7cc4564c06351c975e8f9301eefd9aa05a2c d5f66d11d13d6282e02f

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
LSC-1	`subscribeChain()` , `isUpdatable()` & `isLocked()` will not work properly for Arbitrum or Optimism due to `block.number`	Logical	Medium	Declined	0xWeb3boy
LSC-2	Validation can be bypassed in the `addOperator` function	Logical	Medium	Fixed	crjr0629, Xi_Zi
LSC-3	Merkle tree signature could be replayed.	Signature Forgery or Replay	Medium	Declined	n16h7m4r3
LSC-4	operator can subscribe to the same chain multiple times	Logical	Low	Fixed	crjr0629
LSC-5	Inconsistent way of calculating next epoch	Logical	Low	Fixed	crjr0629

LSC-6	`VoteWeigher` should emit events for updates on quorums, operators can react by participate in new staking opportunities.	Logical	Informational	Fixed	ravikiran_web3
LSC-7	StakeManager contract does not restrict the ERC20 tokens that can be staked and unstaked, leaving unintended tokens also locked.	Logical	Informational	Fixed	ravikiran_web3
LSC-8	Gas Optimization in `LagrangeCommittee:_initCommittee()` function	Gas Optimization	Informational	Fixed	newway55

LSC-1: `subscribeChain()` , `isUpdatable()` & `isLocked()` will not work properly for Arbitrum or Optimism due to `block.number`

Category	Severity	Client Response	Contributor
Logical	Medium	Declined	0xWeb3boy

Code Reference

- code/src/protocol/LagrangeCommittee.sol#L81
- code/src/protocol/LagrangeCommittee.sol#L120
- code/src/protocol/LagrangeCommittee.sol#L234
- code/src/protocol/LagrangeCommittee.sol#L242

```
81: committeeParams[chainID] = CommitteeDef(block.number, _duration, _freezeDuration, _quorumNumber);
```

```
120: if (param.blockNumber > 0 && param.blockNumber >= block.number) {
```

```
234: return block.number > epochEnd - freezeDuration;
```

```
242: uint256 epochNumber = getEpochNumber(chainID, block.number);
```

Description

0xWeb3boy: According to the documentation provided `` `At launch, the Lagrange Protocol will be compatible with all public Ethereum Virtual Machine (EVM) compatible L1s, L2s and rollups. Soon after, the Lagrange Protocol will also support non-EVM compatible chains including Solana, Sui, Aptos and popular Cosmos SDK based chains` ``, this means it will also be compatible with optimism and arbitrum chains.
According to Arbitrum Docs, `` `block.number` `` returns the most recently synced L1 block number. Once per minute, the block number in the Sequencer is synced to the actual L1 block number. Using block.number as a clock can lead to inaccurate timing.
It also presents an issue for Optimism because each transaction is it's own block.

```
function _initCommittee(uint32 chainID, uint256 _duration, uint256 _freezeDuration, uint8 _quorumNumber) internal {
        require(committeeParams[chainID].startBlock == 0, "Committee has already been initialized.");

        committeeParams[chainID] = CommitteeDef(block.number, _duration, _freezeDuration, _quorumNumber);
        committees[chainID][0] = CommitteeData(0, 0, 0);

        emit InitCommittee(chainID, _duration, _freezeDuration, _quorumNumber);
    }
```

While subscribing the chain the code is using `` `block.number` `` which could lead to inaccurate timings on optimism and arbitrum chains

```
    function subscribeChain(address operator, uint32 chainID) external onlyService {
        (bool locked,) = isLocked(chainID);
        require(!locked, "The dedicated chain is locked.");

        OperatorStatus storage opStatus = operators[operator];

        for (uint256 i = 0; i < opStatus.unsubscribedParams.length; i++) {
            UnsubscribedParam memory param = opStatus.unsubscribedParams[i];
            if (param.chainID == chainID) {
 @>             if (param.blockNumber > 0 && param.blockNumber >= block.number) {
                  revert("The dedciated chain is while unsubscribing.");
                }
            }
        }
```

Recommendation

0xWeb3boy: Use `` `block.timestamp` `` rather than `` `block.number` ``

Client Response

0xWeb3boy: Declined.the term epoch is used in the ethereum, there is no need to refer L2 block number.
Secure3: We honor this issue.

