Geode Withdrawal Contracts

Date	October 2023
Auditors	Sergii Kravchenko, Dominik Muhs

1 Executive Summary

This report presents the results of our engagement with **Geode Finance** to review the **Withdrawal contracts** for their **Liquid staking protocol**.

The review was conducted over one weeks, from **Oct 9, 2023** to **Oct 13, 2023**, by **Sergii Kravchenko** and **Dominik Muhs**. A total of 10 person-days were spent.

This is the third review performed for this protocol. The previous two can be found here: one, two.

The security assessment was focused on the withdrawal contract, a new addition to the protocol. The code for this contract is both easily understandable and thoroughly documented. As usual, the development team was highly responsive and collaborative during the evaluation process. Despite the contract’s relatively small codebase, the underlying logic is rather intricate and carries significant security implications, especially since it handles all user funds. During our review, we identified several critical and major issues. As a result, we recommended that the team allocate more time to conduct an in-depth internal system review, as our engagement had time constraints. We also provided some general suggestions to enhance the overall security of the contract’s architecture.

2 Scope

Our review focused on the commit hash `83ecdd1ba0ae2ceaeb1cca6977e8b3202c88efa3`. The scope only includes the withdrawal functionality, the list of files in scope can be found in the Appendix.

2.1 Objectives

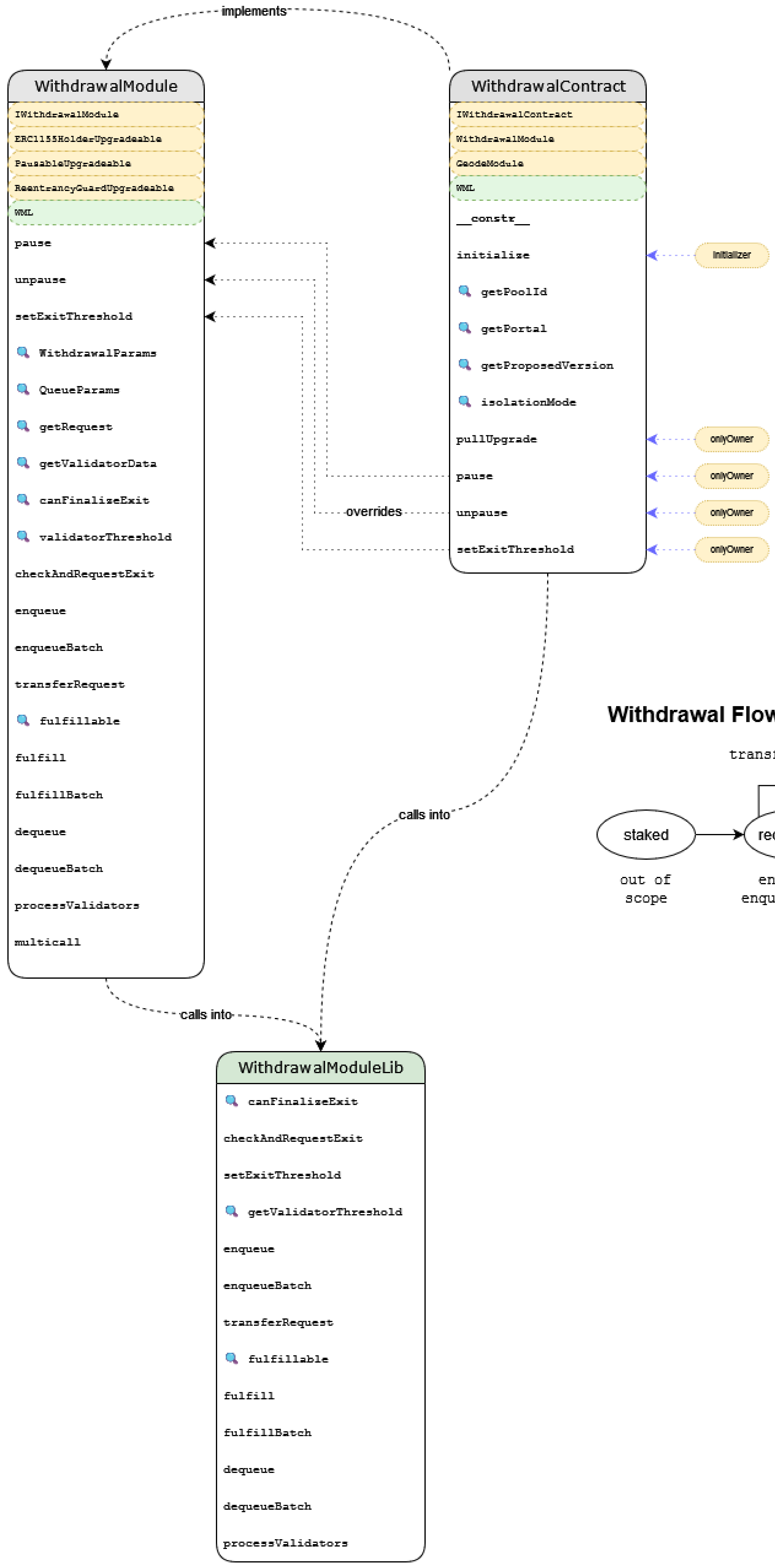
Together with the **Geode Finance** team, we identified the following priorities for our review:

