Geodefi

1 Executive Summary
2 Scope
2.1 Objectives
3 System Overview
3.1 Actors
3.2 Governing Model
3.3 Trust Model
4 Recommendations
4.1 Review the Code Quality recommendations in Appendix 1
5 Findings
5.1 Oracle's `_sanityCheck` for prices will not work with slashing Critical
5.2 Multiple calculation mistakes in the `_findPricesClearBuffer` function Critical
5.3 New interfaces can add malicious code without any delay or check Major
5.4 MiniGovernance - `fetchUpgradeProposal` will always revert Major
5.5 `reportOracle` can be sandwiched for profit. Medium
5.6 Updating interfaces of derivatives is done in a dangerous and unpredictable manner. Medium
5.7 A sandwich attack on `fetchUnstake` Medium
5.8 Only the `GOVERNANCE` can initialize the `Portal` Medium
5.9 The maintainer of the MiniGovernance can block the `changeMaintainer` function Medium
5.10 Entities are not required to be initiated Medium
5.11 Node operators are not risking anything when abandoning their activity or performing malicious actions Medium
5.12 Planets should not act as operators Medium
5.13 The `blameOperator` can be called for an alienated validator Medium
5.14 Latency timelocks on certain functions can be bypassed Medium
5.15 MiniGovernance's senate has almost unlimited validity Medium
5.16 Proposed validators not accounted for in the monopoly check. Medium
5.17 Comparison operator used instead of assignment operator Medium
5.18 `initiator` modifier will not work in the context of one transaction Minor
5.19 Incorrect accounting for the burned gEth Minor
5.20 Boost calculation on fetchUnstake should not be using the cumBalance when it is larger than debt. Minor

Date	November 2022
Auditors	Sergii Kravchenko, Christian Goll, Chingiz Mardanov

1 Executive Summary

This report presents the results of our engagement with **Geode Finance** to review **Geodefi ETH Portal**.

The review was conducted over six weeks, from **November 1st, 2022** to **December 9th, 2022**, by **Sergii Kravchenko**, **Christian Goll** and **Chingiz Mardanov**. A total of 60 person-days were spent.

During this engagement, we audited Geode's liquid staking solution which allows anyone to launch configurable staked ETH 2.0 derivatives. The Geode team's priority was to deliver an alternative staking solution that is trustless and decentralized. We believe that this is a good application and an important problem that the Geode team is trying to solve, helping us all achieve a more decentralized staking state. As a result of that goal, two main components can be identified in the codebase: **Governance Logic** and **Staking Logic**.

While going over the governance portion of the code we identified several issues of varying severity. We believe that some of those issues could have been avoided if the entire architecture was simplified. The code complexity makes it hard to audit or make any definitive statements about the safety of the codebase. While we do understand that Geode team is trying to solve a complex problem, we overall see this codebase as over-engineered. We have provided some examples and recommendations in the Code Quality Appendix.

While going over the staking portion of the protocol, it becomes apparent that there is a big dependency on the off-chain oracle component to operate correctly and fairly. This off-chain infrastructure was not in the scope of this audit so we can not speak for it's security. We also would like to mention that a lot of the aspects of Geode's systems are not finalized for various reasons, some outside of the Geode team's control, such as ETH 2.0 withdrawals still not being enabled or finalized, some required EIPs still in development and simply unfinished implementations such as in the case of Comets (private staking pools). That being said, this system will change shape many times to come, which made it hard to audit.

It is also important to mention that while there are some tests written, there are not enough of them for the system of this scale. Some of the issues we found would have been found with more extensive testing. For that reason, we believe that more bugs could arise as the system is tested further or operates on testnet.

We'd like to note that the Geode team was extremely responsive and open to suggestions. They have been in constant contact with the auditing team working on fixes as the issues were reported. We believe that with some work this codebase will become a lot more auditable and as a result - secure.

2 Scope

Our review focused on the commit hash 8b07c0723d2a655a20d26620d4c3962cb9de4b00. The list of files in scope can be found in the Appendix.