LSC-2:Validation can be bypassed in the `addOperator` function

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	crjr0629, Xi_Zi

Code Reference

- code/src/protocol/LagrangeCommittee.sol#L88-L92
- code/src/protocol/LagrangeCommittee.sol#L88-L92

```
88: function addOperator(address operator, uint256[2] memory blsPubKey) public onlyService {
89:         OperatorStatus storage opStatus = operators[operator];
90:         require(opStatus.blsPubKey[0] == 0, "Operator is already registered.");
91:         opStatus.blsPubKey = blsPubKey;
92:     }
```

```
88: function addOperator(address operator, uint256[2] memory blsPubKey) public onlyService {
89:         OperatorStatus storage opStatus = operators[operator];
90:         require(opStatus.blsPubKey[0] == 0, "Operator is already registered.");
91:         opStatus.blsPubKey = blsPubKey;
92:     }
```

Description

crjr0629: The contract uses **`uint256[2]`** to store bls public keys, BLS public keys are 48 bytes long. being 32 bytes stored in the first position and 16 bytes in the second position or the other way around. Operators are not allowed to perform updates on pubkey through this check :

```
    require(opStatus.blsPubKey[0] == 0, "Operator is already registered.");
```

the check should be done on the second position of the array as well
Xi_Zi: Using blspubkey[0] == 0 in addOperator does not prove that operator is registered, because no nonzero judgment is made on the blsPubKey parameter. If blsPubKey[0] =0 blsPubKey[1] = key is set, the require judgment can be bypassed and the operator can register twice

```
function addOperator(address operator, uint256[2] memory blsPubKey) public onlyService {
        OperatorStatus storage opStatus = operators[operator];
        require(opStatus.blsPubKey[0] == 0, "Operator is already registered.");
        opStatus.blsPubKey = blsPubKey;
    }
```

Recommendation

crjr0629: - use bytes instead of uint256[2] similar as the beacon deposit contract

- check for the right length of the public key.
- use a mapping to check if this public key was already used.

Xi_Zi: To prevent this, you can modify the require statement to ensure that both elements are zero. For example:

```
require(opStatus.blsPubKey[0] == 0 && opStatus.blsPubKey[1] == 0, "Operator is already registere
d.");
```

With this modification, blsPubKey will only pass validation if both elements are zero. This makes it more effective to prevent the same operator from being registered twice.

Client Response

crjr0629: Fixed.
Xi_Zi: Fixed.

LSC-3:Merkle tree signature could be replayed.

Category	Severity	Client Response	Contributor
Signature Forgery or Replay	Medium	Declined	n16h7m4r3

Code Reference

- code/src/protocol/LagrangeCommittee.sol#L196
- code/src/protocol/LagrangeCommittee.sol#L334-L342

```
196: return keccak256(abi.encodePacked(INNER_NODE_PREFIX, left, right));
```

```
334: return keccak256(
335:             abi.encodePacked(
336:                 LEAF_NODE_PREFIX,
337:                 opStatus.blsPubKey[0],
338:                 opStatus.blsPubKey[1],
339:                 opAddr,
340:                 opStatus.subscribedChains[chainID]
341:             )
342:         );
```

Description

n16h7m4r3: The Merkle tree root constructed using the function _leafHash() and _innerHash() does not include any unique parameters such as Nonce. chain id, etc. An previously valid Merkle proof can be replayed by an attacker to the verifier and across different chains.

Recommendation

n16h7m4r3: Consider including an nonce and chain id in in the keccak256 hash computed and validate the same when the signature is consumed by the verifier.

Client Response

n16h7m4r3: Declined. Only deploy in Mantle.
Secure3: We honored this issue.

LSC-4:operator can subscribe to the same chain multiple times

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	crjr0629

Code Reference

- code/src/protocol/LagrangeCommittee.sol#L127

```
127: opStatus.subscribedChains[chainID] =
```

Description

crjr0629: By logic this contract shouldnt allow an operator to subscribe to the same chain multiple times. However, the contract allows this to happen. if an operator has a voting weight of 0, it can subscribe many times to the same chain. this increases the amount of leaves in the contract.