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.

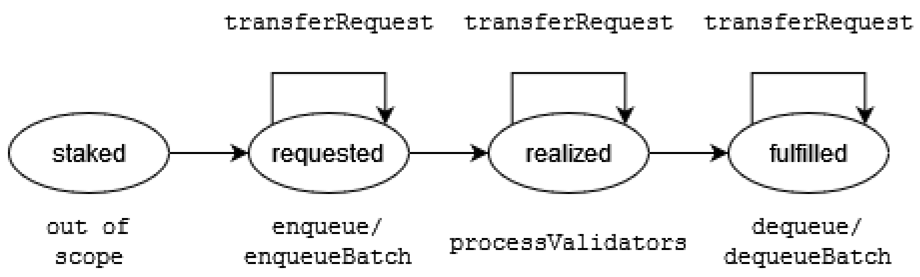
3 System Overview

Overall, the system represents a complex liquid staking protocol that allows users to create custom staking pools. In the scope of this review, we inspect a new part of the code that manages staking withdrawals.

The code’s withdrawal part consists of the `WithdrawalContract`, a proxy for which is deployed for every staking pool. This contract inherits `WithdrawalModule`, which contains most of the public functions, and most of the logic is stored in the `WithdrawalModuleLib` library.



Withdrawal Flow



4 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under audit. The relevant actors are listed below with their respective abilities:

- Pool Operator
  - Pause/Unpause the pool affecting it's isolation mode
  - Set an exit threshold between 60% and 100%
  - Upgrade the pool to a new, authorized version
- User
  - Enqueue a withdrawal requests
  - Enqueue a batch of withdrawal requests
  - Transfer ownership of a withdrawal request to another owner
  - Trigger the fulfillment of a request
  - Trigger the fulfillment of a request batch
  - Dequeue and finalize a request
  - Dequeue and finalize a request batch
- Anyone
  - Trigger the `processValidators` function to pay out rewards

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 The `processValidators` Can Process Validators From Other Pools Critical

Description

The `WithdrawalContract` has the `processValidators` function that helps the contract to keep track of all the validators and the funds that are received from them. These funds are coming in the form of staking fees and withdrawn balances of exited validators.

contracts/Portal/modules/WithdrawalModule/libs/WithdrawalModuleLib.sol:L712-L736

```
function processValidators(
  PooledWithdrawal storage self,
  bytes[] calldata pubkeys,
  uint256[] calldata beaconBalances,
  uint256[] calldata withdrawnBalances,
  bytes32[][] calldata balanceProofs
) external {
  uint256 pkLen = pubkeys.length;
  require(
    pkLen == beaconBalances.length &&
      pkLen == withdrawnBalances.length &&
      pkLen == balanceProofs.length,
    "WML:invalid lengths"
  );

  bytes32 balanceMerkleRoot = self.PORTAL.getBalancesMerkleRoot();
  for (uint256 i; i < pkLen; ) {
    // verify balances
    bytes32 leaf = keccak256(
      bytes.concat(keccak256(abi.encode(pubkeys[i], beaconBalances[i], withdrawnBalances[i])))
    );
    require(
      MerkleProof.verify(balanceProofs[i], balanceMerkleRoot, leaf),
      "WML:NOT all proofs are valid"
    );
```

Anyone can submit a set of validators with their current states to process the funds coming from these validators. The caller has to provide proof that the validators exist, have the correct balances, and belong to this withdrawal contract. The most recent root of the Merkle tree with all the validators and all the balances are stored as `self.PORTAL.getBalancesMerkleRoot()` in the `Portal` contract. So, the security of this contract relies on the fact that all the validators are stored in this tree, including the exited ones, and that their balances are up to date. The security also relies on the fact that no other “fake” validators or validators with other withdrawal credentials are stored in this Merkle tree.

The issue is that one Merkle tree contains the validators from different Geode pools with different withdrawal contracts. So, one validator can be accounted for in every `WithdrawalContract` of every pool, which will create a lot of “fake” ETH accounted in the contract and break all the accounting.