2.1 Objectives

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

1. Ensure that the system is implemented consistently 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

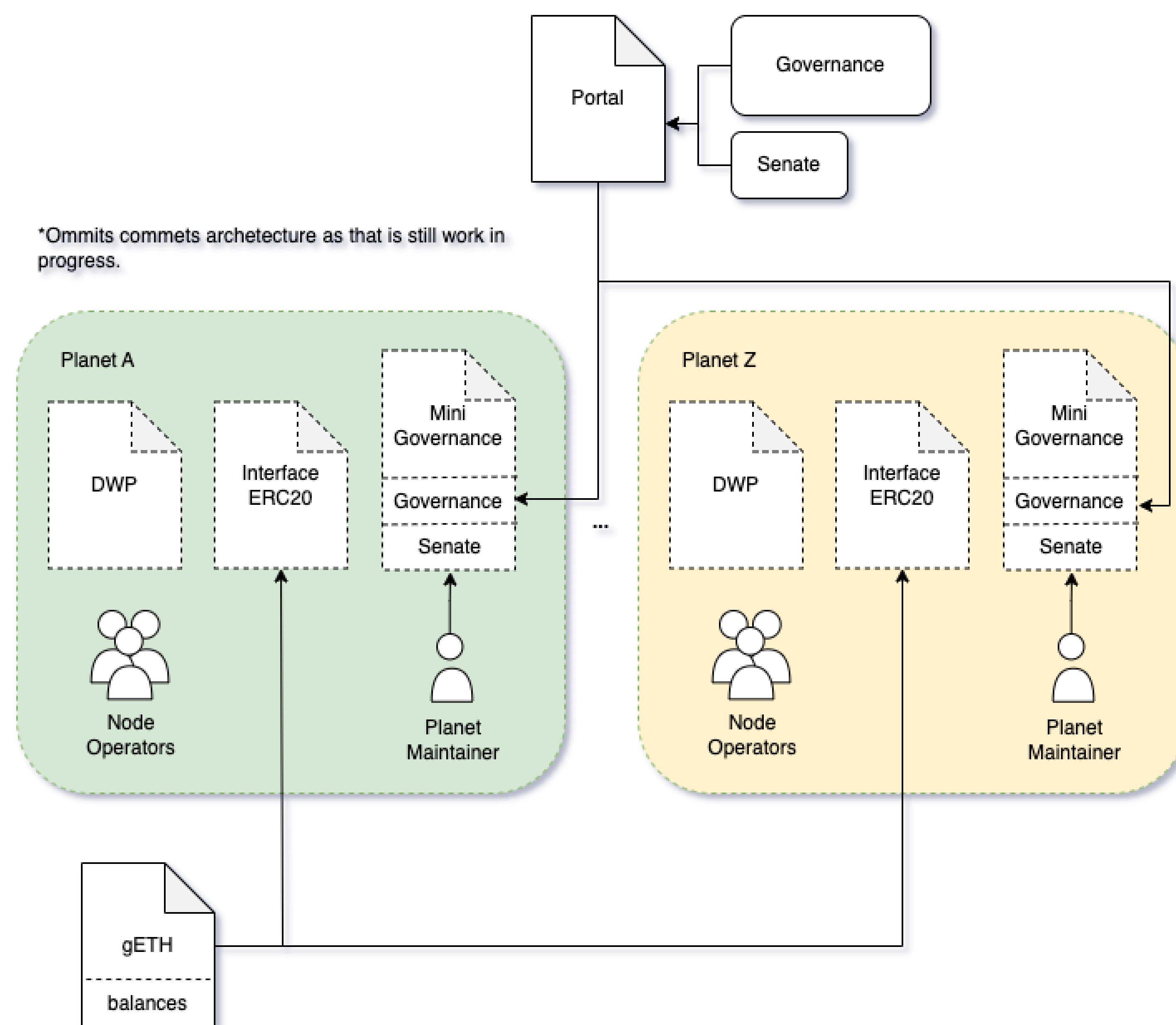
3.1 Actors

Geode Eth Portal is a complex system with many players involved, in this section we outline of the actors as well as briefly describe their responsibilities.

5.21 DataStore struct not having the _gap for upgrades. Minor

Appendix 1 - Code Quality Recommendations

A.1.1 Use Enums for different proposal types as well as states across the system.



- **Governance** - First line of defense in the Geode Universe. At the moment of writing this report, the Geode team is intending that their token will act as governance and that any holder will be able to participate in voting. Governance is capable of creating on-chain proposals but can not approve them.
- **Senate** - is the second line of defense in Geode Universe. At the time of writing Senate is a multisig with signers selected by the Geode team. Senate has the power to approve proposals created by the Governance. Senate has no power to elect Senate. Senate is elected by controllers of the entities of approved types.
- **Node Operators** - are responsible for creating and maintaining validator nodes for the planets. Each new node operator is first proposed by the Governance and then approved by Senate. After that, the node operator is given an allowance for the maximum amount of validator nodes they can create. It is worth mentioning that Planets are also Node Operators for themselves. So Each planet can create validators for itself without any additional proposals to create a node operator.
- **MiniGovernance** - is not an actor per se. It is a governance structure that replicates the main Geode Governance but in the scope of one planet. Just like the rest of Geode, it contains two parts as well: Governance and Senate. Where Geode Portal acts as governance and planet maintainer acts as the senate. So Portal can create new proposals and the maintainer can choose to approve them.
- **Validators** - actual validator nodes represented by BLS12-381 public keys and maintained by node operators. Planet's MiniGovernance is supposed to be selected as the Withdrawal Credentials for the validators associated with the planet.
- **Oracle** - is an off-chain infrastructure that is responsible for monitoring the behavior of node operators and the balance of the validators on the beacon chain.

3.2 Governing Model

Geode protocol has an interesting Governance model that is replicated on two different scales: protocol's main entry point a.k.a Portal level and planet level. On the Portal level, those actors are Governance and Senate that we have introduced in the earlier section. At the moment of writing the Governance will be the protocol's token and the senate will be the multisig. Governance can create proposals while Senate can approve them. One such proposal that can be created is the creation of the new staking Planet (staking derivative). On a planet level, a new MiniGovernance will be created specifically for that planet which essentially replicates all the functionality of the main governing model, with again two actors: Governance and Senate. In the case of the MiniGovernance, the Governance role is taken by the Portal and the senate role is filled by a maintainer of the Planet passed in the proposal for planet creation.

3.3 Trust Model

In any system, it's important to identify what trust is expected/required between various actors. For this audit, we established the following trust assumptions:

- Geode Oracle is trusted and implemented correctly. As of right without an on-chain verification Geode Oracle has the ability to:
  - Imprison Node Operators.
  - Unstake/Stake validator nodes.
  - Dictate the order in which validators unstake thus distributing the boost to node operators.
  - Update the prices.

  It is crucial for all those steps to be performed regularly and correctly in order to maintain a healthy state of the protocol.
- Geode Governance is a token and Geode team controls 100% of its supply. This eliminates a possibility of a malicious proposal being created.
- Geode Team itself is considered to be a trusted actor.

4 Recommendations

4.1 Review the Code Quality recommendations in Appendix 1

Other comments related to readability and best practices are listed in Appendix 1

5 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.

5.1 Oracle's `_sanityCheck` for prices will not work with slashing Critical

Description

The `_sanityCheck` is verifying that the new price didn't change significantly:

code/contracts/Portal/utils/OracleUtilsLib.sol:L405-L417

```
uint256 maxPrice = curPrice +
    ((curPrice *
        self.PERIOD_PRICE_INCREASE_LIMIT *
        _periodsSinceUpdate) / PERCENTAGE_DENOMINATOR);

uint256 minPrice = curPrice -
    ((curPrice *
        self.PERIOD_PRICE_DECREASE_LIMIT *
        _periodsSinceUpdate) / PERCENTAGE_DENOMINATOR);

require(
    _newPrice >= minPrice && _newPrice <= maxPrice,
    "OracleUtils: price is insane"
```

While the rewards of staking can be reasonably predicted, the balances may also be changed due to slashing. So any slashing event should reduce the price, and if enough ETH is slashed, the price will drop heavily. The oracle will not be updated because of a sanity check. After that, there will be an arbitrage opportunity, and everyone will be incentivized to withdraw as soon as possible. That process will inevitably devaluate gETH to zero. The severity of this issue is also amplified by the fact that operators have no skin in the game and won't lose anything from slashing.