Recommendation

crjr0629: - consider registering an operator only when the weight is greater than 0.

```
    uint96 operatorVoteWeight =
        voteWeigher.weightOfOperator(committeeParams[chainID].quorumNumber, operator);
    require(operatorVoteWeight > 0, "Operator has no voting weight.");
    opStatus.subscribedChains[chainID] = operatorVoteWeight;
```

Client Response

crjr0629: Fixed.

LSC-5:Inconsistent way of calculating next epoch

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	crjr0629

Code Reference

- code/src/protocol/LagrangeCommittee.sol#L180-L181
- code/src/protocol/LagrangeCommittee.sol#L257-L258

```
180: uint256 nextEpoch = getEpochNumber(chainID, blockNumber + 1);
181:         currentCommittee = committees[chainID][epochNumber];
```

```
257: uint256 nextEpoch = epochNumber + COMMITTEE_NEXT_1;
```

Description

crjr0629: in contract `` `LagrangeCommittee` `` , the function `` `getCommittee()` `` calculates the next epoch by adding 1 to the current block and calculating its respective epoch number. However function `` `_updateCommittee` `` calculates the next epoch by adding 1 to the current epoch. This inconsistency can lead to unexpected behavior.

```
    uint256 nextEpoch = epochNumber + COMMITTEE_NEXT_1; // _updateCommittee
    uint256 nextEpoch = getEpochNumber(chainID, blockNumber + 1); // getCommittee
```

Recommendation

crjr0629: - use a consistent way or different terminology to calculate the next epoch.

Client Response

crjr0629: Fixed.

LSC-6: `VoteWeigher` should emit events for updates on quorums, operators can react by participate in new staking opportunities.

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	ravikiran_web3

Code Reference

- code/src/protocol/VoteWeigher.sol#L32
- code/src/protocol/VoteWeigher.sol#L39
- code/src/protocol/VoteWeigher.sol#L43

```
32: function addQuorumMultiplier(uint8 quorumNumber, TokenMultiplier[] memory multipliers) external
onlyOwner {
```

```
39: function removeQuorumMultiplier(uint8 quorumNumber) external onlyOwner {
```

```
43: function updateQuorumMultiplier(uint8 quorumNumber, uint256 index, TokenMultiplier memory multip
lier) external onlyOwner {
```

Description

ravikiran_web3: Every time there is a change in the tokenMultiplier, the operators may be interested to listen and react.
Currently, there are no events being fired for adding, removing or updating the token multipliers.
Operators who are participating in staking their tokens are qualified to learn about the changes so that they can react, especially when there are new tokens supported,
the operators can stake more tokens into the staking contract and increase their weight.

Recommendation

ravikiran_web3: Add events for all the below changes.
Event emits on
a) adding new Token Multiplier for a quorum
b) update the Multiplier rate for a token Multiplier for a quorum
c) remove a token Multiplier

Client Response

ravikiran_web3: Fixed.

LSC-7:StakeManager contract does not restrict the ERC20 tokens that can be staked and unstaked, leaving unintended tokens also locked.

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	ravikiran_web3

Code Reference

- code/src/library/StakeManager.sol#L37-L42
- code/src/library/StakeManager.sol#L44-L51

```
37: function deposit(IERC20 token, uint256 amount) external {
38:         token.safeTransferFrom(msg.sender, address(this), amount);
39:         operatorShares[msg.sender][address(token)] += amount;
40:
41:         emit Deposit(msg.sender, address(token), amount);
42:     }
```

```
44: function withdraw(IERC20 token, uint256 amount) external {
45:         require(stakeLockedBlock[msg.sender] < block.number, "Stake is locked");
46:         require(operatorShares[msg.sender][address(token)] >= amount, "Insufficient balance");
47:         operatorShares[msg.sender][address(token)] -= amount;
48:         token.safeTransfer(msg.sender, amount);
49:
50:         emit Withdraw(msg.sender, address(token), amount);
51:     }
```

Description

ravikiran_web3: StakingManager contract does not validate the tokens that will actually be staked in the protocol. So, a caller can deposit any ERC20 token they hold
into the staking manager by calling deposit.

```
  function deposit(IERC20 token, uint256 amount) external {
        token.safeTransferFrom(msg.sender, address(this), amount);
        operatorShares[msg.sender][address(token)] += amount;

        emit Deposit(msg.sender, address(token), amount);
    }
```