Recommendation

Ensure only valid validators can be processed in the `WithdrawalContract` and all the exited validators are still in the tree.

5.2 Staking Deposits Are Locked in the `WithdrawalContract` Critical

Description

After a validator exits, the `processValidators` is called to process the retrieved funds. The initial deposit of 32 ETH should be returned to the WithdrawalContract along with the rewards if not slashed. When processing, the following code is executed:

contracts/Portal/modules/WithdrawalModule/libs/WithdrawalModuleLib.sol:L748-L764

```
self.validators[pubkeys[j]].beaconBalance = beaconBalances[j];
self.validators[pubkeys[j]].withdrawnBalance = withdrawnBalances[j];

if (beaconBalances[j] == 0) {
  // exit
  processed += _distributeFees(
    self,
    pubkeys[j],
    withdrawnBalances[j],
    oldWitBal + DCL.DEPOSIT_AMOUNT
  );
  _finalizeExit(self, pubkeys[j]);
} else {
  // check if should request exit
  processed += _distributeFees(self, pubkeys[j], withdrawnBalances[j], oldWitBal);
  commonPoll = checkAndRequestExit(self, pubkeys[j], commonPoll);
}
```

This code is supposed to process two scenarios: complete exit and partial withdrawals of the rewards. When the `beaconBalances[j]` is zero, the validator is excited, and the deposit is returned. In that case, the 32 ETH initial deposit is added to the old processed balance (`oldWitBal + DCL.DEPOSIT_AMOUNT`) to avoid paying fees to the validator and the protocol from that amount:

contracts/Portal/modules/WithdrawalModule/libs/WithdrawalModuleLib.sol:L655-L672

```
function _distributeFees(
  PooledWithdrawal storage self,
  bytes memory pubkey,
  uint256 reportedWithdrawn,
  uint256 processedWithdrawn
) internal returns (uint256 extra) {
  if (reportedWithdrawn > processedWithdrawn) {
    uint256 profit = reportedWithdrawn - processedWithdrawn;
    Validator memory val = self.PORTAL.getValidator(pubkey);

    uint256 poolProfit = (profit * val.poolFee) / PERCENTAGE_DENOMINATOR;
    uint256 operatorProfit = (profit * val.operatorFee) / PERCENTAGE_DENOMINATOR;
    extra = (profit - poolProfit) - operatorProfit;

    self.PORTAL.increaseWalletBalance{value: poolProfit}(val.poolId);
    self.PORTAL.increaseWalletBalance{value: operatorProfit}(val.operatorId);
  }
}
```

The issue is that in the `_distributeFees` function, the contract pays fees and acknowledges the rest of the funds to be "realized" (will be added to `self.queue.realized`) to be distributed to the withdrawal queue. So these 32 ETH (`DCL.DEPOSIT_AMOUNT`) won't be acknowledged anywhere and will be stuck in the contract.

5.3 Anyone Can Enforce a New Validator to Exit Before They Processed Major

Description

When the validator becomes active, it's not added to the `WithdrawalContract` by default. Because of that, its initial `beaconBalance` will be zero and will remain zero before the first `processValidators` call that includes this validator. While that's the case, anyone can call the `checkAndRequestExit` function:

contracts/Portal/modules/WithdrawalModule/libs/WithdrawalModuleLib.sol:L241-L264

```
function checkAndRequestExit(
  PooledWithdrawal storage self,
  bytes calldata pubkey,
  uint256 commonPoll
) public returns (uint256) {
  (uint256 threshold, uint256 beaconBalancePriced) = getValidatorThreshold(self, pubkey);
  uint256 validatorPoll = self.validators[pubkey].poll;

  if (commonPoll + validatorPoll > threshold) {
    // meaning it can request withdrawal

    if (threshold > validatorPoll) {
      // If Poll is not enough spend votes from commonPoll.
      commonPoll -= threshold - validatorPoll;
    } else if (validatorPoll > beaconBalancePriced) {
      // If Poll is bigger than needed, move the extra votes instead of spending.
      commonPoll += validatorPoll - beaconBalancePriced;
    }

    _requestExit(self, pubkey);
  }

  return commonPoll;
}
```

The `getValidatorThreshold` function will return zero for that validator, so the `checkAndRequestExit` will succeed, and the validator will be forced to exit:

contracts/Portal/modules/WithdrawalModule/libs/WithdrawalModuleLib.sol:L293-L300

```
function getValidatorThreshold(
  PooledWithdrawal storage self,
  bytes calldata pubkey
) public view returns (uint256 threshold, uint256 beaconBalancePriced) {
  uint256 price = self.gETH.pricePerShare(self.POOL_ID);
  beaconBalancePriced = ((self.validators[pubkey].beaconBalance * gETH_DENOMINATOR));
  threshold = (beaconBalancePriced * self.EXIT_THRESHOLD) / PERCENTAGE_DENOMINATOR / price;
  beaconBalancePriced = beaconBalancePriced / price;
```

Recommendation

Make sure all validators are "initialized" within the `WithdrawalContract` before any actions can be performed with them.