Recommendation

Make sure that slashing can be adequately processed when updating the price.

5.2 Multiple calculation mistakes in the `_findPricesClearBuffer` function Critical

Description

The `_findPricesClearBuffer` function is designed to calculate the gETH/ETH prices. The first one (oracle price) is the price at the reference point, for ease of calculation let's assume it is midnight. The second price is the price at the time the `reportOracle` is called.

code/contracts/Portal/utils/OracleUtilsLib.sol:L388

```
return (unbufferedEther / unbufferedSupply, totalEther / supply);
```

To calculate the oracle price at midnight, the current ETH balance is reduced by all the minted gETH (converted to ETH with the old price) and increased by all the burnt gETH (converted to ETH with the old price) starting from midnight to the time transaction is being executed:

code/contracts/Portal/utils/OracleUtilsLib.sol:L368-L374

```
uint256 unbufferedEther = totalEther -
    (DATASTORE.readUintForId(_poolId, _dailyBufferMintKey) * price) /
    self.gETH.totalSupply(_poolId);

unbufferedEther +=
    (DATASTORE.readUintForId(_poolId, _dailyBufferBurnKey) * price) /
    self.gETH.denominator();
```

But in the first calculation, the `self.gETH.totalSupply(_poolId)` is mistakenly used instead of `self.gETH.denominator()`. This can lead to the `unbufferedEther` being much larger, and the eventual oracle price will be much larger too.

There is another serious calculation mistake. In the end, the function returns the following line:

code/contracts/Portal/utils/OracleUtilsLib.sol:L388

```
return (unbufferedEther / unbufferedSupply, totalEther / supply);
```

But none of these values are multiplied by `self.gETH.denominator();` so they are in the same range. Both values will usually be around `1`. While the actual price value should be multiplied by `self.gETH.denominator();`.

5.3 New interfaces can add malicious code without any delay or check Major

Description

Geode Finance uses an interesting system of contracts for each individual staked ETH derivative. At the base of it all is an ERC1155 gETH contract where planet id acts as a token id. To make it more compatible with the rest of DeFi the Geode team pairs

it up with an ERC20 contract that users would normally interact with and where all the allowances are stored. Naturally, since the balances are stored in the gETH contract, ERC20 interfaces need to ask gETH contract to update the balance. It is done in a way where the gETH contract will perform any transfer requested by the interface since the interface is expected to do all the checks and accountings. The issue comes with the fact that planet maintainers can whitelist new interfaces and that process does not require any approval. Planet maintainers could whitelist an interface that will send all the available tokens to the maintainer's wallet for example. This essentially allows Planet maintainers to steal all derivative tokens in circulation in one transaction.

Examples

code/contracts/Portal/utils/StakeUtilsLib.sol:L165-L173

```
function setInterface(
    StakePool storage self,
    DataStoreUtils.DataStore storage DATASTORE,
    uint256 id,
    address _interface
) external {
    DATASTORE.authenticate(id, true, [false, true, true]);
    _setInterface(self, DATASTORE, id, _interface);
}
```

Recommendation

`gETH.sol` contract has a concept of avoiders. One of the ways to fix this issue is to have the avoidance be set on a per-interface basis and avoiding new interfaces by default. This way users will need to allow the new tokens to access the balances.

5.4 MiniGovernance - `fetchUpgradeProposal` will always revert Major

Description

In the function `fetchUpgradeProposal()`, `newProposal()` is called with a hard coded `duration` of 4 weeks. This means the function will always revert since `newProposal()` checks that the proposal duration is not more than the constant `MAX_PROPOSAL_DURATION` of 2 weeks. Effectively, this leaves MiniGovernance non-upgradeable.

Examples

code/contracts/Portal/MiniGovernance/MiniGovernance.sol:L183

```
GEM.newProposal(proposal.CONTROLLER, 2, proposal.NAME, 4 weeks);
```

code/contracts/Portal/utils/GeodeUtilsLib.sol:L328-L331

```
require(
    duration <= MAX_PROPOSAL_DURATION,
    "GeodeUtils: duration exceeds MAX_PROPOSAL_DURATION"
);
```

Recommendation

Switch the hard coded proposal duration to 2 weeks.

5.5 `reportOracle` can be sandwiched for profit. Medium

Description

The fact that price update happens in an on-chain transaction gives the searches the ability to see the future price and then act accordingly.

Examples

MEV searcher can find the `reportOracle` transaction in the mem-pool and if the price is about to increase he could proceed to mint as much gETH as he can with a flash loan. They would then bundle the `reportOracle` transaction. Finally, they would redeem all the gETH for ETH at a higher price per share value as the last transaction in the bundle.

This paired with the fact that oracle might be updated less frequently than once per day, could lead to the fact that profits from this attack will outweigh the fees for performing it.

Fortunately, due to the nature of the protocol, the price fluctuations from day to day will most likely be smaller than the fees encountered during this arbitrage, but this is still something to be aware of when updating the values for DWP donations and fees. But it also makes it crucial to update the oracle every day not to increase the profit margins for this attack.

5.6 Updating interfaces of derivatives is done in a dangerous and unpredictable manner. Medium

Description

Geode Finance codebase provides planet maintainers with the ability to enable or disable different contracts to act as the main token contract. In fact, multiple separate contracts can be used at the same time if decided so by the planet maintainer. Those contracts will have shared balances but will not share the allowances as you can see below:

code/contracts/Portal/helpers/ERC1155SupplyMinterPauser.sol:L47

```
mapping(uint256 => mapping(address => uint256)) private _balances;
```

code/contracts/Portal/gETHInterfaces/ERC20InterfaceUpgradable.sol:L60

```
mapping(address => mapping(address => uint256)) private _allowances;
```

Unfortunately, this approach comes with some implications that are very hard to predict as they involve interactions with other systems, but is possible to say that the consequences of those implications will most always be negative. We will not be able to outline all the implications of this issue, but we can try and outline the pattern that they all would follow.

