Geode Liquid Staking

Date	May 2023
Auditors	Chingiz Mardanov, Sergii Kravchenko

1 Executive Summary

This report presents the results of our engagement with **Geode Finance** to review their **Liquid Staking Library contracts**.

The review was conducted over two weeks, from **May, 8th** to **May, 19th**, by **Chingiz Mardanov** and **Sergii Kravcheko**. A total of 20 person-days were spent.

This represents the second review of the protocol, following significant changes made to the code subsequent to the previous review. We are pleased to note substantial enhancements in code size, quality, and business logic, resulting in improved comprehensibility and audibility of the codebase. However, it is important to acknowledge that a significant amount of code remains unimplemented, and the majority of contracts lie outside the current scope. Due to these factors, alongside the major changes in the codebase, it proved impracticable to fully isolate the code and conduct a thorough review in an isolated manner. So there may still be issues related to the details of unimplemented code or the code out of scope.

2 Scope

Our review focused on the commit hash `4a4d0b5cd402ceb12fcca03d15a4c948e6bf0830`. Due to the complexity of the system and the time restrictions, only two contnracts are in scope:

File	SHA-1 hash
OracleExtensionLib.sol	30eacd8702f566e06b135c6bc1df9ad6ae4f8dd4
StakeModuleLib.sol	20b6a04e6b1ca5ebfdf9c76d3f3416fcdec0fe9b

2.1 Objectives

Together with the **Geode Finance** team, we identified the following priorities for our review:

1. Correctness of the implementation, consistent with the intended functionality and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.

3 System Overview

The codebase represents a highly modular liquid staking protocol. We only reviewed two libraries that will both be used by the **StakingModule**:

- **StakeModuleLib** - the library contains the code to manage and interact with the staking pools and operators:
    - Node operators can be initialized and managed by the controllers and maintainers.
    - Node operators can create validators and stake funds from the pool if they have enough allowance.
    - Pools can be created and then managed by anyone.
    - Pools can manage the allowance of the node operators.
    - Users can deposit funds to the pool.
- **OracleExtensionLib** - the library contains the code the centralized Oracle actor calls. The main actions that are done through this library:
    - Updating balances and prices of the pool shares.
    - Verifying that the validators are initiated with the correct credentials.
    - Imprisoning malicious or faulty node operators.

3.1 Actors

The relevant actors are listed below with their respective abilities:

- **Node Operators** - actors that are running validators with staked funds borrowed from pools. Represented by controller and maintainer addresses.
- **Pools** - actors that are creating and managing liquidity pools needed to gather funds for staking. Represented by controller and maintainer addresses.
- **Depositors** - any user that deposits to the pools and holds `gETH` tokens (shares of the pools).
- **Oracle** - a centralized entity controlled by Geode. Maintaining correct `gETH` token prices and banning malicious node operators.

4 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

4.1 Node Operators Can Stake Validators That Were Not Proposed by Them. Major

In GeodeFi system node operators are meant to add the new validators in two steps:

- Proposal step where 1 ETH of the pre-stake deposit is committed.
- Stake step, where the 1 ETH pre-stake is reimbursed to the node operator, and the 32ETH user stake is sent to a validator.

The issue itself stems from the fact that node operators are allowed to stake the validators of the other node operators. In the `stake()` function there is no check of the validator's `operatorId` against the operator performing the stake. Meaning that node operator A can stake validators of node operator B.