For the first deposit, If the operator or anybody calls **LagrangeCommittee::updateOperatorAmount()** function, the deposited tokens are locked for a very large window.
Since the stakeLockedBlock[msg.sender] is maintained at operator level rather than token level, it applies to all tokens deposited.

```
function withdraw(IERC20 token, uint256 amount) external {
        require(stakeLockedBlock[msg.sender] < block.number, "Stake is locked");
        require(operatorShares[msg.sender][address(token)] >= amount, "Insufficient balance");
        operatorShares[msg.sender][address(token)] -= amount;
        token.safeTransfer(msg.sender, amount);

        emit Withdraw(msg.sender, address(token), amount);
    }
```

But the problem is critical from subsequent deposits, where even unintended tokens will be locked for a very large time window.
There is no easy way to withdraw those tokens unless the operator/user decides to un-register.
Example:
Protocol accepts: DAI, USDC
Protocol does not accepts: WBTC, WETH
Operator A deposits DAI into the staking contract and the stake lock is applied.
After that if the operator deposits WETH, then he will not be able to withdraw until the timelock or initiate unregister process.

Recommendation

ravikiran_web3: The solution is to accept deposits only for enabled tokens.
The list of enabledTokens is in **VoteWeigher** contract in the form of **TokenMultiplier.address**.
Since the stakingManager contract has access to LagrangeService contract, which can reach the **VoteWeigher** contract instance via the committe contract.
So, the below State variables/functions should be added.
VoteWeigher contract:

1. Add a state variable as below.
   EnumerableSet.AddressSet listedToken;
2. Add owner functions to maintain the listedToken.
3. A function to take token address as parameter to check if it is listed or not.
   VoteWeigher::IsListedToken(address token) returns (bool){
   return listedToken.contains(token);
   }

LagrangeCommittee contract
a) IsListedToken() that delegates the call to VoteWeigher contract.
LagrangeService contract
a) IsListedToken() that delegates the call to LagrangeCommittee contract.
Benefits:
a) the check can be done in Staking contract while making deposit.
b) check can be done also during maintenance functions of Quorum Multipliers in VoteWeigher

Client Response

ravikiran_web3: Fixed.yes, will whitelist token

LSC-8:Gas Optimization in `LagrangeCommittee:_initCommittee()` function

Category	Severity	Client Response	Contributor
Gas Optimization	Informational	Fixed	newway55

Code Reference

- code/src/protocol/LagrangeCommittee.sol#L82

```
82: committees[chainID][0] = CommitteeData(0, 0, 0);
```

Description

newway55: the _initCommittee function, a new committee is initialized by setting the committeeParams for a given chainID and creating a new CommitteeData entry with all fields set to 0. This initialization, while necessary, incurs gas costs for storage operations, particularly for the uint256 fields.
This struct is updated to (0,0,0) and each one is a storage slot which costs 20k x 3 = 60k gas.

Recommendation

newway55: I suggest reducing Data Type Sizes for CommitteeData into :

```
struct CommitteeData {
      bytes32 root;
      uint16 leafCount;
      uint16 totalVotingPower;
  }
```

Since totalVotingPower has a maximum value of 100, it does not require the full 256-bit storage space allocated by uint256. Reducing this field to a smaller data type, such as uint16 or even uint8, can significantly reduce the storage space and thus the gas costs. uint8 suffices for values up to 255, which covers the maximum required range.
The leafCount might also be optimized based on its expected range. If its maximum value is within a smaller range, it could similarly be downsized to a smaller integer type like uint32 or uint64, depending on the maximum expected number of leaves.
By adjusting the data types to smaller sizes, it becomes possible to pack these variables within fewer storage slots. The Ethereum EVM storage model aligns with 256-bit slots, and smaller variables can be packed together into a single slot if their combined size does not exceed 256 bits.
This packing reduces the number of storage operations required when initializing or updating the CommitteeData struct, leading to lower gas costs.
One storage slot requires 20K of gas, so reducing from 60k (3 storage slots) to 40k is a considerable improvements in EVM, it will lead to approximately reducing gas cost around 2 to 3$ currently which cannot be neglected

Client Response

newway55: Fixed.

Disclaimer