Examples

There are really two ways to update an interface: set the new one and immediately unset the old one, or have them both run in parallel for some time. Let's look at them one by one.

in the first case, the old interface is disabled immediately. Given that interfaces share balances that will lead to some very serious consequences. Imagine the following sequence:

1. Alice deposits her derivatives into the DWP contract for liquidity mining.
2. Planet maintainer updates the interface and immediately disables the old one.
3. DWP contract now has the old tokens and the new ones. But only the new ones are accounted for in the storage and thus can be withdrawn. Unfortunately, the old tokens are disabled meaning that now both old and new tokens are lost.

This can happen in pretty much any contract and not just the DWP token. Unless the holders had enough time to withdraw the derivatives back to their wallets all the funds deposited into contracts could be lost.

This leads us to the second case where the two interfaces are active in parallel. This would solve the issue above by allowing Alice to withdraw the old tokens from the DWP and make the new tokens follow. Unfortunately, there is an issue in that case as well.

Some DeFi contracts allow their owners to withdraw any tokens that are not accounted for by the internal accounting. DWP allows the withdrawal of admin fees if the contract has more tokens than `balances[]` store. Some contracts even allow to withdraw funds that were accidentally sent to the contract by people. Either to recover them or just as a part of dust collection. Let's call such contracts "dangerous contracts" for our purposes.

1. Alice deposits her derivatives into the dangerous contract.
2. Planet maintainer sets a new interface.
3. Owner of the dangerous contract sees that some odd and unaccounted tokens landed in the contract. He learns those are real and are part of Geode ecosystem. So he takes them.
4. Old tokens will follow the new tokens. That means Alice now has no claim to them and the contract that they just left has broken accounting since numbers there are not backed by tokens anymore.

One other issue we would like to highlight here is that despite the contracts being expected to have separate allowances, if the old contract has the allowance set, the initial 0 value of the new one will be ignored. Here is an example:

1. Alice approves Bob for 100 derivatives.
2. Planet maintainer sets a new interface. The new interface has no allowance from Alice to Bob.
3. Bob still can transfer new tokens from Alice to himself by transferring the old tokens for which he still has the allowance. New token balances will be updated accordingly.

Alice could also give Bob an allowance of 100 tokens in the new contract since that was her original intent, but this would mean that Bob now has 200 token allowance.

This is extremely convoluted and will most likely result in errors made by the planet maintainers when updating the interfaces.

Recommendation

The safest option is to only allow a list of whitelisted interfaces to be used that are well-documented and audited. Planet maintainers could then choose the once that they see fit.

5.7 A sandwich attack on `fetchUnstake` Medium

Description

Operators are incentivized to withdraw the stake when there is a debt in the system. Withdrawn ETH will be sold in the DWP, and a portion of the arbitrage profit will be sent to the operator. But the operators cannot unstake and earn the arbitrage boost instantly. Node operator will need to start the withdrawal process, signal unstake, and only then, after some time, potentially days, Oracle will trigger `fetchUnstake` and will take the arbitrage opportunity if it is still there.

code/contracts/Portal/utils/StakeUtilsLib.sol:L1276-L1288

```
function fetchUnstake(
    StakePool storage self,
    DataStoreUtils.DataStore storage DATASTORE,
    uint256 poolId,
    uint256 operatorId,
    bytes[] calldata pubkeys,
    uint256[] calldata balances,
    bool[] calldata isExit
) external {
    require(
        msg.sender == self.TELESCOPE.ORACLE_POSITION,
        "StakeUtils: sender NOT ORACLE"
    );
```

In reality, the DWP contract's swap function is external and can be used by anyone, so anyone could try and take the arbitrage.

code/contracts/Portal/withdrawalPool/Swap.sol:L341-L358

```
function swap(
    uint8 tokenIndexFrom,
    uint8 tokenIndexTo,
    uint256 dx,
    uint256 minDy,
    uint256 deadline
)
    external
    payable
    virtual
    override
    nonReentrant
    whenNotPaused
    deadlineCheck(deadline)
    returns (uint256)
{
    return swapStorage.swap(tokenIndexFrom, tokenIndexTo, dx, minDy);
}
```

In fact, one could take this arbitrage with no risk or personal funds. This is due to the fact that `fetchUnstake()` could get sandwiched. Consider the following case:

1. There is a debt in the DWP and the node operator decides to withdraw the stake to take the arbitrage opportunity.
2. After some time the Oracle will actually finalize the withdrawal by calling `fecthUnstake`.
3. If debt is still there MEV searcher will see that transaction in the mem-pool and will take an ETH loan to buy cheap gETH.
4. `fetchUnstake()` will execute and since the debt was repaid in the previous step all of the withdrawn ETH will go into `surplus`.
5. Searcher will redeem gETH that they bought for the oracle price from surplus and will get all of the profit.

At the end of the day, the goal of regaining the peg will be accomplished, but node operators will not be interested in withdrawing early later. This will potentially create unhealthy situations when withdrawals are required in case of a serious de-peg.

5.8 Only the `GOVERNANCE` can initialize the `Portal` Medium

Description

In the `Portal`'s `initialize` function, the `_GOVERNANCE` is passed as a parameter:

code/contracts/Portal/Portal.sol:L156-L196

```
function initialize(
    address _GOVERNANCE,
    address _gETH,
    address _ORACLE_POSITION,
    address _DEFAULT_gETH_INTERFACE,
    address _DEFAULT_DWP,
    address _DEFAULT_LP_TOKEN,
    address _MINI_GOVERNANCE_POSITION,
    uint256 _GOVERNANCE_TAX,
    uint256 _COMET_TAX,
    uint256 _MAX_MAINTAINER_FEE,
    uint256 _BOOSTRAP_PERIOD
) public virtual override initializer {
    __ReentrancyGuard_init();
    __Pausable_init();
    __ERC1155Holder_init();
    __UUPSUpgradeable_init();

    GEODE.SENATE = _GOVERNANCE;
    GEODE.GOVERNANCE = _GOVERNANCE;
    GEODE.GOVERNANCE_TAX = _GOVERNANCE_TAX;
    GEODE.MAX_GOVERNANCE_TAX = _GOVERNANCE_TAX;
    GEODE.SENATE_EXPIRY = type(uint256).max;

    STAKEPOOL.GOVERNANCE = _GOVERNANCE;
    STAKEPOOL.gETH = IgETH(_gETH);
    STAKEPOOL.TELESCOPE.gETH = IgETH(_gETH);
    STAKEPOOL.TELESCOPE.ORACLE_POSITION = _ORACLE_POSITION;
    STAKEPOOL.TELESCOPE.MONOPOLY_THRESHOLD = 20000;

    updateStakingParams(
        _DEFAULT_gETH_INTERFACE,
        _DEFAULT_DWP,
        _DEFAULT_LP_TOKEN,
        _MAX_MAINTAINER_FEE,
        _BOOSTRAP_PERIOD,
        type(uint256).max,
        type(uint256).max,
        _COMET_TAX,
        3 days
    );
```