contracts/Portal/modules/StakeModule/libs/StakeModuleLib.sol:L1478-L1558

```
function stake(
  PooledStaking storage self,
  DSML.IsolatedStorage storage DATASTORE,
  uint256 operatorId,
  bytes[] calldata pubkeys
) external {
  _authenticate(DATASTORE, operatorId, false, true, [true, false]);

  require(
    (pubkeys.length > 0) && (pubkeys.length <= DCL.MAX_DEPOSITS_PER_CALL),
    "SML:1 - 50 validators"
  );

  {
    uint256 _verificationIndex = self.VERIFICATION_INDEX;
    for (uint256 j = 0; j < pubkeys.length; ) {
      require(
        _canStake(self, pubkeys[j], _verificationIndex),
        "SML:NOT all pubkeys are stakeable"
      );

      unchecked {
        j += 1;
      }
    }
  }

  {
    bytes32 activeValKey = DSML.getKey(operatorId, rks.activeValidators);
    bytes32 proposedValKey = DSML.getKey(operatorId, rks.proposedValidators);
    uint256 poolId = self.validators[pubkeys[0]].poolId;
    bytes memory withdrawalCredential = DATASTORE.readBytes(poolId, rks.withdrawalCredential);

    uint256 lastIdChange = 0;
    for (uint256 i = 0; i < pubkeys.length; ) {
      uint256 newPoolId = self.validators[pubkeys[i]].poolId;
      if (poolId != newPoolId) {
        uint256 sinceLastIdChange;

        unchecked {
          sinceLastIdChange = i - lastIdChange;
        }

        DATASTORE.subUint(poolId, rks.secured, (DCL.DEPOSIT_AMOUNT * (sinceLastIdChange)));
        DATASTORE.subUint(poolId, proposedValKey, (sinceLastIdChange));
        DATASTORE.addUint(poolId, activeValKey, (sinceLastIdChange));

        lastIdChange = i;
        poolId = newPoolId;
        withdrawalCredential = DATASTORE.readBytes(poolId, rks.withdrawalCredential);
      }

      DCL.depositValidator(
        pubkeys[i],
        withdrawalCredential,
        self.validators[pubkeys[i]].signature31,
        (DCL.DEPOSIT_AMOUNT - DCL.DEPOSIT_AMOUNT_PRESTAKE)
      );

      self.validators[pubkeys[i]].state = VALIDATOR_STATE.ACTIVE;

      unchecked {
        i += 1;
      }
    }
    {
      uint256 sinceLastIdChange;
      unchecked {
        sinceLastIdChange = pubkeys.length - lastIdChange;
      }
      if (sinceLastIdChange > 0) {
        DATASTORE.subUint(poolId, rks.secured, DCL.DEPOSIT_AMOUNT * (sinceLastIdChange));
        DATASTORE.subUint(poolId, proposedValKey, (sinceLastIdChange));
        DATASTORE.addUint(poolId, activeValKey, (sinceLastIdChange));
      }
    }

    _increaseWalletBalance(DATASTORE, operatorId, DCL.DEPOSIT_AMOUNT_PRESTAKE * pubkeys.length);

    emit Stake(pubkeys);
  }
```

This issue can later be escalated to a point where funds can be stolen. Consider the following case:

- The attacker creates 10 validators directly through the ETH2 deposit contract using himself as the withdrawal address.
- Attacker node operator proposes to add 10 validators and adds the 10ETH as pre-stake deposit. Since validators already exist withdrawal credentials will remain those of the Attacker. As a result of those actions, we have inflated the number of proposed validators the attacker has inside the Geode system.
- Attacker then takes the validator keys proposed by someone else and stakes them. Since there is no check described above that is allowed. His proposed validators count will also decrease without a revert due to steps above.
- As a result of that step, attacker will receive the pre-stake of the operator that actually proposed those validators. The attacker will immediately proceed to call `decreaseWallet()` to withdraw the funds.
- The attacker will then withdraw the pre-stake he deposited in the initial validators with faulty withdrawal credential.

This way an attacker could profit 10ETH.

This can be prevented by making sure that validator's operatorId is checked on the `stake()` function call.

4.2 Cannot Blame Operator for Proposed Validator Medium

In the current code, anyone can blame an operator who does not withdraw in time:

contracts/Portal/modules/StakeModule/libs/StakeModuleLib.sol:L931-L946

```
function blameOperator(
  PooledStaking storage self,
  DSML.IsolatedStorage storage DATASTORE,
  bytes calldata pk
) external {
  require(
    self.validators[pk].state == VALIDATOR_STATE.ACTIVE,
    "SML:validator is never activated"
  );
  require(
    block.timestamp > self.validators[pk].createdAt + self.validators[pk].period,
    "SML:validator is active"
  );

  _imprison(DATASTORE, self.validators[pk].operatorId, pk);
}
```

There is one more scenario where the operator should be blamed. When a validator is in the `PROPOSED` state, only the operator can call the `stake` function to actually stake the rest of the funds. Before that, the funds of the pool will be locked under the `rks.secured` variable. So the malicious operator can lock up 31 ETH of the pool indefinitely by locking up only 1 ETH of the attacker. There is currently no way to release these 31 ETH.

We recommend introducing a mechanism that allows one to blame the operator for not staking for a long time after it was approved.