5.4 Anyone Can Force Every Validator to Exit Major

Description

Let's look at the following the following function in the library (`checkAndRequestExit`):

contracts/Portal/modules/WithdrawalModule/libs/WithdrawalModuleLib.sol:L241-L264

```
function checkAndRequestExit(
  PooledWithdrawal storage self,
  bytes calldata pubkey,
  uint256 commonPoll
) public returns (uint256) {
  (uint256 threshold, uint256 beaconBalancePriced) = getValidatorThreshold(self, pubkey);
  uint256 validatorPoll = self.validators[pubkey].poll;

  if (commonPoll + validatorPoll > threshold) {
    // meaning it can request withdrawal

    if (threshold > validatorPoll) {
      // If Poll is not enough spend votes from commonPoll.
      commonPoll -= threshold - validatorPoll;
    } else if (validatorPoll > beaconBalancePriced) {
      // If Poll is bigger than needed, move the extra votes instead of spending.
      commonPoll += validatorPoll - beaconBalancePriced;
    }

    _requestExit(self, pubkey);
  }

  return commonPoll;
}
```

This function forces a chosen validator to exit if the `commonPoll + validatorPoll` is large enough. The `checkAndRequestExit` function of the library returns the new updated amount of `commonPoll` that is supposed to be written to the `WITHDRAWAL.queue.commonPoll` field. That is properly done when used in other parts of the code except for the direct call by the user here:

contracts/Portal/modules/WithdrawalModule/WithdrawalModule.sol:L201-L203

```
function checkAndRequestExit(bytes memory pubkey) external virtual override returns (uint256) {
  return WITHDRAWAL.checkAndRequestExit(pubkey, WITHDRAWAL.queue.commonPoll);
}
```

So once the `commonPoll` is large enough, a malicious user can call this function for every validator and force all of them to exit.

Recommendation

Update all the relevant variables inside the `checkAndRequestExit` function of the library or the module.

5.5 Operators Can “Attack” Other Operators Medium

Description

There is competition in the system between different operators. Most are incentivized to create as many validators as possible, and nobody is incentivized to exit when the exit `Queue` is increasing. However, someone has to exit at some point, and there are two mechanisms in place to define which validator should be kicked out:

1. When `gETH` holders want to exit, which means burn `gETH` and get underlying `ETH` in return, they can call the `enqueue` function of the `WithdrawalContract`. As an option, they can choose which validator they want to exit from staking. That creates a new attack vector for operators to target each other. An operator can deposit 32 ETH to the pool and then try to exit with `WithdrawalContract`; there are no additional fees that they are paying for that process. While exiting, they can choose a validator of the competitor, effectively kicking them from earning the profit and taking their spot.
2. If the person who is exiting doesn’t choose a specific validator to kick from the system, once the queue is large enough, anyone can kick any validator. A function called `checkAndRequestExit` can be called by anyone with an arbitrary parameter of which validator should exit next. That creates a race condition for the operators because everyone is incentivized to force the opponent to exit before the opponent does the same to them.

Recommendation

Consider creating a mechanism that doesn’t generate race conditions and will not allow operators to attack each other.

5.6 Validators Can Have Increasing `withdrawnBalances` Even After Exiting Medium

Description

Here is the code that processes balance and withdrawal updates from the oracle for each validator:

contracts/Portal/modules/WithdrawalModule/libs/WithdrawalModuleLib.sol:L748-L764

```
self.validators[pubkeys[j]].beaconBalance = beaconBalances[j];
self.validators[pubkeys[j]].withdrawnBalance = withdrawnBalances[j];

if (beaconBalances[j] == 0) {
  // exit
  processed += _distributeFees(
    self,
    pubkeys[j],
    withdrawnBalances[j],
    oldWitBal + DCL.DEPOSIT_AMOUNT
  );
  _finalizeExit(self, pubkeys[j]);
} else {
  // check if should request exit
  processed += _distributeFees(self, pubkeys[j], withdrawnBalances[j], oldWitBal);
  commonPoll = checkAndRequestExit(self, pubkeys[j], commonPoll);
}
```

This code acts under the assumption that once the `beaconBalances` of a validator become zero, the `withdrawnBalances` of this validator can't increase anymore. However, according to the official staking documentation, this statement is wrong. There is an exception when more funds are deposited to the exited validator. These funds are then automatically transferred to the `WithdrawalContract`.