But then it calls the `updateStakingParams` function, which requires the `msg.sender` to be the governance:

code/contracts/Portal/Portal.sol:L651-L665

```
function updateStakingParams(
    address _DEFAULT_gETH_INTERFACE,
    address _DEFAULT_DWP,
    address _DEFAULT_LP_TOKEN,
    uint256 _MAX_MAINTAINER_FEE,
    uint256 _BOOSTRAP_PERIOD,
    uint256 _PERIOD_PRICE_INCREASE_LIMIT,
    uint256 _PERIOD_PRICE_DECREASE_LIMIT,
    uint256 _COMET_TAX,
    uint256 _BOOST_SWITCH_LATENCY
) public virtual override {
    require(
        msg.sender == GEODE.GOVERNANCE,
        "Portal: sender not GOVERNANCE"
    );
```

So only the future governance can initialize the `Portal`. In the case of the Geode protocol, the governance will be represented by a token contract, making it hard to initialize promptly. Initialization should be done by an actor that is more flexible than governance.

Recommendation

Split the `updateStakingParams` function into public and private ones and use them accordingly.

5.9 The maintainer of the MiniGovernance can block the `changeMaintainer` function Medium

Description

Every entity with an ID has a controller and a maintainer. The controller tends to have more control, and the maintainer is mostly used for operational purposes. So the controller should be able to change the maintainer if that is required. Indeed we see that it is possible in the MiniGovernance too:

code/contracts/Portal/MiniGovernance/MiniGovernance.sol:L224-L246

```
function changeMaintainer(
    bytes calldata password,
    bytes32 newPasswordHash,
    address newMaintainer
)
    external
    virtual
    override
    onlyPortal
    whenNotPaused
    returns (bool success)
{
    require(
        SELF.PASSWORD_HASH == bytes32(0) ||
            SELF.PASSWORD_HASH ==
            keccak256(abi.encodePacked(SELF.ID, password))
    );
    SELF.PASSWORD_HASH = newPasswordHash;

    _refreshSenate(newMaintainer);

    success = true;
}
```

Here the `changeMaintainer` function can only be called by the Portal, and only the controller can initiate that call. But the maintainer can pause the MiniGovernance, which will make this call revert because the `_refreshSenate` function has the `whenNotPaused` modifier. Thus maintainer could intentionally prevent the controller from replacing it by another maintainer.

Recommendation

Make sure that the controller can always change the malicious maintainer.

5.10 Entities are not required to be initiated Medium

Description

Every entity (Planet, Comet, Operator) has a 3-step creation process:

- Creation of the proposal.
- Approval of the proposal.
- Initiation of the entity.

The last step is crucial, but it is never explicitly checked that the entity is initialized. The initiation always includes the `initiator` modifier that works with the `"initiated"` slot on `DATASTORE`:

code/contracts/Portal/utils/MaintainerUtilsLib.sol:L46-L72

```
modifier initiator(
    DataStoreUtils.DataStore storage DATASTORE,
    uint256 _TYPE,
    uint256 _id,
    address _maintainer
) {
    require(
        msg.sender == DATASTORE.readAddressForId(_id, "CONTROLLER"),
        "MaintainerUtils: sender NOT CONTROLLER"
    );
    require(
        DATASTORE.readUintForId(_id, "TYPE") == _TYPE,
        "MaintainerUtils: id NOT correct TYPE"
    );
    require(
        DATASTORE.readUintForId(_id, "initiated") == 0,
        "MaintainerUtils: already initiated"
    );

    DATASTORE.writeAddressForId(_id, "maintainer", _maintainer);

    _;

    DATASTORE.writeUintForId(_id, "initiated", block.timestamp);

    emit IdInitiated(_id, _TYPE);
}
```

But this slot is never actually checked when the entities are used. While we did not find any profitable attack vector using uninitiated entities, the code will be upgraded, which may allow for possible attack vectors related to this issue.

Recommendation

Make sure the entities are initiated before they are used.

5.11 Node operators are not risking anything when abandoning their activity or performing malicious actions Medium

Description

During the staking process, the node operators need to provide 1 ETH as a deposit for every validator that they would like to initiate. After that is done, Oracle needs to ensure that validator creation has been done correctly and then deposit the remaining 31 ETH on chain as well as reimburse 1 ETH back to the node operator. The node operator can then proceed to withdraw the funds that were used as initial deposits. As the result, node operators operate nodes that have 32 ETH each and none of which originally belonged to the operator. They essentially have no skin in the game to continue managing the validators besides a potential share in staking rewards. Instead, node operators could stop operation, or try to get slashed on purpose to create turmoil around derivatives on the market and try to capitalize while shorting the assets elsewhere.

Recommendation

Senate will need to be extra careful when approving operator onboarding proposals or potentially only reimburse the node operators the initial deposit after the funds were withdrawn from the MiniGovernance.

5.12 Planets should not act as operators Medium

Description

The system stores every entity (e.g., planet, comet, and operator) separately in `DATASTORE` under different IDs. But there is one exception, every planet can also act as an operator by default. This exception bypasses the general rule and goes against some expectations readers might have about the code:

- Every entity with ID has fees; they are stored in `DATASTORE` for each entity `DATASTORE.readUintForId(id, "fee")`. The fees for a planet and an operator should be able to be different. But if a planet acts like an operator, both fees are stored under the same variable.
- The same problem arises with the maintainer address. Since there will probably be different scripts for maintaining a planet and an operator, having separate addresses for the maintainers would make sense.
- Every operator should be initialized before usage, but it is impossible to initialize a planet as an operator. There are two reasons behind it. First, only the original “Operator type” can call `initiateOperator`, while the planet will have a “Planet type”. Second, an entity cannot be initialized twice; even different initialization functions use the same “initiated” storage slot.