4.3 Validators Array Length Has to Be Updated When the Validator Is Alienated. Medium

In GeodeFi when the node operator creates a validator with incorrect withdrawal credentials or signatures the Oracle has the ability to alienate this validator. In the process of alienation, the validator status is updated.

contracts/Portal/modules/StakeModule/libs/OracleExtensionLib.sol:L111-L136

```
function _alienateValidator(
  SML.PooledStaking storage STAKE,
  DSML.IsolatedStorage storage DATASTORE,
  uint256 verificationIndex,
  bytes calldata _pk
) internal {
  require(STAKE.validators[_pk].index <= verificationIndex, "OEL:unexpected index");
  require(
    STAKE.validators[_pk].state == VALIDATOR_STATE.PROPOSED,
    "OEL:NOT all pubkeys are pending"
  );

  uint256 operatorId = STAKE.validators[_pk].operatorId;
  SML._imprison(DATASTORE, operatorId, _pk);

  uint256 poolId = STAKE.validators[_pk].poolId;
  DATASTORE.subUint(poolId, rks.secured, DCL.DEPOSIT_AMOUNT);
  DATASTORE.addUint(poolId, rks.surplus, DCL.DEPOSIT_AMOUNT);

  DATASTORE.subUint(poolId, DSML.getKey(operatorId, rks.proposedValidators), 1);
  DATASTORE.addUint(poolId, DSML.getKey(operatorId, rks.alienValidators), 1);

  STAKE.validators[_pk].state = VALIDATOR_STATE.ALIENATED;

  emit Alienated(_pk);
}
```

An additional thing that has to be done during the alienation process is that the validator's count should be decreased in order for the monopoly threshold to be calculated correctly. That is because the length of the `validators` array is used twice in the `OpeartorAllowance` function:

contracts/Portal/modules/StakeModule/libs/StakeModuleLib.sol:L975

```
uint256 numOperatorValidators = DATASTORE.readUint(operatorId, rks.validators);
```

contracts/Portal/modules/StakeModule/libs/StakeModuleLib.sol:L988

```
uint256 numPoolValidators = DATASTORE.readUint(poolId, rks.validators);
```

Without the update of the array length, the monopoly threshold as well as the time when the fallback operator will be able to participate is going to be computed incorrectly.

It could be beneficial to not refer to `rks.validators` in the operator allowance function and instead use the `rks.proposedValidators` + `rks.alienatedValidators` + `rks.activeValidators`. This way allowance function can always rely on the most up to date data.

4.4 A Potential Controller Update Issue. Minor

We identified a potential issue in the code that is out of our current scope. In the `GeodeModuleLib`, there is a function that allows a controller of any ID to update the controller address:

contracts/Portal/modules/GeodeModule/libs/GeodeModuleLib.sol:L299-L309

```
function changeIdCONTROLLER(
  DSML.IsolatedStorage storage DATASTORE,
  uint256 id,
  address newCONTROLLER
) external onlyController(DATASTORE, id) {
  require(newCONTROLLER != address(0), "GML:CONTROLLER can not be zero");

  DATASTORE.writeAddress(id, rks.CONTROLLER, newCONTROLLER);

  emit ControllerChanged(id, newCONTROLLER);
}
```

It's becoming tricky with the upgradability mechanism. The current version of any package is stored in the following format: `DATASTORE.readAddress(versionId, rks. CONTROLLER)`. So the address of the current implementation of any package is stored as `rks.CONTROLLER`. That means if someone can hack the implementation address and make a transaction on its behalf to change the controller, this attacker can change the current implementation to a malicious one.

While this issue may not be exploitable now, many new packages are still to be implemented. So you need to ensure that nobody can get any control over the implementation contract.

4.5 The Price Change Limit Could Prevent the Setting of the Correct Price. Minor

In the share price update logic of OracleExtensionLib, there is a function called `sanityCheck`. As part of that function, a maximum share price change is checked. In reality, for smaller pools, this can result in an inability to update prices. For example, if the pool was serviced by a single node operator who gets severely slashed, the price being reported by the oracle could easily be more different than the maximum threshold. In this case, the pool would operate at the incorrect price for about 24 hours. Given that we do not have the whole code base in the scope and that some part of the codebase is not complete we can not estimate the risk of such an event. We believe that the Withdrawal Module could be one of the affected modules in such a case.

4.6 Potential for a Cross-Site-Scripting When Creating a Pool. Minor

When creating a new staking pool, the creator has the ability to name it. While it does not present many issues on the chain, if this name is ever displayed on the UI it has to be handled carefully. An attacker could include a malicious script in the `name` and that could potentially be executed in the victim's browser.

contracts/Portal/modules/StakeModule/libs/StakeModuleLib.sol:L358

```
DATASTORE.writeBytes(poolId, rks.NAME, name);
```

We suggest that proper escaping is used when displaying the names of the pool on the UI. We do not recommend adding string validation on the chain.

Appendix 1 - Disclosure