5.7 Ownership Transfers on Dequeued Requests Minor

Description

After a request has been completed and removed from the queue, the owner retains the capability to invoke `WithdrawalModuleLib.transferRequest`. This allows the owner to transfer the concluded withdrawal request to a different owner:

contracts/Portal/modules/WithdrawalModule/libs/WithdrawalModuleLib.sol:L430-L442

```
function transferRequest(
  PooledWithdrawal storage self,
  uint256 index,
  address newOwner
) external {
  address oldOwner = self.requests[index].owner;
  require(msg.sender == oldOwner, "WML:not owner");
  require(newOwner != address(0), "WML:cannot transfer to zero address");

  self.requests[index].owner = newOwner;

  emit RequestTransfer(index, oldOwner, newOwner);
}
```

Recommendation

We recommend disallowing any modifications to withdrawal requests that have already been processed.

5.8 `fulfillable` View Parameters Can Be Spoofed Minor

Description

The `WithdrawalModuleLib.fulfillable` function, marked as an external view, is also used internally. The parameters `qRealized` and `qFulfilled` can be passed as arbitrary values. This could mislead users and off-chain systems into believing that a request in the queue is ready for fulfillment and ultimately submit transactions destined to fail.

contracts/Portal/modules/WithdrawalModule/libs/WithdrawalModuleLib.sol:L457-L481

```
function fulfillable(
  PooledWithdrawal storage self,
  uint256 index,
  uint256 qRealized,
  uint256 qFulfilled
) public view returns (uint256) {
  if (qRealized > qFulfilled) {
    uint256 rTrigger = self.requests[index].trigger;
    uint256 rSize = self.requests[index].size;
    uint256 rFulfilled = self.requests[index].fulfilled;

    uint256 rFloor = rTrigger + rFulfilled;
    uint256 rCeil = rTrigger + rSize;

    if (qRealized > rCeil) {
      return rSize - rFulfilled;
    } else if (qRealized > rFloor) {
      return qRealized - rFloor;
    } else {
      return 0;
    }
  } else {
    return 0;
  }
}
```

Recommendation

The function should source its values directly from the request corresponding to the user-supplied index for increased, rather than relying on externally provided parameters.

5.9 Enqueueing Functions Do Not Return the Request Index

Description

In the Geode withdrawal process, the request ID of a withdrawal is determined based on its index in the `PooledWithdrawal` storage struct. This ID is important, as it must be used in subsequent function calls to pinpoint the correct withdrawal request throughout its lifecycle:

contracts/Portal/modules/WithdrawalModule/libs/WithdrawalModuleLib.sol:L74-L83

```
struct PooledWithdrawal {
  IgETH gETH;
  IPortal PORTAL;
  uint256 POOL_ID;
  uint256 EXIT_THRESHOLD;
  Queue queue;
  Request[] requests;
  mapping(bytes => ValidatorData) validators;
  uint256[9] __gap;
}
```

The current system design does not return this ID in any enqueueing functions, making it challenging to reference in future operations:

Recommendation

We recommend returning the withdrawal request ID (or IDs if multiple) upon submitting a new withdrawal request. This will enable other smart contracts and off-chain software to quickly determine their submissions' position in the queue. Consequently, it will simplify future function calls and interactions with the system.

5.10 Add a State Machine for Withdrawal Requests

Description

Withdrawal requests undergo a specific lifecycle. The management and enforcement of this lifecycle can be improved if each request encapsulates its present state.

Recommendation

We recommend that every smart contract function that modifies the request be treated as a state transition function. By conducting the necessary checks at the start of each function, the code becomes clearer, more explicit, and reduces the possibility of bugs related to unauthorized state transitions.

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
`contracts/Portal/modules/WithdrawalModule/WithdrawalModule.sol`	`8e795624e593653f6f67cc9df3298184f021e950`
`contracts/Portal/modules/WithdrawalModule/libs/WithdrawalModuleLib.sol`	`3dc393cb808a3931242348ce025b566c21271113`
`contracts/Portal/packages/WithdrawalContract.sol`	`2b6d16d197f3ed6c6378daf1859843a7159cb39b`

Appendix 2 - Disclosure

A.2.1 Purpose of Reports

A.2.2 Links to Other Web Sites from This Web Site

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.

A.2.3 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.