Recommendation

Do not allow planets to be operators in the code. If every planet should be able to act as an operator simultaneously, it is better to create separate operator entities for every planet.

5.13 The `blameOperator` can be called for an alienated validator Medium

Description

The `blameOperator` function is designed to be called by anyone. If some operator did not signal to exit in time, anyone can blame and imprison this operator.

code/contracts/Portal/utils/StakeUtilsLib.sol:L1205-L1224

```
/**
 * @notice allows improsening an Operator if the validator have not been exited until expectedExit
 * @dev anyone can call this function
 * @dev if operator has given enough allowence, they can rotate the validators to avoid being prisoned
 */
function blameOperator(
    StakePool storage self,
    DataStoreUtils.DataStore storage DATASTORE,
    bytes calldata pk
) external {
    if (
        block.timestamp > self.TELESCOPE._validators[pk].expectedExit &&
        self.TELESCOPE._validators[pk].state != 3
    ) {
        OracleUtils.imprison(
            DATASTORE,
            self.TELESCOPE._validators[pk].operatorId
        );
    }
}
```

The problem is that it can be called for any state that is not `3` (`self.TELESCOPE._validators[pk].state != 3`). But it should only be called for active validators whose state equals `2`. So the `blameOperator` can be called an infinite amount of time for alienated or not approved validators. These types of validators cannot switch to state `3`.

The severity of the issue is mitigated by the fact that this function is currently unavailable for users to call. But it is intended to be external once the withdrawal process is in place.

Recommendation

Make sure that you can only blame the operator of an active validator.

5.14 Latency timelocks on certain functions can be bypassed Medium

Description

The functions `switchMaintainerFee()` and `switchWithdrawalBoost()` add a latency of typically three days to the current timestamp at which the new value is meant to be valid. However, they don't limit the number of times this value can be changed within the latency period. This allows a malicious maintainer to set their desired value twice and effectively make the change immediately. Let's take the first function as an example. The first call to it sets a value as the `newFee`, moving the old value to `priorFee`, which is effectively the fee in use until the time lock is up. A follow-up call to the function with the same value as a parameter would mean the "new" value overwrites the old `priorFee` while remaining in the queue for the switch.

Examples

code/contracts/Portal/utils/MaintainerUtilsLib.sol:L311-L333

```
function switchMaintainerFee(
    DataStoreUtils.DataStore storage DATASTORE,
    uint256 id,
    uint256 newFee
) external {
    DATASTORE.writeUintForId(
        id,
        "priorFee",
        DATASTORE.readUintForId(id, "fee")
    );
    DATASTORE.writeUintForId(
        id,
        "feeSwitch",
        block.timestamp + FEE_SWITCH_LATENCY
    );
    DATASTORE.writeUintForId(id, "fee", newFee);

    emit MaintainerFeeSwitched(
        id,
        newFee,
        block.timestamp + FEE_SWITCH_LATENCY
    );
}
```

code/contracts/Portal/utils/MaintainerUtilsLib.sol:L296-L304

```
function getMaintainerFee(
    DataStoreUtils.DataStore storage DATASTORE,
    uint256 id
) internal view returns (uint256 fee) {
    if (DATASTORE.readUintForId(id, "feeSwitch") > block.timestamp) {
        return DATASTORE.readUintForId(id, "priorFee");
    }
    return DATASTORE.readUintForId(id, "fee");
}
```

Recommendation

Add a check to make sure only one value can be set between time lock periods.

5.15 MiniGovernance's senate has almost unlimited validity Medium

Description

A new senate for the MiniGovernance contract is set in the following line:

code/contracts/Portal/MiniGovernance/MiniGovernance.sol:L201

```
GEM._setSenate(newSenate, block.timestamp + SENATE_VALIDITY);
```

The validity period argument should not include `block.timestamp`, because it is going to be added a bit later in the code:

code/contracts/Portal/utils/GeodeUtilsLib.sol:L496

```
self.SENATE_EXPIRY = block.timestamp + _senatePeriod;
```

So currently, every senate of MiniGovernance will have much longer validity than it is supposed to.

Recommendation

Pass only `SENATE_VALIDITY` in the `_refreshSenate` function.

5.16 Proposed validators not accounted for in the monopoly check. Medium

Description

The Geode team introduced a check that makes sure that node operators do not initiate more validators than a threshold called `MONOPOLY_THRESHOLD` allows. It is used on call to `proposeStake(...)` which the operator would call in order to propose new validators. It is worth mentioning that onboarding new validator nodes requires 2 steps: a proposal from the node operator and approval from the planet maintainer. After the first step validators get a status of `proposed`. After the second step validators get the status of `active` and all eth accounting is done. The issue we found is that the proposed validators step performs the monopoly check but does not account for previously proposed but not active validators.

Examples

Assume that `MONOPOLY_THRESHOLD` is set to 5. The node operator could propose 4 new validators and pass the monopoly check and label those validators as `proposed`. The node operator could then suggest 4 more validators in a separate transaction and since the monopoly check does not check for the proposed validators, that would pass as well. Then in `beaconStake` or the step of maintainer approval, there is no monopoly check at all, so 8 validators could be activated at once.

code/contracts/Portal/utils/StakeUtilsLib.sol:L978-L982

```
require(
    (DATASTORE.readUintForId(operatorId, "totalActiveValidators") +
        pubkeys.length) <= self.TELESCOPE.MONOPOLY_THRESHOLD,
    "StakeUtils: IceBear does NOT like monopolies"
);
```

Recommendation

Include the `(DATASTORE.readUintForId(poolId,DataStoreUtils.getKey(operatorId, "proposedValidators"))` into the require statement, just like in the check for the node operator allowance check.

code/contracts/Portal/utils/StakeUtilsLib.sol:L983-L995

```
require(
    (DATASTORE.readUintForId(
        poolId,
        DataStoreUtils.getKey(operatorId, "proposedValidators")
    ) +
        DATASTORE.readUintForId(
            poolId,
            DataStoreUtils.getKey(operatorId, "activeValidators")
        ) +
        pubkeys.length) <=
        operatorAllowance(DATASTORE, poolId, operatorId),
    "StakeUtils: NOT enough allowance"
);
```

5.17 Comparison operator used instead of assignment operator Medium

Description

A common typo is present twice in the `OracleUtilsLib.sol` where `==` is used instead of `=` resulting in incorrect storage updates.

Examples

code/contracts/Portal/utils/OracleUtilsLib.sol:L250

```
self._validators[_pk].state == 2;
```

code/contracts/Portal/utils/OracleUtilsLib.sol:L269

```
self._validators[_pk].state == 3;
```

Recommendation

Replace `==` with `=`.

5.18 `initiator` modifier will not work in the context of one transaction Minor

Description

Each planet, comet or operator must be initialized after the onboarding proposal is approved. In order to make sure that these entities are not initialized more than once `initiateOperator`, `initiateComet` and `initiatePlanet` have the `initiator` modifier.

code/contracts/Portal/utils/MaintainerUtilsLib.sol:L135-L147

```
function initiatePlanet(
    DataStoreUtils.DataStore storage DATASTORE,
    uint256[3] memory uintSpecs,
    address[5] memory addressSpecs,
    string[2] calldata interfaceSpecs
)
    external
    initiator(DATASTORE, 5, uintSpecs[0], addressSpecs[1])
    returns (
        address miniGovernance,
        address gInterface,
        address withdrawalPool
    )
```

code/contracts/Portal/utils/MaintainerUtilsLib.sol:L184-L189

```
function initiateComet(
    DataStoreUtils.DataStore storage DATASTORE,
    uint256 id,
    uint256 fee,
    address maintainer
) external initiator(DATASTORE, 6, id, maintainer) {
```

code/contracts/Portal/utils/MaintainerUtilsLib.sol:L119-L124

```
function initiateOperator(
    DataStoreUtils.DataStore storage DATASTORE,
    uint256 id,
    uint256 fee,
    address maintainer
) external initiator(DATASTORE, 4, id, maintainer) {
```

Inside that modifier, we check that the `initiated` flag is 0 and if so we proceed to initialization. We later update it to the current timestamp.

code/contracts/Portal/utils/MaintainerUtilsLib.sol:L46-L72

```
modifier initiator(
    DataStoreUtils.DataStore storage DATASTORE,
    uint256 _TYPE,
    uint256 _id,
    address _maintainer
) {
    require(
        msg.sender == DATASTORE.readAddressForId(_id, "CONTROLLER"),
        "MaintainerUtils: sender NOT CONTROLLER"
    );
    require(
        DATASTORE.readUintForId(_id, "TYPE") == _TYPE,
        "MaintainerUtils: id NOT correct TYPE"
    );
    require(
        DATASTORE.readUintForId(_id, "initiated") == 0,
        "MaintainerUtils: already initiated"
    );

    DATASTORE.writeAddressForId(_id, "maintainer", _maintainer);

    _;

    DATASTORE.writeUintForId(_id, "initiated", block.timestamp);

    emit IdInitiated(_id, _TYPE);
}
```

Unfortunately, this does not follow the checks-effects-interractions pattern. If one for example would call `initiatePlanet` again from the body of the modifier, this check will still pass making it susceptible to a reentrancy attack. While we could not find a way to exploit this in the current engagement, given that system is designed to be upgradable this could become a risk in the future. For example, if during the initialization of the planet the maintainer will be allowed to pass a custom interface that could potentially allow reentering.

Recommendation

Bring the line that updated the `initiated` flag to the current timestamp before the `_;`.

code/contracts/Portal/utils/MaintainerUtilsLib.sol:L69

```
DATASTORE.writeUintForId(_id, "initiated", block.timestamp);
```

5.19 Incorrect accounting for the burned gEth Minor

Description

Geode Portal records the amount of minted and burned gETH on any given day during the active period of the oracle. One case where some gETH is burned is when the users redeem gETH for ETH. In the burn function we burn the `spentGeth - gEthDonation` but

in the accounting code we do not account for `gEthDonation` so the code records more assets burned than was really burned.

Examples

code/contracts/Portal/utils/StakeUtilsLib.sol:L823-L832

```
DATASTORE.subUintForId(poolId, "surplus", spentSurplus);
self.gETH.burn(address(this), poolId, spentGeth - gEthDonation);

if (self.TELESCOPE._isOracleActive()) {
    bytes32 dailyBufferKey = DataStoreUtils.getKey(
        block.timestamp - (block.timestamp % OracleUtils.ORACLE_PERIOD),
        "burnBuffer"
    );
    DATASTORE.addUintForId(poolId, dailyBufferKey, spentGeth);
}
```

Recommendation

Record the `spentGeth - gEthDonation` instead of just `spentGeth` in the burn buffer.

code/contracts/Portal/utils/StakeUtilsLib.sol:L831

```
DATASTORE.addUintForId(poolId, dailyBufferKey, spentGeth);
```

5.20 Boost calculation on fetchUnstake should not be using the cumBalance when it is larger than debt. Minor

Description

The Geode team implemented the 2-step withdrawal mechanism for the staked ETH. First, node operators signal their intent to withdraw the stake, and then the oracle will trigger all of the accounting of rewards, balances, and buybacks if necessary. Buybacks are what we are interested in at this time. Buybacks are performed by checking if the derivative asset is off peg in the Dynamic Withdrawal Pool contract. Once the debt is larger than some ignorable threshold an arbitrage buyback will be executed. A portion of the arbitrage profit will go to the node operator. The issue here is that when simulating the arbitrage swap in the `calculateSwap` call we use the cumulative un-stake balance rather than ETH debt preset in the DWP. In the case where the withdrawal cumulative balance is higher than the debt node operator will receive a higher reward than intended.

Examples

code/contracts/Portal/utils/StakeUtilsLib.sol:L1353-L1354

```
uint256 arb = withdrawalPoolById(DATASTORE, poolId)
    .calculateSwap(0, 1, cumBal);
```

Recommendation

Use the `debt` amount of ETH in the boost reward calculation when the cumulative balance is larger than the debt.

5.21 DataStore struct not having the _gap for upgrades. Minor

Description

Geode Finance codebase follows a structure where most of the storage variables are stored in the structs. You can see an example of that in the `Portal.sol`.

code/contracts/Portal/Portal.sol:L152-L154

```
DataStoreUtils.DataStore private DATASTORE;
GeodeUtils.Universe private GEODE;
StakeUtils.StakePool private STAKEPOOL;
```

It is worth mentioning that Geode contracts are meant to support the upgradability pattern. Given that information, one should be careful not to overwrite the storage variables by reordering the old ones or adding the new once not at the end of the list of variables when upgrading. The issue comes with the fact that structs seem to give a false sense of security making it feel like they are an isolated set of storage variables that will not override anything else. In reality, struts are just tuples that are expanded in storage sequentially just like all the other storage variables. For that reason, if you have two struct storage variables listed back to back like in the code above, you either need to make sure not to change the order or the number of variables in the structs other than the last one between upgrades or you need to add a `uint256[N] _gap` array of fixed size to reserve some storage slots for the future at the end of each struct. The Geode Finance team is missing the gap in the `DataStrore` struct making it non-upgradable.

code/contracts/Portal/utils/DataStoreUtilsLib.sol:L34-L39

```
struct DataStore {
    mapping(uint256 => uint256[]) allIdsByType;
    mapping(bytes32 => uint256) uintData;
    mapping(bytes32 => bytes) bytesData;
    mapping(bytes32 => address) addressData;
}
```

Recommendation

We suggest that gap is used in DataStore as well. Since it was used for all the other structs we consider it just a typo.

Appendix 1 - Code Quality Recommendations

After the engagement, we aggregated a few suggestions on the code quality and style.

A.1.1 Use Enums for different proposal types as well as states across the system.

For anyone unfamiliar with the codebase it will be very difficult to read the code that has a considerable amount of magic numbers. It would have helped if in the code we could see `PROPOSAL_TYPE.NEW_PLANET` instead of just 5 or `VALIDATOR_STATE.EXITED` instead of 3. We spent a long time cross-referencing the documentation, comments, and code to get familiar with all the magic numbers.

A.1.2 Consider not extending a struct with more than one library.

Finding whether the function is in `MaintainerUtils` or `DataStoreUtils` when DataStore is referenced is slow and frustrating for anyone unfamiliar with the codebase. Consider combining those libraries if possible. Potentially some functions from `MaintainerUtils` could go to `StakeUtils` and then the rest of them could migrate to `DataStoreUtils`. We do understand that this could be a relatively big change, but this is something to consider.

A.1.3 Data duplication

Right now in Geode data is stored in storage several times. Once when the proposal is created in a form of the `Proposal` struct and then the second time once the proposal is approved individually by the properties of the proposal, such as `TYPE`, `CONTROLLER` etc. This paired with the fact that data can be fetched from both of these places later on makes it hard to keep track of whether the state is correctly updated.

A.1.4 Access Controls are hard to find in the code

Access Control checks are spread across different files and are hard to find. That paired with the fact that there are a lot of embedded calls makes it hard to check access control modifiers. For example, initiating an operator takes calling Portal, then StakeUtils, then MaintainerUtils with different checks happening in different files without a strong or intuitive relationship to the file name.

A.1.5 Confusing Naming

When auditing the `gETH.sol` file it was confusing to see an `ORACLE_ROLE` that was assigned to Portal rather than to Oracle. This led to some incorrect initial conclusions about the codebase. We agree that it is possible that Portal can take on many different roles and since that is the case it would help to add inline comments outlining what actors should and should not have certain roles at the top of the file. https://github.com/ConsenSysDiligence/geodefi-audit-2022-10/blob/14433e9e94f57973f86c1a2a2b64169ccf147212/code/contracts/Portal/gETH.sol#L230

Appendix 2 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
code/contracts/Portal/MiniGovernance/MiniGovernance.sol	`3ef6d711e39ebda1498e5584286a76f95999abe1`
code/contracts/Portal/Portal.sol	`3e2fddb0a93c8923a2b33f28a8149c4d237e1a9f`
code/contracts/Portal/gETH.sol	`0c149c8ee0dac19469be4ea62c215b80d5ea2516`
code/contracts/Portal/gETHInterfaces/ERC20InterfacePermitUpgradable.sol	`1aa5cf595a4704d0a20c1937d694b8ffd51011d4`
code/contracts/Portal/gETHInterfaces/ERC20InterfaceUpgradable.sol	`8bf2ca0abaa10cb2f913c61bd14827deb6139b31`
code/contracts/Portal/helpers/ERC1155SupplyMinterPauser.sol	`d228841d7c6fdcdc64352288759e2091f3f65e8e`
code/contracts/Portal/utils/DataStoreUtilsLib.sol	`680b860433c4f3d7fd9441f884dad363f4f72e44`
code/contracts/Portal/utils/GeodeUtilsLib.sol	`db36c1cd3a615f80dc4a0d80d432add016083aa5`
code/contracts/Portal/utils/MaintainerUtilsLib.sol	`f171c4ea1e4817f77f3e6e3d0a66b501e8f9c9c6`
code/contracts/Portal/utils/OracleUtilsLib.sol	`7b606059bb4aacddf8e4661ee7102a3e399476b1`
code/contracts/Portal/utils/StakeUtilsLib.sol	`dd33fa8867447adb8787b97b73a3c34dde5773dc`
code/contracts/Portal/withdrawalPool/LPToken.sol	`66124ef6d237a2707fbadfe0f4853a738e34a088`
code/contracts/Portal/withdrawalPool/Swap.sol	`ca1fad6d30111952183987067744700db462904af`
code/contracts/Portal/withdrawalPool/utils/AmplificationUtils.sol	`f5afa0fb65a93e2262a4e2cdff7cc7b7d3d0eec5`
code/contracts/Portal/withdrawalPool/utils/MathUtils.sol	`897743675618bb7bbdaa306b4356160bb904f860`
code/contracts/Portal/withdrawalPool/utils/SwapUtils.sol	`2736852fd4b7d006cd79058c10de0893ef7f4ee3`

Appendix 3 - Disclosure

ConsenSys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as

investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., "third parties") – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.